软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3332−3354 [doi: 10.13328/j.cnki.jos.006923]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 eBPF 的内核堆漏洞动态缓解机制*

王子成[1,2], 郭迎港[1,2], 钟炳南[1,2], 陈越琦[3], 曾庆凯[1,2]

[1](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机科学与技术系, 江苏 南京 210023)

[3](Department of Computer Science, University of Colorado Boulder, Boulder 80309, USA)

通信作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘　要: 内核堆漏洞是目前操作系统安全的主要威胁之一, 用户层攻击者通过触发漏洞能够泄露或修改内核敏感信息, 破坏内核控制流, 甚至获取 root 权限. 但是由于漏洞的数量和复杂性剧增, 从漏洞首次被报告到开发者给出修复补丁 (patch) 往往需要较长时间, 而内核现阶段采用的缓解机制均能被稳定绕过. 为此提出一种基于 eBPF 的内核堆漏洞动态缓解框架, 用于在修复时间窗口中降低内核安全风险. 动态缓解框架采取数据对象空间随机化策略, 在每次分配时为漏洞报告中涉及的数据对象分配随机地址, 并充分利用 eBPF 的动态、安全特性将空间随机化对象在运行时注入内核, 使得攻击者无法准确放置攻击负载, 堆漏洞几乎无法被利用. 评估 40 个真实内核堆漏洞, 并收集 12 个绕过现有缓解机制的攻击程序进行进一步分析和实验, 证实动态缓解框架提供充足的安全性. 性能测试表明, 即使在严苛情况下, 大量分配的 4 类数据对象仅对系统造成约 1% 的性能损耗和可以忽略不计的内存损耗, 同时增加保护对象的数量几乎不引入额外性能损耗. 所提机制对比相关工作适用范围更广, 安全性更强, 而且无需安全专家发布的漏洞补丁, 可以根据漏洞报告生成缓解程序, 具备广阔应用前景.

关键词: 系统安全; 漏洞缓解; eBPF

中图法分类号: TP311

中文引用格式: 王子成, 郭迎港, 钟炳南, 陈越琦, 曾庆凯. 基于eBPF的内核堆漏洞动态缓解机制. 软件学报, 2024, 35(7): 3332–3354. http://www.jos.org.cn/1000-9825/6923.htm

英文引用格式: Wang ZC, Guo YG, Zhong BN, Chen YQ, Zeng QK. Dynamic Mitigation Solution Based on eBPF Against Kernel Heap Vulnerabilities. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3332–3354 (in Chinese). http://www.jos.org.cn/1000-9825/6923.htm

Dynamic Mitigation Solution Based on eBPF Against Kernel Heap Vulnerabilities

WANG Zi-Cheng[1,2], GUO Ying-Gang[1,2], ZHONG Bing-Nan[1,2], CHEN Yue-Qi[3], ZENG Qing-Kai[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[3](Department of Computer Science, University of Colorado Boulder, Boulder 80309, USA)

Abstract: Kernel heap vulnerability is currently one of the main threats to operating system security. User-space attackers can leak or modify sensitive kernel information, disrupt kernel control flow, and even gain root privilege by triggering a vulnerability. However, due to the rapid increase in the number and complexity of vulnerabilities, it often takes a long time from when a vulnerability is first reported to when the developer issues a patch, and kernel mitigation mechanisms currently adopted are usually steadily bypassed. Therefore, this study proposes an eBPF-based dynamic mitigation framework for kernel heap vulnerabilities, so as to reduce kernel security risks during the time window fixing. The framework adopts data object space randomization to assign random addresses to the data objects involved in

* 基金项目: 国家自然科学基金 (61772266, 61431008)
收稿时间: 2022-09-05; 修改时间: 2022-11-18, 2023-01-01; 采用时间: 2023-02-14; jos 在线出版时间: 2023-08-30
CNKI 网络首发时间: 2023-08-31

vulnerability reports at each allocation. In addition, it takes full advantage of the dynamic and secure features of eBPF to inject space-randomized objects into the kernel during runtime, so the attacker cannot place any attack payload accurately, and the heap vulnerabilities are almost unexploitable. This study evaluates 40 real kernel heap vulnerabilities and collects 12 attacks that bypass the existing mitigation mechanisms for further analysis and tests. As a result, it verifies that the dynamic mitigation framework provides sufficient security. Performance tests show that even under severe conditions, the four types of data objects only cause performance loss of about 1% and negligible memory loss to the system, and there is almost no additional performance loss when the number of protected objects increases. Compared with related work, the mechanism in this study has a wider scope of application and stronger security, and it does not require vulnerability patches issued by security experts. Furthermore, it can generate mitigation procedures according to vulnerability reports and has a broad application prospect.

Key words: system security; vulnerability mitigation; eBPF

内存损坏漏洞 (memory corruption) 是由不安全的 C/C++语言编写的程序的主要安全威胁之一, 有报告显示近年来内存损坏漏洞的主要形式为堆漏洞[1,2], 常见的包括空间型 (spatial) 溢出漏洞[3]和时间型 (temporal) 释放后使用漏洞[4]. 攻击者能够通过“堆风水”攻击手段对应用程序堆进行布局, 将攻击目标数据对象放置在漏洞数据对象对应位置, 进而触发漏洞破坏系统的机密性和完整性[5]. 例如, 利用溢出漏洞获取邻接数据对象的机密信息、修改敏感数据, 或将攻击负载放置在释放后使用漏洞被释放的地址, 并解引用悬空指针触发[6,7]. 操作系统 Linux 内核的小数据对象分配器 slab 也受到相同的威胁, 近 3 年公开的 57 个内核漏洞中有 40 个属于堆漏洞, 用户层的攻击者通过一系列精心构造的系统调用, 触发内核堆的内存损坏漏洞获取核心权限, 进而控制整个系统[8,9].

然而从发现漏洞到开发者给出补丁 (patch) 并应用修复往往需要较长时间, 因为近些年来漏洞数量和复杂性剧增, 但确认并修复漏洞要求开发者必须同时理解漏洞的根源和发现漏洞子系统的功能, 否则对漏洞的修复可能无效或影响内核运行效率, 这导致 Linux 中平均一个漏洞存在的时间超过 1 300 天[10], 存在漏洞的内核难以保证系统安全性, 因此, 在这一时间窗口中内核需要缓解 (mitigation)[11,12]机制降低内核漏洞的安全威胁性.

本文研究对比了目前开源 Linux 内核采用的堆漏洞缓解机制, 发现其中存在安全保障不足、无法灵活选择保护对象两类缺陷, 导致漏洞缓解效果未达到预期, 无法应对真实环境中复杂多样的内核堆漏洞. 1) 安全保障不足是指现有缓解机制存在确定的绕过手段, 例如 freelist 随机化[13]和 slab quarantine[14]可以被堆喷射绕过、数据结构成员随机化[15]种子易于被获取、autoslab 难以抵抗同类型对象攻击[16]等; 2) 无法灵活选择保护对象的原因有两方面, 一方面是缓解机制依赖编译前静态配置, 无法在系统运行时灵活增减, 另一方面部分缓解机制在原理层面仅针对小范围安全风险, 例如 freelist 指针模糊化[17]仅保护元数据、成员布局随机化和 autoslab 仅对固定成员数量和长度的数据结构生效, 对于可伸缩和缓冲区对象均无能为力等.

有效的内核堆漏洞缓解机制应该达到以下目标: 1) 无法被确定性攻击; 2) 不依赖静态配置, 支持运行时对任意存在威胁的数据对象实施; 3) 性能开销可以被操作系统接受. 为满足以上目标, 动态缓解机制需要合理设计更复杂的随机化方案, 提供更强的安全保障, 但不引入大量开销, 同时应该具备部分类似热补丁的动态载入的特性, 不影响系统正常运行.

本文提出了一种基于 eBPF[18]的内核堆漏洞动态缓解框架 ERA (eBPF-based randomization allocator), 用于在内核堆漏洞被修复前的时间窗口内, 动态、高效缓解操作系统面临的安全风险, 既无需等待安全专家发布修复补丁, 又避免了内核探测追踪引入的巨大开销, 而且可以根据漏洞报告中的安全威胁数据对象生成缓解程序, 具备强大的易用性. ERA 动态缓解框架相比其他内核缓解机制, 不存在确定性的绕过手段, 为内核提供了更充分的安全保障; 同时可以在运行时选择任意数据对象进行保护, 不局限于运行前配置和数据结构类型.

ERA 采用了数据对象空间随机化方案提供了充足的安全保障. 每次内存分配时在空间更大的 slab cache 中请求超过所需的内存, 并在分配的内存中随机放置数据对象. 这使得每一个空间随机化的数据对象均不存在有关联的地址, 攻击者无从预测数据对象所在地址和偏移量, 很难准确放置攻击负载, 内核堆漏洞利用的难度急剧放大, 其他内核缓解机制中提及的堆喷射、同类型对象攻击、cross-cache[19]攻击等绕过手段均无法突破 ERA 防护.

ERA 能够灵活地在运行时对任意数据对象施加保护, 足以应对真实环境中复杂多变的堆漏洞. 首先 ERA 充分利用 eBPF 动态、安全特性, 可以在运行时将经过空间随机化的数据对象注入内核中, 无需预先配置或重新编

译内核. 其次, ERA 采用的数据对象空间随机化不受数据对象成员的限制, 包括可伸缩对象、缓冲区对象在内的任意数据对象, 均能够在内核地址空间中隐藏自身位置, 相比其他内核堆漏洞缓解机制具备更广阔的适用范围.

本文贡献如下.

(1) 提出了一种内核堆漏洞动态缓解框架. 采用动态的数据对象空间随机化模型, 能够在内核堆漏洞修补前的攻击窗口降低安全威胁, 同时利用 eBPF 技术动态、灵活、安全地将空间随机化应用于任意数据对象, 具备易用性、灵活性、安全性和高效性.

(2) 研究了支持内核堆漏洞动态缓解的关键技术. 采用静态分析技术获取了精确的分配点上下文, 控制数据对象随机化的开启和关闭; 设计了更加安全高效的数据对象空间随机化方案, 解决了现有缓解机制的缺陷; 充分利用 eBPF 提供的动态、安全特性将被保护的数据对象重新注入内核, 不引入额外风险.

(3) 实现了原型系统 ERA, 从收集到的漏洞报告出发, 提取存在安全威胁的数据对象并生成 eBPF 程序载入内核, 缓解内核堆漏洞安全风险.

(4) 实验验证了 ERA 的有效性、高效性、创新性和易用性. 首先验证 ERA 的有效性, 本文评估了 40 个具有代表性的内核堆漏洞, 并选取其中 12 个漏洞的 EXP 攻击程序进一步确认 ERA 具备快速缓解堆利用风险的能力; 其次为验证 ERA 的高效性, 本文选取了内核中大量分配的数据对象, 并对比了原始内核 (编译时成员随机化), 仅增加了约 1% 的性能开销, 同时内存消耗也因为回收机制微不足道; 此外, 与成员随机化的优化工作 SALAD 和 POLAR 的对比结果凸显了 ERA 机制的创新性; 最后, 本文确认了 ERA 的易用性, 即使非安全专家的系统管理员, 也无需等待漏洞修补补丁, 提前缓解内核堆漏洞利用风险.

1 背 景

1.1 威胁模型

本文重点针对操作系统内核中的堆分配器 (slab) 内存损坏漏洞, 主要包括溢出 (out-of-bound) 和释放后使用 (use-after-free) 两类. 堆漏洞是目前内核的主要安全威胁之一, 本文统计了近 3 年来公开的 57 个内核漏洞, 其中有 40 个属于堆漏洞. 其他类型的内核漏洞, 例如 TOCTTOU、竞争和未初始化等作为攻击的一环[19,20], 最后可能也会诱发或辅助堆漏洞利用. 因此本文假设攻击者了解内核漏洞, 但不具备核心 root 权限, 系统开启了 freelist 随机化、指针模糊化、naïve check 和数据结构成员随机化等缓解机制提升堆漏洞攻击难度, 开启了 SMAP、SMEP、NX 等硬件机制避免 ret2usr 和代码注入攻击, 测试的 EXP (exploit 攻击利用程序) 通过堆风水、堆喷射一系列攻击绕过上述安全防护, 试图在用户空间通过执行系统调用触发漏洞, 读取或修改内核信息, 进而获取 root 权限.

但是暂时不考虑的 ERA 方案依赖的 eBPF 机制本身存在的漏洞, 并且假设 ERA 载入前漏洞没有被触发. 内核中不存在 rootkits 形式的恶意代码[21], 不具备任意写原语任意破坏内核数据, 系统的运行平台也不存在硬件漏洞或恶意硬件[22,23].

1.2 内核堆内存漏洞利用

Linux 内核使用 slab/slub 分配器实现用户程序堆的功能, 为内核动态分配小块内存 (小于 2 页, 8 192 字节), 原理是从 buddy 系统中分配连续的、整页的内存, 划分成大小相同的数据对象进行分配, 其中相同大小的一类数据对象由一个 slab cache 管理, 每个 slab cache 中的数据对象以链表 (freelist) 的形式连接, 分配时从 freelist 中取出数据对象, 释放后再重新加入 freelist.

为了便于数据管理, 内核根据分配数据的大小划分了 13 个通用 slab cache, 分别是 8、16、32、64、96、128、192、256、512、1 024、2 048、4 096 和 8 192 字节, 常见的内核堆内存损坏漏洞通常发生在这些 cache 中间. 除此之外, 内核还为 task_struct、cred 和 inode 等携带重要信息的安全敏感数据对象分配了专用的 slab cache 与通用 cache 隔离, 由于只存在 1 类数据对象, 因此漏洞利用风险相对较小.

● 溢出利用

空间型内存损坏漏洞的根源是指针 ptr 指向的地址范围 [ptr+$\text{offset}_{\text{ptr}}$, ptr+$\text{offset}_{\text{ptr}}$+$\text{size}_{\text{access}}$−1], 超出了当前指

针的合法范围 [obj, obj+$size_{obj}$–1], 让攻击者获得了解引用超出指针合法范围的能力[3,24]. 基本原理模型如图 1 所示.

在大多数空间型堆内存损坏漏洞利用的过程中, 攻击者会想办法将受害者数据对象 (victim object) 放在发生溢出的指针指向的漏洞数据对象 (vulnerable object) 后面, 控制发生溢出的指针读写受害者数据对象. 受害者数据对象与漏洞数据对象大小一致, 在同一 slab cache 中, 而且会包含机密信息或安全敏感的代码指针和权限[25].

如图 2 所示, 本文选取了 netfilter 包过滤器中 CVE-2022-34918 作为实际案例, 在该漏洞中, 5 347 行的 elem 缓冲区的分配大小发生了整数溢出, 使得分配了小于预期的数据对象, 而之后通过 5 359 行的 memcpy 函数溢出写了超过范围的数据对象. 攻击者使用堆喷射和堆风水等攻击技术, 如图 3 所示, 通过大量分配 user_key_payload 数据对象, 绕过现有的内核缓解机制, 将其放置在发生溢出的 elem 数据对象之后, 攻击者控制溢出指针修改 user_key_payload 的 data_len 长度, 恶意扩大 user_key_payload 的读取范围, 实现任意读攻击. 此外, slab 分配器元数据 freelist 由于保存在未分配数据对象中, 也会作为溢出攻击的目标之一, 但开发者使用了 freelist 模糊机制[17], 通过 XOR 加密的方式进行了缓解.

对于一个区间的访问行为 (ptr, $offset_{ptr}$, $sizec_{access}$)
被访问区间 (obj, $size_{obj}$)
如果 (ptr+$offset_{ptr}$+$size_{acess}$>(obj+$size_{obj}$), 则存在一个溢出漏洞
即, $offset_{ptr}$+$size_{access}$>$size_{obj}$

图 1 溢出漏洞模型

```
5339    void*nft_set_elem_init(...)
|  |  |    //整数溢出，分配长度小于预期
5347        elem=kzalloc(set->ops->elemsize+tmpl->len, gfp);
5351        ext=nft_set_elem_ext(set, elem);
|  |  |  |    //此处elem地址+偏移量发生溢出
5359            memcpy(nft_set_ext_data(ext), data, set->dlen);
```

图 2 CVE-2022-34918 漏洞根源代码片段

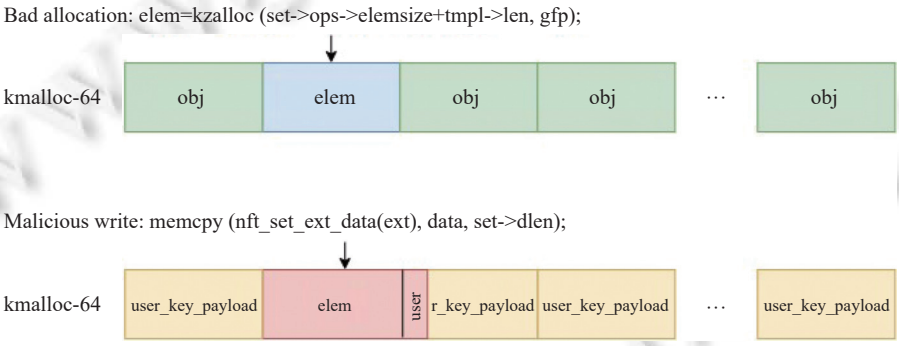


图 3 CVE-2022-34918 信息泄漏内核堆喷射攻击布局

● 释放后使用利用

时间型内存损坏漏洞的根源在于 (悬空) 指针 p 指向了被释放的数据对象 O, 释放前 O 的起始地址为 m, 范围是 [m, $m+size-1$]. 当程序解引用悬空指针 p 时能够泄露内核机密数据, 或触发任意由攻击者构造的攻击负载[26]. 基本原理模型如图 4 所示.

在时间型堆内存损坏漏洞的攻击中, 攻击者会在悬空指针指向的数据对象释放后, 快速将精心设计的攻击负载放置在被释放的位置, 在放置成功后解引用悬空指针触发攻击. 与空间型漏洞类似, 悬空指针指向的漏洞数据对象 (vulnerable object) 和攻击者选择的攻击负载喷射对象 (spray object) 也大小相同, 在同一个 slab cache 中, 喷射对象中也包含机密信息或安全敏感的代码指针和权限.

如图 5 所示, 本文选择了网络包调度中的 CVE-2021-3715 作为实际案例, 在该漏洞中开发者错将新分配的 route4_filter 指针在 539 行删除, 而本应删除的旧 route4_filter 指针保留在 route4_bucket 中, 当旧 route4_filter 数据结构在 554 行释放时, 被错误保留的指针成为悬空指针.

```
一个指针变量p是悬空指针, 当且仅当,
一个堆数据对象O, 地址的范围是：∀m, size: [m, m+size−1], 且这个对象已经被释放,
指针变量p∈[m, m+size−1]
```

图 4　释放后使用漏洞模型

```
465 static int route4_change(...)
493 f=kzalloc(sizeof(struct route4_fliter), GFP_KERNEL);
536 for (pfp=rtnl_dereference(*fp); pfp;
|     |     |     |      fp=&pfp->next, pfp=rtnl_dereference(*fp)) {
539             if (pfp=f) {
|     |     |     |      *fp=f->next;
|     |     |     } // f为新分配route4_filter, 破错误从bucket中删除
|     |
|     |     // fold为本应删除的route4_filter对象, 指针被错误保存,此处释放了fold内存
554       tcf_queue_work(&fold->rworkr,route4_delete_filter_work);
```

图 5　CVE-2021-3715 漏洞根源代码片段

攻击者选择了常见的可伸缩数据对象[27]的 msg_msg 实现信息泄露攻击. msg_msg 数据对象在悬空指针产生后快速填充了被释放 route4_filter 的位置. 之后攻击者调用 kfree 函数, 发起释放后使用攻击, 释放了悬空指针指向的刚刚填充的 msg_msg 数据对象, 因为 msg_msg 能够完整读取自身的 security 成员后面的全部内存, 因此可以将任意携带安全敏感信息的数据对象再次填充到 msg_msg 被恶意释放后的位置, 例如可以直接通过 route4_filter.rwork.func 成员泄露 route4_delete_filter_work 函数地址, 绕过 KASLR 地址空间随机化泄露内核代码基地址. 为了提升攻击的稳定性, 攻击者同样使用堆喷射和堆风水等攻击技术, 如图 6 所示, 预先大量分配 route4_filter 数据对象, 保证 msg_msg 数据对象能够成功覆盖到至少 1 个悬空指针指向的位置.

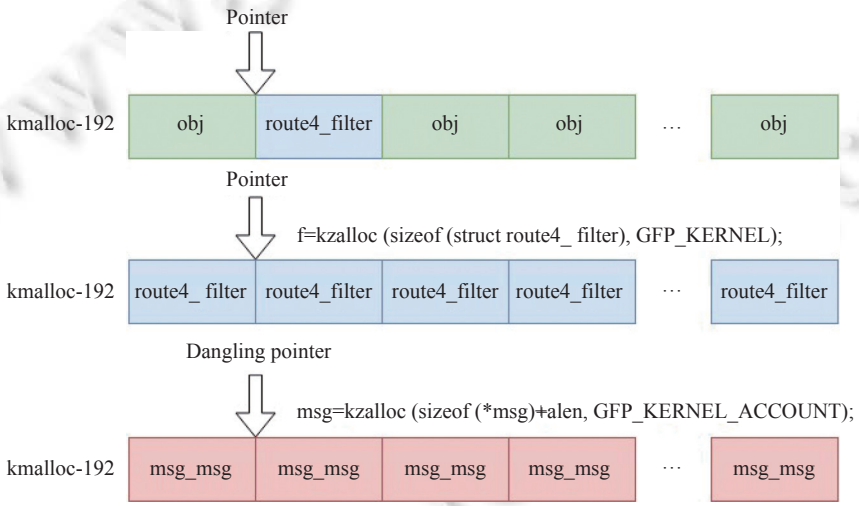


图 6　CVE-2021-3715 信息泄漏内核堆喷射攻击布局

此外, 双重释放 double free 也是 UAF 利用的一种类型, 但是内核中给出了特定检查 (naïve check)[28], 使得连续释放相同数据对象, 操纵 slab freelist 的攻击行为被阻止.

1.3　内核堆漏洞缓解机制

内核采用堆漏洞缓解机制, 在漏洞修复前的时间窗口中降低操作系统安全风险, 但第 1.2 节给出的攻击案例绕过现有的缓解机制, 因此本文对缓解机制进行总结, 从基本原理出发分析缓解机制被恶意绕过的原因. 如表 1 所示.

表 1 内核堆漏洞缓解机制缺陷

缓解机制	缺陷	部署	绕过
Freelist随机化	堆喷射攻击绕过	是	是
Freelist指针模糊化	XOR加密易于破解, 无完整性保护静态配置, 仅对slab元数据	是	是
Autoslab	同类型对象攻击, cross-cache攻击绕过静态配置, 对可伸缩、缓冲区对象无效	是, 仅敏感对象	是
Slab quarantine	堆喷射攻击绕过	否, 未进入内核	是
Naïve check	不连续释放绕过静态配置, 仅对双重释放攻击生效	是	是
数据结构成员随机化	随机种子易获取静态配置, 对可伸缩、缓冲区对象、用户交互UAPI无效	是	是
ERA空间随机化动态缓解	—	—	否

(1) Freelist 随机化[13], 在每次 slab cache 从内核中获取连续页内存拆分成固定长度小数据对象时, 对拆分好的数据对象分配顺序进行洗牌 (shuffle), 使得攻击者难以预测连续分配对象是否相邻, 因此攻击负载放置的难度被提升. 但 freelist 随机化无法应对堆喷射攻击, 即攻击者通过大量分配的方式接触到攻击负载, 存在安全保障不足的问题.

(2) Freelist 指针模糊化[17], 为 slab 元数据 freelist 指针进行两次 XOR 加密, 读取和修改必须采用 slab 分配器的专用函数, 攻击者无法通过明文修改的方式操纵数据对象分配位置. 但 XOR 加密易于破解, 因为 freelist 指针由于系统架构固定以 0xffff888 开头, 同一个 slab cache 的相邻数据对象指针仅有几个 bit 差别, 攻击者可以尝试使用位翻转 (bit flipping) 攻击绕过[29,30]. 而且 freelist 指针模糊化无法保护指针完整性, 此外仅能保护元数据, 对其他类型的堆攻击无效.

(3) Autoslab[16], 由 grsecurity 内核安全公司开发, 借鉴了内核中的专用 slab cache 思想, 使内核中所有类型的数据对象均由专门的 slab cache 分配, 不与相同长度的其他数据对象共享. 这一机制很大程度上提升了内核堆风水难度, 使得攻击负载难以放置. 但此机制需要手动修改内核代码的所有 slab 分配位置, 而且无法部署在无类型的缓冲区, 对于 cross-cache 攻击、同类型对象攻击无法防御, 因此目前仅作为付费软件, 并未合并进入内核.

(4) Slab quarantine[14], 数据对象释放后将其加入隔离列表, 等待一段时间后再还给内核, 试图以此让攻击者无法放置释放后使用攻击的攻击负载, 提升攻击难度. 此机制类似释放后使用漏洞探测采用的 quarantine & sweep 技术[31−33], 但仅隔离而缺乏扫描, 没有从根源上消灭悬空指针, 攻击者依旧可以通过堆喷射攻击绕过此机制.

(5) Naïve check[28], 检查连续分配的两个数据对象地址是否相同, 有效避免了连续双重释放攻击的发生, 攻击者仍然可以不连续释放相同的数据对象地址, 进而构造任意释放类型漏洞.

(6) 数据结构成员布局随机化[15], 在编译时对数据对象的成员的偏移量进行随机化洗牌. 这样当内核堆内存损坏漏洞的漏洞数据对象和受害者/喷射数据对象的成员被随机化之后, 攻击者无法获得准确的机密数据或安全敏感数据的偏移量, 但存在随机性易被打破和随机化范围受限两种缺陷. 1) 由于内核需要支持可扩展模块, 因此随机种子被保存在编译工程文件中易于获取. SALADS[34]和 POLAR[35]分别采用了固定周期和每次分配再随机化提升随机化强度, 但需要将相应取地址指令替换为查询偏移量函数, 引入额外开销. 2) 成员随机化仅能编译时选择成员长度和类型确定的对象, 无法保护用户交互 UAPI、可伸缩和缓冲区对象等. 本文统计了 v5.15 内核版本成员随机化的使用情况, 内核现有超过 50 000 种数据结构, 但仅随机化了其中 67 种, 不足以真正提升漏洞利用难度.

1.4 eBPF 技术

eBPF (extended Berkley packet filter) 是一个运行在内核中的指令虚拟机 (in-kernel VM), 能够将用户编写的程序安全高效地运行在内核地址空间[18]. 其中有 3 大组件极大地释放了 eBPF 的潜能, eBPF 验证器具备大量指令检查策略, 保证注入内核的代码安全稳定运行[36]; JIT (just-in-time) 编译器能够将 eBPF 指令转化为机器码, 保证了代码执行的高效性; eBPF 还在内核中实现了功能复杂的帮助函数 (helper function), 保证了代码强大的功能性.

在最新的各个发行版本的内核中, eBPF 已经成为了默认开启的功能选项, 其除了强大的 XDP 网络包过滤功能[37], 还与内核的 kprobe 监控、perf 性能、LSM 安全、error injection 调试等多个功能组件连接, 进一步扩展了

eBPF 的应用范围, 展现出良好的未来前景. 学术界也开始广泛利用 eBPF 机制, 将编写的策略安全地载入内核, 其中对于性能提升的工作包括, xrp 优化 NVMe 驱动[38]、SynCord 定制锁机制优化同步性能[39]、Sryup 优化系统调度性能[40]. eBPF 对系统安全的探索包括, LBM 拦截恶意硬件对 USB 驱动的攻击[23]、rapidpatch 为固件提供热补丁机制等[41].

2 内核堆漏洞的动态缓解

针对从内核漏洞发现到修复时间窗口缺乏可靠缓解机制的需求, 本文提出了一种动态的内核堆漏洞缓解框架 ERA, 如图 7 所示, 系统管理员输入漏洞报告中存在威胁的数据对象, ERA 即可输出相应的缓解程序并载入内核降低内核安全风险. ERA 结合了数据对象空间随机化技术和 eBPF 技术, 对任意数据对象均能够部署数据对象空间随机化保护, 而且无需重新编译内核, 在运行时即可将安全数据对象注入内核进行替换.

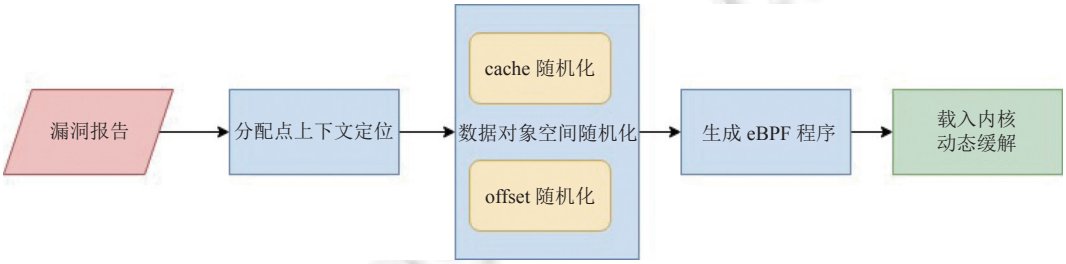


图 7 ERA 工作流程

在动态替换过程中, ERA 首先定位漏洞数据对象的分配点上下文, 即漏洞数据对象分配函数的调用点, 当内核执行到分配点上下文时, ERA 开启数据对象空间随机化, 当前分配点上下文执行过程中所有 slab 分配的数据对象地址均无法预测. ERA 将 cache 和 offset 随机化之后的地址返回给分配点上下文, 同时记录该地址用于合法释放空间随机化数据对象. 在当前分配点上下文执行完毕后, ERA 结束空间随机化, 在此之后的数据对象恢复正常分配. 当随机化数据对象生命周期结束后, ERA 在内存释放函数将随机化地址替换为记录的 cache 随机化分配的起始地址, 正确释放随机化数据对象.

为了实现内核堆漏洞动态缓解框架, 需要面对并解决以下挑战.

● 挑战 1: 分配点上下文定位.

分配点上下文决定了数据对象空间随机化的开启和关闭, 但上下文不易获取. 首先漏洞报告是对内核 bug/漏洞发生位置的描述, 因此常见的漏洞报告形式包括: 1) 自然语言描述; 2) 内核的崩溃信息; 3) 复现 bug 的 Proof-of-concept 程序; 4) 等待审核的漏洞修复补丁; 5) SYZBOT fuzzing 工具的报告[42]. 报告中通常能够确定存在漏洞的数据对象, 但除 fuzzing 报告外, 一般不包含该对象在何处分配. 其次分配点上下文的地址随着内核地址空间布局随机化机制 (KASLR) 在每次启动时不断变化, 而且发行版内核携带调试信息有限, 调试信息也存在大量 bug, ERA 也很难根据调试信息准确定位分配点上下文. 为保证随机化机制及时开启和关闭, 分配点上下文的定位需要尽量避免地址变化的干扰.

● 挑战 2: 数据对象空间随机化方案.

堆漏洞动态缓解框架的核心在于数据对象的空间布局随机化, 第 1.3 节中被绕过的内核缓解机制也采用了随机化方案, 然而现有方案仅作用于类型明确的数据对象, 而对于无类型的缓冲区和可伸缩数据对象则无能为力. 因此新的随机化方案应该适应所有类型的数据对象, 并提供更充足的随机性避免被绕过.

● 挑战 3: 基于 eBPF 技术的随机化数据对象动态注入.

eBPF 技术不提供动态分配释放内存和修改内核数据的功能, 而且还为此设计了一系列验证器避免用户进行相关操作, 因为其设计初衷是在维护内核稳定运行的前提下提供强大的观测能力. 但为了将数据对象空间随机化引入内核, 本文不得不额外实现了两个 eBPF 帮助函数[18]为 eBPF 程序增加了内存管理功能, 并充分利用了 eBPF

的验证器功能, 用 eBPF 程序实现了空间随机化算法[43–45]来分配安全内存. 同样的, 本文使用了 eBPF 的 debug 子系统将空间随机化的内存注入内核, 避免了对内核增加额外开销或引入漏洞.

下面本文详细研究和分析 3 个挑战的解决思路.

2.1 分配点上下文定位

ERA 需要使用 eBPF 探测分配点上下文, 决定数据对象空间随机化的开启和关闭, 但分配点上下文不易获取, 为了保证分配点上下文的准确定位, 本文采用静态分析定位分配函数的调用点 (caller)[46], 再结合 eBPF 的 BTF (bpf type format) 格式探测信息的 CO-RE (compile-once run-everywhere) 特性, 将整个调用点所在的函数作为分配点上下文, 规避 KASLR 带来的地址变化和不准确的调试信息[47].

本文采用基于 LLVM-IR 的静态分析工具来定位分配点调用者函数. 漏洞数据对象分配的标志是调用内存分配函数 (kmalloc, sock_alloc, bio_kmalloc 等), 因此 ERA 静态分析工具首先搜索 kmalloc 等分配函数的调用点作为备选结果[48].

而分配的结果主要有如图 8 所示两种情况, 1) 有类型 struct 分配点 (第 2 行), 2) 无类型缓冲区分配点, 但结果保存在另一 struct 成员 (第 12 行). 而 LLVM-IR 携带了充足的类型信息, 故本文通过分析分配结果的类型变化 (bitcast IR 指令) 和保存位置 (store IR 指令目标地址类型) 确定漏洞对象分配点的调用者 (caller).

```
1   //有类型struct分配点,直接分配
2   struct seq_operations*op=kmalloc(sizeof(*op)，GFP_KERNEL_ACCOUNT);
3   %4=call fastcc i8* @kmalloc(i64 32, 132 4197568) #15, !dbg !4498
4   %5=bitcast i8* %4 to %struct.seq_operations*, !dbg !4498
5
6   //无类型缓冲区分配点,且分配结果为另一struct成员
7   struct legacy_fs_context *ctx;
8   struct legacy_fs_context {
9       char              *legacy_data;/* Data page for legacy fi lesystems */
10  …
11  };
12  ctx->legacy_data=kmalloc(PAGE_SIZE, GFP_KERNEL);
13  %89=getelementptr inbounds %struct.legacy_fs_context,
|   |    |    |    |     %struct.legacyfscontext* %5, 132 O, 132 O, !dbg !6676
14  %93=call fastcc i8* @kmallocl(i64 4096, i32 3264) #12, !dbg !6680
15  store i8* %93, i8** %89, align 8, !dbg !6682
```

图 8 两种代表性的数据对象分配点代码片段及 IR 表示

在有类型 struct 分配 IR 中 (第 3, 4 行), %4 临时变量保存了 kmalloc 分配 32 字节内存的返回地址, 之后 i8*类型的 %4 临时变量被 bitcast 指令转换为 struct seq_operations*类型指针, 因此可以判断在当前函数 seq_operations 类型数据对象被分配; 同理, 在无类型的缓冲区分配 IR (第 13–15 行), %89 临时变量保存了 legacy_fs_context 数据对象第 1 个成员 legacy_data 的地址, 内存分配后地址被保存在 %89 指向的内存中, 由此可以判断在当前函数分配了 legacy_fs_context 的缓冲区成员的数据对象.

ERA 结合静态分析和 BTF 文件实现了数据对象空间随机化的及时开启和关闭, ERA 在分配点上下文, 即分配点的调用者函数开始执行时, 标记当前进程开启了数据对象空间随机化, 在内存分配的核心函数 kmalloc 检测当前进程是否开启随机化, 如开启则返回随机化结果, 否则跳过随机化过程返回正常结果, 在函数结束时取消对当前进程的标记, 关闭随机化. 如后文图 9 所示, struct seq_operations 携带了 4 个代码指针, 经常作为 kmalloc-32 cache 泄露内核代码地址的攻击负载, seq_operation 的分配点上下文为 single_open 函数, ERA 在 single_open 函数开始执行时加入探测点 SEC(‘kprobe/single_open’) 开启随机化, 结束时插入 SEC(‘kretprobe/single_open’) 进行关闭.

2.2 数据对象空间随机化

空间随机化的基本原理之一是额外分配超过需求的内存[43,49], 利用多分配的内存进行随机化, 从而避免攻击

者获取准确的数据对象在地址空间中的位置, 防止恶意利用发生. 类似现有的安全分配器机制, 但是不对所有数据对象进行随机化, 避免夸张的性能和内存开销. 本文在现有的 slab 分配器的基础上设计数据对象的空间随机化, 而非另起炉灶.

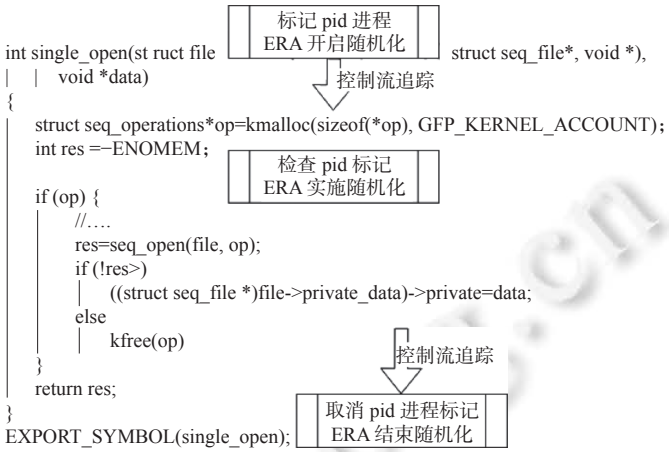


图 9 ERA 数据对象空间随机化开启关闭

根据空间随机化的基本原理, 本文提出了两种随机化方案, 如图 10 所示.

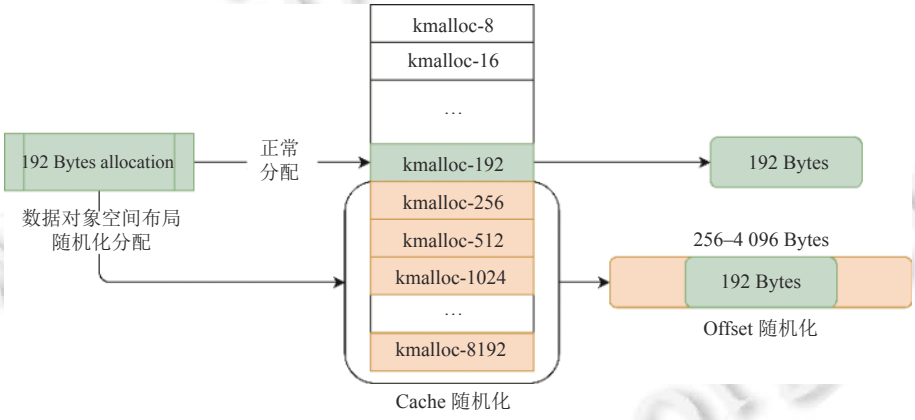


图 10 数据对象空间布局随机化模型

- slab cache 随机化: 随机分配选择空间更大的 slab 缓存, 让漏洞数据对象不在原本的 slab 缓存中.
- offset 随机化: 在分配的大块内存中再进行一次随机化, 让漏洞数据对象的起始点不在这块内存开始的位置 (由于 CPU 的 ALIGNMENT CHECK, 起始点需要 8-byte 对齐).

ERA 的数据对象空间随机化方案提供了充足的安全保障. 在未进行空间随机化的系统中, 内核根据请求在对应大小的 slab cache 分配空闲数据对象, 返回地址为该数据对象的起始位置. 而 ERA 在每次分配时随机选择更大的 slab cache 中分配空闲数据对象, 并在数据对象内的空闲空间中再进行一次随机化, 使得地址无法预测. 随机化范围如表 2 所示, 表示漏洞数据对象在开启 ERA 后, 可选的随机化 slab cache, 以及对应 cache 可选择的 offset 随机化数量.

值得注意的是, 在开启 ERA 后, 即使攻击者使用堆喷射攻击, 也无从知晓攻击程序控制的某个漏洞数据对象所在的具体 cache, 因此虽然表 2 中出现了可选 offset 随机化数量较少的情况, 例如 kmalloc-8 在随机化后选择 kmalloc-16 只有 1 个 offset 可选项, 但是攻击者无法在大量喷射的数据对象中准确锁定 kmalloc-16 上的漏洞数据

对象, 因此 ERA 的随机化安全性体现为表 2 每行可选 cache 及其对应可选 offset 数量的叠加, 例如 kmalloc-8 的安全性体现为 12 个可选 cache 中的 2 070 个可选 offset.

表 2　数据对象空间随机化范围 (开启 ERA 随机化可选 cache 及对应 cache 中的可选 offset 数量)

随机前kmalloc-	随机后kmalloc-													
	8	16	32	64	96	128	192	256	512	1 024	2 048	4 096	8 192	合计
8	—	1	3	7	11	15	23	31	63	127	255	511	1 023	2 070
16	—	—	2	6	8	14	22	30	62	126	254	510	1 022	2 056
32	—	—	—	4	6	12	20	28	60	124	252	508	1 020	2 034
64	—	—	—	—	2	8	16	24	56	120	248	504	1 016	1 994
96	—	—	—	—	—	5	13	21	53	117	245	501	1 013	1 968
128	—	—	—	—	—	—	8	16	48	112	240	496	1 008	1 928
192	—	—	—	—	—	—	—	8	40	104	232	488	1 000	1 872
256	—	—	—	—	—	—	—	—	32	96	224	480	992	1 824
512	—	—	—	—	—	—	—	—	—	64	192	448	960	1 664
1 024	—	—	—	—	—	—	—	—	—	—	128	384	896	1 408
2 048	—	—	—	—	—	—	—	—	—	—	—	256	768	1 024
4 096	—	—	—	—	—	—	—	—	—	—	—	—	512	512
8 192	—	—	—	—	—	—	—	—	—	—	—	—	—	—

对比第 1.3 节相关工作中的缓解机制, 在安全性上, 数据对象空间随机化不存在被确定性绕过的攻击途径, 即使攻击者使用堆喷射、堆风水大量分配数据对象, 甚至最新型的 cross-cache 替换整个 slab cache 页, 也无法准确放置攻击负载达成攻击目标; 在功能上, 数据对象空间随机化机制能够同时应用于长度确定的 struct 对象和长度不确定的可伸缩对象、缓冲区对象, 能够覆盖溢出和释放后使用多种情况. 由此可见随机化既能够用于被漏洞报告确定的漏洞数据对象, 还支持部署在常见的攻击负载对象预防攻击的发生.

本文以第 1.2 节中 CVE-2021-3715 释放后使用漏洞为例, 论证 ERA 数据对象空间随机化提供了充足的安全保证. 攻击者在 kmalloc-192 cache 中大量分配 route4_filter 数据对象, 并在释放后将原地址替换为相同大小的 msg_msg 数据对象. 而在部署数据对象空间随机化机制后, 如表 2 所示, 原本一定在 kmalloc-192 cache 中分配的数据对象可能出现在 kmalloc-256 到 kmalloc-8192 的任意 cache 中. 如图 11 所示, route4_filter 被分配在 kmalloc-1024 中, 且不在该 cache 管理的数据对象起始地址, 不易被攻击者预测, 而且由于悬空指针指向的地址实际为随机化后的地址, 该数据对象的起始地址在第 1 次释放后即从 ERA 机制的哈希表中移除, 如果再次使用 kfree 函数触发释放后使用攻击, 极大概率会导致内核直接崩溃, 因为释放地址无法恢复到非随机化地址.

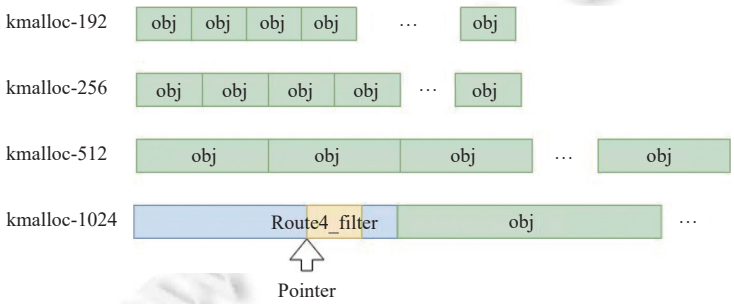


图 11　CVE-2021-3715 漏洞对象空间随机化

假设攻击者为了突破 ERA 使用堆喷射技术请求了大量 route4_filter 对象, 随机化后的数据对象布满 kmalloc-256 到 kmalloc-8192 的 6 个 cache 中, 同时攻击者利用 msg_msg 数据对象可变长的特点 (64–4 096 字节) 占据

route4_filter 释放之后的空间. 但 offset 随机化保证了 route4_filter 的悬空指针指向的地址很难与 msg_msg 构造的攻击负载完全重叠, 只有当 route4_filter 的悬空指针正好位于数据对象的起始地址, 即 offset 随机化为 0 时才能够成功实现攻击, 否则通过 kfree 解引用悬空指针极有可能造成内核崩溃.

本文估算了在 ERA 机制仅随机化 route4_filter 一类数据对象时, 利用堆喷射、堆风水等技术攻击的绕过 ERA 所需的分配数据对象的数量和成功触发概率. 尤其要注意的是, kfree 函数释放非 slab 对象的起始地址时会极大概率导致内核直接崩溃, 因此在此漏洞中释放后使用漏洞仅能够触发 1 次. 假设在 8 核 16 GB 硬件的计算机上, 锁定攻击程序只运行在 1 个 CPU 上, 只使用 slab 的 1 组 per-cpu cache. 攻击程序需要首先采用堆风水攻击填满 per-cpu 局部的半满和全满 cache 页, 才能让 cache 从 buddy 系统继续获得新的内存页, 使用堆喷射攻击填满该内存页绕过 freelist 随机化, 乐观估计 ERA 未部署时单个 cache 达到攻击条件需要至少 126 个 route4_filter 对象 (6 页内存), 如果考虑到 ERA 的 cache 随机化, route4_filter 对象会出现在 6 个 cache 中, 那么攻击程序至少需要连续分配 $126 \times 6 = 756$ 次, 才能勉强保证每个 cache 上都有 route4_filter. 如果再考虑到 ERA 的 offset 随机化, route4_filter 在 kmalloc-256 到 kmalloc-8192 可能出现的位置有 1 872 个, 想要保证 route4_filter 刚好出现在数据对象的起始位置, 乐观估计可能也需要将分配次数扩大 3 个数量级达到 75 万次分配. 然而攻击者只能从 75 万悬空指针中选择一个触发释放后使用漏洞, 可见被利用的风险已经被极大降低, 如果 msg_msg 数据对象也部署 ERA 机制, 那么攻击风险将会被进一步缓解.

值得思考的是对于 CVE-2021-3715, 如果 ERA 的 offset 随机化直接避免了 route4_filter 对象出现在 offset 为 0 的位置, 那么使用 kfree 触发悬空指针类型的释放后使用漏洞就能被避免. 然而此种思路降低了 offset 随机化的随机熵, 而且 ERA 还支持 cache 随机化, 已经具备较强的随机性, 因此本文选择保留 offset 随机化为 0 的可能性. 此外, 在其他同类型漏洞中, 如果通过溢出或修改悬空指针指向内存, 可能尝试次数稍多, 但指针被破坏仍然容易导致内核崩溃; 但如果仅读悬空指针指向内存, 威胁性相对较小.

2.3 基于 eBPF 的动态注入

eBPF 技术的设计初衷是在不影响内核稳定运行的情况下提供强大的观测能力, 因此 eBPF 并不具备内存分配释放和内核数据修改功能, 同时还设计了一系列验证器防止用户编写相关 eBPF 程序. 但是 ERA 需要在尽可能不破坏这一设计规范的前提下, 将空间随机化方案和随机化后的数据对象注入内核, 缓解安全威胁.

2.3.1 空间随机化的 eBPF 实现

由于 eBPF 机制不具备动态分配和释放内存的功能, 因此本文在内核中增加了额外的帮助函数 (helper function), 将内核的分配函数和释放函数导出给 eBPF, 使其依旧能够支持分配参数 (GFP_FLAG). 为了避免探测嵌套, 导出的函数被设置禁止探测. 同时动态缓解机制充分利用了 eBPF 的内核态验证器功能, 主要使用 eBPF 程序实现了空间随机化模型, 对内核代码的修改量非常小约 100 代码行, 几乎不会引入额外的安全漏洞.

eBPF 动态缓解程序在执行到分配点上下文时开启空间随机化, 将 cache 和 offset 随机化之后的地址返回给分配点上下文, 并将该地址和 cache 随机化的数据对象起始地址记录在 eBPF 的哈希表中; 当释放时, 动态缓解程序首先检测释放地址是否属于随机地址, 如果属于, 就从 eBPF 的哈希表中找到 cache 随机化后的数据对象起始地址并合法释放, 再将记录移出哈希表. eBPF 的哈希表是线程安全的, 构成其基础的 bucket 使用自旋锁保护, 而 bucket 里面的成员使用 RCU 机制读写.

2.3.2 随机化数据对象注入

本文借助了内核调试使用的错误注入机制 (error_injection) 来对内核函数返回值进行修改. 通过在编译时标记 ALLOW_ERROR_INJECTION 宏将函数加入错误注入白名单, 用 eBPF 探测该函数起始点, 不执行函数直接返回指定的结果. 错误注入已经作为 eBPF 的子系统之一加入内核, 只需在编译时额外开启 CONFIG_FUNCTION_INJECTION 等几个编译选项即可安全使用 eBPF 提供的 bpf_override_return 帮助函数实现相关功能.

3 实验分析

本文设计了 4 组实验分别对 ERA 的有效性、性能开销、相关工作对比和易用性进行验证. 首先为证明 ERA

的有效性, 本文定性分析了 ERA 方案的安全性, 并从公开渠道收集并评估了 40 个内核堆漏洞, 同时进一步测试了其中 12 个绕过现有缓解机制的攻击程序 (EXP); 其次为了体现 ERA 的具体开销, 本文通过采样内核运行过程, 选取了其中分配量较大的 4 类数据对象, 分别测试了基准测试程序的性能和内存开销, 以验证即使在严苛条件下, ERA 仍能够表现出较好的性能和空间消耗; 此外, 本文对比了同样采用空间随机化技术的数据对象成员随机化机制, 体现了 ERA 在随机性、适用范围和性能上的优势; 最后, 本文通过分析一位系统管理员需要付出多大努力才能使用 ERA 缓解内核堆漏洞, 展示 ERA 工具易于部署的特性. 测试过程中 ERA 的 cache 和 offset 随机化全部开启.

3.1 有效性

● 实验设置

本文从 NVD、SYZBOT、GitHub 等公开渠道收集了 2010–2022 年的 40 个内核堆漏洞, 漏洞类型覆盖了溢出 (OOB) 和释放后使用 (UAF) 两类代表性的内核堆内存损坏, 并涵盖了 slab、buddy 和 vmalloc 这 3 类内核堆分配器. 在此基础上收集了其中 12 个内核堆漏洞的公开 EXP 程序, 这些程序为确保攻击的稳定性, 均采用了堆喷射、堆风水的攻击技术绕过第 1.3 节中内核堆漏洞缓解机制.

首先, 本文对 ERA 缓解溢出、释放后使用、双重释放和非法释放这 4 种最常见的堆漏洞的能力和 ERA 框架自身的安全性进行定性分析, 并根据数据对象空间随机化的原理归纳出 ERA 缓解方案适用的两个条件: 1) 由 slab 分配器分配, 2) 漏洞的攻击方式是读写攻击者控制的其他数据对象, 并以这两个条件作为标准评估收集的 40 个内核堆漏洞, 如果满足标准则说明该漏洞能够被 ERA 机制缓解.

其次, 本文将选取的 12 个内核堆漏洞全部导入到 v5.15 版本内核, 使用 qemu-kvm 虚拟机构建测试环境, 测试环境符合第 1.1 节威胁模型假设, 分别执行漏洞对应的 EXP 攻击程序, 测试是否能够在满足第 1.2 节威胁模型的前提下, 同时绕过现有缓解机制和 ERA 缓解机制, 成功实现攻击负载准确放置, 进而泄露内核代码地址、提升程序执行权限. 本文认为由于运行时的数据对象随机化程度不同, 在攻击发生后如果系统维持运行、提示错误或崩溃, 而攻击目标未能达成, 那么 ERA 缓解程序对此漏洞是有效的.

● 实验结果

对于溢出 (out-of-bound) 漏洞: ERA 方案通过数据对象的空间随机化, 让攻击者无法确定漏洞数据对象和受害者数据对象到底在哪个 slab cache 中, 图 1 溢出漏洞模型中的 *ptr* 和指向的 *obj* 均存在随机性, 因此无法将被攻击对象正好放在漏洞数据对象后面, 如果按照 $offset_{\text{ptr}} + size_{\text{access}}$ 进行溢出, 则无法预测溢出结果, 因此无法进行攻击.

对于释放后使用 (use-after-free) 漏洞: ERA 方案能够成功地随机化分配漏洞数据对象在内核地址空间的位置, 图 4 释放后使用模型中的悬空指针 p 的位置存在随机性, 同理 p 指向的 O 的位置 $\forall m, size : [m, m + size - 1]$ 很难被攻击者准确定位, 因此构造的攻击负载无从放置, 即使触发悬空指针 p 指向的地址, 也很难引起负载中引入的下一步攻击.

对于双重释放 (double-free) 漏洞: 内核的 slab 机制本身提供了 naive check, 使得这类漏洞无法通过连续释放相同地址控制 slab 的 freelist, 即使攻击者可能将其转化 kfree 函数释放后使用漏洞, 也无法绕过 ERA 实现攻击利用.

对于非法释放 (invalid free) 漏洞: 内核 slab 机制在释放时检查地址是否由 slab 分配, 因此这类漏洞在内核中无法利用.

此外, 攻击者对 ERA 框架的攻击尝试也无法生效, 首先 ERA 载入内核的代码通过了验证器检查, 并设置为只读, 因此不存在代码完整性问题; 其次 ERA 使用的数据和空间随机化分配的数据天然具备了一定随机性, 攻击者也很难通过漏洞利用找到 eBPF 使用的数据位置; 最后, eBPF 程序的运行和载入需要 root 权限, 攻击者无法注入恶意的 eBPF 程序破坏系统安全.

内核堆漏洞的评估结果如表 3 所示, 在公开收集的 40 个漏洞中, 36 个漏洞能够被 ERA 缓解机制成功缓解, 这些堆漏洞均由 slab 分配器分配, 且攻击方式为读写攻击者控制的其他数据对象, ERA 采用空间随机化技术使得

这些漏洞数据对象的分配点无从预测, 攻击者无法准确放置或解引用负载, 因此内核堆漏洞被利用的风险显著降低.

表 3 内核堆漏洞评估结果

编号	类型	分配器	攻击时读写其他对象	ERA有效	编号	类型	分配器	攻击时读写其他对象	ERA有效
CVE-2010-2959	溢出	slab	√	√	CVE-2021-33909	释放后使用	slab	√	√
CVE-2017-7184	溢出	slab	√	√	CVE-2017-7533	释放后使用	slab	√	√
CVE-2022-0185	溢出	slab	√	√	CVE-2016-8655	释放后使用	slab	√	√
CVE-2022-34918	溢出	slab	√	√	CVE-2021-26708	释放后使用	slab	√	√
CVE-2016-6187	溢出	slab	√	√	CVE-2017-15649	释放后使用	slab	√	√
CVE-2021-22555	溢出	slab	√	√	CVE-2021-20226	释放后使用	slab	√	√
CVE-2021-43276	溢出	slab	√	√	CVE-2021-27365	释放后使用	slab	√	√
CVE-2017-1000112	溢出	slab	×	×	CVE-2021-22600	释放后使用	slab	√	√
CVE-2021-27365	溢出	slab	√	√	CVE-2022-1786	释放后使用	slab	√	√
CVE-2017-7308	溢出	page	√	×	CVE-2022-2602	释放后使用	slab	√	√
CVE-2022-27666	溢出	page	√	×	CVE-2017-11176	释放后使用	slab	√	√
CVE-2020-14386	溢出	vmalloc	√	×	CVE-2022-1116	释放后使用	slab	√	√
CVE-2017-8824	释放后使用	slab	√	√	CVE-2022-29581	释放后使用	slab	√	√
CVE-2020-16119	释放后使用	slab	√	√	CVE-2022-25220	释放后使用	slab	√	√
CVE-2021-23134	释放后使用	slab	√	√	CVE-2020-14356	释放后使用	slab	√	√
CVE-2022-2586	释放后使用	slab	√	√	CVE-2022-29582	释放后使用	slab	√	√
CVE-2021-3715	释放后使用	slab	√	√	CVE-2017-10661	释放后使用	slab	√	√
CVE-2021-4154	释放后使用	slab	√	√	CVE-2016-10150	释放后使用	slab	√	√
CVE-2019-18683	释放后使用	slab	√	√	SYZBOT-1e2ff6d	释放后使用	slab	√	√
CVE-2022-2588	释放后使用	slab	√	√	SYZBOT-ea6a322	释放后使用	slab	√	√

无法应用的情况包括 CVE-2017-7308、CVE-2022-27666 和 CVE-2020-14386, 此处 3 个漏洞的漏洞数据对象由 buddy 或 vmalloc 分配器分配, 而非 slab 分配器, 但是本文认为数据对象空间随机化的思想对于这两类分配器依旧有效, 我们将在未来的工作中将 ERA 机制部署在这两类分配器上. CVE-2017-1000112 则属于特殊情况, 溢出发生在数据对象内部. 该漏洞中的漏洞数据对象 sk_buff 虽然也由 slab 分配器分配, 但分配长度远大于 sk_buff 成员所需长度, 在额外分配的长度中, struct sk_shared_info 数据结构位于数据对象的结尾, sk_buff 成员的结尾到 sk_shared_buff 数据结构开始的区域为保存 sk_buff 信息的缓冲区, 该缓冲区可能发生溢出, 破坏位于数据对象结尾处的 sk_shared_info 成员, 造成数据对象内溢出. 攻击时没有破坏其他攻击者控制的数据对象, 因此 ERA 无法缓解此类漏洞. 虽然 ERA 和现有缓解机制均无法直接应对上述两种情况, 但对于第 1 类 buddy 或 vmalloc 分配器分配漏洞数据对象情况, ERA 可以间接随机化攻击过程中用到的受害者/喷射数据对象, 由于这类对象可选范围通常较固定且符合 ERA 适用范围, 故 ERA 依旧能够缓解安全风险. 12 个漏洞的 EXP 攻击测试结果如表 4 所示[50–61], 本文分析了 EXP 攻击中漏洞数据对象的类型和攻击负载的数据对象类型, 并利用 ERA 的分配点上下文定位技术确定了 ERA 部署位置. ERA 机制启动了漏洞数据对象和攻击者使用的攻击负载数据对象的空间随机化, 测试结果显示均能够成功缓解, 攻击目标无法达成, 即使采用堆喷射、堆风水等能够稳定绕过现有缓解机制的 EXP 程序仍无法绕过 ERA 的数据对象空间随机化, 进一步说明了 ERA 的有效性.

3.2 高效性

● 实验设置

性能和内存开销实验在 intel core i5 12600K, 内存 16 GB DDR4, 1 TB NVMe 固态硬盘, 内核版本 v5.15 物理机上进行. 由于漏洞的触发路径并非实际内核的常用执行路径[62], 常规的内存密集型测试实际上几乎不触发

ERA 的空间随机化机制, 无法测试出 ERA 机制的额外开销, 因此本文对内核的运行过程中的数据对象分配数量采样, 随机选取一定时间内分配较为频繁的 4 类数据对象部署 ERA 空间随机化机制, 体现严苛条件下 ERA 的性能损耗. 采样工具选取 eBPF 的命令行工具 bpftrace, 执行“bpftrace -e ‘tracepoint:kmem:kmalloc { @[kstack()]= count(); }’”, 通过统计内存分配函数的调用栈数量, 选取测试的数据对象.

表 4　面对真实 CVE 漏洞的动态缓解技术有效性 (EXP 均能绕过现有缓解机制)

CVE编号	类型	威胁对象	分配上下文	ERA有效
CVE-2010-2959[50]	溢出	缓冲区 shmid_kernel	bcm_sendmsg newseg	是 是
CVE-2017-7533[51]	溢出	缓冲区 iovec	inotify_handle_event iovec_from_user	是 是
CVE-2021-22555[52]	溢出	xt_table_info pipe_buffer	xt_alloc_table_info alloc_pipe_info	是 是
CVE-2022-34918[53]	溢出	缓冲区 simple_xattr	nft_set_elem_init simple_xattr_alloc	是 是
CVE-2017-7184[54]	溢出	xfrm_replay_state_esn cred	xfrm_alloc_replay_state_esn cred_alloc_blank	是 是
CVE-2016-8655[55]	释放后使用	packet_sock user_key_payload	packet_create user_prepare	是 是
CVE-2021-26708[56]	释放后使用	vsock_transport msg_msg	vmci_transport_socket_init load_msg	是 是
CVE-2020-16119[57]	释放后使用	sock msg_msg	sk_alloc load_msg	是 是
CVE-2017-10661[58]	释放后使用	timerfd_ctx msg_msg	__x64_sys_timerfd_create load_msg	是 是
CVE-2016-10150[59]	释放后使用	kvm_device key	kvm_vm_ioctl key_alloc	是 是
CVE-2017-11176[60]	释放后使用	netlink_sock 缓冲区	__netlink_create ____sys_sendmsg	是 是
CVE-2017-15649[61]	释放后使用	packet_sock msg_msg	packet_create load_msg	是 是

性能开销重复多次取平均值, 本文首先选取了 lmbench 微基准测试程序 (micro benchmark)[63]进行测试, 测试范围涵盖系统调用上下文切换、文件系统、本地通信延迟和带宽 4 类基本功能, 测试了随机选取的 4 类数据对象分别部署的开销和同时部署的开销, 判断单个数据对象和多个数据对象累加对系统性能造成的影响. 同时本文使用 phoronix-test-suites 宏基准测试程序 (macro benchmark)[64]集合, 测试了 ERA 在具体应用程序下的表现, 测试包含 PostMark、OSBench、IPC_benchmark、HackBench、OpenSSL、BenchmarkMutex、PyBench 和 Apache HTTP Server, 该测试主要针对全部 4 类数据对象累加的性能开销.

由于内存开销随系统启动时间和正在执行的程序不断发生波动, 而且由于系统中断、网络通信等功能的偶然性, 很难像性能开销测试一样横向对比, 因此本文聚焦于内存占用的增量而非具体内存开销, 即采样的每组内存占用量减去采样过程中的最小内存占用量. 表明了内核在执行测试任务的内存分配和释放情况. 本文将内存增量的差值作为对比标准, 用于比较 ERA 开启后的额外内存开销, 同时注重分析差值的最大值, 即 ERA 开启后的最大额外内存开销. 本文编写了系统内存采样程序, 每秒采集 1 次当前系统的内存使用情况, 分别测试了执行 lmbench 和 Chrome 浏览器在线播放相同视频 5 min 两项指标, 其中后者反映了 ERA 应对复杂任务时的表现, 同时更好控制采样时间变量, 主要测试了全部 4 类对象累加的内存开销.

最后, 本文在工作日开启 ERA 空间随机化 24 h, 期间使用者正常进行日常研究工作, 用于测试 ERA 长期工作给系统和使用者带来的影响.

● 实验结果

根据采样结果, 本文选取了两类长度确定的数据对象 kernfs_open_file (kof) 和 seq_operations (so), 还选择了两类长度不确定的缓冲区数据对象, 分别由 load_elf_phdrs (lep) 函数和 inotify_handle_inode_event (ihie) 函数分配. 以上对象在采样过程中均分配 2 000 次以上.

性能测试的结果均以未开启 ERA 内核的测试结果进行基准化, 1.0 表示性能开销相同, 大于 1.0 则表示 ERA 性能开销大于未开启 ERA 内核, 小于则开销好于未开启 ERA 内核. lmbench 基准测试程序的性能损耗如图 12 所示, 单项测试的最坏结果约造成 3% 的额外开销, 同一数据对象 4 项性能损耗的几何平均数仅 1.01 上下, 代表平均性能开销仅为约 1%. 而且单独的数据对象和全部数据对象累加的性能开销几乎没有显著变化. 宏基准测试程序的各类性能也仅仅导致了微不足道的约 1% 的性能开销, 如图 13 所示.

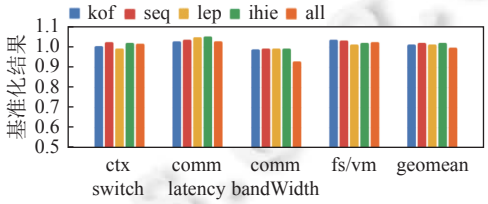


图 12 lmbench 性能测试结果对比

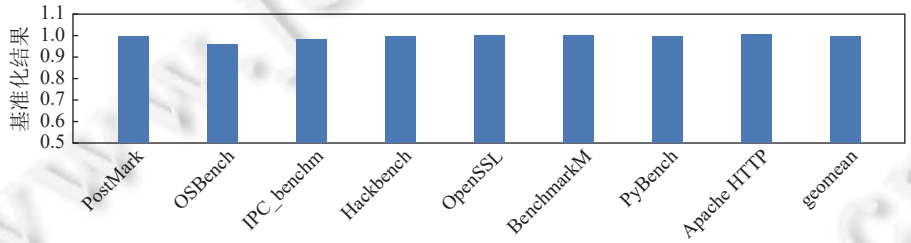


图 13 phoronix 测试结果对比

导致 ERA 性能表现优秀的原因主要有两点, 一是漏洞数据对象的分配和释放, 即使在大量分配的情况下仍然在内核的整条执行路径上占比极低, 因此原本的开销较低. 其二是, ERA 基于 eBPF 插入的探测点仅有分配点上下文开启、结束、slab 分配函数和释放函数这 4 处, 其中前两处探测点执行数量较少, 而且负责开启关闭当前进程的随机化, 指令较少; 而后两者在所有调用分配和释放函数时都会触发, 执行次数较多, 但首先 eBPF 插入探测点使用高效的 ftrace 函数跳转机制而非 int3 中断跳转; 其次在跳转后第 1 步采用的是 eBPF 的哈希表查找, 查找 key 为 32 位长度的进程 pid 或 64 位的随机化地址, 时间复杂度较低; 最后随机化设计比较巧妙, 随机化仅发生在查找哈希表存在匹配项的条件下, cache 和 offset 随机化机制仅需要获取 1 个随机数, 而且原有的内存分配逻辑被跳过, 无需额外分配, 因此开销同样非常低. 同理, 增加保护对象的数量并没有显著放大 ERA 在内核执行路径上面的占比, 故性能损耗几乎没有增加.

在部分测试指标中, 系统在开启缓解机制后获得了更好的性能表现, 例如 lep 的 ctx 切换性能好于未开启 ERA 内核, 而 4 类数据对象累加的带宽性能甚至增加了约 7%. 类似情况在内核性能测量工作中也有出现[65]. 究其原因可能是由于 CPU 采用了大小核调度和变频架构, 测试时 CPU 的工作状态和系统后台程序运行的噪声干扰了测试结果, 导致应的部署 ERA 测试时主频较高同时 IO 设备通信速度快带宽大, 性能表现更好.

内存开销增量结果如图 14 所示, 图中横坐标为执行相同任务的内存采样点, 纵坐标表示内存占用的增量. 其中 lmbench 执行和 Chrome 浏览器在线播放视频的内存占用增量和其走势基本吻合, 相同采样点 ERA 开启和未

开启时内存增量的差值不大, 说明内存使用情况基本一致, 因为测试时执行的任务相同. 而在两组测试中 slab 内存占用情况走势基本吻合, 但测试过程中相同采样点 ERA 开启和未开启时内存增量的差值较为明显, 说明了 ERA 开启时 slab 内存消耗略高于未开启时, 经过计算差值为 5–150 MB, 最高的额外内存损耗占比 0.9%, 该现象符合 ERA 的数据对象空间随机化原理. 需要额外强调的是测试所选 4 类对象在内核中大量分配 (远高于真实漏洞情况[62]), 同时 slab 分配器占用的内存并未全部分配, 而是作为缓存由 slab cache 管理, 数据对象生命周期较短, 因此本文认为 ERA 在严苛的测试环境下开销能够被接受, 且 slab 内存的循环使用不会对系统带来过大压力.

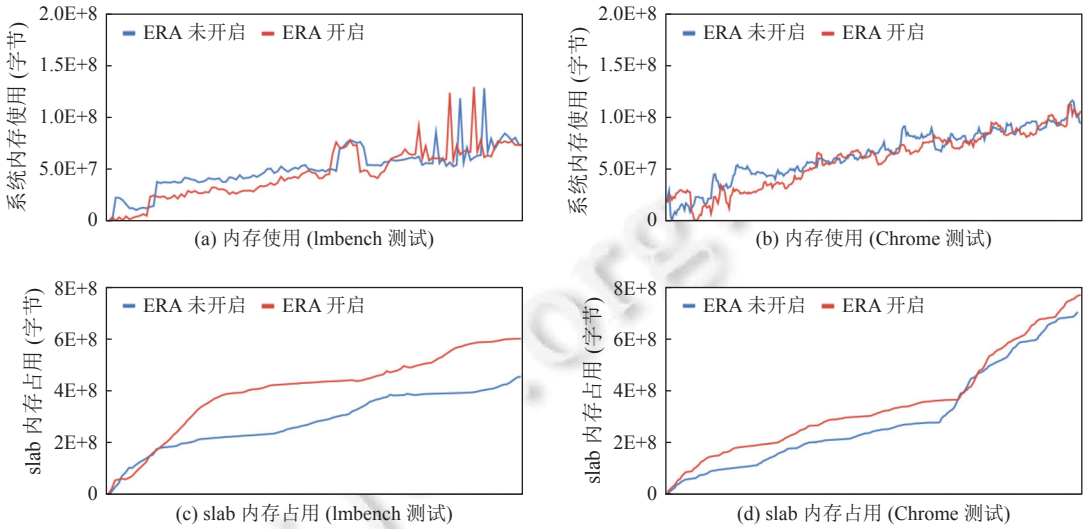


图 14 lmbench 和 Chrome 播放在线视频的系统内存使用、slab 占用增量

在工作日 24 h 开启 ERA 的测试中, 系统始终保持正常运行, 而系统使用者很难感知 ERA 带来的影响, 因此本文认为 ERA 能够被应用在研究和生产环境下有效缓解内核堆漏洞风险.

3.3 相关工作对比

● 实验设置

本文选择了同样采用空间随机化技术的数据对象成员布局随机化 (randstruct) 及其后续改进 SALAD 和 POLAR, 对比 ERA 设计的 cache 和 offset 两类数据对象空间随机化. 成员随机化的基本原理在于打乱数据对象内成员的布局, 使得攻击者无法准确放置攻击负载, 不能准确溢出或解引用数据对象的安全敏感成员, 从而提升攻击利用难度. 然而目前内核中的成员随机化 (randstruct) 仅在编译时通过编译器扩展随机化 1 次, 而且为了支持可扩展模块载入, 随机种子公开可见, 攻击者可以轻易得到安全敏感成员的准确偏移量. 为了提升成员随机化的机密性, 相关工作 SALAD 和 POLAR 在随机化频率上进行优化, 前者在固定周期更换数据对象的随机种子, 后者则在每次分配时重新随机化数据对象布局, 不过都需要将取数据对象成员取地址指令替换为专用的取地址函数, 增加了额外开销.

由于成员随机化不具备动态增加保护对象的能力, 故无法对类型多变的漏洞数据对象及时提供保护, 与 ERA 工作的安全背景和目标不同, 安全性远弱于 ERA 方案, 因此本文仅在原理层面上对随机性、随机频率、功能、性能和内存消耗上对比 ERA 和成员随机化, 首先设置了一个度量指标表示随机化强度, 即一个 8 字节成员能够出现在分配的数据对象中的位置数量, 代表攻击者想要读写指定安全敏感指针的难度, 数量越多则代表随机性越强. 其次, 为了验证性能开销, 本文采用 LLVM IR Transform Pass[66]和 Fisher & Yates 洗牌算法[67]模拟了成员随机化工作, 对比了 kernfs_open_file (kof) 和 seq_operations (so) 两类长度和成员确定的数据对象, 在 ERA 和成员随机化情况下的基准测试程序性能开销, 并测量 Chrome 浏览器播放在线视频 5 min 的内存使用增量, 其中成员随机化仅选择随机化频率最高的 POLAR 方案.

● 实验结果

随机性度量, 功能和内存消耗的对比如表 5 所示. 表 5 中的 *size* 表示分配的数据对象的大小, *n* 表示数据对象成员的数量, *entropy* 表示随机化时额外增加的空间. 在随机性方面, randstruct 方案的 8 字节成员仅能出现在长度为 *size*/8 个位置. 而 SALAD 和 POLAR 可以人为设置一个额外的 *entropy* 区间, 一般 *entropy* 最大等于 *size*, 那么这两类成员随机化方案的强度最大为 randstruct 的 2 倍. 而 ERA 采用的空间随机化因为将分配内存放在更大空间的 cache 中, 在通用的 slab cache 里, 多数情况下后一个 cache 数据对象大小是前一个的 2 倍, 例如 kmalloc-128/256, 而特殊情况下为 1.33–1.5 倍 (kmalloc-64/96/128, kmalloc-128/192/256), 因此 ERA 空间随机化的强度多数情况下能够达到 2 倍以上优于成员随机化, 最大为常数 1 024 个可行位置, 远超过成员随机化. 而对比随机化频率, randstruct 的每次编译时随机化频率最低, 其次是 SALAD 固定周期随机化, 如果 SALAD 的周期调整为每次分配, 则认为与 POLAR 和 ERA 随机频率一致, 显然分配次数越多攻击难度越大.

表 5　数据对象空间随机化与成员随机化对比

随机化方案	随机性	度量	随机频率	动态长度支持	运行时开销	内存开销
Randstruct	成员随机化	(*size*)/8	每次编译	否	无	无
SALAD	成员随机化	(*size*+*entropy*)/8	固定周期	否	周期shuffle+取地址	*entropy*+*n*×4+8
POLAR	成员随机化	(*size*+*entropy*)/8	每次分配	否	每次分配shuffle+取地址	*entropy*+*n*×4+8
ERA	空间随机化	[(*size*×1.33)/8, 1024)	每次分配	是	每次分配随机化	[0.33×*size*, 8184)

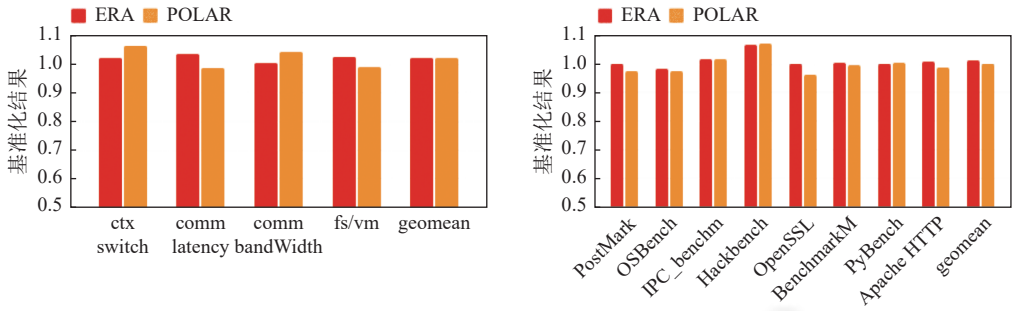
在功能上, 成员随机化必须依赖重新编译内核增加被保护数据对象, 且仅支持确定长度、有类型的数据结构, 而 ERA 则能够动态增加数据对象, 且不存在数据对象类型限制. 显然 ERA 在适用范围上有毋庸置疑的优势.

在运行时性能消耗方面, randstruct 因为仅在编译时随机化, 因此性能几乎没有损耗, 而 SALAD, POLAR 及 ERA 则普遍存在每次分配时随机化的开销, 但不同点在于 SALAD 和 POLAR 采用 shuffle 算法需要获取多个随机数, 而 ERA 虽然有 cache 和 offset 两种随机化, 但仅需要获取一个随机数, 此外 SALAD 和 POLAR 需要将数据对象成员的取地址指令替换为专用的取地址函数, 根据成员的需要返回准确的地址, ERA 并不存在此类开销. 对于额外的内存消耗, randstruct 同样没有运行时消耗, 而 SALAD 和 POLAR 则需要专门的元数据保存随机化前后的映射关系, 因此对于一个数据对象所需要的额外内存开销则包括: 1) 8 字节指定数据对象地址, 2) *n* 个成员对应的偏移量 (*n*×4), 3) 额外的 *entropy* 区间, 最大为 *size*. ERA 的原理是通过增加空间的分配降低地址的可预测性, 因此对于内存的额外消耗量大于 SALAD 和 POLAR, 至少为分配数据对象的 0.33 倍, 最多为 8 184 字节 (8 字节内存被随机化到 kmalloc-8192). 但根据上文结论, 数据对象的生命周期较短, 因此此类工作的内存能够快速循环使用, 造成的内存开销可以接受.

POLAR 和 ERA 对比基准测试程序的结果如图 15 所示, 在 lmbench 测试中, 上下文切换和通信带宽测试 ERA 好于 POLAR, 而其余两项 POLAR 表现更好, 几何平均测试结果二者几乎持平. phoronix 测试组件的测试结果也体现二者开销基本持平, 但原理上 ERA 的性能开销应该明确好于 POLAR, 本文认为造成二者性能基本持平的主要原因是, 选取的 kernfs_open_file 和 seq_operations 数据对象成员数量较少, 而且在内核中仅替换了 176 个取地址指令, 且替换后取地址函数仅负责根据成员编号获取偏移量, 相比复杂的内核程序, 取地址函数的指令和执行次数较少, 因此造成的开销几与原理上性能更好的 ERA 一致.

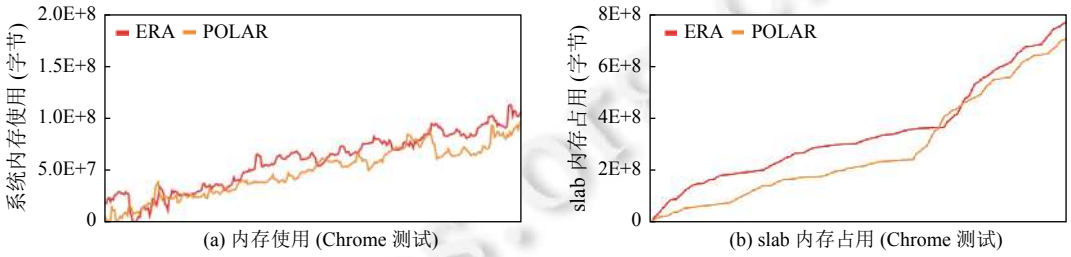
POLAR 和 ERA 对比 Chrome 浏览器在线播放视频 5 min 的结果如图 16 所示, 因为测试任务相同, ERA 和 POLAR 的使用内存增量基本一致, 在 slab 占用指标上 ERA 稍高于 POLAR 符合上文原理分析, 但因为数据对象生命周期较短, 且 slab 占用内存并未被全部分配, 因此本文认为对比相关工作 ERA 内存开销劣势并不显著.

综上所述, ERA 相比同样采取空间随机化的数据对象成员随机化, 在安全性上提供了更强的随机性; 功能上适应范围更广; 而性能损耗方面 ERA 在原理上也具备优势, 同时空间随机化设计原理上存在的内存损耗在实际测试中额外开销有限, 因此本文认为 ERA 采用的数据对象空间随机化具备较为显著的创新性



(a) Lmbench (b) Phoronix

图 15 ERA 和 POLAR 执行 lmbench 和 phoronix 测试结果对比



(a) 内存使用 (Chrome 测试) (b) slab 内存占用 (Chrome 测试)

图 16 Chrome 播放在线视频的系统内存使用、slab 占用情况, 图中单位为字节

3.4 易用性

为了证明 ERA 系统的易用性, 本文具体分析使用者得到漏洞报告后使用 ERA 的流程.

首先, 根据内核堆漏洞发生的位置, 找到漏洞数据对象的类型仍需要 ERA 使用者手动分析, 但是本文认为从源代码中找出漏洞数据对象的类型并不困难. 常见的 IDE 集成开发环境都能够提示选定数据结构的类型和声明位置, 即使是对于没有数据结构类型的缓冲区, 通常也被其他数据结构的指针成员指向, 因此使用者可以利用静态分析工具找到分配点上下文. 其次, ERA 已经提供了 eBPF 程序生成的功能, 使用者只需要将第一步提取的漏洞数据对象分配点上下文输入, ERA 即可输出相关的安全缓解程序. 最后, 使用者必须使用 root 权限启动 eBPF 程序将漏洞缓解策略载入内核中, 从此刻开始攻击者将很难利用漏洞破坏内核完整性.

鉴于近 3 年公开的 57 个内核漏洞中有 40 个属于内核堆漏洞, ERA 具备广阔的应用前景. 以 CVE-2017-7533[51] 为例, 漏洞报告中已经给出溢出的根源在于 event=kmalloc(alloc_len, GFP_KENEL); 处分配的内存不足以保存用户传递的文件名数组, 那么无需进行额外分析, 只需将分配点上下文, 即 inotify_handle_event 函数交给 ERA 原型系统, 生成对应 eBPF 程序并载入内核, 即可实现堆漏洞的动态缓解.

4 讨 论

ERA 采用了分配点上下文控制数据对象空间随机化机制开启和关闭, 然而实际测试中本文发现这一方案还存在一定提升空间. 例如 CVE-2010-2959[50]中存在两个问题, 首先内核编译优化将漏洞数据对象分配点 bcm_rx_setup 被内联在 bcm_sendmsg 函数中, 并且可能被优化成 bcm_sendmsg.cold 缩减指令数量, 因此在选择探测点时需要选择上层的 bcm_sendmsg 函数. 但是 memorizor[68]已经给出了精准的运行时指令到数据结构的关系, 我们将在未来的工作中将其整合进分配点上下文定位模块中.

其次 bcm_rx_setup 函数中不止调用了 1 次内核分配函数, 更何况该函数被内联到了上层函数 bcm_sendmsg 中, 但 ERA 采用的方案会随机化 bcm_sendmsg 函数中所有分配的数据对象, 可能会造成短时间内较多的无效额外开销, 但是这也使得 ERA 拥有了远大于相关工作的随机熵, 抵御未知的安全威胁, 而且测试结果表明, 随机化数量的增加对系统运行时性能和内存的开销均未造成太大影响, 因此这些开销是可以接受的.

此外, ERA 数据对象空间随机化中的 slab cache 随机化方案最大支持 4096 字节, slab cache 中最大的 kmalloc-8192 cache 暂不支持 ERA, ERA 可以通过创建更大的 slab cache 实现对 kmalloc-8192 的支持, 但根据本文在内存密集测试下的统计结果, kmalloc-8192 仅占全部分配量的 0.5%, 而且此 cache 中较少存在包含安全敏感信息的对象, 因此我们将在未来的工作中实现该功能.

内核堆除了文中重点提到的 slab 分配器, 还有 buddy 和 vmalloc 两种分配器, 例如 CVE-2017-7308[69]页缓冲区溢出, 我们强调 ERA 的随机化设计思想对这两类较为少见的漏洞也有效, 并在未来的工作中会将 ERA 拓展到这两类分配器中. CVE-2017-1000112 数据对象内溢出漏洞超出了 ERA 的适用范围, 为实现对此类漏洞的缓解, 我们将探索 eBPF 对漏洞触发更直接的监控和阻止.

5 相关工作

热补丁技术是一种在程序运行时动态修复代码 bug 的技术, 面向 Linux 的 livepatch[70], ksplice[71], kgraft[72], kpatch[73]等热补丁技术, 也利用了 kprobe, ftrace 等[74]动态追踪机制, 将安全专家发布的 bug 修复补丁应用在内核中, 而无需重新编译内核并载入. ERA 在表现形式上与热补丁具备了相似的动态特性, 但不需要安全专家发布的修复补丁, 可以在修复前时间窗口内生效.

针对内存损坏漏洞根源的探测和防御也可以用来降低系统安全威胁, 例如针对溢出型漏洞的数据对象边界检查和数据指针边界检查[3,75], 针对释放后使用型漏洞的悬空指针限制和指针解引用检查[1,31−33,76–80]. 然而二者通常都需要静态插桩插入指针追踪、数据对象追踪和运行时检查, 占用大量系统资源, 巨大的性能和内存损耗很难被操作系统内核接受.

监视器 (reference monitor) 通过部署安全策略监控内核安全的重要目标保证系统安全, 例如安全敏感数据完整性[81], 数据流完整性[82], 控制流完整性[83]等, 然而相关工作通常不针对漏洞根源, 而且占用虚拟化等硬件资源, 仍未被开源 Linux 系统采用.

安全分配器, 在内存分配和释放时随机化或调整数据对象的布局[43–45,84], 同样能够提升漏洞利用难度, 但通常需要对系统中的所有数据对象进行随机化, 运行时性能和内存损耗较大, 操作系统负责管理全部内存资源很难接受过大资源损耗.

6 总 结

本文提出了一种针对内核堆漏洞的动态缓解技术, 在漏洞修复前的较长时间窗口内降低内核面对的安全威胁. 动态缓解采用了数据对象空间随机化技术, 通过在更大空间的 slab cache 中分配大于所需的内存, 并在内存中随机放置数据对象, 使得原本存在安全威胁的数据对象地址难以被攻击者预测, 并在此基础上利用 eBPF 技术的特性将空间随机化的数据对象安全、高效地注入内核.

本文强调基于 eBPF 的动态缓解技术的安全性、高效性和易用性. ERA 能够动态对任意存在安全威胁的数据对象部署空间随机化, 使得攻击者无法准确预测数据对象的位置或偏移量, 很难放置攻击负载破坏整个系统. 动态缓解机制相比原始内核仅引入约 1% 性能开销, 内存开销由于快速分配释放也几乎与原始内核一致. 系统管理员无需等待安全专家发布的补丁或重新编译内核即可部署缓解程序, 因此我们认为本文的动态缓解技术有广阔的使用前景.

References:

[1] Lee B, Song CY, Jang Y, Wang TL, Kim T, Lu L, Lee W. Preventing use-after-free with dangling pointers nullification. In: Proc. of the 22nd Annual Network and Distributed System Security Symp. San Diego: NDSS, 2015. [doi: 10.14722/ndss.2015.23238]

[2] NVD. National Vulnerability Database. 2022. https://nvd.nist.gov/

[3] Nagarakatte S, Zhao JZ, Martin MMK, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. In: Proc. of the 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Dublin: ACM, 2009. 245–258. [doi: 10.1145/1542476.1542504]

[4] Nagarakatte S, Zhao JZ, Martin MMK, Zdancewic S. CETS: Compiler enforced temporal safety for C. In: Proc. of the 2010 Int'l Symp. on Memory Management. Toronto: ACM, 2010. 31–40. [doi: 10.1145/1806651.1806657]

[5] Liu X, Tong W, Liu JN, Feng D, Chen JL. A review of dynamic memory allocator research. Chinese Journal of Computers, 2018, 41(10): 2359–2378 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.02359]

[6] Zeng K, Chen YQ, Cho H, Xing XY, Doupé A, Shoshitaishvili Y, Bao T. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 71–88.

[7] Xu W, Li JR, Shu JL, Yang WB, Xie TY, Zhang YY, Gu DW. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. Denver: ACM, 2015. 414–425. [doi: 10.1145/2810103.2813637]

[8] Chen YQ, Xing XY. SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 1707–1722. [doi: 10.1145/3319535.3363212]

[9] Wu W, Chen YQ, Xu J, Xing XY, Gong XR, Zou W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 781–797.

[10] Alexopoulos N, Brack M, Wagner JP, Grube T, Mühlhäuser M. How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 359–376.

[11] Yang ST, Chen KX, Wang Z, Zhang C. Exploit-oriented automated information leakage. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2082–2096 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6570.htm [doi: 10.13328/j.cnki.jos.006570]

[12] Palit T, Monrose F, Polychronakis M. Mitigating data leakage by protecting memory-resident sensitive data. In: Proc. of the 35th Annual Computer Security Applications Conf. San Juan: ACM, 2019. 598–611. [doi: 10.1145/3359789.3359815]

[13] mm: SLAB freelist randomization. 2021. https://lwn.net/Articles/685047/

[14] Linux kernel heap quarantine versus use-after-free exploits. 2022. https://a13xp0p0v.github.io/2020/11/30/slab-quarantine.html

[15] Introduce struct layout randomization plugin. 2021. https://lwn.net/Articles/723997/

[16] Lin ZP. How AUTOSLAB changes the memory unsafety game. 2021. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game

[17] Improve bit diffusion for freelist ptr obfuscation. 2022. https://lore.kernel.org/lkml/202003051623.AF4F8CB@keescook/

[18] Gregg B. BPF internals tracing examples (eBPF). 2021. https://www.brendangregg.com/Slides/LISA2021_BPF_Internals.pdf

[19] Lin ZP, Wu YH, Xing XY. DirtyCred: Escalating privilege in Linux kernel. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 1963–1976. [doi: 10.1145/3548606.3560585]

[20] Gens D, Schmitt S, Davi L, Sadeghi AR. K-Miner: Uncovering memory corruption in Linux. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2018. [doi: 10.14722/ndss.2018.23326]

[21] Manès VJM, Jang D, Ryu C, Kang BB. Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions. Computers & Security, 2018, 74: 130–143. [doi: 10.1016/j.cose.2018.01.009]

[22] Göktas E, Razavi K, Portokalidis G, Bos H, Giuffrida C. Speculative probing: Hacking blind in the Spectre era. In: Proc. of the 2020 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2020. 1871–1885. [doi: 10.1145/3372297.3417289]

[23] Tian DJ, Hernandez G, Choi JI, Frost V, Johnson PC, Butler KR. LBM: A security framework for peripherals within the Linux kernel. In: Proc. of the 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 967–984. [doi: 10.1109/SP.2019.00041]

[24] Szekeres L, Payer M, Wei T, Song D. SoK: Eternal war in memory. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 48–62. [doi: 10.1109/SP.2013.13]

[25] Chen WT, Zou XC, Li GR, Qian ZY. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In: Proc. of the 29th USENIX Conf. on Security Symp. USENIX Association, 2020. 1093–1110.

[26] Caballero J, Grieco G, Marron M, Nappa A. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proc. of the 2012 Int'l Symp. on Software Testing and Analysis. Minneapolis: ACM, 2012. 133–143. [doi: 10.1145/2338965.2336769]

[27] Chen YQ, Lin ZP, Xing XY. A systematic study of elastic objects in kernel exploitation. In: Proc. of the 2020 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2020. 1165–1184. [doi: 10.1145/3372297.3423353]

[28] Add naive detection of double free. 2022. https://lore.kernel.org/lkml/20200625215548.389774-3-keescook@chromium.org/

[29] Weaknesses in Linux kernel heap hardening. 2021. https://blog.infosectcbr.com.au/2020/03/weaknesses-in-linux-kernel-heap.html

[30] Bit flipping attacks against free list pointer obfuscation. 2021. https://blog.infosectcbr.com.au/2020/04/bit-flipping-attacks-against-free-list.html

[31] Erdős M, Ainsworth S, Jones TM. MineSweeper: A “clean sweep” for drop-in use-after-free prevention. In: Proc. of the 27th ACM Int’l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 212–225. [doi: 10.1145/3503222.3507712]

[32] Liu DP, Zhang MW, Wang HN. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 1635–1648. [doi: 10.1145/3243734.3243826]

[33] Ainsworth S, Jones TM. MarkUs: Drop-in use-after-free prevention for low-level languages. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 578–591. [doi: 10.1109/SP40000.2020.00058]

[34] Chen P, Xu J, Lin ZQ, Xu DY, Mao B, Liu P. A practical approach for adaptive data structure layout randomization. In: Proc. of the 20th European Symp. on Research in Computer Security. Vienna: Springer, 2015. 69–89. [doi: 10.1007/978-3-319-24174-6_4]

[35] Kim J, Jang D, Jeong Y, Kang BB. POLaR: Per-allocation object layout randomization. In: Proc. of the 49th Annual IEEE/IFIP Int’l Conf. on Dependable Systems and Networks. Portland: IEEE, 2019. 505–516. [doi: 10.1109/DSN.2019.00058]

[36] Gershuni E, Amit N, Gurfinkel A, Narodytska N, Navas JA, Rinetzky N, Ryzhyk L, Sagiv M. Simple and precise static analysis of untrusted Linux kernel extensions. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Phoenix: ACM, 2019. 1069–1084. [doi: 10.1145/3314221.3314590]

[37] Xhonneux M, Duchene F, Bonaventure O. Leveraging eBPF for programmable network functions with IPv6 segment routing. In: Proc. of the 14th Int’l Conf. on emerging Networking EXperiments and Technologies. Heraklion: ACM, 2018. 67–72. [doi: 10.1145/3281411.3281426]

[38] Zhong YH, Li HY, Wu YJ, Zarkadas I, Tao J, Mesterhazy E, Makris M, Yang JF, Tai A, Stutsman R, Cidon A. XRP: In-kernel storage functions with eBPF. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: OSDI, 2022. 375–393.

[39] Park S, Zhou DY, Qian YC, Calciu I, Kim T, Kashyap S. Application-informed kernel synchronization primitives. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: OSDI, 2022. 667–682.

[40] Kaffes K, Humphries JT, Mazières D, Kozyrakis C. Syrup: User-defined scheduling across the stack. In: Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles. ACM, 2021. 605–620. [doi: 10.1145/3477132.3483548]

[41] He Y, Zou ZH, Sun K, Liu ZT, Xu K, Wang Q, Shen C, Wang Z, Li Q. RapidPatch: Firmware hotpatching for real-time embedded devices. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 2225–2242.

[42] Sun H, Shen YH, Liu JZ, Xu YR, Jiang Y. KSG: Augmenting kernel fuzzing with system call specification generation. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 351–366.

[43] Novark G, Berger ED. DieHarder: Securing the heap. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. Chicago: ACM, 2010. 573–584. [doi: 10.1145/1866307.1866371]

[44] Silvestro S, Liu HY, Liu TY, Lin ZQ, Liu TP. Guarder: A tunable secure allocator. In: Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 117–133.

[45] Silvestro S, Liu HY, Crosser C, Lin ZQ, Liu TP. FreeGuard: A faster secure heap allocator. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2389–2403. [doi: 10.1145/3133956.3133957]

[46] Di Luna GA, Italiano D, Massarelli L, Österlund S, Giuffrida C, Querzoni L. Who’s debugging the debuggers? Exposing debug information bugs in optimized binaries. In: Proc. of the 26th ACM Int’l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM, 2021. 1034–1045. [doi: 10.1145/3445814.3446695]

[47] eBPF: Introduction, Tutorials & Community Resources. 2022. https://ebpf.io/

[48] Emamdoost N, Wu QS, Lu KJ, McCamant S. Detecting kernel memory leaks in specialized modules with ownership reasoning. In: Proc. of the 28th Annual Network and Distributed System Security Symp. NDSS, 2021. [doi: 10.14722/ndss.2021.24416]

[49] Heelan S, Melham T, Kroening D. Automatic heap layout manipulation for exploitation. In: Proc. of the 27th USENIX Conf. on Security Symp. Baltimore: USENIX Association, 2018. 763–779.

[50] CVE. CVE-2010-2959. 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959

[51] CVE. CVE-2017-7533. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533

[52] CVE. CVE-2021-22555. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22555

[53] CVE. CVE-2022-34918. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-34918

[54] CVE. CVE-2017-7184. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7184

[55] CVE. CVE-2016-8655. 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655

[56] CVE. CVE-2021-26708. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26708

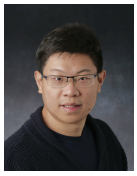
[57] CVE. CVE-2020-16119. 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-16119
[58] CVE. CVE-2017-10661. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10661
[59] CVE. CVE-2016-10150. 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10150
[60] CVE. CVE-2017-11176. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11176
[61] CVE. CVE-2017-15649. 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15649
[62] Li YW, Dolan-Gavitt B, Weber S, Cappos J. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In: Proc. of the 2017 USENIX Conf. on USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2017. 1–13.
[63] McVoy L, Staelin C. LMBench: Portable tools for performance analysis. In: Proc. of the 1996 Conf. on USENIX Annual Technical Conf. San Diego: USENIX Association, 1996. 23.
[64] Linux Hardware Reviews & Performance Benchmarks, Open-source News. 2022. https://www.phoronix.com/
[65] Ren X, Rodrigues K, Chen LY, Vega C, Stumm M, Yuan D. An analysis of performance evolution of Linux's core operations. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 554–569. [doi: 10.1145/3341301.3359640]
[66] Suchy B, Campanoni S, Hardavellas N, Dinda P. CARAT: A case for virtual memory through compiler-and runtime-based address translation. In: Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation. London: ACM, 2020. 329–345. [doi: 10.1145/3385412.3385987]
[67] Fisher-Yates shuffle. 2022. https://algorithm-wiki.csail.mit.edu/wiki/Fisher%E2%80%93Yates_Shuffle
[68] Roessler N, Chien Y, Atayde L, Yang PR, Palmer I, Gray L, Dautenhahn N. Lossless instruction-to-object memory tracing in the Linux kernel. In: Proc. of the 14th ACM Int'l Conf. on Systems and Storage. Haifa: ACM, 2021. 2. [doi: 10.1145/3456727.3463767]
[69] CVE. CVE-2017-7308. 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7308
[70] Ubuntu Livepatch Service. 2022. https://ubuntu.com/security/livepatch
[71] Oracle Ksplice. 2022. https://ksplice.oracle.com/
[72] Live Kernel Patching Using kGraft. 2022. https://documentation.suse.com/sles/12-SP4/html/SLES-kgraft/index.html
[73] Introducing kpatch: Dynamic kernel patching. 2022. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching
[74] An introduction to KProbes. 2022. https://lwn.net/Articles/132196/
[75] Yun I, Kapil D, Kim T. Automatic techniques to systematically discover new heap exploitation primitives. In: Proc. of the 29th USENIX Security Symp. USENIX Association, 2020. 1111–1128.
[76] Van Der Kouwe E, Nigade V, Giuffrida C. DangSan: Scalable use-after-free detection. In: Proc. of the 12th European Conf. on Computer Systems. Belgrade: ACM, 2017. 405–419. [doi: 10.1145/3064176.3064211]
[77] Wang Y, Gao FJ, Ma KX, Situ LY, Wang LZ, Chen BH, Liu Y, Zhao JH, Li XD. Detecting and preventing dangling pointers. Ruan Jian Xue Bao/Journal of Software, 2020, 31(6): 1600−1618 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5994.htm [doi: 10.13328/j.cnki.jos.005994]
[78] Cho H, Park J, Oest A, Bao T, Wang RY, Shoshitaishvili Y, Doupé A, Ahn GJ. ViK: Practical mitigation of temporal memory safety violations through object ID inspection. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 271–284. [doi: 10.1145/3503222.3507780]
[79] Dang THY, Maniatis P, Wagner DA. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In: Proc. of the 26th USENIX Security Symp. Vancouver: USENIX Association, 2017. 815–832.
[80] Shen ZK, Dolan-Gavitt B. HeapExpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities. In: Proc. of the 2020 Annual Computer Security Applications Conf. Austin: ACM, 2020. 454–465. [doi: 10.1145/3427228.3427645]
[81] Proskurin S, Momeu M, Ghavamnia S, Kemerlis VP, Polychronakis M. xMP: Selective memory protection for kernel and user space. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 563–577. [doi: 10.1109/SP40000.2020.00041]
[82] Song CY, Lee B, Lu KJ, Harris W, Kim T, Lee W. Enforcing kernel security invariants with data flow integrity. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego: NDSS, 2016. [doi: 10.14722/ndss.2016.23218]
[83] Abubakar M, Ahmad A, Fonseca P, Xu DY. SHARD: Fine-grained kernel specialization with context-aware hardening. In: Proc. of the 30th USENIX Security Symp. USENIX Association, 2021. 2435–2452.
[84] Berger ED, Zorn BG. DieHard: Probabilistic memory safety for unsafe languages. ACM SIGPLAN Notices, 2006, 41(6): 158–168. [doi: 10.1145/1133255.1134000]

附中文参考文献:

[5] 刘翔, 童薇, 刘景宁, 冯丹, 陈劲龙. 动态内存分配器研究综述. 计算机学报, 2018, 41(10): 2359–2378. [doi: 10.11897/SP.J.1016.2018.

02359]
[11] 杨松涛, 陈凯翔, 王准, 张超. 面向缓解机制评估的自动化信息泄露方法. 软件学报, 2022, 33(6): 2082–2096. http://www.jos.org.cn/1000-9825/6570.htm [doi: 10.13328/j.cnki.jos.006570]
[77] 王豫, 高凤娟, 马可欣, 司徒凌云, 王林章, 陈碧欢, 刘杨, 赵建华, 李宣东. 垂悬指针检测与防御方法. 软件学报, 2020, 31(6): 1600–1618. http://www.jos.org.cn/1000-9825/5994.htm [doi: 10.13328/j.cnki.jos.005994]

王子成(1996－), 男, 博士生, 主要研究领域为操作系统安全, 漏洞攻击利用.



陈越琦(1995－), 男, 博士, 助理教授, 主要研究领域为系统安全, 软件安全.

郭迎港(1997－), 男, 博士生, 主要研究领域为操作系统安全, 形式化建模.

曾庆凯(1963－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为信息安全, 分布计算.

钟炳南(1991－), 男, 博士生, 主要研究领域为信息安全, 操作系统.