软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2082−2096 [doi: 10.13328/j.cnki.jos.006570]

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

面向缓解机制评估的自动化信息泄露方法*

杨松涛[1], 陈凯翔[2], 王 准[2], 张 超[2]

[1](清华大学 计算机科学与技术系, 北京 100084)

[2](清华大学 网络科学与网络空间研究院, 北京 100084)

通信作者: 张超, E-mail: chaoz@tsinghua.edu.cn

摘 要: 自动生成漏洞利用样本(AEG)已成为评估漏洞的最重要的方式之一, 但现有方案在目标系统部署有漏洞缓解机制时受到很大阻碍. 当前主流的操作系统默认部署多种漏洞缓解机制, 包括数据执行保护(DEP)和地址空间布局随机化(ASLR)等, 而现有 AEG 方案仍无法面对所有漏洞缓解情形. 提出了一种自动化方案 EoLeak, 可以利用堆漏洞实现自动化的信息泄露, 进而同时绕过数据执行保护和地址空间布局随机化防御. EoLeak 通过动态分析漏洞触发样本(POC)的程序执行迹, 对执行迹中的内存布局进行画像并定位敏感数据(如代码指针), 进而基于内存画像自动构建泄漏敏感数据的原语, 并在条件具备时生成完整的漏洞利用样本. 实现了 EoLeak 原型系统, 并在一组夺旗赛(CTF)题目和多个实际应用程序上进行了实验验证. 实验结果表明, 该系统具有自动化泄露敏感信息和绕过 DEP 及 ASLR 缓解机制的能力.

关键词: 信息泄漏; 自动生成漏洞利用样本; 动态分析; 污点分析; 内存画像

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/6570.htm

Exploit-oriented Automated Information Leakage

YANG Song-Tao[1], CHEN Kai-Xiang[2], WANG Zhun[2], ZHANG Chao[2]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Institute for Network Science and Cyberspace, Tsinghua University, Beijing 100084, China)

Abstract: Automated exploit generation (AEG) has become one of the most important ways to demonstrate the exploitability of vulnerabilities. However, state-of-the-art AEG solutions in general assume that the target system has no mitigations deployed, which is not true in modern operating systems since they often deploy mitigations like data execution prevention (DEP) and address space layout randomization (ASLR). This paper presents an automated solution EoLeak, able to exploit heap vulnerabilities to leak sensitive data and bypass ASLR and DEP at the same time. At a high level, EoLeak analyzes the program execution trace of the POC input that triggers the heap vulnerability, characterizes the memory profile from the trace and locates important data (e.g., code pointers), constructs leak primitives that discloses sensitive data, and generates exploits for the entire process when possible. A prototype of EoLeakis implemented and it is evaluated on a set of CTF binary programs and several real-world applications. Evaluation results show that EoLeak is effective in terms of leaking data and generating exploits.

Key words: information leakage; automated exploit generation; dynamic analysis; taint analysis; memory profiling

自动生成漏洞利用样本(automatic exploit generation, AEG)[1,2]已经成为评估漏洞的最重要方式之一. 给定

* 基金项目: 国家重点研发计划(2021YFB2701000); 国家自然科学基金(61972224, U1736209)
本文由“系统软件安全”专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.
收稿时间: 2021-09-05; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

一个有漏洞的二进制程序以及触发漏洞的 POC (proof-of-concept)样本, AEG 系统可以自动分析目标二进制程序并生成漏洞利用样本. AEG 不仅可以辅助生成攻击, 同样可以辅助防御. 例如, 软件供应商可以使用 AEG 工具来评估软件漏洞的威胁级别, 并确定漏洞修复的紧迫性.

近年来, 研究人员已提出许多 AEG 方案. 早期工作[3−8]主要关注对栈和格式化字符串漏洞的分析, 其漏洞利用模式相对固定、有效. 而近年来的工作[9−15]更注重堆等复杂类型漏洞, 需要更复杂的利用技巧, 例如堆内存布局操控, 以构建可行的漏洞利用样本. 然而, 现有 AEG 工作很少考虑目标环境中部署有漏洞缓解等防御机制的场景, 这些漏洞缓解机制为漏洞利用带来极大的挑战. 事实上, 随着现代操作系统广泛部署各种漏洞缓解机制, 若攻击者想在如今的生产服务环境中进行实际攻击, 突破目标环境中的防御机制是必须解决的重要问题.

现代操作系统中广泛部署了 3 种著名的防御机制, 包括数据执行保护(NX/DEP)[16]、栈保护变量(Canary/Cookie)[17]和地址空间布局随机化(address space layout randomization, ASLR)[18]. 数据执行保护 DEP 的目标是, 防止内存中写入的数据被当作代码来执行. 设置栈保护变量 Canary 可以在栈缓冲区溢出漏洞覆盖栈帧中的函数返回地址时检测到该破坏行为. 地址空间布局随机化 ASLR 则将内存中的代码段、数据段、堆栈等的基地址随机化, 使攻击者难以找到可用的重要数据和代码的地址, 从而使得攻击变得更加困难. 除此之外, 还有一些其他漏洞利用缓解措施, 包括控制流完整性解决方案[19−21]等, 也可以有效缓解漏洞利用. 但是由于性能和兼容性等各种问题, 厂商尚未广泛部署这些保护措施. 因而, 自动生成漏洞利用样本的 AEG 方案目前主要需要考虑 DEP、ASLR、Canary 等防御机制绕过即可.

在当前的 AEG 工作中, 部分方案[4,5,22,23]假设目标环境和程序没有启用防御机制, 部分方案[8,12−14]可以绕过数据执行保护但无法对抗地址空间布局随机化, 部分方案[3,6,7,24−26]可以通过栈漏洞绕过地址空间布局随机化保护, 或在没有数据执行保护的情况下通过堆漏洞绕过地址空间布局随机化. 目前, 尚未有工作研究通过堆漏洞绕过数据执行保护和地址空间布局随机化的防御.

绕过地址空间布局随机化防护的关键在于泄漏被随机化的内存地址. 目前部署在现代操作系统中的地址空间布局随机化防御基于大尺寸的内存段, 包括栈、堆和共享库, 粒度相对较粗. 其中, 每个段内的偏移是不受随机化影响的固定值. 绕过地址空间布局随机化的最有效策略是泄漏某个段的地址, 从这个地址可以推断出攻击者所需的所有在同一段内的其他地址, 其位于固定偏移处. 因此, 我们需要找到一个包含随机化后地址的指针进行泄漏. 常见的方法包括借用目标二进制程序本身的语义或构建新的输出功能, 触发输出函数以打印出该地址. 通过这个泄漏的随机地址, 攻击者可以推断其他代码地址并基于此完成漏洞利用.

本文提出了一种面向缓解机制评估的自动化信息泄漏系统 EoLeak, 可以通过堆漏洞同时绕过数据执行保护和地址空间布局随机化, 达成利用效果. EoLeak 首先通过对触发漏洞的 POC 的运行时内存执行过程进行动态分析, 定位敏感信息的变量位置, 然后自动地为敏感信息变量构建信息泄漏, 并根据泄漏的信息生成漏洞利用样本. 为了定位内存中的重要变量, 方案通过构建内存画像来记录所有有关信息并搜索可能的泄漏路径. 为了构建泄漏能力, 方案通过堆漏洞分析模型来实现对堆漏洞的初步攻击, 扩展了内存利用能力. 方案还执行了一种轻量级动态污点策略, 通过监控内存缓冲区的传输操作来搜索具有用户可控参数的库函数调用, 降低了插桩开销. 对于最终的漏洞利用, 方案遵循相似的策略来构建用户可控的库函数调用, 并妥善处理了泄漏失败的特殊情形.

我们在基于 QEMU 的记录和重放平台 PANDA[27]上实现了该系统, 并通过 17 个 CTF 堆漏洞二进制程序和 5 个真实世界软件对其进行了评估. 结果表明, 系统成功生成了 15 个自动泄漏样本和 14 个最终利用样本. 对于真实软件, 系统可以自动分析、定位敏感信息的运行时内存地址和相关指针.

1 研究案例

针对堆漏洞绕过数据执行保护和地址空间布局随机化的 AEG 解决方案面临着与分析人员手工构造利用样本相同的问题. 本节中, 我们通过对一个实际的堆漏洞利用案例进行研究, 来概述所面临的挑战和本文所

提出的泄漏系统解决方案.

1.1 堆漏洞利用案例

如图 1 所示, 目标程序逻辑简化后可以表示为图 1 所示代码(level 参数确保合法), 其中存在一个释放后使用(use after free)的堆漏洞. 程序在第 20 行调用堆块释放函数时未检查指向目标堆块的 buf 指针是否合法, 也未在堆块被释放后将指向堆块的 buf 指针置零.

```
1  int sizes[3] = {0x28, 0xfa0, 0x61a80};
2  char* buf[3];
3  bool exist[3];
4
5  void create(int level) {
6    if (!exist[level]) {
7      buf[level] = calloc(1, sizes[level]);
8      read(buf[level], sizes[level]);
9      exist[level] = true;
10   }
11 }
12
13 void renew(int level) {
14   if (exist[level]) {
15     read(buf[level], sizes[level]);
16   }
17 }
18
19 void remove(int level) {
20   free(buf[level]);
21   exist[level] = false;
22 }
```

图 1 堆漏洞示例代码

POC 通过构造重叠的堆块内存布局来触发漏洞. 如图 2 所示, 矩形代表堆块头部和堆块区域, 侧方的矩形条代表对应堆块指针 buf 的当前指向区域. 当依次执行: (a) 创建小堆块; (b) 释放小堆块; (c) 创建大堆块; (d) 释放小堆块; (e) 创建小堆块; (f) 创建中堆块之后, 不仅所有堆块所对应的存在指示都被置为真, 且大堆块指针和小堆块指针发生了重叠, 都指向了(f)中小堆块的起始地址, 还让大堆块指针指向的区域可以对包括小堆块区域、中堆块头部、中堆块区域在内的内存区域进行覆盖. 这样, 对大堆块的写入可以造成对中堆块头部的堆溢出.

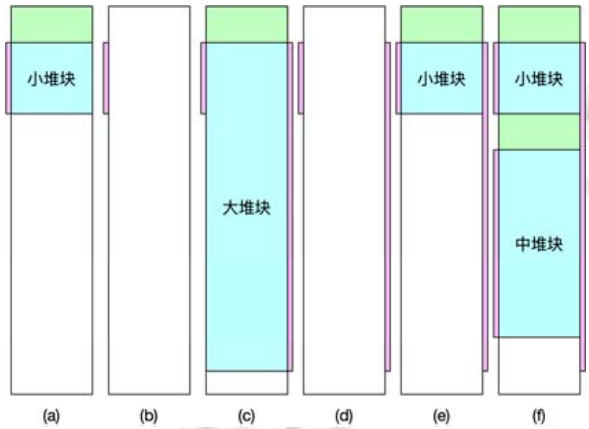

图 2 输入漏洞触发样本时程序执行过程中堆内存布局变化示意

当攻击者试图利用这类启用了地址空间布局随机化防护的漏洞时, 其首先尝试泄漏 libc 库的随机化后的地址. 通过在小堆块中创建一个伪造的堆块并执行 safe unlink 攻击, 指向小堆块的指针将被篡改为一个略低于该指针本身地址的值, 小堆块包含了指向大堆块的指针. 此时, 便可以通过依次修改小堆块和大堆块的内容, 先后篡改大堆块指针的值及其指向内存区域的数据, 实现任意地址写. 攻击者可以借助任意地址写原语用输出库函数的过程链接表地址替换全局偏移表中的某个 libc 库函数地址, 并在触发该函数调用时以任意

libc 库函数指针为第 1 个参数, 从而实现了泄漏 libc 库函数地址. 借助该泄漏地址, 攻击者可以计算 libc 库中 *system* 函数和“/bin/sh”字符串等信息的实际运行时地址, 将其写回堆块中并触发 *system* 函数调用, 以完成漏洞利用.

1.2 攻击模型

本文假定在目标环境中启用了 3 种广泛部署的防御机制, 包括数据执行保护[16]、栈保护变量[17]和地址空间布局随机化[18]. 同时还假定目标二进制程序中存在一个常见的堆漏洞可以利用, 例如释放后使用或堆溢出.

此外, 本文假定攻击者拥有一个可以触发堆漏洞的漏洞触发样本, 广泛发展的漏洞发现工具可以满足这一要求. 由于现代 AEG 解决方案通常都允许用户提供对应的漏洞利用输入模板[28], 本文允许攻击者对漏洞触发样本做一些划分以辅助跟踪分析. 本文还假定攻击者可以通过最近的一些堆布局操纵工作[9−11]掌握堆风水能力, 以便通过漏洞触发样本提供一个相对方便而确定的堆布局, 辅助进一步分析.

1.3 研究挑战

在漏洞利用过程中, 攻击者需要找到一个随机化后的 libc 库函数地址进行泄漏, 然后依据泄漏的地址信息生成漏洞利用样本. 泄漏和利用生成都要求以攻击者控制的参数调用某些特定的库函数(包括打印数据和系统函数). 为了实现这一目标, 需要解决以下多个挑战.

(1) 挑战 1: 哪些重要的敏感信息值得泄漏, 又如何定位其在内存中的位置? 例如, 要绕过地址空间布局随机化, 需要通过信息泄漏来获得一个随机化后的地址. 在仅给定目标二进制文件和 POC 时, 需要对运行过程中的内存结构构建内存画像;

(2) 挑战 2: 如何泄漏所定位的敏感信息? 用户只能通过向程序提供输入来影响二进制程序的执行过程. 为了实现信息泄漏, 攻击者需要确定正确的输入以触发打印特定位置数据的功能;

(3) 挑战 3: 如何根据泄漏的信息生成漏洞利用样本? 即便信息遭到泄漏, 仍与漏洞利用存在一定的距离, 需要额外工作来跨越这其中的障碍.

1.4 自动泄漏方案

为了解决上述挑战, 本文提出了一种新颖的解决方案 EoLeak, 来自动执行面向漏洞利用的敏感信息泄漏和相应的漏洞利用样本生成. 总的来说, 方案对二进制程序执行迹进行动态分析, 对运行时内存结构构建内存画像, 定位有价值的数据变量, 构建用户可控的读写能力, 并生成绕过地址空间布局随机化和数据执行保护的堆漏洞利用样本.

方案使用给定的 POC 来分析二进制文件的运行过程, 并识别参与计算的指针和敏感变量, 将相关内存信息及时间戳记录在图中, 从而可以找到通往指定内存位置的一条嵌套指针链条.

为了构建泄漏能力, 方案首先基于堆内存模型, 利用堆漏洞来获得更广泛的内存操纵能力. 通过执行轻量级动态污点分析来跟踪用户输入字节, 方案分析用户可控内存区域, 从二进制程序执行路径中提取抽象的堆操作信息, 并从预先设定的堆利用模板列表中检查相符合的堆漏洞利用条件是否被满足. 为了减少污点分析性能开销, 并直接获得用户输入和内存数据之间的联系, 本文采用的污点分析只考虑内存传输操作作为传播策略.

构建漏洞利用样本遵循与构建泄漏能力相似的策略. 方案通过任意或可控的内存写来以指定的参数调用目标库函数, 执行漏洞利用. 对于自动泄漏失败或二进制文件受到其他高级防御机制保护的情形, 方案还尝试使用相应的堆漏洞利用技术来直接生成漏洞利用样本.

2 系统设计

本节介绍了 EoLeak 方案的设计细节. 如图 3 所示, 主要有 3 个步骤.

(1) 敏感信息定位. 给定一个有漏洞的二进制程序和一个触发堆漏洞的 POC, 方案首先分析程序执行迹, 提取指令语义, 通过恢复指针和有价值的内存对象来定位它们的位置. 我们通过内存画像来记

录变量地址并维护一个指针图来搜索到特定地址的泄漏路径;

(2) 构建信息泄漏. 方案利用堆模型, 通过堆漏洞利用模板来实现漏洞推断, 以扩展内存操纵能力并执行轻量级动态污点分析, 来研究用户输入与库函数调用参数之间的关系. 通过构建读写功能来实现参数受控, 方案可以构建泄漏原语来打印出敏感信息;

(3) 生成利用样本. 方案通过与构建泄漏能力类似的策略来依据泄漏信息生成目标二进制文件的最终利用样本. 如果无法触发选定参数的漏洞利用库函数调用, 方案还会使用相应的模板来处理特定的堆漏洞利用情形作为补充.

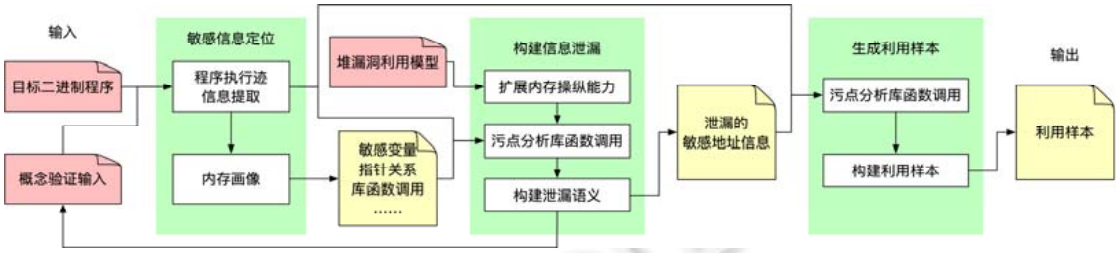

图 3 自动化信息泄漏系统总览

2.1 敏感信息定位

EoLeak 方案的第 1 步是使用 POC 运行给定的有漏洞的二进制程序, 并在执行过程中探索敏感信息.

2.1.1 重要数据

在程序的整个执行过程中会产生许多指针和内存对象参与计算. 其中, 我们认为以下 3 种信息值得注意.

(1) 程序执行中使用的代码指针, 尤其是位于可写内存区域的代码指针;

(2) 内存对象指针, 包括由堆分配函数返回的堆块指针;

(3) 程序执行中频繁访问的变量, 或作为函数调用参数的变量.

可写代码指针在漏洞利用中非常宝贵, 因为通过篡改这些指针, 攻击者可以劫持控制流. 并且可写代码指针在随机化后的代码空间中为推理共享库的基地址提供了参考. 敏感数据通常作为成员变量被存储在内存对象中, 内存对象的地址空间在堆管理器控制之下, 我们通过堆块指针来掌握运行时堆的状态, 从而推理内存对象在内存空间中的分布. 部分变量, 如数据结构的起始指针经常参与后续数据结构的运算, 或常作为参数被应用程序接口调用. EoLeak 方案认为这些变量对漏洞利用生成也很重要, 记录了这类变量的访问频率.

2.1.2 内存画像

为了识别和定位上述敏感信息, EoLeak 方案在执行动态程序执行迹分析时构建了内存画像: 对每条指令进行反汇编以获取其操作码和操作数信息, 并从执行迹中提取对应操作数的运行时变量值. 通过分析寄存器和内存操作数的寻址模式, 方案识别指针并对访问进行计数, 然后将数据添加到全局内存指向映射中, 该映射保存了所有记录的内存位置和变量声明周期时间戳.

为了定位保存 libc 等库函数地址的指针, EoLeak 方案首先查找所有属于该库内存段的运行时地址变量值. 通过一个单独的进程监控器来获取内存段的地址区域信息后, EoLeak 方案记录所有运行时间接调用和跳转目标, 检查并过滤出属于该内存段的那些地址, 并执行一个单独的验证过程来比对确认每个库地址确实对应于一个库函数符号.

如图 4 所示, 样例程序的指针内存画像中包含了两个指针的嵌套: 位于 0x6020b0 处的 pS 指针指向包含指针 pM 和 pH 的内存区域 0x602098, 而 pM 和 pH 这两个指针又分别指向单独的内存区域. 当 pH 指向的以 0x175e010 为起始地址的内存区域中包含一个有价值的敏感数据时, 便可以通过使用两个连续的泄漏读取来构建对该重要数据的泄漏: 先读取*pS+0x10 处 pH 指针的值, 再读取*pH+0x8 处的敏感信息变量. 由于内存中的偏移量是相对固定的, 只要知道第 1 个位于 0x6020b0 处的 pS 指针地址, 就可以沿着链条打印出整个链上

的每一个节点.

图 4 样例漏洞程序部分运行时指针内存画像

在收集到的内存分析图中, 我们将内存描述为一系列由指针指向的嵌套内存区域, 每个指针都位于特定内存区域的某个偏移处. 给定一个用户可以读取的已知地址指针和一个漏洞时间范围, EoLeak 方案可以自动搜索到指定目标变量的指针泄漏链条, 从而通过连续任意读功能应用这条泄漏路径.

2.2 构建信息泄漏

定位敏感信息的位置后, EoLeak 方案构建泄漏能力来打印出敏感信息. 正如在上一小节中所阐释的, 对特定变量的泄漏可以被拆分为没有冲突的连续任意读取. 因此, 方案首先构造单个任意读取的泄漏功能, 然后将其拼接以形成任意读取链条. 更具体地说, 方案分析目标二进制程序的语义, 结合堆漏洞构建泄漏能力, 以目标变量地址作为参数调用打印输出库函数, 最后触发函数调用完成泄漏过程.

2.2.1 扩展内存操纵能力

开发者编写的程序通常对用户输入的数据有严格的校验, 避免其破坏内存造成漏洞攻击, 导致目标二进制程序本身往往没有足够的执行语义来实现可控的库函数调用. 因此, 必要的步骤是通过结合堆漏洞攻击来提供更多的可利用原语以扩展可用的内存操纵范围. 攻击者可以手动调整 POC 输入来实现对堆漏洞的初始攻击, EoLeak 也提供了一套通过简单的堆模型和漏洞利用模板来自动推导和构建这类堆利用攻击的方案.

现有的堆漏洞攻击基本上来源于成熟的堆漏洞利用技术, 针对各堆管理器的攻击方案也在发展中变得特化. glibc 当前采用的 ptmalloc2 为最常用的堆管理器之一, 通过 arena 结构及其中的 bin 数组为堆上动态分配的内存提供高效的管理. 在 ptmalloc2 中, 每个 malloc 分配出来的内存都有统一的 chunk 堆块结构, 其中包含头部数据用于管理堆结构, 存储了前一堆块的大小 prev_size 和该堆块的大小 size. 空闲的堆块复用了主体部分的首段字节作为 FD 和 BK 指针(分别指向下一个和上一个空闲的堆块). 每个分配出来的内存堆块依据其大小和使用状态被归为以下 4 类之一: (1) fast bin; (2) small bin; (3) large bin; (4) unsorted bin.

EoLeak 针对 ptmalloc2 堆管理器抽象出了典型堆漏洞利用方案的模板, 包括触发漏洞攻击所需的条件和执行攻击后的利用效果. 具体来说, EoLeak 提供了一个堆漏洞利用模板列表, 每个模板对应一种堆利用攻击方案, 包含了所需条件(如堆块分配大小限制、堆块头部溢出可能性)和攻击效果(如任意地址的堆分配).

表 1 中列出了当前支持的主要利用模板. 我们以样例程序中执行的 safe unlink 攻击为例. 在未执行 safe unlink 攻击之前, 目标二进制程序内存中没有攻击者可写的指针, 无法实现进一步的泄漏. safe unlink 攻击是一种释放后使用攻击, 它通过可控的 FD 和 BK 指针错误地释放一个已经被释放的 small bin 或 unsorted bin. 在堆利用模板中, EoLeak 要求必须满足以下条件才能实现攻击.

(1) 在已知地址处有一个指向某堆块的指针;

(2) 该堆块可布置为一个可以通过检查的被释放的堆块, 其 FD 和 BK 对应于偏移处都可具有的受控的内容;

(3) 该堆块的相邻堆块可以触发释放操作.

对于目标二进制程序而言, 条件(1)是静态内存状态, 可以通过内存画像的简单检查来获得. 条件(2)要求

该堆块具有满足约束条件的写能力. 条件(3)则需要对相邻的堆块执行释放, 可以通过对程序执行迹进行分析而获得.

该模板的攻击效果则可被总结为: 已知指针被设置为指向略低于指针所在地址处.

表 1 EoLeak 当前支持的堆漏洞利用模板列表

堆漏洞攻击种类	攻击条件	利用效果	可扩展原语
Fastbin 攻击	(1) 存在两个 fastbin 堆块可被按顺序释放; (2) 先释放堆块, 释放后可修改 FD; (3) 可再次按顺序分配 fastbin 堆块	后释放 fastbin 堆块可被分配至重叠堆地址	扩展指针指向目标内存区域范围, 制造重叠堆块
Unlink 攻击	(1) 在已知地址处存在指向某堆块的指针; (2) 该堆块可被设置为已释放堆块, FD 和 BK 可控; (3) 该堆块的相邻堆块可触发释放操作	已知指针被设置为指定值	扩展指针指向目标内存区域范围, 制造可控指针
Safe Unlink 攻击	(1) 在已知地址处存在指向某堆块的指针; (2) 该堆块可被设置为已释放堆块, FD 和 BK 可控; (3) 该堆块的相邻堆块可触发释放操作	已知指针被设置为指向略低于指针所在地址处	扩展指针指向目标内存区域范围, 制造可控指针
Unsorted Bin 泄漏	(1) 存在堆块可被释放进 unsorted bin; (2) 该堆块被释放后位于 unsorted bin 链表尾部; (3) 该堆块释放后可访问 FD	unsorted bin 堆块 FD 指针指向 main arena 偏移处	制造指向 libc 地址的指针
伪造内存写	(1) 存在两个指针可以对其指向区域进行覆写; (2) 其中一个指针可被修改至指向低于另一指针所在地址处, 且偏移不超过可写内存长度	另一指针值可被一次写入覆盖	制造可控任意写指针

在分析阶段, EoLeak 方案已经记录了指针及对应的内存区域. 通过分析包括堆块头部信息和双向链表指针结构的堆块元数据, EoLeak 方案维护了一个表示运行时堆布局结构的简易堆模型. 方案通过分析程序执行迹来提取抽象的堆操作信息, 收集程序路径中对堆状态的修改语义, 并将这些修改与堆模型结合起来进行比较, 以检查其是否满足任何堆漏洞攻击模板的条件. 如果某个堆漏洞攻击模板的所有条件都成立, 方案便会根据模板执行堆攻击, 验证这一攻击的有效性, 并更新 POC 输入和程序追踪状态. 若后续的分析表明此攻击无效, 则方案回退到攻击之前, 并以此尝试其他攻击模板.

在执行完 POC 后, 样例程序执行到了如图 2(f)所示的堆布局状态, 方案开始检查其是否满足 safe unlink 堆攻击模板的条件. 对于条件(1), 方案在 bss 段找到了几个固定地址的堆块指针. 对于条件(2), 方案发现了程序路径, 其中存在图 4 所示的 pS 指针, 指向的小堆块中可以填充用户伪造的假堆块, 且其长度可以大于小堆块容许的大小, 允许通过堆溢出覆盖后续相邻堆块头部的大小字段和存在标识比特. 虽然小堆块和大堆块都可以填充用户可控的内容, 但只有 pS 满足从指向内存区域起始的可写长度大于剩余块大小的要求, 因此, prev_size 字段和 prev_in_use 比特可以根据需要被覆盖为指定内容(标明前一伪造块为已释放状态, 同时通过大小匹配检查). 对于第 3 个条件, 另一条程序路径满足了触发堆块释放的要求. 3 个条件都得以满足, 因此 EoLeak 方案根据模板构造对应的攻击输入, 依次执行相应的程序路径, 成功更新了当前程序状态.

2.2.2 污点分析可控库函数调用

收集更多内存操纵原语后, EoLeak 方案尝试着构造单个泄漏能力. 在获得库函数实际地址之前, 我们仍然依靠目标二进制程序本身来完成泄漏过程. 因此, 需要找到目标二进制程序中已经存在的输出能力, 并以欲将泄漏的数据地址为参数进行触发.

EoLeak 方案首先在目标二进制程序中查找具有打印输出语义的库函数, 这在构建内存画像的过程中已经收集获得. 进而, 方案分析程序执行迹以确定是否存在合适的程序路径来用可控的参数调用该输出库函数, 这包括两种可控参数情形: 参数直接来源于当前 POC 输入中的某些字节, 或参数来源于当前程序路径中某些可控内存区域的变量传播. 前者可以通过修改当前输入进行简单的调整, 而后者则需要额外的写入程序路径来将所需的地址数据写入相应的内存区域. 另一方面, 如果没有合适的程序路径来调用库函数, 则 EoLeak 方案必须首先通过劫持其他库函数符号来构造泄漏原语, 然后再以可控参数对该函数进行调用. 总之, EoLeak 方案需要了解用户输入可以影响哪些内存区域, 也需要了解函数参数来源于哪些内存区域.

为了跟踪来自用户输入的数据流, 除了反汇编二进制指令并获取运行时数据外, EoLeak 方案还根据指令

语义执行一个轻量级的污点传播分析来跟踪用户输入. 用户输入库函数的结果被标记为污点源, 目标库函数调用参数被标记为污点接受器. 由于动态污点传播是一个比较繁重的过程, 分析工作量大而无法保证精度, 为了降低性能开销并专注于由用户输入字节直接控制的内存区域, EoLeak 方案仅以不超过一阶的算术或位操作的内存传输操作作为污点传播策略. 其背后的思路在于: 如果目标地址完全被攻击者所控制, 则构成该地址的连续字节有很高的概率会作为一个整体在内存中进行传输或执行偏移操作. 在这种情况下, 污点分析仅跟踪原始用户输入, 降低了维护的开销, 并且使后续漏洞利用生成分析更加简洁.

2.2.3 构建泄漏语义

EoLeak 方案对上述内容的分析结果存在多种可能性, 包括可能存在多个读取语义的程序路径可以实现泄漏能力, 打印库函数的参数可能来源于多个可控内存区域, 而每个内存区域也可以支持多种写入方式.

方案在这些候选组成部分中进行选取, 拼接出一个完整的泄漏原语. 由于通过指针链条的连续任意读取泄漏能力要求每个任意读取泄漏之间不能发生冲突, 方案在拼接泄漏路径后对其执行迹进行分析, 计算其带来的副作用, 包括其他内存位置的写入、对堆布局的额外操作等, 对每个生成的库函数调用给予一个副作用评分, 用以衡量副作用的影响, 优先选择副作用较小的泄漏路径. 具体而言, 方案提取每个执行迹中的内存读写和分配释放操作, 并分成两部分进行处理. 对于执行迹中的内存读写, 方案对每个不同地址的内存写入计 3 分, 并对每一组逆序的执行迹中内存读写(即存在某个地址执行迹先读后写)计 1 分. 对于执行迹中的内存分配和释放操作, 方案对每个内存分配和释放计 1 分, 并对每一组不匹配的分配和释放(匹配即一对分配释放操作目标地址相同)计 1 分. 在选择泄漏路径时, 方案按内存分配释放分数和内存读写分数依次优先选择分数较小的泄漏路径.

目标程序的内存画像中记录了所有与随机化后库函数地址相关的指针位置, EoLeak 方案自动地为每个库函数指针计算可能的泄漏链条, 尝试对任一指针进行泄漏, 以获得在地址空间布局随机化后的运行时库函数地址. 分析人员也可以在内存画像图中手动标记目标敏感信息, EoLeak 方案可以尝试为该变量构建相应的泄漏路径.

2.3 生成利用样本

在完成信息泄漏之后, 我们尝试利用泄漏出来的 libc 库函数地址和此前通过模板扩展出的可控内存范围来实现目标二进制程序的利用, 通过使用和构建泄漏语义相似的策略来构建利用语义中的库函数调用, 例如 *system* (“/bin/sh”), 在下文中将以此作为示例. 也有其他可用的库函数, 如 *execve*, 只需要不同的参数组合. 自动构建利用样本过程同样可以分为两个步骤: 使一个可被调用的代码指针具有随机化后的运行时 *system* 函数地址, 并确保第 1 个或相应的参数被填充为“/bin/sh”的地址.

由于在不同的随机化基地址中, 相同库中的两个符号之间总具有相同的偏移量, 所以, *system* 函数的运行时地址可以通过泄漏出的 libc 库函数地址进行计算. 被调用的代码指针可以选取典型代表, 如全局偏移表, 其保存程序中使用的真实库函数地址. libc 库中包含“/bin/sh”这一字符串, 其地址亦可通过泄漏的地址进行计算. 在某些情形下, *system* 库函数调用的第 1 个参数无法由用户输入值直接控制, EoLeak 方案则寻求替代选项: 将参数设置为指向具有可写内存区域的指针, 通过在原始 POC 中附加或修改字节来触发写入过程, 将“/bin/sh”字符串本身写入目标内存区域而非其运行时所在地址. 当 *system* 函数地址和参数都准备好后, 便可通过篡改 libc 库函数符号来触发对 *system* 函数的库函数调用, 实现对目标二进制文件的利用.

EoLeak 方案基于泄漏函数地址的方案来绕过地址空间布局随机化防护. 在泄漏和生成利用样本阶段, 无需注入 shellcode, 而是基于最初堆漏洞利用模板对可用内存范围的扩展, 通过对可控库函数调用目标地址和参数进行分析劫持库函数调用, 通过复用代码以实现攻击, 可以对抗数据执行保护. 在堆漏洞利用的过程中, 一般不依赖于栈溢出功能, 且可以通过内存画像指定栈保护变量来构造对应的泄漏路径, 因此可以对抗栈保护变量的防护.

3 实现方案

本文基于 PANDA 项目实现了 EoLeak 系统. 具体来说, 我们用 C++实现了部分分析代码, 并将整个系统包装为一个基于 Python 的服务器/客户端架构.

3.1 记录和重放

针对随机化保护的分析, 一直是一项艰巨的工作. 随机化值仅在实际执行过程中生成, 在此之前无法观察到相关的随机化程序行为. 动态分析解决了这个问题, 但每次执行时产生的随机结果有所不同, 从而给研究人员带来了新的困难.

EoLeak 系统借助了记录程序执行迹并以确定方式进行重放的技术. 在记录阶段, 所有运行时的环境变量照常生成并被记录在日志文件中. 当重放执行迹时, 每条指令都具有与最初记录时完全相同的执行行为. 一致的运行环境有助于研究人员轻松地比较和分析多次重放之间同一程序执行迹的执行过程, 并准确识别随机化后的库函数地址和相应的内存结构.

3.2 PANDA插件

PANDA 基于 QEMU 模拟器, 在 TCG 生命周期中有多个回调函数点, 如 *PANDA_CB_INSN_TRANSLATE* 在基本块首次转换为 TCG IR 之前被调用, 而 *PANDA_CB_AFTER_INSN_EXEC* 在一条指令执行后被触发. 通过这些回调函数有选择地插入分析逻辑代码, EoLeak 系统能够记录和分析特定用户输入的目标二进制程序的执行过程. 例如: 在 Capstone 反汇编工具的帮助下, 将指令在执行前进行反汇编, 并在全局指针映射结构中记录识别到的指针. 污点和传播分析在这里进行.

3.3 服务器/客户端架构

PANDA 是一个构建在 QEMU 之上的分析平台, 其在启动目标操作系统镜像实例的新虚拟机时面临严重的性能问题和时间开销. EoLeak 系统需要动态决定目标二进制程序的下一个用户输入, 无法承担频繁的启动和关闭虚拟机的时间成本. 为此, 我们将 EoLeak 设计为服务器/客户端架构, 客户端是一个 Python 控制器, 用户控制 PANDA 虚拟机并接受来自服务器的命令. 其包装了常见的 PANDA 操作, 例如记录给定目标二进制程序和对应输入的新程序执行过程记录、重放程序执行迹并运行分析插件进行分析以及将服务器传来的 ISO 文件注入/弹出虚拟机等. 当分析操作完成后, 客户端将分析结果发送回服务器. 为了降低虚拟机启动的时间开销, 客户端在后台保留一个运行中的客户操作系统实例来接受和执行来自服务器的命令, 并根据需要自动重启虚拟机.

EoLeak 系统的服务器则是另一个 Python 项目, 也是系统的核心分析器. 它向客户端发送 PANDA 操作和分析指令, 从客户端接收分析结果, 并驱动自动构建泄漏和生成漏洞利用样本过程. 当生成新的程序输入时, 服务器将输入和二进制文件一起发送给客户端, 以进行另一次执行跟踪分析. 通过将漏洞利用分析模块和 PANDA 执行模块拆分为服务器和客户端架构, EoLeak 系统可以通过连接一个服务器和多个执行器客户端来并行加速.

4 评估验证

本节介绍针对 EoLeak 系统的实验评估和验证, 主要回答了如下几个研究问题.

(1) 研究问题 1: EoLeak 系统能否成功定位敏感信息?

(2) 研究问题 2: EoLeak 系统采用的堆攻击模板是否有效?

(3) 研究问题 3: EoLeak 系统能否为堆漏洞程序生成自动泄漏敏感信息以及能否自动实现利用的样本?

4.1 敏感信息定位

如表 2 所示, 我们从著名的 CTF 赛事和网站中收集了 17 个堆漏洞程序. 所有案例提供的 POC 都不可利

用. 系统的 libc 版本为 2.23, 启用的防御类型在表中以 SNP 表示(S 表示栈保护变量, N 表示数据执行保护, P 表示 PIE, 字母存在表示目标程序部署了对应防御, –表示 3 种防御均不存在), 系统采用全随机模式地址空间随机化保护.

表 2 EoLeak 测试的 CTF 堆漏洞二进制程序

二进制程序	CTF 赛事	漏洞类型	部署防御
aiRcraft	RCTF 2017	释放后使用	SNP
b00ks	ASIS 2016	单字节溢出	NP
babyheap	0CTF 2017	堆缓冲区溢出	SNP
freenote	0CTF 2015	两次释放	SN
mario	Defcon 2018	释放后使用	SNP
message_me	ASIS 2018	释放后使用	SN
minesweeper	CSAW 2017	堆缓冲区溢出	–
note3	ZCTF 2016	堆缓冲区溢出	SN
RNote	ZCTF 2017	单字节溢出	N
RNote2	RCTF 2017	堆缓冲区溢出	SNP
SecretHolder	HITCON 2016	释放后使用	SN
secret-of-my-heart	Pwnable.tw	单字节溢出	SNP
secure_keymanager	SECCON 2017	堆缓冲区溢出	SN
SleepHolder	HITCON 2016	释放后使用	SN
stkof	HITCON 2014	堆缓冲区溢出	SN
vote	N1CTF 2018	释放后使用	SN
zone	CSAW 2017	单字节溢出	SN

为了回答研究问题 1, 我们收集了 EoLeak 在动态分析过程中记录的 libc 库函数地址指针作为敏感信息的代表. 如表 3 所示, EoLeak 在所有 CTF 二进制文件中都成功定位到了可供泄漏的 libc 库函数地址. 左列的 libc 地址识别数代表原始 POC 下识别到的存储随机化后 libc 库函数相关地址值的内存位置数量, 由动态分析模块给出. 右列的攻击后 libc 地址识别数则代表在实现了初步堆攻击后识别到的对应内存位置数量, 体现了堆攻击对内存操纵能力的提升. libc 库相关指的是与 libc 库基址有固定偏移量的地址, 其不仅包括 libc 函数符号, 也包括了 libc 数据段内部的某些内存结构字段, 这些字段也可以起到泄漏 libc 运行时地址的作用.

表 3 EoLeak 当前支持的堆漏洞利用模板列表

二进制程序	POC 长度	POC 执行指令数	引用 libc 符号数	libc 地址指针识别数	攻击后 libc 地址指针识别数	堆地址识别数	栈地址识别数	程序地址识别数
aiRcraft	32	1 729	17	14	17	2	75	37
b00ks	154	2 786	12	12	12	3	82	23
babyheap	75	1 342	19	19	22	1	68	2
freenote	68	1 435	13	11	13	7	72	2
mario	847	7 604	40	44	50	44	79	46
message_me	213	1 861	15	10	13	2	88	1
minesweeper	12	2 568	28	13	13	5	123	28
note3	138	2 873	15	10	10	4	67	1
RNote	164	1 490	16	10	13	3	60	2
RNote2	1 374	1 677	18	18	22	8	64	2
SecretHolder	68	2 104	13	8	8	3	52	1
secret-of-my-heart	43	2 554	19	19	22	1	72	27
secure_keymanager	378	1 992	13	8	11	3	68	2
SleepHolder	41	2 381	16	11	11	3	52	3
stkof	18	1 636	16	7	10	2	55	2
vote	36	2 442	21	10	10	3	64	3
zone	86	3 783	15	11	11	43	79	1

我们还评估了 5 个真实世界程序的敏感信息定位能力. 从表 4 的结果可以看出, EoLeak 成功地在所有应用程序中定位到了 libc 地址. 值得注意的是: 原始 POC 下识别到的 libc 库相关地址数量与目标二进制文件的大小没有明显关系, 但与目标二进制文件中引用的 libc 符号数量大致相同. 这表明, 几乎所有可能的用于绕过地址空间布局随机化的 libc 相关地址都仅来自于全局偏移表. 另一方面, 通过初步堆攻击所扩展得到的 libc 相关地址不受此限制, 是稳定的泄漏来源选项.

表 4 真实软件中对 libc 库相关地址的识别

真实软件	部署防御	引用 libc 符号数	libc 地址指针识别数	堆地址识别数	栈地址识别数	程序地址识别数
ProFTPd	SN	198	122	13 451	88	1 186
nginx	N	123	86	6 746	138	4 675
nullhttpd	SN	78	59	27	122	4
apache	SN	149	92	583	300	1 364
smbclient	SNP	238	171	4 836	376	20

4.2 堆攻击模板

为了回答研究问题 2, 我们在 17 个 CTF 堆漏洞程序上验证 EoLeak 方案所采用的攻击模板列表的有效性. 结果见表 5. 17 个程序中, 有 5 个程序可以成功匹配 Fastbin 攻击模板; 有 8 个程序可以匹配 Unlink 攻击模板; 有 2 个程序可以通过伪造数据结构来实现内存写; 还有 2 个程序无法匹配到对应的攻击模板. 即, 有 88.2%的目标程序可以由 EoLeak 自动探索从堆布局不过分复杂的漏洞触发点到遵循模板的堆漏洞初步攻击.表 3 中, 初步攻击后新增识别到的 libc 库相关地址体现了这一初步攻击所带来的内存可利用性的提升. 我们也调研了 2 个无法成功匹配初步攻击模板的程序案例以及在真实软件中的情形,具体缘由在下一小节中加以分析.

表 5 CTF 堆漏洞二进制程序适用的堆攻击模板

二进制程序	漏洞类型	适用攻击模板	二进制程序	漏洞类型	适用攻击模板
aiRcraft	释放后使用	Fastbin 攻击	RNote2	堆缓冲区溢出	Unlink 攻击
b00ks	单字节溢出	伪造内存写	SecretHolder	释放后使用	Unlink 攻击
babyheap	堆缓冲区溢出	Fastbin 攻击	secret-of-my-heart	单字节溢出	Fastbin 攻击
freenote	两次释放	Unlink 攻击	secure_keymanager	堆缓冲区溢出	Unlink 攻击
mario	释放后使用	Fastbin 攻击	SleepHolder	释放后使用	Unlink 攻击
message_me	释放后使用	Unlink 攻击	stkof	堆缓冲区溢出	Unlink 攻击
minesweeper	堆缓冲区溢出	–	vote	释放后使用	–
note3	堆缓冲区溢出	Unlink 攻击	zone	单字节溢出	伪造内存写
RNote	单字节溢出	Fastbin 攻击	–	–	–

4.3 泄漏和漏洞利用生成

为了回答研究问题 3, 我们使用 17 个 CTF 堆漏洞程序来评估 EoLeak 系统, 尝试生成泄漏和漏洞利用样本. 对于每个堆漏洞程序, 会准备一个能够触发堆漏洞但未能利用的 POC.

如表 6 所示: EoLeak 系统为 17 个堆漏洞程序中的 15 个成功构建了泄漏, 其中 14 个生成了最终利用样本. 也就是说, EoLeak 系统在构建 libc 库函数地址泄漏方面成功率达到了 88.2%, 而 82.4%的目标程序得到了利用.

表 6 CTF 堆漏洞二进制程序的信息泄漏构建和利用样本生成效果

二进制程序	构建信息泄漏	生成利用样本	二进制程序	构建信息泄漏	生成利用样本
aiRcraft	√	√	RNote2	√	√
b00ks	√	√	SecretHolder	√	√
babyheap	√	√	secret-of-my-heart	√	√
freenote	√	√	secure_keymanager	√	√
mario	√	–	SleepHolder	√	√
message_me	√	√	stkof	√	√
minesweeper	–	–	vote	–	–
note3	√	√	zone	√	√
RNote	√	√	–	–	–

我们进一步调查了无法构建泄漏的 3 个案例和未能生成漏洞利用的 1 个案例, 分析了失败的原因.

(1) 多线程. vote 程序是一个多线程程序, 对于当前的分析模块来说过于复杂. 目前, EoLeak 仅支持对单线程的漏洞程序进行分析. 出于同样的原因, EoLeak 暂时无法理解条件竞争类漏洞;

(2) 自定义堆结构. minesweeper 程序实现了自定义的堆结构和管理器. EoLeak 的堆模型推断依赖于标准的 ptmalloc2 实现, 因此目前无法处理自定义堆, 也无法为程序顺利生成泄漏或最终漏洞利用;

(3) 尚未支持的堆利用技术. mario 程序需要利用_IO_file 结构来实现堆利用攻击, 当前堆模型推断机制

尚未能支持该类利用技巧. 因此, 虽然可以正常使用 FD 指针构造泄漏, 但在生成漏洞利用样本时仍是失败的;

(4) 利用窗口. 实际中, 软件路径复杂, 大多漏洞仅允许有限的攻击窗口, 通常仅有 1 次写入的能力, 很难复用. 即使部分漏洞符合利用模板条件, 后续亦因复杂的堆布局变化而难以构造无冲突的连续读取链条, 因而难以自动实现泄漏.

5 相关工作

5.1 自动生成漏洞利用样本

早期的自动生成漏洞利用样本解决方案没有考虑防御机制. APEG[22]是第一个 AEG 解决方案, 它将补丁前后出现差异的条件检查与目标二进制文件进行比较, 使用符号执行来自动构建在补丁之前未能通过条件检查的输入. AEG[4]基于栈上的控制流劫持对源代码进行符号执行, 以利用栈溢出和格式化字符串漏洞. Mayhem[5]设法为二进制文件中的栈溢出和格式化字符串漏洞生成漏洞利用, 通过控制流劫持符号执行实现执行 ret2stack-shellcode 和 ret2libc 攻击. PolyAEG[23]通过污点分析用户可控的控制流方向, 为具有一个异常输入的二进制文件生成多种漏洞利用. 这些 AEG 解决方案都无法绕过数据执行保护或地址空间布局随机化.

一些研究工作设法突破防御机制, 但仍无法自动绕过地址空间布局随机化. HI-CFG[8]将良性输入转换为缓冲区之间的漏洞点, 生成带有缓冲区信息的控制流图(control flow graph), 并通过读写操作搜索缓冲区传输. 该工作的核心目标是生成可以造成异常状态的 POC 样本, 如果想绕过地址空间布局随机化防御的利用样本, 则需要人工引导的介入. Revery[12]、KOOBE[13]和 FUZE[14]研究如何使用模糊测试技术生成针对堆漏洞的攻击. Revery 在用户态程序中搜索与崩溃执行路径相似的内存布局, 尝试缝合替换程序路径. KOOBE 专注于 Linux 内核中的越界写入漏洞. 另一方面, FUZE 分析用于创建和取消引用悬空指针的时间戳. 这些解决方案将绕过地址空间布局随机化, 被视为超出研究范围的内容, 因而未予考虑.

一些研究工作试图绕过地址空间布局随机化, 但只能通过对栈漏洞的利用实现绕过, 而无法在堆漏洞上成功绕过. Heelan[3]提出了一种使用符号执行来生成已知栈溢出漏洞的解决方案, 通过跳转寄存器引导向未随机地址来绕过地址空间布局随机化. Q[6]使用少量非随机代码强化现有漏洞利用, 以绕过地址空间布局随机化. 它通过二进制分析平台 BAP[29]构建面向返回编程(return-oriented programming)的配件, 并匹配所提出的 QooL 语言以进行栈利用. CRAX[7]通过对用户控制的程序计数器进行污染, 为大型真实世界的二进制文件生成漏洞利用. 尽管它可以使用 ret2libc 和 jmp2reg 技术绕过栈上的地址空间布局随机化, 但它无法实现堆漏洞上的绕过. ShellSwap[24]通过使用符号跟踪以及漏洞利用代码布局修复和路径拼接的组合, 将现有的旧漏洞利用移植到一个有效的新漏洞利用中. 其通过一个具有可控寄存器的特定 jmp 指令覆盖指令指针来绕过地址空间布局随机化, 因而无法同时绕过数据执行保护. R2dlAEG[26]利用符号执行探索利用 return2dl-resolve 攻击技术来为栈溢出漏洞生成绕过数据执行和地址空间布局随机化的利用样本. KEPLER[25]通过发现多个漏洞利用链来生成绕过现代缓解技术的面向返回编程的漏洞利用. 它需要劫持控制流并利用控制流劫持原语(control-flow hijacking primitive)来构建漏洞利用的 POC.

数据流分析自然可以对抗地址空间布局随机化保护. FLOWSTITCH[30]自动生成面向数据的漏洞利用, 在不违反控制流完整性的情况下, 拼接信息泄漏或特权升级. DOP[31]更进一步, 通过构建数据流配件来执行面向数据的编程, 但是它们没有在堆漏洞上绕过地址空间布局随机化. BOPC[32]假设所有控制流缓解都已启用. 它在给定的入口点上搜索内存布局, 通过多个任意内存写入原语来设置内存布局(如果绕过地址空间布局随机化, 则亦需要读取原语), 使得后续数据流满足所需的 SPL 逻辑.

解释器上的 AEG 研究工作也受到了一些关注. SHRIKE[9]对 PHP 进行回归测试以提取堆操作. 此外, 它会搜索溢出块与目标块相邻的特定内存布局. Gollum[10]自动解决了寻找数据结构溢出并以某种特定方式使用的问题. 它首先检查可利用的堆布局, 然后以惰性方法搜索相应的堆操作过程. 由于需要确定的堆分配地址, 它不能绕过地址空间布局随机化.

5.2 污点分析

污点分析技术是用于分析程序数据流的一种重要方法, 可以直接判断输入源和目标点之间是否存在关联性, EoLeak 执行轻量级动态污点分析来研究用户输入与库函数调用参数之间的关系. 静态污点分析很难保证结果的正确性, 现行的污点分析系统往往采用动态分析的方法, 并与动态符号执行和模糊测试等方法相结合.

Dytan[33]给出了一个通用的污点分析框架, 并采用动态插桩的方法进行污点分析, 但它在污点标记和插桩效率上存在很大的局限性. Libdft[34]使用影子内存(shadow memory)方法, 提高了动态数据流追踪的效率, 加强了动态污点分析技术的实用性. 现行的动态污点分析方法都不可避免地存在过污染(over-taint, 污点标记过多, 污点变量大量扩散)或欠污染(under-taint, 污点传播分析不当, 本应被标记的变量没有被标记)的问题[35]. 并且, 由于污点分析需要对每一条指令指定污点传播规则, 它们对目标程序的运行平台有着很强的依赖, 迁移到不同的平台需要付出很大的工程代价. 迄今为止, 没有一个动态污点分析系统可以同时在 x86 和 x86-64 平台上分别对 32 位和 64 位程序进行分析. 为了缓解过污染和欠污染问题, 从而加强动态污点分析的可靠性和完备性, TaintInduce[36]可以只利用对应架构最少的语义信息来自动分析得到传播规则, 具有较好的跨平台迁移能力. 另外, 还有一些结合二进制分析的动态污点分析工具, 比如 TEMU[37]和 Triton[38].

传统的污点分析方法工程量大, 运行效率低, 因此一些前沿的研究尝试使用机器学习的手段, 用神经网络模拟指令的输入和输出, 尝试为指令自动建立传播规则, 从而达到降低工程量和跨平台的目的. 如 Neutaint[39]利用神经网络来直接估计输入源和目标点之间的关联性.

6 总 结

对于自动生成漏洞利用样本解决方案来说, 将部署在目标系统中的缓解策略涵盖进考虑范围是一个现实的挑战. 我们提出了一种有效的自动解决方案 EoLeak, 用于通过堆漏洞同时绕过地址空间布局随机化和数据执行保护. 通过使用漏洞触发样本对程序执行迹进行动态分析, EoLeak 方案利用对内存传输的轻量级污点分析和简单的堆利用模型来推断可能的库函数指针和漏洞利用目标. 我们在 17 个夺旗赛堆漏洞二进制程序上测试了 EoLeak, 其中 15 个可以自动构建泄漏, 14 个可以生成最终利用样本. EoLeak 还可以成功分析真实世界程序中的敏感信息和内存可控性.

References:

[1] Liu J, Su PR, Yang M, He L, Zhang Y, Zhu XY, Lin HM. Software and cyber security—A survey. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 42−68 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[2] Zhao SR, Li XJ, Fang Y, Yu YP, Huang WH, Chen K, Su PR, Zhang YQ. A survey on automated exploit generation. Computer Research and Development, 2019, 56(10): 2097−2111 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2019.20190655]

[3] Heelan S. Automatic generation of control flow hijacking exploits for software vulnerabilities [MS. Thesis]. Oxford: University of Oxford, 2009.

[4] Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D. Automatic exploit generation. Communications of the ACM, 2014, 57(2): 74−84. [doi: 10.1145/2560217.2560219]

[5] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 380−394. [doi: 10.1109/SP.2012.31]

[6] Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. In: Proc. of the 20th USENIX Security Symp. San Francisco: USENIX Association, 2011.

[7] Huang SK, Huang MH, Huang PY, Lai CW, Lu HL, Leong WM. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: Proc. of the 6th IEEE Int'l Conf. on Software Security and Reliability. Gaithersburg: IEEE, 2012. 78−87. [doi: 10.1109/SERE.2012.20]

[8] Caselden D, Bazhanyuk A, Payer M, Szekeres L, McCamant S, Song D. Transformation-aware exploit generation using a HI-CFG. University Berkeley, Department of Electrical Engineering and Computer Science, 2013. [doi: 10.21236/ADA587051]

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

[10] Heelan S, Melham T, Kroening D. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 1689−1706. [doi: 10.1145/3319535.3354224]

[11] Wang Y, Zhang C, Zhao Z, Zhang B, Gong X, Zou W. MAZE: Towards automated heap feng shui. In: Proc. of the 30th USENIX Security Symp. Virtual: USENIX Association, 2021. 1647−1664.

[12] Wang Y, Zhang C, Xiang X, Zhao Z, Li W, Gong X, Liu B, Chen K, Zou W. Revery: From proof-of-concept to exploitable. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 1914−1927. [doi: 10.1145/3243734.3243847]

[13] Chen W, Zou X, Li G, Qian Z. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In: Proc. of the 29th USENIX Security Symp. Virtual: USENIX Association, 2020. 1093−1110.

[14] Wu W, Chen Y, Xu J, Xing X, Gong X, Zou W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In: Proc. of the 27th USENIX Security Symp. Baltimore: USENIX Association, 2018. 781−797.

[15] Ning G, Zhang T, Wen W, Mei R. Study of non-heapspray IE's vulnerability exploitation technique. Netinfo Security, 2014(6): 39−42 (in Chinese with English abstract). [doi: 10.3969/j.issn.1671-1122.2014.06.007]

[16] van de Ven A. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. Raleigh: Red Hat, 2004.

[17] Cowan C, Pu C, Maier D, Walpole J, Bakke P. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symp. San Antonio: USENIX Association, 1998. 63−78.

[18] PaX T. PaX address space layout randomization (ASLR). 2003. http://pax.grsecurity.net/docs/aslr.txt

[19] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security, 2009, 13(1): 1−40. [doi: 10.1145/1609956.1609960]

[20] Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, Payer M. Control-flow integrity: Precision, security, and performance. ACM Computing Serveys, 2017, 50(1): 1−33. [doi: 10.1145/3054924]

[21] Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson U, Lozano L, Pike G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Proc. of the 23rd USENIX Security Symp. San Diego: USENIX Association, 2014. 941−955.

[22] Brumley D, Poosankam P, Song D, Zheng J. Automatic patch-based exploit generation is possible: Techniques and implications. In: Proc. of the 2008 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2008. 143−157. [doi: 10.1109/SP.2008.17]

[23] Wang M, Su P, Li Q, Ying L, Yang Y, Feng D. Automatic polymorphic exploit generation for software vulnerabilities. In: Zia T, ed. Proc. of the Security and Privacy in Communication Networks. Cham: Springer Int'l Publishing, 2013. 216−233.

[24] Bao T, Wang R, Shoshitaishvili Y, Brumley D. Your exploit is mine: Automatic shellcode transplant for remote exploits. In: Proc. of the 2017 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2017. 824−839. [doi: 10.1109/SP.2017.67]

[25] Wu W, Chen Y, Xing X, Zou W. KEPLER: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX Association, 2019. 1187−1204.

[26] Fang H, Wu L, Wu Z. Automatic return-to-dl-resolve exploit generation method based on symbolic execution. Computer Science, 2019, 46(2): 127−132 (in Chinese with English abstract). [doi: 10.11896/j.issn.1002-137X.2019.02.020]

[27] Dolan-Gavitt B, Hodosh J, Hulin P, Leek T, Whelan R. Repeatable reverse engineering with PANDA. In: Proc. of the 5th Program Protection and Reverse Engineering Workshop. Los Angeles: ACM, 2015. [doi: 10.1145/2843859.2843867]

[28] Chen K, Zhang C, Yin T, Chen X, Zhao L. VScape: Assessing and escaping virtual call protections. In: Proc. of the 30th USENIX Security Symp. Virtual: USENIX Association, 2021. 1719−1736.

[29] Brumley D, Jager I, Avgerinos T, Schwartz EJ. BAP: A binary analysis platform. In: Gopalakrishnan G, ed. Proc. of the Computer Aided Verification. Berlin: Springer, 2011. 463−469. [doi: 10.1007/978-3-642-22110-1_37]

[30] Hu H, Chua ZL, Adrian S, Saxena P, Liang Z. Automatic generation of data-oriented exploits. In: Proc. of the 24th USENIX Security Symp. Washington: USENIX Association, 2015. 177−192.

[31] Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z. Data-oriented programming: On the expressiveness of non-control data attacks. In: Proc. of the 2016 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2016. 969−986. [doi: 10.1109/SP.2016.62]

[32] Ispoglou KK, AlBassam B, Jaeger T, Payer M. Block oriented programming: Automating data-only attacks. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. Toronto: ACM, 2018. 1868−1882.

[33] Clause J, Li W, Orso A. Dytan: A generic dynamic taint analysis framework. In: Proc. of the 2007 Int'l Symp. on Software Testing and Analysis. London: ACM, 2007. 196−206. [doi: 10.1145/1273463.1273490]

[34] Kemerlis VP, Portokalidis G, Jee K, Keromytis AD. libdft: Practical dynamic data flow tracking for commodity systems. In: Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments. New York: ACM, 2012. 121−132.

[35] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (bug might have been afraid to ask). In: Proc. of the 2010 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2010. 317−331. [doi: 10.1109/SP.2010.26]

[36] Chua ZL, Wang Y, Baluta T, Saxena P, Liang Z, Su P. One engine to serve'em all: Inferring taint rules without architectural semantics. In: Proc. of the Network and Distributed System Security Symp. San Diego: Internet Society, 2019. [doi: 10.14722/ndss.2019.23339]

[37] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang Z, Newsome J, Poosankam P, Saxena P. BitBlaze: A new approach to computer security via binary analysis. In: Sekar R, ed. Proc. of the Information Systems Security. Berlin: Springer, 2008. 1−25. [doi: 10.1007/978-3-540-89862-7_1]

[38] Saudel F, Salwan J. Triton: Concolic execution framework. In: Proc. of the Rennes: Symp. sur la Sécurité des Technologies de l' Information et des Communications, 2015.

[39] She D, Chen Y, Shah A, Ray B, Jana S. Neutaint: Efficient dynamic taint analysis with neural networks. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1527−1543. [doi: 10.1109/SP40000.2020.00022]

附中文参考文献:

[1] 刘剑, 苏璞睿, 杨珉, 和亮, 张源, 朱雪阳, 林惠民. 软件与网络安全研究综述. 软件学报, 2018, 29(1): 42−68. http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[2] 赵尚儒, 李学俊, 方越, 余媛萍, 黄伟豪, 陈恺, 苏璞睿, 张玉清. 安全漏洞自动利用综述. 计算机研究与发展, 2019, 56(10): 2097−2111. [doi: 10.7544/issn1000-1239.2019.20190655]

[15] 宁戈, 张涛, 文伟平, 梅瑞. 一种非堆喷射的 IE 浏览器漏洞利用技术研究. 信息网络安全, 2014(6): 39−42. [doi: 10.3969/j.issn.1671-1122.2014.06.007]

[26] 方皓, 吴礼发, 吴志勇. 基于符号执行的 Return-to-dl-resolve 利用代码自动生成方法. 计算机科学, 2019, 46(2): 127−132. [doi: 10.11896/j.issn.1002-137X.2019.02.020]

杨松涛(1996－), 男, 博士生, 主要研究领域为系统软件安全, 二进制攻防.

王准(1999－), 男, 工程师, 主要研究领域为软件安全, 二进制攻防.

陈凯翔(1995－), 男, 博士生, 主要研究领域为安全攻防技术.

张超(1986－), 男, 博士, 长聘副教授, 博士生导师, CCF 高级会员, 主要研究领域为软件与系统安全.