



基于符号执行的堆溢出 fastbin 攻击检测方法

张 超¹, 潘祖烈¹, 樊 靖²

(1. 国防科技大学 电子对抗学院, 合肥 230037; 2. 北海舰队, 山东 青岛 266000)

摘 要: 为弥补当前软件漏洞自动检测系统无法对含堆溢出漏洞的程序进行自动检测的缺陷, 提出一种 Linux 平台下面向堆溢出的 fastbin 攻击的自动检测方法。基于已有的 fastbin 攻击实例, 利用 fastbin 攻击特征, 建立 fastbin 攻击检测模型, 并基于该模型给出一种 fastbin 攻击检测方法。运用污点分析和符号执行技术, 通过监控符号数据到达漏洞触发点的关键信息构建路径约束以及触发 fastbin 攻击的数据约束, 基于对约束的求解, 判断程序是否存在 fastbin 攻击的可能, 并生成测试用例。实验结果表明, 面向堆溢出的 fastbin 攻击检测方法能够实现对 fastbin 攻击的准确检测。

关键词: 堆溢出; fastbin 攻击; 符号执行; 污点分析; 约束构建

开放科学(资源服务)标志码(OSID):



中文引用格式: 张超, 潘祖烈, 樊靖. 基于符号执行的堆溢出 fastbin 攻击检测方法[J]. 计算机工程, 2020, 46(10): 151-158.

英文引用格式: ZHAO Chao, PAN Zulie, FAN Jing. Detection method for heap overflow fastbin attack based on symbolic execution[J]. Computer Engineering, 2020, 46(10): 151-158.

Detection Method for Heap Overflow fastbin Attack Based on Symbolic Execution

ZHAO Chao¹, PAN Zulie¹, FAN Jing²

(1. College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China;

2. Beihai Fleet, Qingdao, Shandong 266000, China)

[Abstract] The existing automatic detection systems for software vulnerabilities fail to automatically detect the programs with heap overflow vulnerabilities. To address the problem, this paper proposes an automatic detection method for heap overflow fastbin attacks on Linux platforms. Based on the fastbin attack examples, the characteristics of fastbin attacks are used to establish a detection model for fastbin attacks, and on this basis a detection method of fastbin attacks is proposed. The method uses the technique of stain analysis and symbolic execution to monitor the key information of symbol data reaching the vulnerability trigger point, and on this basis constructs path constraints and data constraints that trigger fastbin attacks. Based on the solution of constraints, the possibility of fastbin attacks in the program can be judged and test cases can be generated. Experimental results show that the proposed heap overflow fastbin attack detection method can effectively detect fastbin attacks.

[Key words] heap overflow; fastbin attack; symbolic execution; stain analysis; constraint construction

DOI: 10.19678/j.issn.1000-3428.0055750

0 概述

随着信息技术的发展, 软件漏洞的挖掘与利用成为领域研究热点, 而传统的软件漏洞主要以手工方式构造, 但是手工构造过程需要大量的底层知识和分析经验。随着软件数量的增加及其功能的日益完善, 软件漏洞呈现更加多样且复杂的趋势^[1-2]。尽管目前不少二进制漏洞的自动化调试与检测方法已

经能有效地发现程序错误, 但其中只要有一部分程序错误可以被利用, 就将导致严重后果^[3]。因此, 如何快速、准确地对漏洞的危险性进行评估, 是当前漏洞自动分析与检测领域的关键问题之一^[4-5]。

堆溢出漏洞是一种常见的缓冲区溢出漏洞^[6]。通过对堆溢出漏洞的利用, 可能导致程序控制流劫持以及执行任意代码的后果。其中, fastbin 攻击是一种 Linux 环境下面向堆溢出漏洞的攻击方法^[7]。

基金项目: 国家重点研发计划重点专项“网络空间安全”(2017YFB0802905)。

作者简介: 张 超 (1995—), 男, 硕士研究生, 主研方向为网络安全; 潘祖烈, 副教授、博士; 樊 靖, 学士。

收稿日期: 2019-08-15 **修回日期:** 2019-11-18 **E-mail:** sw_125_c@sina.com

典型的 fastbin 攻击通过堆块的快速分配和释放等操作,修改相邻堆块的堆块头,造成相邻堆块的堆块头数据被覆盖,从而达到任意地址分配以及任意地址写的目的。为保护 fastbin 链表不被攻击, Linux 系统设置了如 size 位检测等保护机制,避免程序流被劫持^[8]。

本文通过分析已有的 fastbin 实例,总结 fastbin 攻击特征,构建 fastbin 攻击检测模型,并在该模型的基础上提出针对 fastbin 攻击的自动检测方法,通过污点分析^[9-10]和符号执行^[11-12]实现 fastbin 攻击的自动检测。

1 软件漏洞自动利用技术发展现状

针对控制流劫持类漏洞的攻击检测及测试例生成,已有大量相关研究和成果。文献[13]提出基于二进制补丁比较的漏洞利用自动生成方法 APEG。APEG 通过比较程序的 bug 版本和补丁版本程序的不同,生成能在补丁版本中增加校验失败的利用。实验结果表明,该方法具有较强的可靠性和实用性。但该方法也有局限性,即存在无法处理补丁程序中不添加过滤判断的情况,以及构造的类型主要属于拒绝服务,只能造成源程序的崩溃,而无法直接造成控制流劫持。

为克服 APEG 对于补丁的依赖以及无法构造控制流劫持的缺陷,文献[14]提出漏洞自动挖掘与测试例生成方法 AEG。AEG 集成了优化后的符号执行和动态指令插桩技术,实现了从软件漏洞自动挖掘到软件漏洞自动利用的整个过程,并且生成的利用样本直接具备控制流劫持能力,是第 1 个真正意义上的面向控制流的漏洞利用自动化构建方案。该方案的局限性主要体现在:需要依赖源代码进行程序错误搜索;所构造的样本主要是面向栈溢出或者字符串格式化漏洞,并且利用样本受限于编译器和动态运行环境等因素。

在 AEG 方法的基础上,文献[15]提出了基于符号执行的 CRAX 方法,其主要构建在 S2E^[16-18]、KLEE^[19]、QEMU^[20] 环境中。CRAX 可以作为 fuzzer 的后端,借助前端生成的 crash 和二进制程序自动生成利用代码。CRAX 使用符号执行技术,通过对程序动态运行过程的监视,检测 EIP 劫持状态。当发现 EIP 存在劫持的可能时,在符号化可控区域中布置 shellcode 并覆盖 EIP 的值为 shellcode 地址,生成可利用代码。其局限性在于检测过程未考虑系统保护机制与动态运行环境对漏洞攻击的影响。

2 面向堆溢出漏洞的 fastbin 攻击原理

对于堆内存管理而言,堆块(chunk)就是最小的操作单位。而 Glibc 针对分配的堆块大小的不同,将堆块分为 3 类:大小 16 Byte 到 80 Byte 的是 fast

chunk;堆块大小 512 Byte 以下的是 small chunk;堆块大小大于 512 Byte 的是 large chunk。

堆块在被释放后,会放入被称为 bin 的数据结构中,以链表的形式存储。Glibc 定义了 4 种链表,分别为 fastbin、smallbin、largebin 和 unsortedbin。Glibc 维护了 136 个 bins 链表,其中,fastbins 10 条,smallbins 62 条,largebins 63 条,unsortedbins 1 条。

而 fastbin 所包含 chunk 的大小为:16 Bytes, 24 Byte, 32 Bytes, ..., 80 Byte,以单向链表的形式存储,每条 fastbin 链中存储的堆块大小都与本链大小相同。而其他 3 种链表以双向链表的形式存储。fastbin 链表结构如图 1 所示。

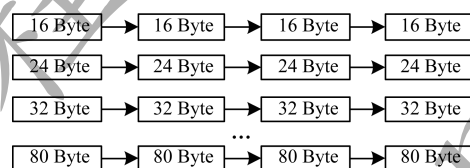


图 1 fastbin 链表结构

Fig. 1 Structure of fastbin chain list

堆块的状态分为已分配堆块(allocated chunk)和已释放堆块(free chunk) 2 种状态,而系统通过堆头的标志位来区分堆块的不同状态。

图 2 为 fastbin chunk 的头部结构。当堆块出入释放状态时,第 1 Byte ~ 第 4 Byte 是 prev_size,表示前一个堆块的大小。第 5 Byte ~ 第 8 Byte 是 size,表示当前堆块的大小。由于每个堆块的大小一定大于等于 8 Byte,因此最后 3 bit 的内存空间没有实际意义,glibc 将这 3 bit 的空间用来做标志位,最后 1 bit 表示前一个堆块的使用状态。当前一个堆块是释放(free)状态时,F 标志位为 0;若为已分配(allocated)状态,F 标志位为 1。第 9 Byte ~ 第 12 Byte 为 fd,指向链表中的前一个堆块。因为是单链结构,所以 fastbin 堆块没有 bk 指针。

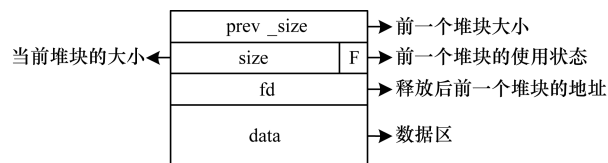


图 2 fastbin 堆块头部结构

Fig. 2 Head structure of fastbin heap block

glibc 为了实现堆块的快速分配和释放,专门设置了 fastbin 链表。fastbin 链表是单链,通过 fd 指针连接起来。当释放某一个堆块时,会首先检查其大小是否落在 fastbin 的范围中。如果是,则直接将其插入到对应的 fastbin 链表中,而且 fastbin 链表采用先进先出(LIFO)原则,即后释放的链表在下一次分配时会优先分配。

当连续分配两块大小相同的 fastbin 堆块时, 2 个堆块在实际内存中是相邻的。因此,当前一个堆

块存在溢出漏洞时,以通过溢出修改相邻堆块的 fd 指针为 fake_fd,使其指向任意地址,然后在目标地址构造 size 位,就可以实现任意地址分配,从而实现任意地址写的目的。如图3所示,通过 fastbin 攻击,实现任意地址分配与地址写,再通过进一步构造,即可劫持控制流。

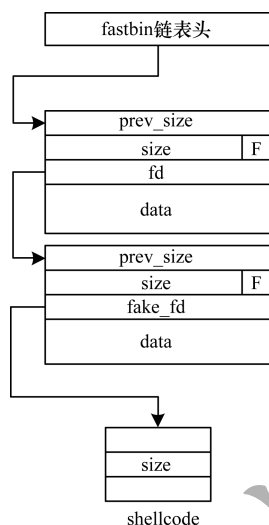


图3 fastbin 攻击过程示意图

Fig.3 Schematic diagram of fastbin attack process

针对传统的 fastbin 利用方法,glibc 目前主要的针对性保护机制是 size 位检测。在 glibc 分配堆块之前,首先会检查待分配堆块的 size 位是否与所在的 fastbin 链的大小相同,如果不同,则系统报异常,无法继续分配。然后 glibc 会检查待分配堆块 fd 指针指向的下一个堆块的 size 位是否与所在的 fastbin 链大小相同,如果不同,同样系统会报错,停止堆块的分配。

3 基于符号执行的 fastbin 攻击检测方法

3.1 整体思路

由于已有的堆溢出漏洞攻击检测技术无法收集系统分配和释放堆块时的相关信息,因此已有的攻击检测技术无法识别面向堆溢出的漏洞的控制流劫持攻击。针对这一问题,本文基于堆溢出中的 fastbin 攻击方法,提出了基于符号执行的 fastbin 攻击检测方法。

在 fastbin 攻击检测中的总体过程分为种子输入、堆块信息收集、堆溢出错误检测、fastbin 攻击检测等。初步方案为将模糊测试产生的 crash 作为种子输入,经符号执行引擎传递至被测程序,使被测程序沿确定路径运行至堆溢出错误代码区域。在堆溢出错误检测过程中,通过 API 函数挂钩进行符号变元引入,将外部输入标记为符号值,通过符号执行技术将程序中的变量表示为符号和常量的表达式,进而对目标程序进行检测,收集程序运行过程中堆块分配与释放的相关信息。同时,通过匹配

fastbin 攻击检测模型,构造相应的数据约束。在被测程序的动态运行过程中,根据数据约束将满足条件的符号化内存区域具体化,消除保护机制对检测过程的影响,生成 fastbin 攻击检测测试用例。本文以 C/C++ 程序为分析对象,选取若干 Linux 环境下的测试集开展测试验证。

3.2 面向堆溢出的 fastbin 攻击检测模型

本文对面向堆溢出的 fastbin 攻击原理进行分析,提出通过堆溢出导致 fastbin 攻击控制流劫持的过程需要满足4个特征:堆块溢出特征,溢出堆块可控特征,fastbin 操作触发特征,指针数据可控特征。

根据面向堆溢出的 fastbin 攻击特征,本文对程序的特征定义如下:

1)堆块溢出特征(ChunkOverflow):表示程序是否存在堆溢出错误的特征。

2)溢出堆块可控特征(ChunkControl):表示堆块溢出的数据可以受外部输入数据控制的特征。

3)fastbin 操作触发特征(IsFastbin):表示程序发生的堆溢出错误是否满足 fastbin 攻击方法的特征。

4)指针数据可控特征(PtrControl):表示指针数据受外部输入数据控制的特征。

5)堆溢出漏洞的 fastbin 攻击方法触发特征(FastbinHijack):表示程序可以通过 fastbin 攻击方法导致控制流劫持。

当程序同时存在堆块溢出特征、溢出堆块可控特征、fastbin 操作触发特征、指针数据可控特征这4种特征,就可以判断程序中存在堆溢出错误,且可以同过错误触发 fastbin 攻击。因此,上述的程序特征间的关系可以用下式表达:

$$\text{FastbinHijack} = \text{ChunkOverflow} \wedge \text{ChunkControl} \wedge \text{IsFastbin} \wedge \text{PtrControl}$$

本文对程序运行过程中堆块状态定义如下:

定义三元组 $\text{header} = (\text{pre_size}, \text{size}, \text{fd})$,描述堆块分配过程中头部 header 值的变化。

前堆大小 pre_size:分配过程中该值为0。

本堆大小 size:描述分配过程中堆块的大小,同时 size 的最后一位表示前一堆块分配状态,在 free 状态时 $F = 0$,在 allocated 状态时 $F = 1$ 。

链表指针 fd:指向了 fastbin 链表中的下一个堆块。

定义函数 $F(\text{header}, t_pre\text{size}, t_size, t_fd)$:描述 fastbin 攻击发生时,目标头部 header 的状态变化规则。其中,参数 $t_pre\text{size}$ 、 t_size 、 t_fd 表示 header 变化后的目标值。 $t_pre\text{size}$ 和 t_size 值为堆块原来的 pre_size 和 size 位相同大小,而 t_fd 则为目标地址的值, $t_fd = \text{target_addr}$ 。

面向堆溢出的 fastbin 攻击主要是通过堆溢出修改相邻堆块的 fd 指针,并保持相邻堆块头的 pre_size

和 size 不变,因此需要具体化 pre_size、size、fd,从而绕过 glibc 的保护机制。此过程中堆块头 header 的数据变化如下:

$$F(\text{header}, t_pre\text{size}, t_size, t_fd) = \begin{cases} \text{pre_size} = t_pre\text{size} \\ \text{size} = t_size \\ \text{fd} = t_fd \end{cases}$$

为实现面向堆溢出的 fastbin 攻击检测,构造测试用例,需要对 fastbin 攻击劫持程序控制流的过程进行监视。在堆块操作过程中,glibc 保护机制对堆块的数据检查会导致程序异常退出,造成检测失败。为消除保护机制对检测过程的影响,本文根据 fastbin 攻击的场景设计相应的数据具体化过程,建立了面向堆溢出漏洞的 fastbin 攻击检测模型,如图 4 所示。

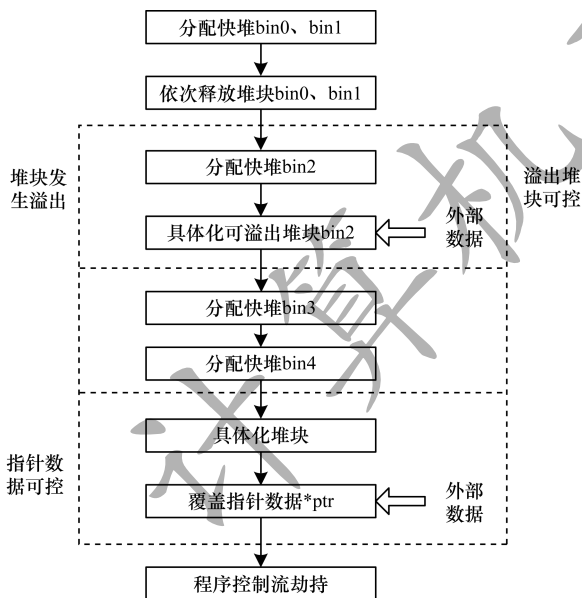


图 4 堆溢出漏洞 fastbin 攻击检测模型

Fig. 4 Detection model of heap overflow vulnerability fastbin attack

检测模型通过程序函数挂钩,获取堆块分配函数参数,记录新建堆块的起始地址与分配长度。在程序动态运行过程中,监视针对已建堆块的写入与释放操作。当程序触发堆块写入操作时,通过符号内存搜索算法,比对内存中符号区域与写入对象堆块区域的地址与长度,检查写入对象堆块的堆溢出触发特征与溢出堆块可控性特征。若堆块溢出且可控,则记录写入对象堆块信息。当溢出堆块触发释放操作时,检查溢出堆块及其相邻堆块是否满足 fastbin 攻击操作条件,判断 fastbin 攻击能否发生。

3.3 面向堆溢出的 fastbin 攻击检测算法

本文在 fastbin 攻击检测模型的基础上,使用符号执行和污点分析技术,设计并实现了堆溢出 fastbin 攻击的检测系统。该系统通过对虚拟机中运行程序的函数进行挂钩,收集程序运行过程中堆块

分配和释放的相关信息。同时通过 fastbin 攻击检测算法,将外部输入数据符号化,然后根据符号值的传播来确定外部数据在程序中的传播路径,并以此为基础,构建可到达漏洞发生点的约束条件,生成测试用例。针对程序动态运行中的符号化内存和堆块状态变化,有如下定义:

二元组 $S = (s_addr, s_size)$: 二元组 S 描述了符号化区域的特征, s_addr 和 s_size 分别表示符号化区域的起始地址和大小。

三元组 $H = (h_addr, h_size, h_state)$: 三元组 H 描述了堆块的特征, h_addr 表示堆块数据区的起始地址, h_size 表示堆块数据区域的大小, h_state 表示堆块状态是处于分配状态还是释放状态。

Symb_map: 表示所有符号化区域的集合。

二元组 Allo_map: 表示所有处于分配状态的堆块的集合。

chunk_overflow: 存在堆溢出特征的堆块的集合。

chunk_exploitable: 可被外部数据控制的堆块的集合。

系统在运行过程中,执行以下步骤:

步骤 1 系统通过对函数进行挂钩,记录了程序运行过程中堆块创建和释放的相关信息,包括堆块创建地址、堆块大小等。当有符号化数据进入程序时,首先使用堆块溢出检测算法,通过判断输入程序的符号化数据是否超过堆块边界,判断堆块是否发生溢出。

步骤 2 使用溢出堆块可利用性检测算法,检测溢出堆块是否可控。

步骤 3 系统通过对堆块的申请和释放的监控,通过 fastbin 攻击检测算法,检测堆块的申请和释放状态是否匹配 fastbin 攻击检测模型。

步骤 4 如果堆块的申请和释放状态符合 fastbin 攻击检测模型,可以通过 fastbin 攻击方法实现任意地址写。系统利用指针数据可控性检测算法对指定的数据指针进行符号化检查,若指针数据是符号值,则满足指针数据可控性要求。

同时要根据上述步骤建立相应的约束,然后与数据约束 data_constraint 进行舍取,建立测试用例约束条件 FastbinHijack_constraint。

检测系统是执行在宿主机中通过对虚拟机模拟执行的程序进行函数挂钩获取相关的信息,来作为检测算法运行的基础。而虚拟机和宿主机的信息传递是通过寄存器实现的。比如对 malloc 函数的返回地址进行挂钩,可以知道新创建堆块的堆块指针和堆块大小以及 malloc 函数的返回地址,这些信息就是通过寄存器传输到宿主机,其中堆块指针存放在寄存器 eax 中,堆块大小存放在寄存器 ecx 中, malloc 的返回地址存放在 edx 中。通过寄存器传输的方式,使算法可以实时获取程序运行过程中各种信息的变化。

3.3.1 堆溢出检测算法

程序存在不安全操作使输入信息长度大于数据区域长度从而导致溢出,是 fastbin 攻击发生的基本条件。本文利用挂钩技术可以获取程序堆块分配的起始地址和数据区长度等信息,而且还能获得外部输入数据的长度。通过检查写操作导致目标区域状态的变化,对比符号化区域长度和堆块数据区长度,可以实现堆溢出检测。若堆块发生溢出,则将溢出堆块加入溢出堆块集合 $\text{chunk}_{\text{overflow}}$ 中。

堆溢出检测算法如算法 1 所示。

算法 1 堆溢出检测算法

输入 符号化区域集合 Symb_map , 分配状态堆块集合 Allo_map
 输出 溢出堆块集合 $\text{chunk}_{\text{overflow}}$
 for($a = 0; a < \text{Symb_map.size}(); a++$)
 for($b = 0; b < \text{Allo_map.size}(); b++$)
 {
 if($s_addr > h_addr \&\& s_addr < (h_addr + h_size) \&\& (s_addr + s_size) > (h_addr + h_size)$)
 {
 $\text{Allo_map} \rightarrow \text{chunk}_{\text{overflow}}$
 break
 }
 }
 end if

算法 1 准确地描述了堆块发生溢出的基本特征,即堆块数据超出了堆块本身的长度,体现在算法中是存在符号区域的起始地址被包含在堆块数据区,但结束地址不被包含在当前堆块的数据区。算法对堆块溢出的特征定义准确。

3.3.2 堆块可利用性检测算法

由于 fastbin 攻击要求通过控制堆块的溢出数据,修改相邻堆块的头部,并构造假的 fd 指针,溢出堆块 chunk1 通过溢出修改相邻堆块 chunk2 头部后的构造如图 5 所示。同时,程序通过溢出堆块可利用性检测算法,建立伪堆块数据约束 $\text{fakechunk_constraint}$ 。

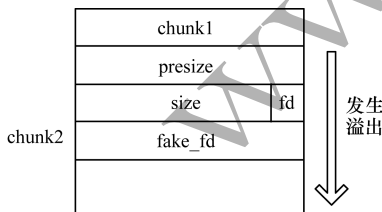


图 5 堆块溢出后的状态

Fig.5 State after heap block overflow

根据图 5 的溢出堆块结构,伪堆块数据约束 $\text{fakechunk_constraint}$ 由伪 pre_size 约束 $\text{pre_size_constraint}$ 、伪 size 约束 size_constraint 、伪 fd 约束 fd_constraint 组成,关系式如下:

$$\text{fakechunk_constraint} = \text{pre_size_constraint} \wedge \text{size_constraint} \wedge \text{fd_constraint}$$

堆块可利用性检测算法如算法 2 所示。

算法 2 堆块可利用性检测算法

输入 堆块集合 Allo_map , 堆溢出堆块集合 $\text{chunk}_{\text{overflow}}$, 溢出后想实现任意地址写的地址 target_addr

输出 可控溢出堆块集合 $\text{chunk}_{\text{exploitable}}$
 伪堆块数据约束 $\text{fakechunk_constraint}$
 foreach($\text{Heap} \in \text{chunk}_{\text{overflow}}$)
 foreach($\text{byte} \in \text{Allo_map}$)
 $\text{fake_start} = h_addr + h_size$
 $\text{size} = \text{Hex2str}(h_size + 0 \times 8 + 0 \times 1)$
 $\text{count} = 0$
 for($a = 0; a < 4; a++$)
 {
 $\text{cover} = \text{EqExpr}::\text{create}(\text{fake_start} + \text{count}, 0 \times 00)$
 $\text{pre_size_constraint} = \text{AndExpr}::\text{create}(\text{pre_size_constraint}, \text{cover})$
 $\text{count}++$
 }
 for($a = 0; a < 4; a++$)
 {
 $\text{cover} = \text{EqExpr}::\text{create}(\text{fake_start} + \text{count}, \text{size}[a])$
 $\text{size_constraint} = \text{AndExpr}::\text{create}(\text{size_constraint}, \text{cover})$
 $\text{count}++$
 }
 for($a = 0; a < 4; a++$)
 {
 $\text{cover} = \text{EqExpr}::\text{create}(\text{fake_start} + \text{count}, \text{target_addr}[a])$
 $\text{fd_constraint} = \text{AndExpr}::\text{create}(\text{fd_constraint}, \text{cover})$
 $\text{count}++$
 }
 $\text{fakechunk_constraint} = \text{pre_size_constraint} \wedge \text{size_constraint} \wedge \text{fd_constraint}$
 if($\text{solve}(\text{fake_constraint}) = \text{true}$)
 $\text{Heap} \rightarrow \text{chunk}_{\text{exploitable}}$
 end if

算法 2 并未直接检测堆块是否可利用,而是通过在符号区域构建约束,然后对约束进行求解,如果约束有解,说明伪堆块可以构建成功,即说明堆块可利用,反之堆块不可利用。

3.3.3 fastbin 攻击检测算法

程序触发 fastbin 攻击首先要满足分配的堆块大小不大于 80 Byte,属于快堆。然后在已释堆块中寻

找是否有物理地址相邻且与当前堆块大小相同的堆块,如果有,则可以通过第 1 个堆的溢出数据修改相邻堆的头部,伪造假的 fd 指针,使系统误认为 fd 指向的地址是下一个待分配快堆,从而实现任意地址写。

本文通过对堆块分配和释放函数挂钩,检查堆块分配和释放过程中堆块在内存中的布局,建立可导致 fastbin 攻击的溢出数据约束,根据溢出数据约束,具体化溢出数据覆盖的内存区域。算法 3 为程序 fastbin 攻击检测算法。

算法 3 fastbin 攻击检测算法

输入 溢出堆块集合 $\text{chunk}_{\text{overflow}}$, 分配堆块集合 Allo_map , 已释放堆块集合 Free_map

输出 fastbin 攻击发生标识 isFastbin

isFastbin = false

foreach(heap in Allo_map)

if(heap \in $\text{chunk}_{\text{overflow}}$)

if(h_size \in (0, 0x38))

{

foreach(chunk in Free_map)

if(chunk - > h_addr = heap - > h_addr +
heap - > h_size + 0x8)

IsFastbin = true

}

算法 3 准确地描述了 fastbin 操作被触发的特征,即当前堆块是快堆且存在溢出错误,并且物理相邻的是一个相同的快堆且处于释放状态。

3.3.4 指针数据可控性检测算法

在程序符合 fastbin 攻击条件的基础上,检查指定的指针数据是否受到外部数据控制,导致程序控制流被劫持。在 32 bit 系统环境中,指针数据是一段连续的 4 Byte 内存区域(64 bit 系统中为 8 Byte)。本文以三元组 $p = (p_addr, p_sym, p_val)$ 描述指针数据的存放地址、符号化状态与数据约束等属性。通过对指针数据建立可控性约束,实现指针数据具体化。算法 4 为指针数据可控性检测算法。

算法 4 指针数据可控性检测算法

输入 指针数据属性 p, 具体化数据集 dataset

输出 指针数据可控性约束 ptr_constraint

if(p_sym == true)

{

p_val = create::Eq(dataset[p], Int8)

ptr_constraint = ptr_constraint \wedge p_val

}

end if

4 实验结果与分析

4.1 结果分析

本文使用了 3 个含有堆溢出缺陷的实验程序进行验证。其中,fastbin. c 来自 ctf 比赛试题,fastbin_dup. c 来自 shellphish/how2heap 测试集,babyheap 来

自 ctf-challenges 测试集。上述所有程序均采用 C 语言编写。

在面向堆溢出的 fastbin 攻击检测过程中,为更好地验证系统对 fastbin 攻击过程的检测效果,实验中关闭了地址随机化,而保留了 glibc 针对堆块操作过程中相关的保护机制。

为体现系统的效果,对比系统与已有的漏洞攻击检测技术的不同,在实验过程中将每个测试程序分别交由 fastbin 攻击检测系统和 CRAX 系统进行测试,其中, t_1 表示系统完成实验样本分析所用时间, t_2 表示系统完成程序代码生成所用时间。测试用例生成情况如表 1 所示。其中,一为不能生成测试用例。

表 1 测试用例生成情况对比

Table 1 Comparison of test case generation situation

测试程序	fastbin 攻击检测系统		CRAX 系统	
	t_1	t_2	t_1	t_2
fastbin. c	0.8	1.0	0.6	—
fastbin_dup. c	1.2	—	0.8	—
babyheap	2.0	—	1.5	—

从表 1 的结果可以看出,CRAX 系统针对测试用例的分析时间比本文提出的检测系统要短,但是 CRAX 系统对 3 个测试用例均不能生成测试用例。而 fastbin 攻击检测系统对其中的 1 个程序生成了测试用例。这表明针对面向堆溢出的 fastbin 攻击,本文的方法有更好的检测效果。

而针对 2 个没有生成测试用例的例子,本文也对其进行了分析与记录,其测试过程中的情况如表 2 所示。

表 2 fastbin 攻击检测系统的约束构建情况

Table 2 Constraint construction situation of fastbin attack detection system

测试程序	堆块溢出特征	溢出堆块可控特征	fastbin 操作触发特征	指针数据可控特征
fastbin. c	有	有	有	有
fastbin_dup. c	无	无	有	无
babyheap	有	有	有	无

从表 2 可以看出,对于 babyheap,系统检测出了堆溢出特征、溢出堆块可控特征、fastbin 操作触发特征这 3 个特征,而无法检测到关键数据指针可控,通过堆源程序的分析发现,babyheap 要想达到覆盖关键数据指针,劫持程序控制流,必须要泄露 libc 基地址,然后才能检测到关键数据指针可控。因此,如果针对 babyheap 在系统中专门添加 libc 基地址记录模块,可以实现 babyheap 的测试用例的生成。

而对 fastbin_dup. c,经过对源程序的分析发现,程序只有 fastbin 操作触发特征,而无对堆块的操作,因此无法触发堆块溢出检测和溢出堆块可控性检测,更无法控制关键指针,因此无法生成测试用例。

4.2 案例分析

为了更直观地表示 fastbin 攻击方法自动利用原型系统的代码自动生成过程,本文以 fastbin 程序的自动分析与利用过程为例进行案例分析。fastbin 程序的漏洞触发关键代码如代码 1 所示。

代码 1 fastbin 程序的 fastbin 操作触发关键代码

```
bin1 = malloc(0x38);
bin2 = malloc(0x38);
free(bin2);
free(bin1);
bin3 = malloc(0x38)
read(handle1, bin3, 0x64); //引入污点数据,发生堆溢出
//错误
```

```
bin4 = malloc(0x38);
bin5 = malloc(0x38); //触发 fastbin 操作
```

为获得程序分配堆块的相关信息,系统通过在 malloc 函数的返回点进行挂钩,可以获取程序新建堆块的相关信息,并将新建堆块信息添加进入 Allo_chunk 集合中,新建堆块信息如下:

```
this is the 1 time malloc() Ret_Hook
heap_addr=0x804b008
heap_size=0x30
heap_state=1
this is the 2 time malloc() Ret_Hook
heap_addr=0x804b040
heap_size=0x30
heap_state=1
```

对 read 函数返回点进行函数挂钩,可以发现程序从外部读入的数据。然后根据通过挂钩获得外部读入数据在内存中的地址和读入数据大小等信息,将读入数据符号化,将相关内存进行污点标记,检测结果如下:

```
linux read( ) return-hook invoked :
fd      : 3
foffset : 0
databuf : 0x804a03c
size    : 8
```

检测到堆块溢出特征后,使用堆块溢出数据可控算法检测溢出数据的可控性,并构建伪堆块数据约束,约束构建过程如下:

```
(Eq (w8 0x0) (Read w8 0x0 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x1 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x2 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x3 v0_data_offset_0_0))
(Eq (w8 0x39) (Read w8 0x4 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x5 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x6 v0_data_offset_0_0))
(Eq (w8 0x0) (Read w8 0x7 v0_data_offset_0_0))
```

通过对 free 函数的入口点进行挂钩,系统可以获取堆块释放的相关信息,然后将新建堆块集合 Allo_chunk 中的相关信息删除,并将堆块释放信息添加到已释放堆块集合 Free_chunk。最后通过

fastbin 操作触发检测算法根据检测溢出数据可控堆块和已释放堆块的相对位置关系来确定 fastbin 操作是否触发,获取 fastbin 操作触发标识 Is_Fastbin, Is_Fastbin 标识获取结果如下:

```
the chunk is fastbin!
chunk 0x804b008 is exploitable !
And Is_Fastbin = true
```

在判断程序触发了 fastbin 操作的基础上,对关键指针变量进行可控性检查,发现指针数据被污点数据覆盖为符号值,生成指针数据可控约束 PtrConstraint,指针可控性检查结果如下:

```
54 [State 0] Found Symbolic Array at 0x804a044, width 48
ptr = 0x804a044 can be controled!
```

系统通过对程序运行过程中生成的约束进行求解,生成的针对 fastbin 程序的二进制利用代码如图 6 所示。将生成的测试用例作为输入输入至程序中,触发 fastbin 攻击,成功劫持控制流,结果如图 7 所示。

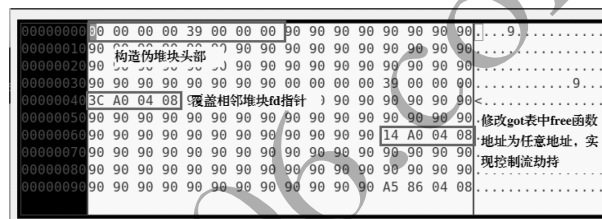


图 6 针对 fastbin 程序自动生成的利用代码

Fig. 6 Utilization code automatically generated for fastbin program

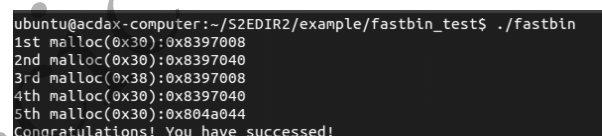


图 7 劫持控制流的结果

Fig. 7 Result of hijacking control flow

5 结束语

本文通过总结面向堆溢出的 fastbin 攻击特征,利用符号执行和污点分析技术,在 fastbin 攻击检测模型和 fastbin 攻击检测算法的基础上设计并实现了 fastbin 攻击检测系统。从堆块溢出特征、溢出堆块可控性特征、fastbin 操作触发特征、数据指针可控性特征 4 个方面来描述面向堆溢出的 fastbin 攻击,判断程序是否符合 fastbin 攻击特征并验证程序的可利用性。基于 3 个程序的实验证明了系统对相关程序检测结果的正确性。但本文 fastbin 攻击检测系统未考虑 ASLR 等系统保护机制对实验的影响,无法实现多漏洞的组合测试,只能单一地检测面向堆溢出的 fastbin 攻击检测。因此,如何实现多漏洞的组合检测将是下一步的研究工作。

参考文献

- [1] LI Zhen, ZOU Deqing, WANG Zeli. Survey on static software vulnerability detection for source code [J]. Chinese Journal of Network and Information Security, 2019, 5(1): 5-18. (in Chinese)
李珍, 邹德清, 王泽丽. 面向源代码的软件漏洞静态检测综述[J]. 网络与信息安全学报, 2019, 5(1): 5-18.
- [2] HE Liang, SU Purui. Research progress on automatic utilization of software vulnerabilities [J]. China Education Network, 2016(21): 46-48. (in Chinese)
和亮, 苏璞睿. 软件漏洞自动利用研究进展[J]. 中国教育网络, 2016(21): 46-48.
- [3] WU Meng. Talking about the research of computer software safety detection technology [J]. Digital Technology and Application, 2012(12): 149-149. (in Chinese)
吴蒙. 浅谈计算机软件安全检测技术研究[J]. 数字技术与应用, 2012(12): 149-149.
- [4] MILLER C, CABALLERO J, BERKELEY U, et al. Crash analysis with BitBlaze [J]. Revista Mexicana de Sociología, 2010, 44(1): 81-117.
- [5] HEELAN S, KROENING D. Automatic generation of control flow hijacking exploits for software vulnerabilities [D]. Oxford, UK: University of Oxford, 2009.
- [6] CHEN Aihong, PENG Weimin. Analysis of the principle and using technology of heap overflow [J]. Computer and Digital Engineering, 2008, 36(9): 117-119. (in Chinese)
陈爱红, 彭伟民. 堆溢出原理及利用技术的分析研究[J]. 计算机与数字工程, 2008, 36(9): 117-119.
- [7] PEI Zhongyu, ZHANG Chao, DUAN Haixin. Several methods of exploiting Glibc heap [J]. Journal of Cyber Security, 2018, 3(1): 1-15. (in Chinese)
裴中煜, 张超, 段海新. Glibc 堆利用的若干方法[J]. 信息安全学报, 2018, 3(1): 1-15.
- [8] CAO Yaobin, WANG Yagang. An overview of the stack protection techniques in the GCC compiler [J]. Information Technology, 2017(7): 23-25. (in Chinese)
曹耀彬, 王亚刚. GCC 编译器中的堆栈保护技术概述[J]. 信息技术, 2017(7): 23-25.
- [9] SHI Dawei, YUAN Tianwei. A dynamic taint analysis method combined with coarse-grained and fine-grained [J]. Computer Engineering, 2014, 40(3): 12-17, 22. (in Chinese)
史大伟, 袁天伟. 一种粗细粒度结合的动态污点分析方法[J]. 计算机工程, 2014, 40(3): 12-17, 22.
- [10] ZHU Zhengxin, ZEENG Fanping, HUANG Xinyi. Dynamic symbolic taint analysis of binary programs [J]. Computer Science, 2016, 43(2): 155-158. (in Chinese)
朱正欣, 曾凡平, 黄心依. 二进制程序的动态符号化污点分析[J]. 计算机科学, 2016, 43(2): 155-158.
- [11] HOWDEN W E. Symbolic testing and the DISSECT symbolic evaluation system [M]. Washington D. C., USA: IEEE Press, 1977.
- [12] ZHAO Yuehua, KAN Junjie. Research and design of test data generation method based on symbolic execution [J]. Computer Applications and Software, 2014(2): 303-306. (in Chinese)
赵跃华, 阚俊杰. 基于符号执行的测试数据生成方法的研究与设计[J]. 计算机应用与软件, 2014(2): 303-306.
- [13] BRUMLUY D, POOSANKAM P, SONG D, et al. Automatic patch-based exploit generation is possible: techniques and implications [C]//Proceedings of IEEE Symposium on Security & Privacy. Washington D. C., USA: IEEE Press, 2008: 158-169.
- [14] AVGERINOS T, SANG K C, HAO B L T, et al. AEG: automatic exploit generation [C]//Proceedings of IEEE Conference on Network and Distributed System Security. Washington D. C., USA: IEEE Press, 2011: 668-679.
- [15] HUANG S K, HUANG M H, HUANG P Y, et al. CRAX: software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations [C]//Proceedings of the 6th IEEE International Conference on Software Security and Reliability. [S. l.]: IEEE Computer Society, 2012: 78-87.
- [16] CHIPOUNOV V, KUZNETSOV V, CANDEA G. S2E: a platform for in-vivo multi-path analysis of software systems [J]. ACM SIGPLAN Notices, 2011, 47(4): 265-278.
- [17] CHIPOUNOV V, KUZNETSOV V, CANDEN G. The S2E platform: design, implementation, and applications [M]. New York, USA: ACM Press, 2012.
- [18] WANG Xue, LI Xuexin, ZHOU Zhipeng, et al. Analysis of the software testing platform: S2E [J]. Netinfo Security, 2012(7): 16-19. (in Chinese)
王学, 李学新, 周智鹏, 等. S2E 测试平台及并行性能分析[J]. 信息网络安全, 2012(7): 16-19.
- [19] CADAR C, DUNBAR D, ENGLER D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs [C]//Proceedings of USENIX Conference on Operating Systems Design and Implementation. [S. l.]: USENIX Association, 2009: 209-224.
- [20] BELLARD F. QEMU, a fast and portable dynamic translator [C]//Proceedings of Annual Conference on USENIX Annual Technical Conference. [S. l.]: USENIX Association, 2005: 396-411.