文章编号:1671-4229(2021)03-0044-15

基于程序动态信息的漏洞自动利用技术研究

孙 起[1],鲁 辉[1],孙彦斌[1],仇计清[2*]
(1.广州大学 网络空间先进技术研究院,广东 广州 510006;2. 河北科技大学 理学院,石家庄 050026)

摘 要:近年来,随着模糊测试技术的发展,漏洞自动挖掘工作取得了较大研究进展,而漏洞自动利用研究进展相对缓慢。文章重点深入研究漏洞利用生成,旨在解决潜在漏洞仍需大量人工辅助进行利用性分析的问题,主要研究工作如下:①以Linux系统下的ELF文件为研究目标,综合考虑各种漏洞利用方式和缓解措施,提出自动化漏洞利用模型;②基于动态插桩获取程序对内存数据的读写情况,并在污点分析基础上提出了一种改进的内存信息获取技术,将程序对内存数据的读写情况记录成能够帮助漏洞利用生成的信息,并在Linux操作系统上实现了一个自动化漏洞利用系统(Exploiter),Exploiter针对目标程序以及使目标程序崩溃的异常输入,可输出一个在目标程序中打开的系统命令解析器。最后的实验结果验证了Exploiter的有效性和整体性能。

关键词:自动漏洞挖掘;自动漏洞利用;内存信息获取;污点分析
中图分类号:TP 311.5 **文献标志码**:A

Research on automatic exploitation of vulnerability technology based on program dynamic information

SUN Qi[1], *LU Hui*[1], *SUN Yan-bin*[1], *QIU Ji-qing*[2*]
(1. Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China;
2. School of Science, Hebei University of Science and Technology, Shijiazhuang 050026, China)

Abstract: In recent years, automatic vulnerability mining has made great progress with the development of fuzzy testing, but the progress of automatic vulnerability exploitation is relatively slow. The main research work is as follows: ① Taking ELF file in Linux system as the research target, considering various ways of vulnerability exploitation and mitigation measures, an automatic vulnerability exploitation model is proposed, ② Based on dynamic stake insertion, an improved memory information acquisition technology is proposed based on Taint Analysis, which records the read and write status of the program to the memory data as the information generated by vulnerability exploitation, and implements an automatic vulnerability exploitation system (Exploiter) on Linux operating system. According to the target program and the abnormal input that causes the target program to crash, the Exploiter can output a system command parser opened in the target program. Finally, the experimental results verify the effectiveness and overall performance of Exploiter.

Key words: automatic vulnerability discoveray; automatic vulnerability exploitation; memory information acquisition; taint analysis

---

基金项目:国家自然科学基金资助项目(61972108);广东省高校创新团队资助项目(2020KCXTD007);广州市高校创新团队资助项目(202032854)

作者简介:孙起(1996—),男,硕士. E-mail:lkjmlkjm@126.com
* 通信作者. E-mail: qiujiqing@263.net

引文格式:孙起,鲁辉,孙彦斌,等. 基于程序动态信息的漏洞自动利用技术研究[J]. 广州大学学报(自然科学版),2021,20(3):44-58.

随着模糊测试技术的发展,现有工具已能够在程序中找到大量的潜在漏洞,如谷歌的 OSS-Fuzz[1]平台对开源应用程序进行持续测试,5 个月即发现了超过 1 000 个潜在漏洞。然而,评估这些潜在漏洞的威胁程度仍然需要大量的人力。如何进一步提高漏洞威胁程度评估的自动化程度成为领域内亟待解决的问题。2016 年,美国国防部高级研究计划局(DARPA)发起“网络安全挑战赛”(CGC)[2],极大推进了全自动网络安全攻防系统的研发,自动化漏洞挖掘和利用技术已经成为新的研究热点。

1　相关工作

为评估潜在漏洞的威胁程度,相关研究工作主要围绕以下两种思路展开。

(1)分析程序崩溃点状态[3]。例如,微软的! exploitable[4]检查崩溃点所在基本块的所有指令,寻找已知可利用模式(如被污染的控制流转移指令)。HCSIFTER[5]采取额外的步骤来恢复堆溢出损坏的数据,使程序能够在崩溃点之后执行更多的代码,从而提供更可靠的评估。这种方法依赖于已有模式,只能大致识别目标状态的可利用性,而不能得到明确可验证结论,存在漏报和误报。

(2)应用多种手段,直接生成一个能够对目标程序达到利用目标的输入,该输入可以重复验证。该思路最早由 Avgerinos 等[6]提出(AEG),通过在预处理阶段将程序源代码分别经 GCC 编译成可执行程序,经 LLVM 生成所需的字节码信息,通过对 LLVM 中间代码的分析,找出存在错误的位置,并建立路径约束和符号执行生成相应的输入;继而利用动态分析方法提取程序在基于异常输入运行时的各类信息,例如,栈上控制流劫持地址,输入被存放的地址等;最后综合符号执行约束条件和动态分析获取的信息构建可利用样本。在利用生成环节,AEG 考虑了两种利用方式:return-to-stack 利用和 return-to-libc 利用。return-to-stack 利用劫持控制流,将返回地址导向可以自定义内容的栈上,并在栈上写入 shellcode 达成利用;return-to-libc 利用 libc 的 execve 函数,并将对应的参数设置为“/bin/sh”来达成利用。AEG 作为第一个自动化漏洞利用生成方案,主要依赖于源码,构造的利用样本局限于栈溢出和字符串格式化漏洞,并且利用样本能否生效还受编译器和动态运行环境的限制。Cha 等[7]在 2012 年的 IEEE S&P 会议上提出的 Mayhem 进一步摆脱了对源码的依赖,并提出两项新技术:混合符号执行和基于索引的内存建模。Mayhem 结合在线/离线符号执行的优势,能够在二进制程序的基础上推理可以被符号化的内存(受用户输入影响)。Mayhem 使用两个可利用性原则来进行约束:①攻击者是否能够劫持 IP,②攻击者能否执行 payload。

高质量、多样性的漏洞利用样本可以帮助分析人员进一步分析程序脆弱性,Wang 等[8]在 2013 年针对控制流劫持类漏洞提出多样性利用样本自动生成方法 PolyAEG,通过动态污点分析找出程序所有控制流劫持的位置,基于不同的控制流转移模式构造出多样的利用样本。PolyAEG 通过组合蹦床指令和有限长度攻击载荷形成的攻击链使得利用生成更加灵活,它对确定可利用性的要求如下:①蹦床指令和 shellcode 可以出现在预期的位置;②在劫持点,控制流可被用户劫持;③蹦床指令能依次执行,直到完整的 shellcode 被执行。DARPA CGC 比赛中缔造 Mechanical Phish 的参赛团队在赛后总结的文章中提到,比赛中的利用生成仍然是依赖于执行 shellcode,通过回顾赛题可以知晓,当时比赛中出现的二进制程序并没有开启任何漏洞缓解措施。

近年来,研究者的目光不再局限于针对栈溢出漏洞自动利用。Hu 等[9]在 2015 年 USENIX Security 会议上首次提出了一种面向数据流利用的自动化构造方法 FlowStitch,其核心思想是在不改变程序控制流前提下,通过篡改程序中的关键变量,达到非预期操作目的。2018 年,Eckert 等[10]提出的 Heaphopper 及 Wang 等[11]提出的 Revery 研究了基于堆溢出的 AEG。Heaphopper 将以堆分配器为目标,能够在存在内存损坏的情况下,自动生成针对堆分配器的利用,从而指导实现和评估堆分配器安全性的改进。Revery 在引起程序崩溃的输入基础上,应用 AFL 发散探索路径寻找可利用状态,拼接可利用状态和崩溃输入生成利用,并在 CTF 程序上验证了想法。在 2019 年的 CCS 会议上,Heelan 等[12]研究了解释器中堆溢出的自动利用生成 Gollum。直接自动生成并完整利用在某些场合比较困难,Wu 等[13]提出了 FUZE,它是一个能够帮助安全研究人员开发内核 UAF 利用的虚

拟机,包括能够自动识别进一步利用 UAF 的系统调用,自动计算安全研究人员需要配射到脆弱对象区域的数据,精确定位何时需要进行堆喷射和漏洞利用,协助绕过安全防御机制。

综上,漏洞自动利用生成的相关研究目前正处于积极探索阶段,还存在以下问题尚待解决:①漏洞自动利用研究针对的问题背景与利用模型没有统一标准,是否可以建立一个清晰的模型,帮助研究者理解现有问题并发现新的问题;②信息获取技术较为单一,是否存在其他动态获取利用信息的方式。基于此,本文对自动化漏洞利用问题的描述与利用模型开展相关研究,并提出一种基于污点分析的内存信息获取技术。

2 自动化漏洞利用背景与利用模型

2.1 程序模型

目前研究的漏洞都是以程序为载体,在研究漏洞之前,首先以编程者的角度对程序进行定义。

定义 1 逻辑程序。逻辑程序是在程序编写之前,编写者对程序预期效果的建模。每个逻辑程序都可以等价于一个特定的图灵机,由一个 6-元组表示 $\theta=(S,s0,\Sigma,\Delta,\delta,\sigma)$。其中,S 是状态的有限集合,s0 是开始状态,它是 S 的元素,Σ 是输入字母表的有限集合,Δ 是输出字母表的有限集合,状态转换函数 $\delta:S\times\Sigma->S$,输出函数 $\sigma:S\times\Sigma->\Delta$。

实际操作中,程序编写者将逻辑程序模型通过编程写入现有的计算机,尽管编程者总是希望实际程序能够有着与逻辑程序完全相同的效果,但由于计算机有自身特定的体系结构,它只是在该结构下对逻辑程序尽可能地模拟,而不能保证完全一致。以将两个字符串进行拼接的程序为例。

(1)δ 对应到具体计算机上不是一个原子操作,而是由多条指令模拟。一般情况下,指令会顺序执行,最终进入一个对应于 S 中的状态,中间每条指令执行的临时状态不会产生影响,但是在一些异常情况下,多条指令带来的状态空间增量使得攻击者在利用漏洞时有了更高的自由度。图 1 中两个字符串的拼接对于逻辑程序来说是一个原子操作,对于实际程序却需要多条指令去完成,可是当攻击者能够控制下一条执行的指令时,就可以选择直接跳转到中间的某一条指令,如 call strcat@ plt 处,使用 strcat 函数完成一些操作。

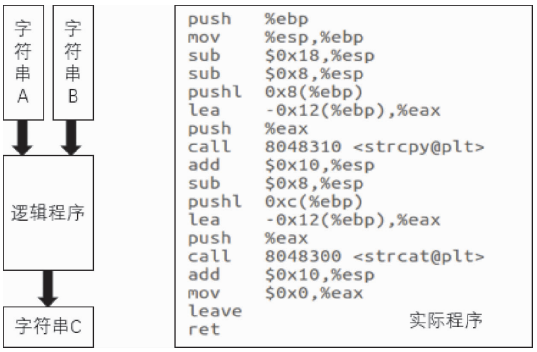

图 1 逻辑程序与实际程序

Fig. 1 Logic program and real program

(2)实际程序在编写过程中产生疏漏,导致程序进入异常状态。在异常状态中,实际程序的状态空间不再符合编程者预期,在某些情况下,程序仍然保持执行,将产生一系列后果。如图 1 的实际程序中,使用了 10 字节长度的缓冲区以保存 2 个字符串拼接的结果,如果拼接结果长度超过 10 字节,程序进入异常状态。

把实际程序在编写过程中产生疏漏的状态空间表示为 S',显然 S' = S' 临时 ∪ S ∪ S' 异常。其中,S' 异常是由于编程者疏漏,程序进入的异常状态,S' 临时是多条指令模拟状态转换产生的中间临时状态,S 是逻辑程序中应该存在的程序状态。如图 1 中所示,一个存在异常状态的实际程序中,S' 异常状态远比编程者预想的大得多。正是由于逻辑程序和实际程序之间存在差异,导致了漏洞的产生,也产生了对漏洞利用的研究。

2.2 漏洞利用模型

2.2.1 通过输入进行编程

在逻辑程序中,从编程者的角度来看,程序根据编程者预设的指令逐条执行,在状态空间 S 中进行转换,用户的输入被当成数据处理;从用户的角度来看,程序根据用户输入的数据在状态空间中进行转换,编程者预设的指令和变量被当作数据。两种视角并无实际意义上的不同,状态空间 S 和编程者的预期是一致的。

图 2 表示一个处理用户输入的程序,用户输入数字、字母及符号时,程序会分别进入 3 种状态。对于编程者来说,程序按照预先编写好的程序,借助于定义好的变量不断地处理来自用户的数据;对于用户来说,程序按照用户的输入不断地

在程序状态中切换,并在此过程中不断修改变量。

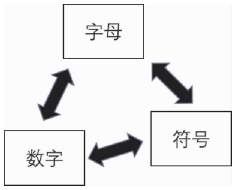

图 2　输入处理程序

Fig. 2　Input handler

在实际程序中,这种视角的不同带来了明显的差异。用户首先通过非预期的操作,控制程序从编程者预想的状态空间中脱离出来,并使用一连串的其他操作达到自己的目的,而这些操作都是编程者无法预知的。对于上述程序,一个明显的非预期操作是用户输入了不可见字符(图 3)。

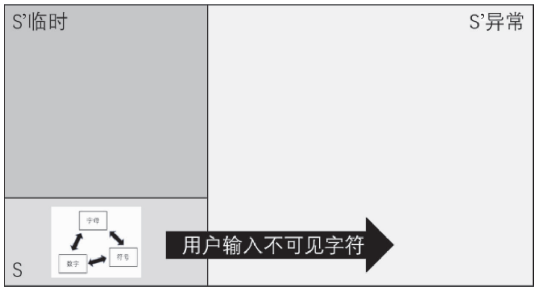

图 3　非预期输入导致程序进入异常状态

Fig. 3　Unexpected input causes the program to enter an abnormal state

2.2.2　漏洞利用过程

在攻击者通过输入对目标状态机进行编程的视角下,漏洞利用的过程根据是否进入异常状态可以分为以下 3 个阶段:

第一阶段:程序正常执行

攻击者通过用户输入使程序状态在 S 与 S' 临时中转换,通过外部输入改写内存上的内容,但是程序仍然忠实地遵循理想程序的状态转换,完成基础的数据处理功能。

第二阶段:进入异常状态

由于攻击者的异常输入,程序执行进入偏离理想程序预期的异常状态,开始以编程者无法预计的方式继续执行。

第三阶段:异常状态编程

程序进入异常状态以后,程序中原本存在的状态转换函数仍然生效,在攻击者的精心布置下,可以通过程序交互进一步影响程序状态转换。整个过程如图 4 所示。

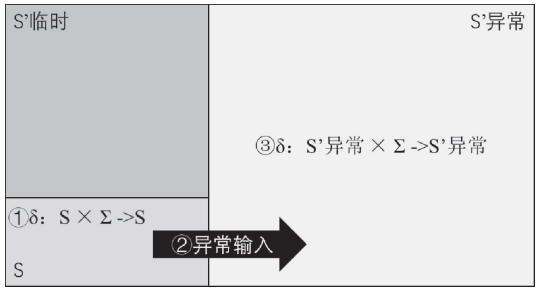

图 4　漏洞利用过程

Fig. 4　Vulnerability exploitation process

2.2.3　利用缓解措施

在有了清晰的程序模型后,可以进一步探究漏洞利用缓解措施的生效原理。本文中归纳的 4 种漏洞利用缓解措施(即数据执行保护、地址随机化、重定向只读保护和栈溢出哨兵保护)分别从约束状态空间 S' 异常和模糊状态转换函数 δ 两个方面来限制漏洞利用。

数据执行保护、重定向只读保护和栈溢出哨兵保护都属于约束状态空间 S' 异常的方式(图 5),开启上述 3 种保护的程序在原本 S' 异常的状态空间中开辟了一块状态空间,专属于异常处理。在异常处理之后还有一个利用状态,在此处表示预先定义的一种利用成功的程序状态。异常处理状态常设定为程序从刚进入异常状态到最终转换成利用状态的中间状态。因此,在漏洞利用时,由于该路径已经被漏洞利用缓解措施拦截而无法贯通。对于这种保护方式,可以采用绕过异常处理状态使保护无法生效。开启了数据执行保护,利用过程中就需避免将数据当作代码执行的操作;对于重定向只读保护,需避免修改重定向表的操作;对于栈溢出哨兵保护,需避免在检查哨兵之前不能使哨兵复位的操作。

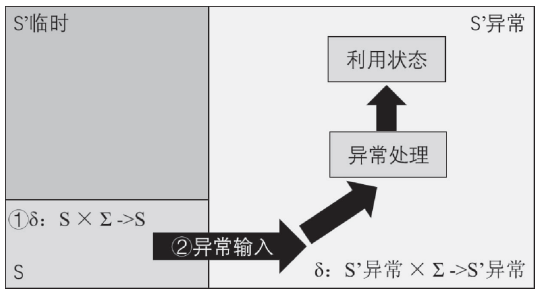

图 5　约束状态空间 S'

Fig. 5　Constrain the state space S'

地址随机化保护属于模糊状态转移函数 δ 的方式,见图 6。开启了地址随机化保护的程序在异

常状态空间中的转移函数δ不再像原来一样清晰,虽然从理论上来说,模糊状态转移函数对于漏洞利用仅有延缓效果,并不能真正阻止任何方式的漏洞利用,但是由于程序使用者获取程序内信息的手段有限,现有的绕过方式不得不借助于一个额外的漏洞来泄露程序内部信息。因此,这种保护方式对程序漏洞利用也起到了很好的缓解作用。

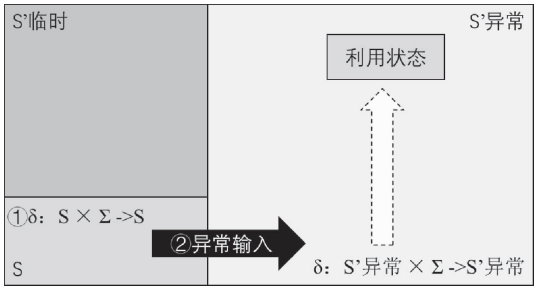

图6 模糊状态转移函数δ

Fig. 6 Blur the state transition function δ

2.3 自动化漏洞利用模型

图7为一般漏洞利用模型。漏洞利用是通过用户输入控制目标程序首先进入异常状态,随后异常状态中在特定的状态转移函数δ的作用下,逐步转换为利用状态的过程(在本章后续内容中,利用状态指目标程序打开一个新的shell的状态)。在上一节中可知,站在使用者的角度,状态的转换可以看作使用者通过输入编程逐步执行的结果。因此,漏洞利用应该是一类程序,这类程序有预先编写好的程序逻辑。而自动化漏洞利用应该是一个系统,它能够根据目标程序的实际情况,自动地选择利用程序,并自动向利用程序提供所需的输入。

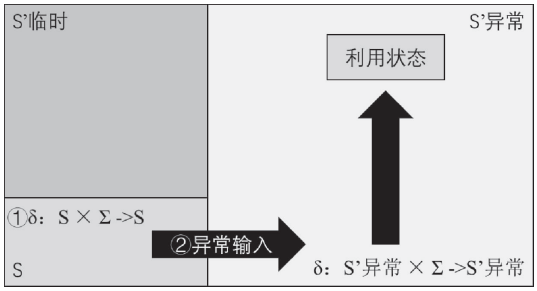

图7 一般漏洞利用模型

Fig. 7 General vulnerability exploitation model

漏洞利用程序是攻击者在利用程序漏洞过程中所采取的可重放的利用思路,每种思路都对应着一种漏洞利用实现的固定操作过程,类似于编程中为某个特定目标编写好的函数。

本文归纳的利用程序有3个:指令数据混淆、重用程序指令和利用动态链接机制。

(1)指令数据混淆。在冯·诺依曼体系结构中用户数据与程序代码未做明确区分,一旦程序的ip指针被用户劫持,指向用户写入数据的位置,该数据就会被当作程序代码执行。因此,用户可以在可控的数据区内写下能够弹出shell的代码,达到获取shell的目的。该利用方式要求目标程序未开启NX保护,程序中存在同时具有写和执行权限的段。

该利用程序操作步骤如下:

1)在可以被用户输入控制的内存中寻找一段可以放下shellcode的空间;

2)通过改写用户输入,将该段内存设置为shellcode;

3)劫持控制流跳转到该段内存的起始地址shellcodeAddr。

输入:劫持ip指针的字符在用户输入中的偏移位置,可以被用户输入控制的内存空间。

输出:payload = … + shellcode + … + shellcodeAddr。

(2)重用程序指令。ROP技术的出现使得用户可以借助于程序的调用返回机制,通过在栈上堆放的指令地址,来重用程序中原本存在的代码片段。该利用方式要求程序中存在足够的代码片段。

该利用程序操作步骤如下:

1)在程序中寻找多个以ret为结尾的代码片段rop gadgets,每个代码片段能够进行一个特定的操作;

2)将多个代码片段的地址及需要的特定数据合理地排列在栈上;

3)栈溢出劫持控制流,逐个执行这些特定的操作。

输入:程序自带的以ret结尾的机器码片段、劫持ip指针的字符在用户输入中的偏移位置。

输出:payload = … + rop gadgets。

(3)利用动态链接机制。通过伪造动态机制中用到的符号信息,可以解析出一个用户自定义的函数,此处想要解析出的函数是system。该利用方式要求程序使用动态链接,并且不完全开启RELRO保护。

该利用程序操作步骤如下:

1)在内存中构造动态链接时使用的数据结构;

2)通过设定偏移,使动态链接机制将伪造的数据结构作为参数进行动态链接,动态链接返回 system 函数,并将已经布置在栈上的/bin/sh 作为参数进行调用。

输入:程序可用 bss 段的地址、程序使用的输入函数、劫持 ip 指针的字符在用户输入中的偏移位置。

输出:payload = … + (write something to bss) + _dl_runtime_resolve + offset + … + (fakeStructure and '/bin/sh')。

3　程序动态信息获取

本文的程序动态信息获取基于污点分析改进而来。传统污点分析的一个典型应用场景是检测程序的关键位置是否被从网络接收到的数据污染,并在影响时发出警报。但是在漏洞利用时,不仅需要知道是否被污染,还要进一步精确地知道是被用户输入的哪个字节污染,从而在此基础上构建可靠的漏洞利用代码。本文改进了污点分析,使得该污点分析系统可以获知影响程序关键位置的数据来源于用户输入数据的哪些字节,从而在漏洞自动利用生成时,针对性地构造用户输入的对应位置。

3.1　自动化漏洞利用模型

传统的污点分析(如 libdft)中,在污染传播之前首先定义一个位图,每个位对应着内存中一个字节,见图 8。对于 32 位虚拟地址空间,共有 4G 的内存空间,该位图的大小应为 512 MB。在大多数内核中,最高的 1G 地址被映射到内核,因此,该位图的大小仅需要 384 MB。在某些情况下,这个位图可以缩减得更小。

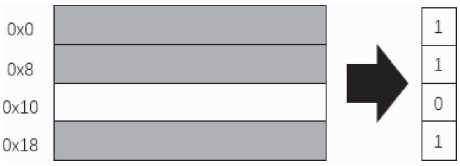

图 8　内存与位图映射关系

Fig. 8　Memory and bitmap mapping

由于对于每个内存字节,仅有一个对应的位,该位只能置为 0 或 1,其中,1 表示被污染,0 表示未被污染,在这种情况下,尽管可以知道内存的某个位置是否被污染,但是却无法区分污染来自哪里。容易理解的是,如果想要区分两种污染的源头,对每个字节需要增加一个位来标记,对于被源头 1 污染的字节,标记为 01,被源头 2 污染的字节,标记为 10,内存占用翻倍。当想要区分 3 种污染时,很容易产生一个误区,认为可以使用 11 来标记第三种污染,但这是错误的。因为当来自不同源头的污染影响同一个字节时,污染会在该内存位置合并,在 2 种颜色的情况下,11 用于表示该内存位置既被源头 1 污染,也被源头 2 污染。当有第三个污染源时,必须使用第三个单独的位来进行跟踪。由此可知,需要区分多少种颜色,就需要多少个位跟踪相应的内存地址。每新增一个污染源,跟踪单个内存字节的位向量长度都会线性增加。

如果想区分内存中 8 种不同颜色的污染,需要对每一个字节的内存使用一个 8 位的位向量来跟踪(图 9)。

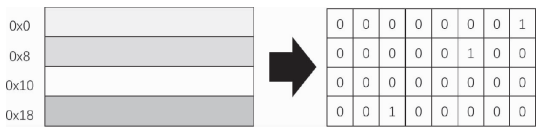

图 9　8 种颜色下的内存与位图映射关系

Fig. 9　Memory and bitmap mapping under eight colors

对于本文想要达成的目的,用户输入的每一个字节都被当成一个独立的污染源,需要使用不同的污染标签(图 10)。使用传统的方法是无法奏效的,主要有 2 个问题:①传统方法中,污染源的数目是在程序执行前就已经固定的,但是程序中用户输入的长度却是变化的,预定一个固定数目位的位向量无法满足要求;②传统方法中,跟踪每个内存字节的位向量长度会随着污染源数目的增加而线性增加,程序中的用户输入可能很长[14],每个内存字节的位向量长度为用户输入字节数,这种情况的内存开销是无法承受的。

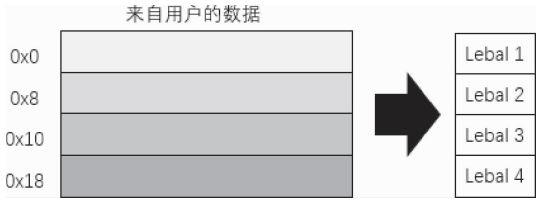

图 10　内存与标签映射关系

Fig. 10　Memory and label mapping

3.2 基于污点分析的内存信息获取技术

如前所述,在以一字节为粒度的情况下,如果仅将内存地址分为2类:被污染和未被污染,需要一次性分配数百MB的内存空间。如果想将内存地址分为更多类,需要的内存空间会随着分类数量的增加而线性增长,占用的空间很快超过内存可用空间。本文考虑从2个方面解决这个问题:①将位图从原本的一次性分配改进为动态分配;②在位图与污染标签中引入一个中间数据结构,专门处理污染合并问题。

3.2.1 地址污染表

地址污染表保存程序中所有内存地址与污染标签的对应关系。在程序实际运行过程中,相对于整个虚拟地址空间的字节数,内存中被污染的字节数只有很小一部分,因此,当该内存地址被污染时,动态分配记录该地址污染所需空间是非常合理的做法。另外,内存污染常常是一次污染一段内存空间,还应考虑每次动态分配带来的开销。因此,本文使用一种3层索引结构。

对于内存污染表,需要提供如下2个操作:

(1)SETB(addr, lebal):将地址addr处的标签设置为lebal。

(2)GETB(addr):获取addr处的标签。

结构中的内存地址数据以下列形式保存。

地址的最高8位作为一级索引,之后的12位作为二级索引,末尾的12位作为三级索引。一级索引表和二级索引表中,存放指向下一级索引表的指针,三级索引处存放该内存地址的具体标签。

一个具体的例子见图11. 0x10203040被来自用户的数据污染标签为1,被放入一个崭新的内存地址污染位图,根据最高位0x10,找到一级索引表中第0x10个位置,发现该位置还没有分配相应的二级索引表,于是分配一个长度为0x1000的二级索引表,根据中间12位0x203,在二级索引表中找到第0x203的位置,像前面一样,分配一个长度为0x1000的三级索引表,在三级索引表中,找到0x040的位置,并将该位置的内容置为1。

3.2.2 污染标签管理结构

污染标签管理结构的初衷是使用一个中间结构来保存污染标签与位向量的映射,此后对每个内存字节的影子内存,都可以保存该污染标签,需要处理合并污染操作时,再取出污染标签对应的位向量进行合并。如图11的索引结构中,最低级的索引表中每个项都是32位无符号整型。如果保存标签,可以表示4 294 967 296种不同的污染标签;如果保存位向量,为支持污染合并问题,仅可以表示32种不同的颜色。

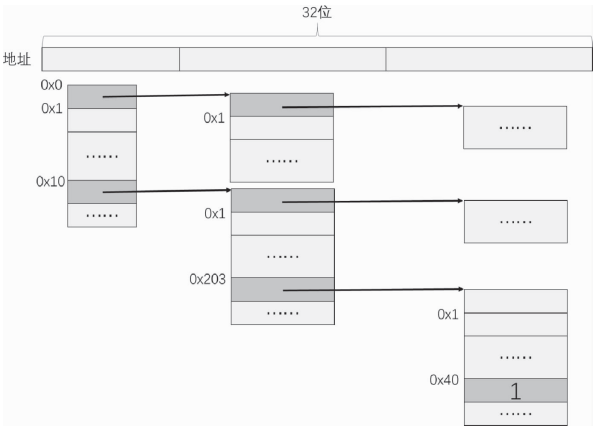

图11 三级索引

Fig. 11 Tertiary index

接下来,以递进的思路来叙述污染标签管理结构的逐步改进。

第一步,污染标签管理结构旨在管理内存标签与位向量的关系。一个最简单的思路就是建立一个数组,以数组的下标作为标签,数组中的每个项存储一个指向位向量的指针(图12)。

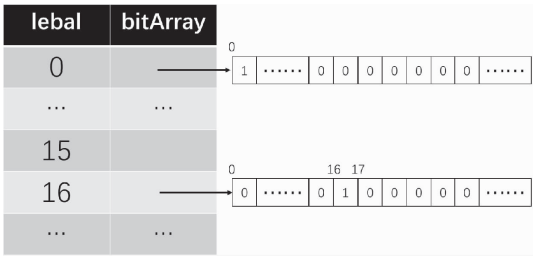

图12 标签位向量映射

Fig. 12 Label bit vector mapping

第二步,考虑到位向量长度是在用户输入的影响下动态变化的,因此,不能使用固定长度的数组储存。而且,位向量中大多数位都是零,保存如此多的零使储存效率极低。回顾位向量与用户输入的联系:用户输入中的第n个字节,对应着的位向量就是一个除了第n位为1,其他位都是0的位向量。因此,本文考虑保存一个数值集合替代位向量,如图13所示。这样做有两个好处:①节约了大量位向量中储存0的不必要空间;②posArray中的pos清晰地表明被该标签标记的内存位置受

到用户输入的第几个字节影响。

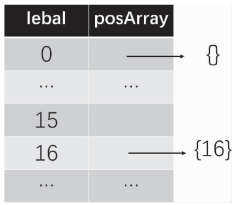

图 13　标签数值集合映射

Fig. 13　Label value set mapping

第三步,完善该数据结构,考虑该结构的数据存储与需要提供的操作。

数据存储:使用一个数组保存标签与 posArray 的映射,数组下标即为污染标签,数组项为一个指向 posArray 的指针。

INSERT(pos):用户输入中的第 pos 个字节作为一个新的污染,插入污染标签管理结构。

COMBINE(lb1, lb2):2 个标签 lb1 和 lb2 对同一个字节产生影响时,这个字节的标签需要被一个合并了 lb1 和 lb2 的新的标签标记,因此,需要提供一个合并操作。

FIND(lb):对于一个标签 lb,在该数据结构中找到能表示它的 posArray。

图 14 中,1 号标签表示被该标签标记的位置受到用户输入中一个字节的影响,29 号标签表示受到用户输入第十五和第十六个字节影响,30 号标签表示受到用户输入第一、第十五和第十六 3 个字节影响,其中,30 号标签是 COMBINE(1,29)的结果。

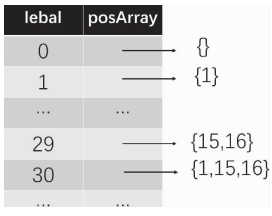

图 14　combine 实例

Fig. 14　Combine example

INSERT 操作和 FIND 操作很简单,可是 COMBINE 操作却存在问题。COMBINE 的操作过程如下:首先通过 FIND 操作,找到 lb1 和 lb2 所对应的 posArray1 和 posArray2,然后将 posArray1 和 posArray2 取并集,放入一个新的 posArrayNew,如果数组中没有与 posArrayNew 相同的项目,则数组增加一项,保存 lbNew 与 posArrayNew 的对应关系,返回 lbNew,否则,找到这个相同的项,返回对应的 lb。对于这个过程,2 次 FIND 操作时间复杂度为 O(1),对 2 个数组取并集,时间复杂度为 O(len(posArray1) + len(posArray2)),简记为 O(m)。再在 nodes 中查找是否有与 posArrayNew 相同的项,由于每个 posArray 项都是数组,比较数组是否相同需要线性的时间复杂度,此操作时间复杂度为O(len(nodes) * len(posArray)),简记为 O(n * m)。综上,整个过程的时间复杂度为 O(n * m)。

第四步,优化 COMBINE 时间复杂度。因为本文希望数组中保存的每个 posArray 各不相同,所以可以使用一个类似于 c ++ 标准模板库中 map 的容器保存每个 posArray,并建立一个 posArray 到 lebal 的逆向查找映射。

通过建立一个并列的映射表结构(图 15),使得每个项可以通过映射表结构逆向找到特定 posArray 对应的 lebal。通过这一点改进,时间复杂度从 O(n * m)改变为 O((lgn) * m),m 表示 posArray 的长度,在实际程序中,m 常常很小,这样时间复杂度变得可以接受。

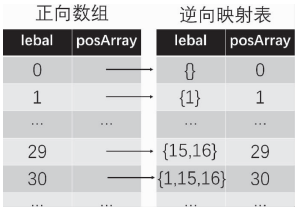

图 15　并列逆向映射

Fig. 15　Parallel inverse mapping

4　漏洞自动利用系统设计与实现

4.1　系统总体框架设计

根据本文提出的漏洞自动利用模型,对应自动漏洞利用的 3 个步骤,分别设计 3 个模块:预处理模块、程序信息获取模块和漏洞自动利用生成模块,系统框架见图 16。

(1)预处理模块负责判断目标程序可能采取的利用方式,判断的主要依据为程序的链接方式、防护机制,同时还要设定输入的方式。

(2)程序信息获取模块负责根据所要采用的利用方式,获取必要的信息,其中包括程序的内存信息。

(3)漏洞自动利用生成模块负责将结合导致程序崩溃的输入、程序中取得的信息以及采用某

种特定利用方式时固有的流程,构造出一个能直接使目标程序执行一个 shell 的有效载荷,并使用该载荷运行目标程序。

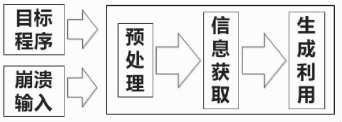

图16 系统框架

Fig. 16 System framework

4.2 预处理模块的实现

预处理模块将程序执行命令、造成程序崩溃输入所在的路径及程序采用的输入方式作为参数读入,具体的例子如下:

python preTreat. py. /rsync-2. 5. 7/rsync -av localhost::src/dest. /rsync-2. 5. 7/payload env

其中,preTreat. py 是自动利用脚本,紧跟的以 rsync 为结尾的字符串表示自动利用的目标程序路径,后面还有几个该目标程序的运行参数,/home 开头、payload 结尾的字符串表示能使该程序崩溃的输入所在的路径,整个字符串末尾的 env 参数指明该程序从环境变量中读取来自用户的数据,造成程序崩溃。

本程序基于 pwntools 获取程序的一些基本信息并与程序进行交互,通过 pwntools 中的 ELF 模块,可以方便地获取程序各项保护机制的开启情况,根据开启情况,在 ret2shellcode、rop 和 ret2dlresolve 3 种利用方式中进行选择。

4.3 程序内存信息获取模块的实现

程序内存信息获取模块基于 pin 动态二进制插桩框架实现,参考动态污点分析的实现原理,并在此基础上改进,形成改进的内存信息获取模块。详细的实现围绕 4 个方面:污染源、污染传播、污染槽和污染存储。

4.3.1 污染源

Exploiter 支持 3 种污染源:来自程序命令行参数的污染、来自标准输入的污染(read 系统调用)和来自环境变量的污染. pin 框架中,提供了 KNOB 类,用户可以通过该类设定一些运行 pintool 时的选项。程序内存信息获取模块中,用户通过设置使用 pintool 时的参数,可以决定是否开启来自参数的污染、来自标准输入的污染和来自环境变量的污染。

此处以对于来自参数的污染为例,使用 RTN_AddInstrumentFunction 进行函数粒度的插桩,在插桩函数中,判断函数名称是否为 main,如果是,则在函数的起始位置插入分析函数 beforeMain,分析函数的算法如算法 1 所示。取得污染参数存储的内存位置,将该段内存位置的内容逐个标记成污染字节,在标记的过程中,每个字节都有一个特定的污染标签,后续可以通过该标签来确定内存中的污染来自于用户输入的第几个字节。

算法 1 设置来自参数的污染

输入: *argv* 程序参数指针数组,*index* 污染参数索引

```
1: function BEFOREMAIN(argv, index)
2:     buf←argv[index]
3:     count←strlen(argv[index])
4:     for each i ∈ [0, count) do
5:         tag←tagAlloc(i)
6:         tagmapSetByte(buf + i, tag)
7:     end for
8:     return
9: end function
```

4.3.2 污染传播

为了保证污染在程序执行过程中的传播,需要对大量指令的系统调用进行插桩,对于这一部分工作,本文主要在 libdft 的基础上实现,在此仅作简单说明。

Exploiter 的污染传播以字节为单位,用户输入的每个字节分别对应一个污染标签,在污染传播过程中,不同的污染标签可能发生合并操作,则生成一个新的标签,包含合并的 2 个污染标签存储的信息。每个内存地址最多有一个污染标签,该标签保存的污染信息能够确定内存中该字节可能受到用户输入的哪几位影响。为了支持可能数量比较大的污染标签,以及标签之间的合并,本文设计了一个特殊的数据结构,用来存储污染标签与污染信息的对应关系,该部分内容将在之后的污染存储小节中详细介绍。

4.3.3 污染槽

Exploiter 在 ret 指令处设置污染槽,在每个 ret 指令处检查当前栈顶指针,如果栈顶指针的内容被污染,说明程序执行的下一条指令受到用户输入的影响,这不符合编程者的预期,程序可能存在栈溢出漏洞,于是结束该程序,并将用户输入对程序内存造成的影响信息返回,供进一步的利用生成使用。

4.3.4 污染存储

为了实现进一步的漏洞利用,Exploiter 专门改

进了污点分析的污染存储,使得动态污点分析技术能够支持标签数量不固定且可能较多的情况。以下分别介绍记录程序中对应地址污染的位图和污染标签管理结构的具体实现。

对于地址污染表,它是一个内存地址到污染标签的映射。其中,分别涉及到 2 个操作,即 SETB(addr, lebal),将地址 addr 处的标签设置为 lebal 和 GETB(addr),获取 addr 处的标签。

SETB(addr, lebal)算法如算法 2 所示。首先判断标签是否为 0(0 标签是一个预留的标签,用来表示没有被污染),如果没被污染就返回。然后在全局的 TableList 中根据 addr 的最高 12 位寻找对应位置的 pageTable 是否为空,如果为空,则创建一个新的 pageTable,并放入该位置。取得已经存在的 pageTable 作为 curTable,并根据 addr 的中间 12 位检查 curTable 对应位置是否为空,为空则创建 page。取得对应 page,根据 addr 的后 8 位,找到存放标签的位置,将该位置赋值为当前标签。

算法 2 设置地址标签
输入: *env* 环境变量地址
```
1: function SETB(addr, lebal)
2:     if lebal == 0 then
3:         return
4:     end if
5:     if Table List[PAGETABL E(addr)] == NULL then
6:         newTable←new TableType()
7:         if newTable≠NULL then
8:             TableList[PAGETABLE(addr)] + -newTable
9:         end if
10:    end if
11:    curTable←TableList[PAGETABLE(addr)]
12:    if curTable[PAGE(addr)] == NULL then
13:        newPage←new    PageType()
14:        if newPage≠NULL then
15:            curTable[PAGE(addr)]←newPage
16:        end if
17:    end if
18:    curPage←curTable[PAGE(addr)]
19:    curPage[OFFSET(addr)]←tag
20: end function
```

GETB(addr)算法如算法 3 所示。根据三级索引逐级检查当前索引项是否为空,如果不为空,则继续查找;如果能查找到最终存在的项,就返回该位置存放的标签,否则直接返回空标签。

算法 3 获取对应地址的污染标签
输入: *addr* 地址
输出: *lebal* 标签
```
1: function GETB(addr)
2:     if TableList[PAGETABL E(addr)] ≠ NULL then
3:         curTable←Table List[PAGETABL E(addr)]
4:         if curTable[PAGE(addr)] ≠ NULL then
5:             curPage←curTable[PAG E(addr)]
6:             if curPage ≠ NULL then
7:                 return cur Page[OF FS ET(addr)]
8:             end if
9:         end if
10:    end if
11:    return NULL
12: end function
```

对于污染标签管理结构,它是一个污染标签到污染信息的映射。其中,污染信息是一个数组,数组中保存着的是用户输入中的偏移,它表示被某个污染标签标记的内存地址受到用户输入中这些偏移的字节影响。例如,对于一个映射{lebal1:[3,5,7]}表示被 lebal1 标记的内存地址的内容受到用户输入中第三个、第五个和第七个字节的影响。

本文使用 c ++进行实现时,用 std::set < tag_off > 结构来保存用户输入偏移的数组,使用一个 std::vector < std::set < tag_off > * > 结构,用来存储污染标签到用户输入偏移数组的映射,其中,vector 的下标索引号即污染标签。为了支持合并污染时通过用户输入偏移数组逆向查找对应污染标签的操作,又使用了 std::map < std::set < tag_off >, lb_type > 结构,保存用户输入偏移数组到污染标签的映射。

该结构有 3 个必要的操作:INSERT(pos)、COMBILE(lb1, lb2)及 FIND(lb)。其中,FIND(lb)操作非常简单,可以直接从标签与用户输入偏移数组中取得对应于 lb 的项,另外 2 个稍微复杂一些。

INSERT(pos)过程描述如下:

输入:pos 当前插入字节在用户输入中的位置

输出:lb 插入项对应的标签

步骤 1:创建一个 std::set < tag_off > 保存该标签对应的用户输入偏移集和 cur;

步骤 2:将 pos 作为一个元素插入集和 cur;

步骤 3:将集和 cur 插入保存输入偏移数组指针的正向映射数组;

步骤 4:将集和 cur 以及当前 lb 组成一组键值对保存在标签与输入偏移数组指针的逆向映射字典中。

COMBINE(lb1,lb2)过程描述如下:

输入:lb1,lb2 需要合并的 2 个标签

输出:lb 合并后的标签

步骤 1:分别检查两个标签是否为空,如果是,则返回另一个;

步骤 2:将 2 个标签对应的输入偏移数组取并集放入一个新的集和 tmp;

步骤 3:在标签与输入偏移数组指针的逆向映射字典中查找该集和 tmp;

步骤 4:如果找到一模一样的集和,则删除该临时集和,并返回该一模一样的集和对应的标签。如果没有找到,则将 tmp 集和分别加入正向和逆向映射中,并返回新创建的标签。

4.4 漏洞自动利用生成模块

漏洞利用生成模块主要基于 pwntools 实现,在前序获取信息的基础上,自动构造新的利用输入,并通过 pwntools 中的 process 模块运行程序进行交互,最终在 pwntools 提供的交互窗口中 get shell,验证利用成果。

5 系统测试与结果分析

本章设置 2 组实验对 Exploiter 系统进行测试和评估。第一组实验使用一些真实的漏洞程序与相应的崩溃输入,重点测试系统基本功能的有效性、健壮性以及各功能模块的性能;第二组实验通过在漏洞程序中进行一些规范化的布置,测试系统在理想环境下的功能实现。

5.1 实验环境与测试对象

本实验运行在 Windows 物理机上通过 vmware 虚拟化构建 ubuntu 16.04 系统中,使用的 CPU 是 Intel(R) Core(TM) i5-8250U CPU@1.60 GHz,虚拟机分配了 2 GB 内存。

实验使用的测试对象的选取参考了漏洞利用相关研究 AEG[6]、MAYHEM[7] 和 CRAX[15],全部来自于真实的漏洞程序,在编译时指定为 32 位,并使用-fno-stack-protector 参数关闭了 canary 栈溢出保护机制,使用-z execstack 参数关闭了 NX 保护机制,并且在系统中关闭了 ASLR 保护机制。其中,canary 与 ASLR 保护机制本系统没有考虑,NX 保护机制使得 ret2shellcode 的利用方式失效,不影响 rop 与 dlresolve 的利用方式,编译多组程序分开实验的意义不大,因此,在实验过程中,使用的是统一的编译方式。

5.2 系统功能测试

5.2.1 真实程序测试

本文选取了一些真实的漏洞程序 aeon-0.2a、aspell-0.50.5、glftpd-LNX_1.24、htget-0.93、iwconfig V.26、ncompress-4.2.4、proftpd-1.3.0a 和 rsync-2.5.7,使用如图 17 中的命令行执行系统。

```
command_list = [
"python preTreat.py ./commentest/aeon-0.2a/aeon ./commentest/aeon-0.2a/payload env",
"python preTreat.py ./commentest/aspell-0.50.5/word-list-compress d ./commentest/aspell-0.50.5/payload read",
"python preTreat.py ./commentest/glftpd-LNX_1.24/dupescan ./commentest/glftpd-LNX_1.24/payload arg",
"python preTreat.py ./commentest/htget-0.93/htget ./commentest/htget-0.93/payload arg",
"python preTreat.py ./commentest/iwconfig/iwconfig ./commentest/iwconfig/payload arg",
"python preTreat.py ./commentest/ncompress-4.2.4/compress ./commentest/ncompress-4.2.4/payload arg",
"python preTreat.py ./commentest/proftpd-1.3.0a/ftpdctl -s ./commentest/proftpd-1.3.0a/payload arg",
"python preTreat.py ./commentest/rsync-2.5.7/rsync -av localhost::src /dest ./commentest/rsync-2.5.7/payload env"
]
```

图 17 测试命令列表

Fig. 17 Test command list

为了直观地看到利用是否成功,本文在系统中进行如图 18 中的设置:当利用成功后,向目标程序发送命令 echo pwned,接受返回结果,如果成功收到“pwned”字符串,说明利用成功,并向标准输出打印“yes”字符串。

```
def test(io):
    if PWNED == 1:
        io.sendline("echo pwned")
        if "pwned" in io.recvuntil("pwned"):
            print "yes"
            exit()
    else:
        io.interactive()
```

图 18 测试配置

Fig. 18 Test configuration

随后,在图 19 这个简单的脚本中,如果捕获到“yes”字符串,说明利用成功,打印“yes”,否则,打印“no”。

```
name_list = [
"aeon-0.2a",
"aspell-0.50.5",
"glftpd-LNX_1.24",
"htget-0.93",
"iwconfig V.26",
"ncompress-4.2.4",
"proftpd-1.3.0a",
"rsync-2.5.7"
]
i = 0
for command in command_list:
    p = os.popen("timeout 100 " + command)

    if "yes\n" in p.readlines():
        print name_list[i].ljust(20) + "yes"
    else:
        print name_list[i].ljust(20) + "no"
    i += 1
```

图 19 测试配置

Fig. 19 Test configuration

分别测试 3 种利用方式,实验结果如图 20。

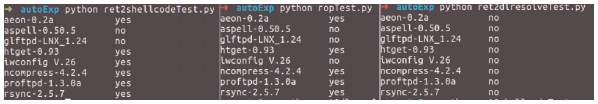

图 20　测试结果

Fig. 20　Test result

最终的实验结果归纳如表 1 中所示。其中,3 种输入方式表示触发漏洞程序的输入分别来自环境变量、程序参数与标准输入。Y 表示利用成功,N 表示利用失败。

表 1　真实程序实验结果表

Table 1　Experimental results of real program

目标程序	输入方式	利用方式		
		ret2shellcode	rop	ret2dlresolve
aeon-0.2a	env	Y	Y	N
aspell-0.50.5	stdin	N	N	N
glftpd-LNX_1.24	arg	N	N	N
htget-0.93	arg	Y	Y	N
iwconfig V.26	arg	Y	N	N
ncompress-4.2.4	arg	Y	Y	N
proftpd-1.3.0a	arg	Y	Y	N
sync-2.5.7	env	Y	Y	N

实验结果表明,Exploiter 的 ret2shellcode 利用方式和 rop 利用方式可以在大多数漏洞程序上生效,ret2dlresolve 的效果并不理想。

5.2.2　理想程序测试

为了进一步验证 Exploiter 系统 ret2dlresolve 利用方式的有效性,本文对实验程序进行理想化修改,将造成程序崩溃的输入统一成通过 read 函数读入,实验情况如图 21。

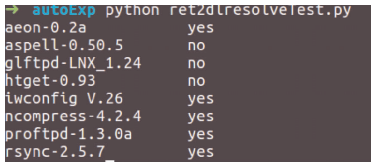

图 21　理想程序结果

Fig. 21　Experimental results of ideal program

经过修改之后的程序,能够被 ret2dlresolve 利用方式成功利用。

对于理想程序的实验结果如表 2,实验结果表明,该系统 ret2dlresolve 的利用方式也能在大多数程序中生效。

表 2　理想程序结果

Table 2　Experimental results of Ideal program

目标程序	输入方式	利用方式		
		ret2shellcode	rop	ret2dlresolve
aeon-0.2a	env	Y	Y	Y
aspell-0.50.5	stdin	N	N	N
glftpd-LNX_1.24	arg	N	N	N
htget-0.93	arg	Y	Y	N
iwconfig V.26	arg	Y	N	Y
ncompress-4.2.4	arg	Y	Y	Y
proftpd-1.3.0a	arg	Y	Y	Y
sync-2.5.7	env	Y	Y	Y

5.3　系统性能测试

在性能测试中,本文分别对内存信息获取模块和漏洞自动利用生成模块进行测试。

5.3.1　内存信息获取模块

(1)基本性能测试

内存信息获取模块中,地址污染表和污染标签管理结构分别管理地址与污染标签的关系以及污染标签和输入字节偏移的关系,两者的内存占用是本文重点关注的因素,本文在程序运行过程中,动态地计算它们的内存占用。

对于地址污染表,它是一个 3 层索引结构:顶层是一个有着 256 项的指针数组,占用的字节数为 256 * 4 bytes;第二层和第三层分别是有着 4 096 项的指针数组和标签数组,每个表的占用都是 4 096 * 4 bytes。因此,假设第二层、第三层分配的表数目分别为 m 和 n,计算内存开销的公式为 mem = 256 * 4 + (m + n) * 4 096 * 4。

对于污染标签管理结构,它分别使用了一个 vector 作为正向的索引和一个 map 作为逆向的索引。2 个索引表的内存占用为表自身的内存开销加上表中每一项的内存开销乘以项数,公式为 mem1 = sizeof(_Rb_tree) + (sizeof(std::set<tag_off>) + sizeof(lb_type) + sizeof(std::priv::_Rb_tree_node_base)) * forward.size()。另外,每一个标签对应的输入字节偏移数组长度都不相同,需要遍历每一项,进行叠加计算,其中,叠加计算每一项内存开销的公式为 mem2 += sizeof(_Rb_tree) + sizeof(std::set<tag_off>*) + (sizeof(*(*nodes[i]).begin()) + sizeof(std::priv::_Rb_tree_node_base)) * (*nodes[i]).size()。最终总体的开销 mem = mem1 + mem2。

除此以外,时间开销也是本文关注的要素,最终实验结果如表 3 所示。

表 3 内存信息获取模块性能

Table 3 Memory information acquisition module performance

目标程序	内存		时间/ms
	地址污染表/bytes	污染标签管理结构/bytes	
aeon-0.2a	50 176	101 968	740
aspell-0.50.5	x	x	x
glftpd-LNX_1.24	33 792	54 936	780
htget-0.93	82 944	140 968	900
iwconfig V.26	33 792	65 176	700
ncompress-4.2.4	82 944	212 136	880
proftpd-1.3.0a	33 792	56 164	880
rsync-2.5.7	50 176	26 704	1 000

从结果中可以发现,地址污染表和污染标签管理结构的内存开销有着相同的数量级别,大致在数十 kb 到数百 kb 之间,相比于传统污点分析动辄数百 mb 来说是非常理想的。另外,所有程序都能在 1 s 内完成工作。

(2)额外输入测试

在前一个实验中,本文使用能使目标程序崩溃的输入驱动程序执行,但是这个长度在很多时候是灵活变化的,本文通过继续增加用户输入长度,来观察用户输入长度增加对地址污染表和污染标签管理结构的内存开销影响。对于 htget-0.93 程序,增加长度会直接改变程序逻辑,不符合本实验目的;对于 aspell-0.50.5 程序,内存信息获取模块无法工作,则对剩下的 6 个程序进行本实验。

图 22 中,x 轴表示在原本能使目标程序崩溃的输入基础上,增加的用户输入个数,y 轴表示地址污染表的内存开销。可以看到用户输入继续增加 50 个字符并不会影响地址污染表的内存开销,原因在于 3 级索引结构一次会开辟 4 096 bytes 的空间,只要用户输入增加长度没有超过已经开辟空间的范围,将不会带来新的内存开销。

图 23 展示了污染标签管理结构的内存消耗随用户输入增加的变化趋势。可以看到,用户输入每增加一位,都会影响污染标签管理结构的内存消耗,原因在于用户输入每多一位,污染标签管理结构都需要一个额外的标签来标记该输入字节。除了线性的增长以外,图中某些位置出现了跳跃式增长,说明随着输入长度的增加,程序中出现了一些新的涉及污点分析的操作。

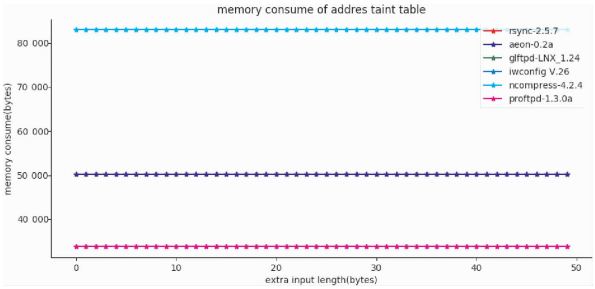

图 22 地址污染表内存消耗随用户输入增加的变化情况

Fig. 22 Change of memory consumption of address pollution table with the increase of user input

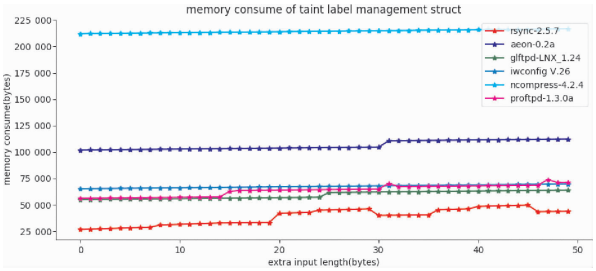

图 23 污染标签管理结构内存消耗随用户输入增加的变化情况

Fig. 23 Change of memory consumption of pollution label management structure with the increase of user input

总体而言,用户输入长度的增加给地址污染表和污染标签管理结构带来的额外内存开销是非常有限的,说明本文所设计的结构能够在更长的用户输入情况下保持可用性。

(3)污染标签管理结构性能对比测试

本文在改进的内存信息获取中设计的污染标签管理结构与 Angora[16] 中设计的一种字节级污点跟踪的思路有相似之处,2 者同样包含了 INSERT 操作和 COMBINE 操作,分别处理污染的添加与合并操作。本文分别从 2 个系统中剥离该数据结构,专门测试 2 个结构在不同次数的 INSERT 操作和 COMBINE 操作下的时间消耗。具体实验中,先执行一定次数的 INSERT 操作,向结构中插入最初的污染,然后执行 COMBINE 操作,随机地将结构中的污染进行合并,得到的结果如图 5 ~ 10 所示。

如图 24 所示,insert times 表示调用 insert 插入新的污染的次数,combine times 表示调用 combine 合并污染的次数。

图中圆圈表示的是本文所设计的数据结构,可以看到该数据结构对 INSERT 操作的增多表现良好,但是随着 COMBINE 操作的增加,时间消耗会有比较明显的上升。

图中三角表示的是 Angora 中所使用数据结构的性能表现,可以看到,随着插入污染字节的数量增多,该结构时间消耗快速上升,而 COMBINE 操作次数的增加对该结构影响较小。

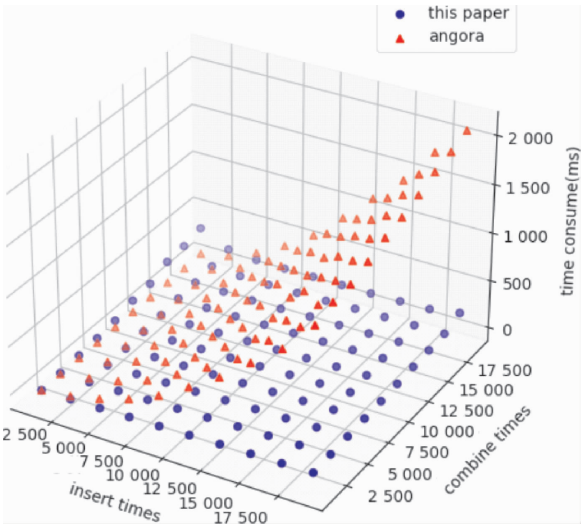

图 24　不同插入次数与合并次数下的时间消耗

Fig. 24　Time consumption under different insertion times and merging times

总体而言,在本文选取的数据范围内,本文提出的污染标签管理结构从时间上来说有着更好的性能表现。

5.3.2　漏洞自动利用生成模块

在对漏洞自动利用生成模块的性能测试中,对于 ret2shellcode 和 rop 2 种利用方式使用了实际程序作为测试目标,ret2dlresolve 利用方式中,使用了理想程序作为测试目标,实验结果如表4 所示。其中,标记为 x 的表示该利用方式无法生效。

表 4　漏洞自动利用模块性能

Table 4　Automatic exploit module performance

目标程序	实际程序		理想程序
	ret2shellcode /ms	rop/ms	ret2dlresolve /ms
aeon-0.2a	525	53 790	342
aspell-0.50.5	x	x	x
glftpd-LNX_1.24	x	x	x
htget-0.93	383	62 248	x
iwconfig V.26	396	63 173	339
ncompress-4.2.4	430	x	380
proftpd-1.3.0a	417	68 103	395
rsync-2.5.7	503	66 710	561

从表 4 中可以看出,3 种利用方式中,ret2shellcode 和 ret2dlresolve 时间开销相近,都比较小,而 rop 由于需要在程序中寻找程序片段用于利用生成,时间开销大得多。

6　总结与展望

本文首先调研漏洞自动利用生成的相关背景,并逐步建立起一个清晰的自动化漏洞利用模型,将自动化漏洞利用中的问题纳入模型中的 3 个环节,即利用程序、程序输入和程序选择。

其次,尝试解决利用程序的输入问题,即应用插桩技术自动获取利用程序成功生成利用所需的信息。本文使用动态二进制插桩方式,借鉴了污点分析的思路,设计出一种基于污点分析的内存信息获取技术。

在以上 2 点研究工作的基础上,本文建立了一个自动漏洞利用系统 Exploiter,并在一些真实的漏洞程序上验证了该系统的有效性。

在整个研究过程中,还存在以下 4 个方面,值得进一步研究:

(1)内存信息获取可分为数据获取和信息组织 2 块,数据获取从技术上已经有了可行性,但是信息组织的研究较少。在程序中,除了污点数据以外,堆的分配与释放、变量的生命周期以及进程间通信等都是容易出现问题的方面,如何将这些信息保存并呈现,是今后工作中一个值得深入的问题。

(2)本文抽象了一个完整的漏洞自动利用模型,但是仅基于该模型归纳出 3 种原理下的漏洞利用程序,还有更多其他原理的漏洞能够基于该模型固化成可复用的程序,在未来的工作中,可以进一步完善。

(3)本文所设计的系统仅考虑运行在 Linux 环境下,没有考虑可移植性,在不同的环境下,可能需要进行相应的修改。

(4)本文设计的系统作为一个实验性质的系统,考虑的情况很少,在使用其他程序作为输入时,可能会遇到更多意想不到的问题,同时也未考虑结合实际应用场景[17-20]。同时,因为时间与能力的限制,无法大规模测试各种程序,因此,本系统的有效性还需在更多的实践中进行验证。

参考文献:

[1] Aizatsky M, Serebryany K, Chang O, et al. Announcing OSS-Fuzz: Continuous fuzzing for open source software[J]. Google Testing Blog, 2016.

[2] 田志宏, 张永铮, 张伟哲, 等. 基于模式挖掘和聚类分析的自适应告警关联[J]. 计算机研究与发展, 2009, 46(8): 1304-1315.

[3] Serebryany K, Bruening D, Potapenko A, et al. Addresssanitizer: A fast address sanity checker[C]//2012 USENIX Annual Technical Conference (USENIX ATC 12). Berkeley: USENIX Association, 2012: 309-318.

[4] Microsoft Corporation. !exploitable crash analyzer-MSEC debugger extensions[EB/OL]. (2013-05-01)[2021-07-03]. http://msecdbg.codeplex.com/.

[5] He L, Cai Y, Hu H, et al. Automatically assessing crashes from heap over? ows[C]//Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. Piscataway: IEEE, 2017:274-279.

[6] Avgerinos T, Cha S K, Rebert A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.

[7] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code[C]//2012 IEEE Symposium on Security and Privacy. Piscataway: IEEE, 2012: 380-394.

[8] Wang M, Su P, Li Q, et al. Automatic polymorphic exploit generation for software vulnerabilities[C]//International Conference on Security and Privacy in Communication Systems. Cham: Springer, 2013: 216-233.

[9] Hu H, Chua Z L, Adrian S, et al. Automatic generation of data-oriented exploits[C]//24th USENIX Security Symposium (USENIX Security 15). Berkeley: USENIX Association, 2015: 177-192.

[10] Eckert M, Bianchi A, Wang R, et al. Heaphopper: Bringing bounded model checking to heap implementation security[C]//Proceedings of the 27th USENIX Security Symposium (USENIX Security 18). Berkeley: USENIX Association, 2018: 99-116.

[11] Wang Y, Zhang C, Xiang X, et al. Revery: From proof-of-concept to exploitable[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2018: 1914-1927.

[12] Heelan S, Melham T, Kroening D. Gollum: Modular and greybox exploit generation for heap overflows in interpreters[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2019: 1689-1706.

[13] Wu W, Chen Y, Xu J, et al. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities[C]//Proceedings of the 27th USENIX Security Symposium (USENIX Security 18). Berkeley: USENIX Association,2018: 781-797.

[14] Wang Y, Tian Z, Zhang H, et al. A privacy preserving scheme for nearest neighbor query[J]. Sensors, 2018, 18(8): 2440.

[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]//2012 IEEE Sixth International Conference on Software Security and Reliability. Piscataway: IEEE, 2012: 78-87.

[16] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). Piscataway: IEEE, 2018: 711-725.

[17] Shafiq M, Tian Z, Bashir A K, et al. IoT malicious traffic identification using wrapper-based feature selection mechanisms [J]. Computers & Security, 2020, doi:10.1016/j.cose.2020.101863.

[18] Shafiq M, Tian Z, Bashir A A, et al. Data mining and machine learning methods for sustainable smart cities traffic classification: A survey[J]. Sustainable Cities and Society, 2020,doi:10.1016/j.scs.2020.102177.

[19] Shafiq M, Tian Z, Sun Y, et al. Selection of effective machine learning algorithm and bot-IoT attacks traffic identification for internet of things in smart city[J]. Future Generation Computer Systems, 2020, 107: 433-442.

[20] Shafiq M, Tian Z, Bashir A K, et al. CorrAUC: A malicious bot-IoT traffic detection method in IoT network using machine-learning techniques[J]. IEEE Internet of Things Journal, 2020, 8(5): 3242-3254.

【责任编辑: 卓祯雨】