计算机科学

COMPUTER SCIENCE

一种基于实时代码装卸载的代码重用攻击防御方法

侯尚文, 黄建军, 梁彬, 游伟, 石文昌

引用本文

侯尚文, 黄建军, 梁彬, 游伟, 石文昌. 一种基于实时代码装卸载的代码重用攻击防御方法[J]. 计算机科学, 2022, 49(10): 279-284.

HOU Shang-wen, HUANG Jian-jun, LIANG Bin, YOU Wei, SHI Wen-chang. Defense Method Against Code Reuse Attack Based on Real-time Code Loading and Unloading[J]. Computer Science, 2022, 49(10): 279-284.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

GDL:一种通用型代码重用攻击 gadget 描述语言

GDL:A Gadget Description Language for General Code Reuse Attack

计算机科学, 2020, 47(6): 284-293. https://doi.org/10.11896/jsjkx.190700109

基于 DirectUI 可扩展应用程序架构的设计与实现

Expanded Application Framework Based on DirectUI

计算机科学, 2012, 39(Z11): 295-300.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220500091

一种基于实时代码装卸载的代码重用攻击防御方法

侯尚文　黄建军　梁　彬　游　伟　石文昌
中国人民大学信息学院　北京 100872
(18810968233@163.com)

摘　要　近年来,代码重用攻击(Code Reuse Attack)已经成为针对二进制程序的一种主流攻击方式。以ROP为代表的代码重用攻击,利用内存空间中存在的指令片段,构建出能实现特定功能的指令序列,达成了恶意目标。文中根据代码重用攻击的基本原理,提出了基于实时装卸载函数代码的防御方法,通过动态装卸载的方式裁剪代码空间,从而达到缩小攻击面以防御代码重用的目的。首先,以静态分析的方式获取受保护程序依赖库的函数信息;以替换库的形式使用这些信息;其次,在Linux动态装载器中引入实时装载函数的操作及自动触发和还原的装卸载流程,为了减小频繁卸载导致的高额开销,设计了随机化批量卸载机制;最后,在真实环境中开展实验,验证了该方案防御代码重用攻击的有效性,展示了随机卸载策略的意义。

关键词:代码重用攻击;实时代码装卸载;面向返回编程;动态链接库;随机卸载
中图法分类号　TP309.5

Defense Method Against Code Reuse Attack Based on Real-time Code Loading and Unloading

HOU Shang-wen,HUANG Jian-jun,LIANG Bin,YOU Wei and SHI Wen-chang
School of Information,Renmin University of China,Beijing 100872,China

Abstract　In recent years,code reuse attack has become a mainstream attack against binary programs. The code reuse attack such as ROP uses the instruction gadgets in the memory space to construct an instruction sequence that can realize specific functions and achieve malicious purposes. According to the basic principle of the code reuse attack,this paper proposes a defense method based on real-time function loading and unloading. More specifically,the method shrinks the code space by the dynamic loading/unloading,to reduce the attack surface and defend the code reuse. First,it extracts sufficient function information in the dependent libraries of the target program by static analysis,and uses this information in the form of replacement libraries. Second,it introduces real-time loading in the dynamic loader in Linux,and proposes an auto-triggerable and auto-restorable loading/unloading. In order to reduce the high overhead caused by frequent unloading,a randomized batch unloading mechanism is designed. Finally,experiments are carried out in a real environment to verify the effectiveness of the scheme against code reuse attacks,and the significance of the randomized unloading strategy is demonstrated.

Keywords　Code reuse attack,Real-time code loading and unloading,Return oriented programming,Dynamic link library,Randomized unloading

1　引言

随着数据执行防护[1]等机制的部署,早期严重威胁计算机安全的代码注入攻击已经被阻止。近年来,代码重用攻击(Code-reuse Attack)已经成为针对二进制软件的一种主流攻击方式。传统的防御方法假设程序内部的代码不会产生攻击行为,恶意行为均由外部引入。而代码重用攻击打破了这一假设,攻击者能够扫描目标可执行程序及其依赖的动态链接库,提取可以利用的指令片段(Gadgets)来达成恶意目标。

Return-into-libc攻击[2]作为最早的代码重用攻击,能够以非预期的方式调用函数。面向返回的编程(Return Oriented Programming,ROP)[3]攻击和面向跳转的编程(Jmp Oriented Programming,JOP)[4]攻击增强了攻击者执行任意功能的能力。研究人员还提出了一些其他先进的代码重用攻击技术[5-8]。代码重用攻击已被证明可以有效地破坏现实世界的程序[9],例如Adobe Reader曾经遭受过真实的pure ROP攻击[10]。此外,随着攻击链自动构造[11]等技术的推广,代码重用攻击的威胁日益增加。

面对威胁,研究者们提出了多种防御机制。控制流完整性方案[12-17]通过确保控制流传输的目标地址有效来减少代码重用攻击,地址随机化[18]通过随机化代码段的地址并阻止敌手预测目标gadget的普遍适用位置来防止代码重用攻击。

到稿日期:2022-05-11　返修日期:2022-07-23
基金项目:国家自然科学基金(U1836209)
This work was supported by the National Natural Science Foundation of China(U1836209).
通信作者:黄建军(hjj@ruc.edu.cn)

这些防御机制的有效性[19]已得到充分验证。然而,考虑到受害者进程的可执行代码依然存在于内存中,为攻击者提供了可乘之机,上述机制仍无法充分保护程序免受各类代码重用攻击的破坏。

受此启发,本文针对受害进程中存在的可执行代码,设计了新的方案,即通过缩小受害进程中可执行代码范围的方式来抵御代码重用攻击。考虑到代码重用攻击的实质是利用代码空间自身存在的代码发起攻击,只要代码空间内存在足够的指令片段,攻击者就有机会构造 gadget 链发起进攻。本文的想法是,通过实时装卸载内存中的函数,对代码空间进行持续的裁剪,保证代码空间中只有最小集合的、仅供当前运行使用的代码。值得注意的是,在动态链接库机制下,系统会加载所有链接库进入内存,远超程序运行所需。因此,本文工作的重点是对动态链接库进行裁剪,最大程度地降低内存空间的代码数量,让攻击者置于"巧妇难为无米之炊"的境地。

本文工作依靠代码空间裁剪展开防御,可以作为同类工作的良好补充,且本文工作直接面向二进制程序,无需受保护程序的源代码,或是重新编译程序,有着广泛的应用前景。

2 相关工作

国内外的研究者做了大量的相关工作,提出了多种防御机制。根据攻击的前置条件,代码重用攻击的防御方法大致可分为两类:以控制流完整性机制为代表的控制防御技术和以地址随机化为代表的内存信息保护方法。

第一类方法致力于判断程序执行流程的异常,如果发现程序执行过程与正常过程不同,则认为发生了攻击。控制流完整性(Control-Flow Integrity, CFI)[12]保护机制的提出是很自然的,既然代码重用攻击伴随着控制流的劫持,那么只要能够限制程序控制流在正确的控制流图中,代码重用攻击就无法进行。Mashtizadeh 等[13]提出了一种加密指针数据来增强 CFI 的方法,称之为 CCFI。该方案将指针细分,每次添加指针时加密相关标识信息并保存,在解引用时将其解密并进行比对。Denis-Courmont 等[14]使用了 ARMv8.3 指针身份验证扩展技术,在现成的处理器上提供了细粒度的硬件辅助 CFI。Hyerean 等[15]为了满足细粒度 CFI 方案对控制流图的精度需求,提出了一种基于索引的位向量方案——IBV-CFI,它为所有间接转移生成独立的位向量,从而实现高效的运行时检查。Qiang 等[16]考虑到性能问题,借助云环境的特点设计了一种上下文敏感的增量式 CFI——CloudCFI。在运行阶段,多个路径检查实例同时运行,且每个实例都能处理发生在其他实例上的控制劫持。CFI 技术还能用于加强物联网设备的安全性,Fu 等[17]使用消息认证码等加密技术,更针对性地维护基于 ARM 的物联网设备的控制流完整性。

另一类方法致力于保护内存中代码片段的信息,干扰攻击者获取内存中目标代码片段位置信息的能力,从而阻止攻击进行。ASLR(Address Space Layout Randomization)[18]是已经在实际中得到广泛应用的防御机制。它的主要思想是对堆、栈、共享库映射等线性内存区域布局进行随机化,使攻击者难以得知某些内存区域的具体位置,从而阻止攻击。隔离和加密也是保护内存信息的重要方式。Pomonis[20]提出了防止操作系统内核代码指针泄漏和损坏的解决方案——kR^X 和 kSplitStack 两个系统。前者使代码布局多样化,后者靠拆分堆栈来增强隔离机制。Mishra 等[21]充分利用英特尔的 SGX 可信执行技术,设计了一种划分主机和飞地的接口专业化工具,该工具能够提供强大的内存隔离和加密能力。

实际上,以上方法虽然优点突出,但还是忽略了可利用代码依然存在于内存中这个问题。本文提出的工作通过缩减代码利用范围,能够在攻击者突破上述机制后有效降低攻击链构建的可能,与上述机制形成良好补充,提高程序抵御代码重用攻击的能力。

3 方案实现

3.1 方案总述

本节根据代码重用攻击的特点,提出了一套完整的代码重用攻击防御方法,该方法能够有效阻止攻击者实践其精心构造的攻击链。本文方法的设计思路通用于现代计算机系统,但由于不同平台的实现机理不同,本文将围绕方案在 Linux 系统下的实现进行介绍。本节将描述整个防御系统的实现过程,重点阐述函数装卸载部分的实现。

本文方案的架构如图 1 所示,分为程序运行前的准备与程序运行过程中的干预。在程序运行前,需要通过静态分析,从目标程序及其依赖的动态链接库中获取需要的知识,这些知识将用于维护图 1 右侧的"实时装卸载"。在程序运行过程中,需要实现自动化的干预,使函数只在程序执行需要时被装载,并在执行结束后的合适时机被卸载,将对应函数代码移出内存。此外,在这两步之间,需要设计信息传递的渠道,使装卸载工作在运行过程中能自由使用静态分析获取的信息。

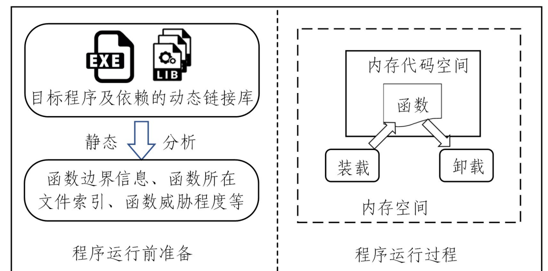

图 1 方案总体架构

Fig. 1 Overall framework

本文设计了动态替换库用于信息传递。动态替换库与动态链接库对应,实际上是清空了实际代码,只留下了少许必要信息的库文件。通过分别加载动态链接库与动态替换库中对应位置的函数内容,本文实现了函数装卸载,如图 2 所示。

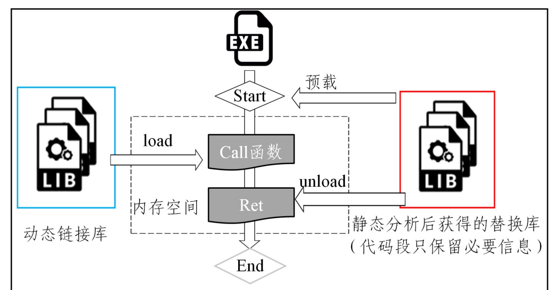

图 2 实时装卸载流程

Fig. 2 Real time loading and unloading process

图 2 中，受保护的程序正在运行，其左右分别为包含真实代码与无效代码的动态链接库和动态替换库。加载左框部分进内存，表示“装载”，即将真实代码填入对应的位置，保持程序正常执行。加载右框部分进入内容，表示“卸载”，即在程序调用依赖函数结束后，将不能被代码复用攻击利用的构造代码置入对应函数的内存位置，可以理解为执行结束后找机会清空该部分代码。程序启动时则直接加载动态替换库，表示其初始依赖代码全部为空，只有实际需要时才发生即时装载。

按照上述运行模式，程序运行过程中，链接库所在的内存空间只会在程序实际运行库函数时，让该函数的真实代码存在于内存中。代码区其余部分则以无法被代码重用攻击利用的形式存在，实现了预期的防御功能。

本文工作主要涉及 4 部分内容：1）函数信息提取；2）动态替换库构造；3）装卸载工作的实现；4）随机卸载策略的实现。以下将对各项工作展开详细介绍。

3.2 函数信息提取

从代码重用攻击的特点来看，以函数为裁剪单位最合适。因此，需要在程序运行前，对动态库文件中代码段的所有函数进行明确的界定，即确定函数在动态链接库中的起始偏移地址以及函数代码所占字节数等能够标定函数位置的信息。当动态链接库不止一个时，需要记录函数所在库的名称。

装卸载部分还需要一些特殊信息，用于防护方案部署后，维持程序正常运行。例如，在动态链接库中，部分函数在结束时会在 ret 指令上带上数字，自动删除通过栈传递的参数。为了在卸载时维持程序正常运行，本文工作必须提前获取、记录这些数字，因此信息提取环节需要对动态库进行更深入的分析。此外，在函数卸载部分，为了衡量一个函数处于内存中时，其代码内容为代码重用攻击的攻击者提供便利的威胁程度，需要计算每个函数的二进制码包含的特定 gadget 的数量，将其作为度量函数威胁程度的衡量指标。

本文通过 ldd 工具获知受保护程序依赖的动态库，通过 objdump 工具获取库文件的汇编表达，在此基础上对汇编文件进行静态分析。此外，本文使用 ROPgadget 工具获取目标库文件的所有 gadget，并记录每个函数的 gadget 数量。

3.3 动态替换库生成

本文需要在程序运行过程中对其进行一定程度的干预，从而实现实时的装卸载工作。本文的核心想法是将函数入口代码替换为调用（call）用于实时装卸载操作的 trampoline 函数的指令，自动实现对实际运行过程中函数调用行为的劫持。

在原始动态链接库的基础上，本文以函数为单位，将函数代码全部清空，换上联系 trampoline 函数的跳转指令、函数信息以及无效填充代码，构成动态替换库。

使用动态替换库，是同时考虑时间和空间开销的结果。若不使用替换库，则需要在每次装、卸载函数时，重新根据需要去查询当前函数信息，再按顺序逐字节写入覆盖的区域；同时，如果不把静态分析得到的函数信息存放在替换库中，就不得不额外开辟大量的内存空间来存放这些函数信息。

构造动态替换库的关键，是设计置于函数入口的“call trampoline function”这样的调用指令。trampoline 函数用于实现函数的实时装载，它的具体功能与实现将在 3.4 节进行说明。按照假设，非控制流劫持攻击情况下，函数应从入口处进入执行。有了入口处的转移指令，就能够在执行该函数前执行装载函数，将实际代码装载进对应的位置。

这条调用指令需要以字节码的形式写入替换库，为了实现正确的跳转，根据 call 指令字节码的构造规则，需要同时知道库函数和 trampoline 函数的地址，而这两个地址需要等待程序实际运行前，ld 装载器加载动态链接库时才能知晓。届时，只需将库函数地址和 trampoline 函数地址的差值，放在 e8（call）的后面，即可完成该库函数的入口调用指令。

然后是函数信息的传递，需要传递的信息已在 3.2 节中给出，这些信息必须由静态分析工作传递给后续的实时装卸载工作。计算好各部分所占字节，让这些内容紧跟 call 指令所占的 5 个字节，放入函数入口位置。其中，入口偏移 32 位地址占 4 字节；函数长度占 2 字节；函数所在库索引使用编号进行标记，占 1 字节；函数 gadget 威胁级别占 1 字节；ret 数字占 1 字节；加上 call 指令的 5 字节，合计需要填入 14 字节的内容。最后用无效指令将该函数位置填充满，替换示例如图 3 所示。

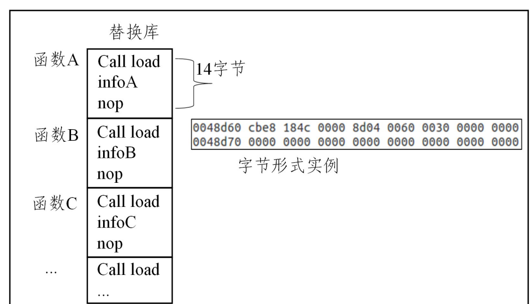

图 3 动态替换库示意图

Fig. 3 Schematic diagram of dynamic replacement library

3.4 实时装卸载

3.4.1 实时装载

做好信息获取与传递的准备后，便要正式使用这些函数信息，实现在程序运行时装卸载函数。实时装载的实现流程如图 4 所示。程序调用库函数时，通过替换库中函数入口的调用指令，自动改变程序控制流，进入实现装载功能的函数，完成真实代码的映射。整个流程可以分为：1）自动触发 trampoline 函数；2）实现装载功能；3）恢复执行。

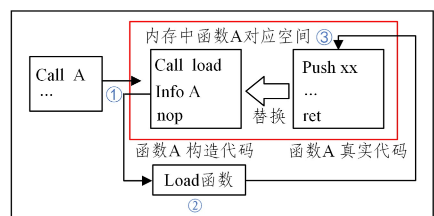

图 4 按需装载流程示意图

Fig. 4 Schematic diagram of on-demand loading process

图 4 中的 Load 函数分为两个部分：实现于 dl-trampoline. S 文件的汇编函数_dl_load 与实现于 dl-runtime. c 文件的 C 语言函数_my_load。控制流首先进入_dl_load 函数，通过 push 指令保存运行环境，并调用_my_load 函数，最后使用 pop 指令还原运行环境，并直接使用 jmp 指令，跳转到目标库函数的入口继续执行。

通过_my_load 函数实现函数装载功能。首先，通过函数

传参的方式,从栈上获取关键的地址信息。如图5所示,连续执行两次call指令后,栈上留下了返回地址和目标函数地址,其中,返回地址用于指导函数卸载后返回,目标函数地址用于指导函数信息的获取、标注真实代码的填充位置以及装载完成后的跳转位置。

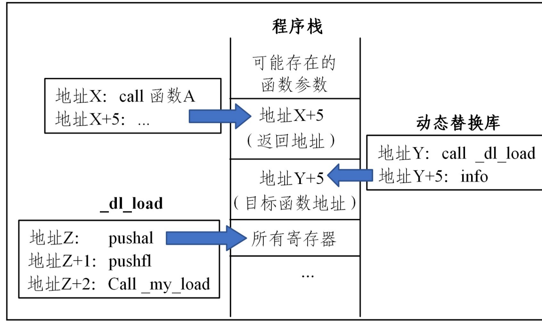

图5 装载前程序栈上栈帧示意图

Fig. 5 Schematic diagram of stack frame before loading

获取替换库中的函数信息后就能轻松打开对应链接库,读取真实函数代码,填入相应位置,完成装载工作。由于此时内存中替换库位置被覆盖,卸载时无法从此获得信息,因此需要开辟新的函数装载栈,按后进先出的方式存放所有装载进内存的库函数信息,供卸载流程使用。

为了让目标函数能被顺利执行,在装载结束前需要恢复部分栈帧,尤其是将esp指向返回地址,使目标函数能正确使用可能存在的参数。为了能够在函数执行结束后自动进入卸载流程,需要将图5中栈上返回地址位置修改为_dl_unload函数的地址。

3.4.2 实时卸载

卸载流程如图6所示。库函数通过ret指令退出时,控制流立刻进入负责卸载功能的_my_unload函数,用替换库对应位置的内容覆盖这块代码空间,然后返回函数调用前的位置恢复程序运行。整个流程可以分为:1)自动触发trampoline函数;2)实现卸载功能;3)恢复执行。

图6的unload函数分为_dl_unload函数和_my_unload函数。卸载过程可以简单理解为装载的逆过程,在3.1节中已经提到,卸载就是将替换库中函数对应的内容填回函数在内存中的位置。卸载操作需要注意的有两点,卸载只能从函数装载栈上获得必需的信息,卸载后的栈恢复需要考虑到函数自带的ret数字。

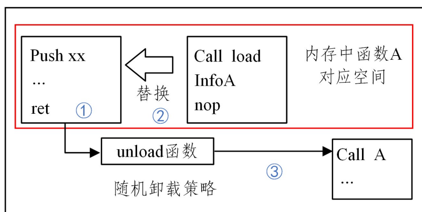

图6 卸载流程示意图

Fig. 6 Schematic diagram of unloading process

3.5 随机批量卸载策略

在实际运行过程中,部分库函数将被频繁、反复调用,如strcmp和memcpy等函数,这将极大程度地增加方案的时间开销,甚至增加到无法接受的地步。但若放弃卸载,内存中残留的函数代码将严格递增至程序依赖的所有库函数,无法满足本文最初阻止攻击者构建攻击链的基本要求。针对上述问题,本文提出取消实时卸载,进行随机化批量卸载的对策,该方法通过随机效果破坏攻击者利用内存残留代码作恶的稳定性,并基于批量卸载减小频繁装卸载带来的大量开销。

在内存中开辟残留函数表来记录未被卸载的函数信息。通过观察与测试,我们决定允许内存中最多存在20个被_my_load装载来的函数,若达到这个上限,则批量卸载其中一半的函数。由于无法跟踪未被卸载的函数,如果其中有正在被执行的函数,暴力卸载必然出错,因此,在批量卸载前,需要实现backtrace函数,回溯栈上记录的函数调用情况,与残留函数表进行比对,排除执行中的函数。

对于剩下的函数,本文选择最应被卸载的函数群进行卸载。函数的选择参考了操作系统常见的进程调度算法。首先,参考最高响应比优先调度算法,将函数在内存中的留存时间纳入考虑。残留函数表中为每个函数记录一个remain_time字段,其初始化为0,每次离开卸载函数前,增加所有残留函数表的记录中remain_time字段的值,表示其在内存中留存的单位时间增加。此外,参考最短进程优先调度算法,将函数gadget数量作为度量函数威胁程度的指标,更倾向于卸载威胁程度高的函数。在两个指标之外,本文还引入了随机数作为调控因素,防止批量卸载流程进入固定模式后,残留函数被攻击者轻易预测。

综上所述,将gadget数量、残留时间以及随机数,都规范化为0～9的数字,并分别乘以权重(设置为0.5,0.2,0.3),从而获得每个卸载候选函数的综合得分。根据分值进行排序,获得得分最高的10个函数。

最后是批量卸载部分,因为有了需要卸载的函数在残留函数表上的下标数组,接下来只需按照3.4.2节介绍的卸载方式对每个函数执行卸载操作,并把残留函数表对应位置置为空,就可以恢复执行环境,退出_my_unload函数。

3.6 原型系统实现

本文实现了一个原型系统。预处理部分,编写了约800行代码的独立C语言程序;运行部分,在glibc2-23的源码中加入了约100行的动态库加载C语言代码、约350行的实时装卸载操作C语言代码以及30行的trampoline函数汇编语言代码,重编译安装的ld装载器即可实现运行时装卸载的功能。

4 实验结果和分析

本节将在不同环境下进行对比实验,主要区别在于装载、卸载和随机策略的应用,环境设置如表1所列。

表1 实验环境描述

Table 1 Experimental environment description

	Setting description
Unprotected	Protection scheme is not deployed, and the program runs with unmodified dynamic loader
Load only	The dynamic library function is loaded only once during the initial execution without unloading
Unload totally	Once the dynamic loader is unloaded, the additional function is strictly executed
Unload randomly	Deploy random unloading strategy based on function loading

4.1 防护功能测试

4.1.1 防御能力测试

本节以包含栈溢出漏洞的程序 Crashmail 1.6 为例，将本研究的防护系统投入实战，直观地体现本文工作的现实价值。我们采用 Exploit-DB 网站提供的 ROP 攻击脚本，已经在无防护环境下完成了攻击。本节的实验设置如下：1）为 crashmail 1.6 软件部署本防护方案的仅装载环境，防御 ROP 攻击；2）对仅装载环境下的 crashmail 软件继续展开 ROP 攻击；3）部署严格卸载环境、随机卸载环境，并开展上述攻击。

首先使用 patchelf 命令改变测试环境，调整 ROP 攻击脚本后再次进行攻击，攻击失败。图 7 给出了攻击失败的原因，可以通过 gdb 工具查看栈溢出前的内存空间，对比两图可以发现，仅装载环境下，gadget 位置的函数都因不被需要而置空，攻击链因此失效。

```
(gdb) x/4x 0xB7EFDC8C
0xb7efdc8c <__lll_lock_wait_private+44>:       0x9066c35a
(gdb) x/4x 0xb7e2e07e
0xb7e2e07e <__ctype_get_mb_cur_max+30>: 0xe855c358
(gdb) x/4x 0xB7E75FAB
0xb7e75fab <_IO_remove_marker+43>:        0x66c30289
```

（a）Unprotected environment

```
(gdb) x/x 0xb7f13ecc
0xb7f13ecc <__lll_lock_wait_private+44>:       0x00000000
(gdb) x/x 0xb7e45eae
0xb7e45eae:     0x7be80000
(gdb) x/x 0xb7e8d1ab
0xb7e8d1ab <_IO_remove_marker+43>:        0x00000000
```

（b）Loading-only environment

图 7 查看相同位置内容

Fig. 7 View content in the same location

然后尝试攻击仅装载保护下的 crashmail。根据 gdb 调试结果，在栈溢出发生前，共有 78 个库函数被装载进内存。可以认为，这些被装载进内存且持续残留的函数代码，为攻击者提供了实现代码重用攻击的可能性。构建新的 gadget 链，在限制 78 个函数的条件下，想要简单重现之前的攻击过于困难，因此这里放宽了限制条件，作为与真实攻击者能力差距的补偿，我们使用了额外的本应被裁剪的部分函数中的 gadget。

最后部署两种实现了函数卸载的环境来防御上述攻击。实际上，因为上述攻击放宽了条件，所以在 78 个函数内选择的 gadget 只有“mov dword ptr [edx]，eax；mov eax，0xffffffff；ret”这一条，其地址 0xb7e3b664 位于__syscall_error 函数的[0xb7e3b650，0xb7e3b66f]范围内。在严格卸载的环境中，__syscall_error 函数在经过_my_load 装载后，立刻通过_my_unload 被卸载，在栈溢出时，该位置 gadget 不存在，防御成功。

而在随机卸载环境中，并不能保证在栈溢出前将该指令所在函数百分之百卸载出内存。重复运行 10 次攻击脚本，只有 6 次完成防御。当然，真实的攻击者只能在被_my_load 装载进内存的函数中寻找 gadget，更多相关函数将参与随机卸载，切断 gadget 链的概率将远超现在的 60%。

4.1.2 裁剪效果测试

有效 gadget 的减少量更能够说明本文工作减弱了攻击者使用这些 gadget 进行攻击的能力，我们设计了 AGR（Average Gadget Reduction）机制作为参考指标，AGR 的计算式如式（1）所示：

$$\frac{1}{n}\sum_{j=1}^{n}\left(1-\frac{T_j}{S}\right) \tag{1}$$

其中，S 表示程序依赖的所有库函数的 gadget 总数；T_j 表示第 j 次观察时，内存中库文件所有内容的 gadget 总数。在程序运行时取 n 次 T，最后计算平均数。

我们选择以下 4 个程序，分别在仅装载环境、严格卸载环境和随机卸载环境下进行测试，得到的结果如表 2 所列。

表 2 不同命令下平均 gadget 减少程度

Table 2 Average gadget reduction with different commands

（单位：%）

Command	Load only	Unload totally	Unload randomly
Touch xx	77.2	98.3	94.5
Cat xx	76.1	98.2	94.3
Date	85.2	98.9	96.4
Crashmail	84.7	98.5	95.6

本文方案能够大幅度削减内存中可供代码重用攻击的 gadget 数量。仅装载的情况和 4.1.1 节的实验测试效果一致，虽然能有效减少内存中的 gadget 数量，但仍存在防护上的缺陷。而严格卸载能够达到超过 98% 的极高的 gadget 削减效果，随机卸载的削减效果同样十分明显，这证明采用了卸载机制的防护方案的安全功能十分强劲。

4.2 性能开销测试

对方案的性能开销进行测试，包括实际程序运行开销测试与构造运行环境的压力测试。

首先，同样对比 4.1.2 节中使用的 4 个命令，分别在仅装载环境、严格卸载环境和随机卸载环境中计算平均运行时间。使用 gettimeofday 函数记录每个命令运行百遍的平均时间，得到如表 3 所列的结果。

如表 3 所列，严格开展卸载工作，相比仅装载环境增加了大量时间开销；而在引入随机批量卸载策略后，虽然相比仅装载环境依然引入了一定开销，但相比完全卸载环境，性能有了 30%～50% 的提升。可以看出，进行随机批量卸载能够有效缓解严格卸载带来的性能开销，具有重要意义。

表 3 不同环境下命令平均用时

Table 3 Average command time in different environments

	Load only/ms	Unload totally/ms	Unload randomly/ms	Performance improvement/%
Touch xx	15.82	24.45	19.35	32.1
Cat xx	19.51	28.32	22.13	31.7
Date	1.57	2.63	1.82	51.6
Crashmail	1.71	2.94	2.11	48.5

为了对系统进行压力测试，分别编写了简单函数单独调用、复杂函数单独调用、I/O 函数调用、函数混杂调用 4 种情况的测试样例进行测试，记录并计算程序运行的平均时间，获得实际运行开销。将运行时间以仅装载为标准进行基准化，获得如表 4 所列的结果。

表 4 压力测试基准化结果

Table 4 Benchmarking results of stress test

	Load only	Unload totally	Unload randomly	Performance improvement/%
Simple call	1	1.83	1.02	97.6
Complex call	1	11.56	2.26	88.1
File operation	1	19.10	3.11	88.3
Hybrid execution	1	24.29	5.32	81.5

在大量依赖库函数的情况下,随机卸载相比严格卸载有着极高的性能提升。在每个循环仅调用一个简单函数的极端情况下,甚至能减少97.6%的性能开销。

另外,实验结果表明,以上测试中本文工作相比无防护的情况增加了10倍以上的时间开销。其原因是本文工作为了预先清理内存空间,在程序加载时重写了链接库位置的所有内容,导致基础开销过大。未来将继续完善,寻找更合适的替换库装载方法。

结束语 本文针对代码重用攻击的基本原理,根据其攻击过程必须依赖内存空间中,尤其是动态链接库中的代码片段的特点,提出了基于实时装卸载函数代码的防御方法。首先以静态分析的方式获取足量的函数信息,并以替换库的形式使用这些信息。通过修改Linux系统下的动态装载器,在装载过程中引入用于实时装载函数、随机批量卸载函数的操作,实现了本文方案的核心工作。最后,在真实环境中开展实验,验证了本文方法防御ROP攻击以及随机卸载策略平衡开销与功能的有效性。但是本文工作相比未受保护的情况,性能开销依然严重,缺少泛用性,提高性能将是未来研究的重点。

参 考 文 献

[1] The PaX Team. Pax: non-executable pages design & implementation[EB/OL]. https://pax.grsecurity.net/docs/noexec.txt.

[2] COntex. Bypassing non-executable-stack during exploitation using return-to-libc[EB/OL]. http://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf.

[3] SHACHAM H. The geometry of innocent flesh on the bone: Returninto-libc without function calls(on the x86)[C]//Proceedings of the ACM Conference on Computer and Communications Security(CCS'07). 2007:552-561.

[4] BLETSCH T, JIANG X, FREH V, et al. Jump Oriented Programming: A New Class of Code-Reuse[C]//Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security(ASIACCS '11). 2011:30-40.

[5] SNOW K Z, MONROSE F, DAVI L, et al. Just-in-time code reuse: On the effective-ness of fine-grained address space layout randomization[C]//IEEE. 2013:574-588.

[6] VEEN V V D, ANDRIESSE D, STAMATOGIANNAKIS M, et al. The dynamics of innocent flesh on the bone: Code reuse ten years later[C]//the 2017 ACM SIGSAC Conference. ACM, 2017:1675-1689.

[7] SADEGHI A, NIKSEFAT S, ROSTAMIPOUR M. Pure-call oriented programming(pcop): chaining the gadgets using call instructions[J]. Journal of Computer Virology and Hacking Techniques, 2018, 14(2):139-156.

[8] HU H, SHINDE S, ADRIAN S, et al. Data-oriented programming: On the expressiveness of non-control data attacks[C]//2016 IEEE Symposium on Security and Privacy(SP). 2016:969-986.

[9] RAINS T, MILLER M, WESTON D. Exploitation trends: From potential risk to actual risk[C]//RSA Conference. 2015.

[10] LI X A, SZOR P. Emerging "stack pivoting" exploits bypass common security[EB/OL]. https://securingtomorrow.mcafee.com/other-blogs/mcafeelabs/emerging-stack-pivoting-exploits-bypass-common-security/.

[11] SCHLOEGEL M, BLAZYTKO T, BASLER J, et al. Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains[C]//ESORICS. 2021.

[12] ABADI M, BUDIU M, ERLINGSSON Ú, et al. Control-flow integrity[C]//Proceedings of the 12th ACM Conference on Computer and Communications Security. ACM, 2005:340-353.

[13] MASHTIZADEH A J, BITTAU A, BONEH D, et al. CCFI: Cryptographically enforced control flow integrity[C]//Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. ACM, NY, USA, 2015:941-951.

[14] DENIS-COURMONT R, LILJESTRAND H, CHINEA C, et al. Camouflage: Hardware-assisted CFI for the ARM Linux kernel[C]//2020 57th ACM/IEEE Design Automation Conference (DAC). 2020:1-6.

[15] HYEREAN J, MOON C P, DONG H L. IBV-CFI: Efficient fine-grained control-flow integrity preserving CFG precision[J]. Computers & Security, 2020, 94:101828.

[16] QIANG W, HUANG Y, JIN H, et al. CloudCFI: Context-Sensitive and Incremental CFI in the Cloud Environment[J]. In IEEE Transactions on Cloud Computing, 2021, 9(3):938-957.

[17] FU A M, DING W J, KUANG B Y, et al. FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices[J]. Computers & Security, 2022, 116:102666.

[18] PAX Team. Address Space Layout Randomization [EB/OL]. http://pax.grsecurity.net/docs/aslr.txt.

[19] BLETSCH T. Code-reuse attacks: New frontiers and defenses [J/OL]. https://repository.lib.ncsu.edu/bitstream/handle/1840.16/6698/etd.pdf;jsessionid=DF7DE65EDFDB8C2D7110D1CA2BB6DEAC?sequence=1.

[20] POMONIS M. Preventing Code Reuse Attacks On Modern Operating Systems[M]. Columbia: Columbia University, 2020.

[21] MISHRA S, POLYCHRONAKIS M. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks[C]//Sixteenth European Conference on Computer Systems(EuroSys'21). 2021.

HOU Shang-wen, born in 1997, postgraduate, is a member of China Computer Federation. His main research interests include software security analysis and so on.

HUANG Jian-jun, born in 1986, Ph.D, assistant professor, master supervisor, is a member of China Computer Federation. His main research interests include program analysis, vulnerability detection and mobile security.

(责任编辑:何杨)