KMBox：基于 Linux 内核改造的进程异构冗余执行系统

马博林[1], 张 铮[1], 邵昱文[1], 李秉政[1], 潘传幸[1], 蒋 鹏[2], 邬江兴[1]

[1]中国人民解放军战略支援部队信息工程大学 郑州 中国 450001
[2]网络通信与安全紫金山实验室 南京 中国 211100

摘要 随机化技术防御进程控制流劫持攻击, 是建立在攻击者无法了解当前内存地址空间布局的基础之上, 但是, 攻击者可以利用内存信息泄露绕过随机化防御获得 gadget 地址, 向程序注入由 gadget 地址构造的 payload, 继续实施控制流劫持攻击, 窃取敏感数据并夺取或破坏执行软件的系统。目前, 异构冗余执行系统是解决该问题的方法之一, 基本思想是同一程序运行多个多样化进程, 同时处理等效的程序输入。随机化技术使冗余的进程对恶意输入做出不同的输出, 同时正常功能不受影响。近年来, 一些符合上述描述的系统已经被提出, 分析进程异构冗余执行系统的表决设计可以发现, 基于 ptrace 的实现方法会引入大量的上下文切换, 影响系统的执行效率。率先直接修改内核设计出一种进程异构冗余执行系统, 表决过程完全在内核中完成, 冗余的进程独立地采用内存地址空间随机化技术, 构建相互异构的内存地址空间布局, 在与内存信息泄露相关的系统调用处进行表决, 发现泄露信息不一致, 阻断进程控制流劫持攻击。即使攻击者跳过内存信息泄露进行漏洞利用, 异构内存空间布局也使得注入由 gadget 地址构造的 payload 无法同时在冗余的进程中有效, 阻断进程控制流劫持攻击。实现了原型系统 KMBox, 实验证明该系统能够有效抵御进程控制流劫持攻击, 性能相较于基于 ptrace 的进程异构冗余执行系统有所提高。

关键词 控制流劫持攻击; 异构冗余执行系统; 随机化
中图法分类号 TP309 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.01.02

KMBox: Linux Kernel-based Heterogeneous Redundant Execution System Designed for Processes

MA Bolin[1], ZHANG Zheng[1], SHAO Yuwen[1], LI Bingzheng[1], PAN Chuanxing[1], JIANG Peng[2], WU Jiangxing[1]

[1]PLA Information Engineering University, Zhengzhou 450001, China
[2]Purple Mountain Laboratories, Nanjing 211100, China

Abstract The randomization technology to defeat process control-flow hijacking attacks is based on the fact that attackers are unable to know about the memory address space layout. However, attackers can exploit information disclosure to bypass the randomization defense and obtain gadget address. So that attackers can still launch process control-flow hijacking attacks to steal sensitive data and to seize or disrupt the system on which the software is executed. At present, the heterogeneous redundant execution system is one of the methods to solve this problem. The underlying idea of heterogeneous redundant execution system is to run several diversified processes of the same program, side by side on equivalent program inputs. The randomization techniques make the redundant processes respond differently to malicious inputs, while leaving the behavior under normal operating conditions unaffected. In recent years, some systems have been proposed that match the above description. The voting designs of heterogeneous redundant execution systems for processes are analyzed, the implementation based on ptrace introduces a large number of context switches, which affects the execution efficiency of the system. It is the first to design a kernel-based heterogeneous redundant execution system which directly modifies the kernel for processes. The redundant processes adopt memory address space layout randomization independently, besides, the system calls related information disclosure will be voted to find abnormality and to defeat process control-flow hijacking attacks. Even if attackers skip information disclosure to exploit other vulnerabilities, the heterogeneous memory address space layouts prevent the injected payload from being effective in redundant processes at the same time, which can also defeat process control-flow hijacking attacks. The prototype system KMBox is implemented and experiments show that the prototype can effectively defeat process control-flow hijacking attacks. Comparative performance tests show that KMBox is better than the heterogeneous redundant execution system based on ptrace.

Key words control-flow hijacking attack; heterogeneous redundant execution system; randomization

---

通讯作者： 张铮, 博士, 副教授, Email: ponyzhang@126.com。
本课题得到国家自然科学基金项目(No. 61521003)与国家重点研发计划项目(No. 2018YF0804003)资助。
收稿日期: 2021-10-10; 修改日期: 2021-12-17; 定稿日期: 2022-11-04

1 引言

近年来, 关于软件漏洞利用的攻防技术在博弈发展, 数据执行保护(Data Execution protection, DEP)[1]、栈溢出保护(Stack Smashing protection, SSP)[2]和内存地址空间布局随机化(Address Space Layout Randomization, ASLR)[3]等可以阻止大部分控制流劫持攻击的发生。这种情况下, 攻击者采取一些高级的攻击技术[4-13], 如 return-to-libc[6]、ROP[7-9]、JOP[10-12]和 SROP[13]等, 利用内存空间中以 ret、call、jmp 指令结束的已有代码片段(称为 gadget)[14]或组合多个 gadget 来获得程序的执行权限, 能够绕过 DEP、SSP 等防御机制, 发起控制流劫持攻击[15-16]。

与此同时, 应用最为广泛的 ASLR 技术暴露出以下问题: (1)虽然代码基地址经过了随机化处理, 但是通过相对偏移可以由一个指针获得所有 gadget 的地址; (2)ASLR 的有效性是建立在内存地址空间布局的保密性之上, 而内存泄露漏洞的普遍存在给 ASLR 带来了挑战。无论是代码注入攻击, 还是通过内存泄露发起的代码复用攻击, 攻击者的目标是劫持程序的控制流。

当前, 软件异构冗余执行是有效防御控制流劫持攻击的方法之一。软件冗余执行[17]是利用故障发生的时空随机性质, 通过表决比较软件副本的执行结果, 对软硬件故障引起的计算错误实现容错处理, 双模冗余的结构如图 1 所示。软件异构冗余执行将软件冗余执行和软件多样化[18]结合, 解决软件漏洞不可避免和同质化安全问题[19-21], 软件副本之间采用异构设计, 在结构和实现上存在着不同, 甚至是完全不同的技术路线, 这就保证了软件副本之间存在相同安全漏洞的概率是极低的。因此, 当网络攻击发生时, 软件副本产生一致的攻击成功结果通过表决是极小概率事件, 软件异构冗余执行从机理上既可感知随机性故障, 也可抵抗利用软件漏洞的网络攻击。

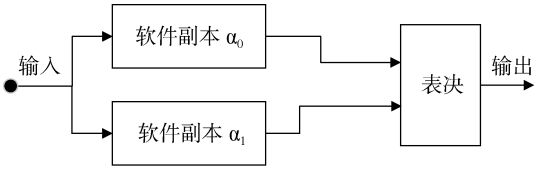

图 1 双模冗余结构

Figure 1 Dual modular redundancy structure

本文借鉴以 ASLR 为代表的随机化防御方法和软件异构冗余执行的技术思路, 设计和实现了一种基于 Linux 内核改造的进程异构冗余执行原型系统——KMBox。本文剩余部分的组织如下: 第 2 章阐述进程异构冗余执行系统防御控制流劫持攻击的原理并总结相关工作为本文的设计方法作背景说明; 第 3 章介绍 KMBox 的设计和实现; 第 4 章进行实验与分析; 第 5 章探讨 KMBox 的局限性和改进思路; 第 6 章进行总结。

2 相关工作

2.1 防御原理

攻击者在实施控制流劫持攻击时需要两个关键步骤, 一是通过信息泄露获取 gadget 内存地址; 二是基于已获得的 gadget 地址, 构造 payload 将栈中的返回地址覆盖为 gadget 地址, 从而劫持控制流。如图 2 所示, 程序中内存对象 A 存在溢出, 攻击者首先通过具有输出功能的系统调用, 例如 write、sendto 等, 获得 func2 的内存地址 0x5008a00; 再构造 payload= A+B+RBP+0x5008a00 进行栈溢出, 来覆盖程序的返回地址指向 func2, 从而劫持控制流。而在开启 Canary 保护机制的情况下, 攻击者还需要泄露 Canary 的值, 加入到 payload 中以通过检查。

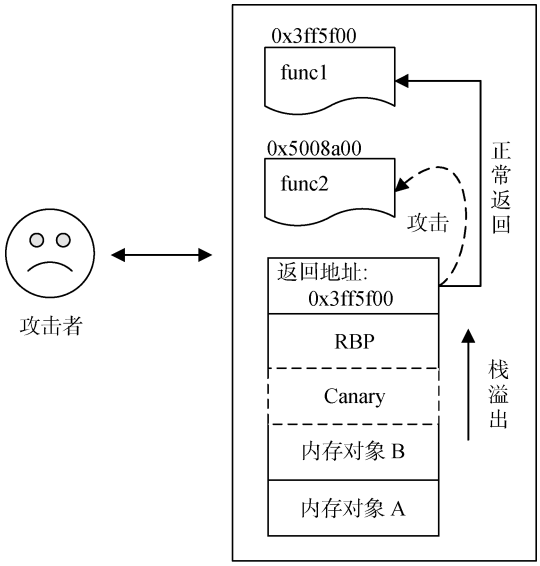

图 2 控制流劫持攻击示意图

Figure 2 Schematic diagram of control-flow hijacking attacks

KMBox 利用异构冗余执行架构和以 ASLR 为代表的随机化防御方法, 在用户空间中构造冗余的进程, 同时改造内核能够支撑冗余进程的启动、主从关系注册、内存地址空间异构、非异构资源共享、系统调用表决及结束等关键环节。图 2 所示的程序利用 KMBox 加载运行, 且操作系统开启 ASLR、程序

编译时开启 PIE 的情况下, 冗余进程的内存布局如图 3 所示, 由于冗余进程具有相互异构的内存地址空间布局, 一方面冗余进程的 gadget(func2)地址不一致, 无法通过表决机制向攻击者泄露, 另一方面使得注入由 gadget 地址构造的 payload 无法同时在冗余进程中有效。

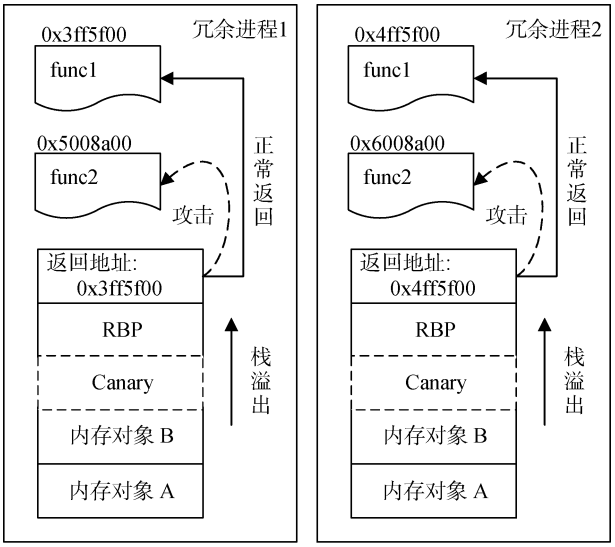

图 3 冗余进程的异构内存布局

Figure 3 Heterogeneous memory layouts of redundant processes

2.2 表决设计方案分析

有关进程异构冗余执行系统的研究成果多集中在国外的多变体技术和国内的拟态防御技术, 其中, 多变体技术以美国加利福尼亚大学研究团队[22-24]和比利时根特大学研究团队[25-28]的成果为代表, 拟态防御技术则是以战略支援部队信息工程大学研究团队[29-32]的成果为代表。无论是多变体技术还是拟态防御技术, 表决的发生位置和实现方法在进程异构冗余执行中起到了至关重要的作用, 表 1 总结了代表性进程异构冗余执行系统的表决设计方案。

表 1 进程异构冗余执行系统的表决设计方案

Table 1 Voting designs of heterogeneous redundant execution systems for processes

原型系统	发生位置	实现方法
Orchestra[22]	CP/US	ptrace
GHUMVEE[25]	CP/US	ptrace
DCL[26]	CP/US	ptrace
MimicBox[31]	CP/US	ptrace
ReMon[27]	CP/US+ IP/KS	ptrace+内核
BUDDY[33]	IP/KS	内核

表决在进程异构冗余执行系统中的发生位置可以按两种方式进行分类, 一是根据表决是否发生在代表软件副本的冗余进程内, 表决发生在冗余进程内的方式称为进程内(In-Process, IP)表决, 而表决由单独的进程去完成, 发生在冗余进程外的方式称为跨进程(Cross-Process, CP)表决。二是根据表决发生在内核空间(Kernel Space, KS)还是用户空间(User Space, US)中进行分类, 因此表决发生的可能位置如图 4 所示。

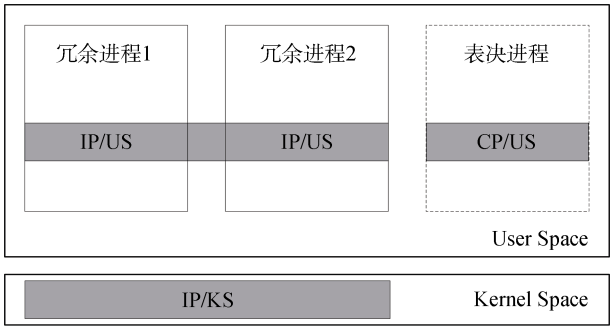

图 4 表决可能的发生位置

Figure 4 Possible placements of voting

表决的常用实现方法也可以按两种方式进行分类, 一是较为常用的基于 ptrace 实现方法, 采用该方法时表决发生位置为 CP/US, 实现的原型系统有 Orchestra[22]、GHUMVEE[25]、DCL[26]、MimicBox[31], ptrace 是操作系统自带的进程跟踪调试工具, 虽然使得原型系统易开发部署, 但是会引入大量的上下文切换, 影响系统的执行效率。二是在内核实现, 相对基于 ptrace 的实现方法, 避免了上下文切换, 减少系统的性能损耗, 实现的原型系统有 BUDDY[33]和 ReMon[27], 其中 ReMon 在实现难易程度和性能损耗之间进行了平衡, 将基于 ptrace 的实现方法和内核实现方法相结合。当前采用内核实现的原型系统均是通过添加内核模块的方法, 虽然与内核相对解耦, 但是该方法需要修改 sys_call_table 拦截系统调用, 过程中会破坏内存的写保护机制。

综合以上分析, 本文率先选择直接改造内核的方式创新地实现进程异构冗余执行原型系统——KMBox, 不仅避免了大量上下文切换, 还不会破坏操作系统自身的内存写保护机制。防御有效性测试和性能对比测试表明, KMBox 能够有效抵御进程控制流劫持攻击, 并且性能要优于基于 ptrace 在用户空间拦截、表决的进程异构冗余执行系统。

3 KMBox 原型系统

进程通过系统调用由用户空间进入内核空间, 因此实现 KMBox 需要对以下关键过程进行设计:

(1)如何启动功能等价的冗余进程; (2)如何将冗余进程建立起主从关系; (3)如何实现冗余进程的内存地址空间异构; (4)如何共享和同步非异构资源; (5)如何对系统调用表决; (6)如何结束主从关系的冗余进程。

3.1 如何启动功能等价的冗余进程

KMBox 在用户空间设置启动模块, 若要程序以异构冗余的形式执行, 使用该启动模块加载目标程序即可, 其他不采用该模块而正常启动的进程不受影响。KMBox 启动模块内两次调用 fork, 创建冗余的进程并返回 *master_pid* 和 *slave_pid*, 每个进程在内核空间中具有独立的 task_struct。为保证等价进程的表决和非异构资源共享能够准确地在主从进程之间开展, KMBox 添加用于注册主从关系的系统调用 kmb_register, *master_pid* 和 *slave_pid* 作为该系统调用的参数, 为冗余进程建立起主从关系(详见 3.2 节)。程序在未采用 KMBox 启动模块加载时, 完成进程创建后使用 execve 系统调用加载可执行程序, 向 execve 中传入可执行程序的全路径名 *filename*、参数 *argv*、环境变量 *envp*, execve 找到相应的可执行文件, 检查可执行文件格式, 根据可执行文件的头部修改 task_struct 和堆栈指针, 进程完成可执行程序的加载后开始执行。KMBox 启动模块加载程序时, 主从进程同样需要完成加载可执行程序的过程, 由于主从进程共享非异构资源(详见 3.3 节), 因此改造添加 kmb_execve 系统调用, 在 execve 基础上增加主从进程间的“等待-唤醒”机制, KMBox 启动冗余进程的过程如图 5 所示。

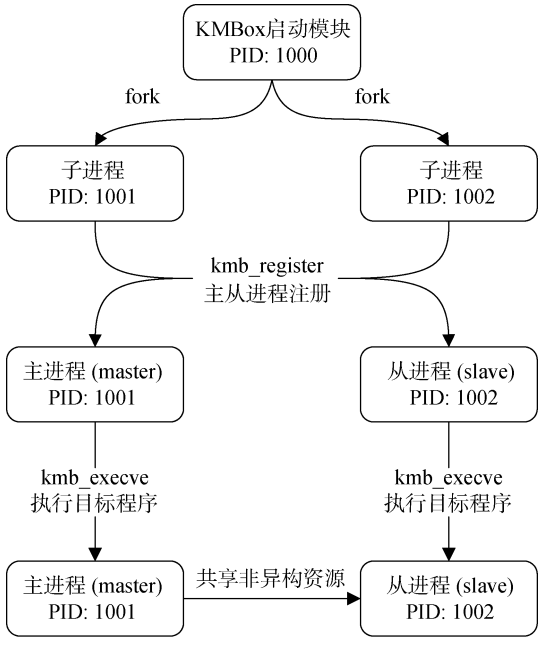

图 5 KMBox 启动冗余进程的过程

Figure 5 Process of KMBox starting the redundant processes

主从进程通过 kmb_execve 加载目标程序的主要过程可总结为: (1)主进程开始 kmb_execve 时, 从进程处于等待状态; (2)主进程在 start_thread 前, 进入等待状态, 并唤醒从进程开始 kmb_execve 加载可执行程序; (3)从进程在 start_thread 前, 将需要与主进程共享的资源指向主进程所对应的内存空间, 唤醒主进程, 主从进程从各自新的 EIP/RIP 处开始独立运行, 流程如图 7 所示。“等待-唤醒”机制只会增加进程加载可执行程序过程的时间消耗, 不会影响进程加载程序后的运行过程。

3.2 如何将冗余进程建立起主从关系

KMBox 启动模块内两次调用 fork 生成冗余进程, 为建立起冗余进程的主从关系, KMBox 在内核空间为主从进程组添加设置 mas_task_struct 数据结构, 如图 6 所示, 由 kmb_register 将冗余进程注册为主从进程组。

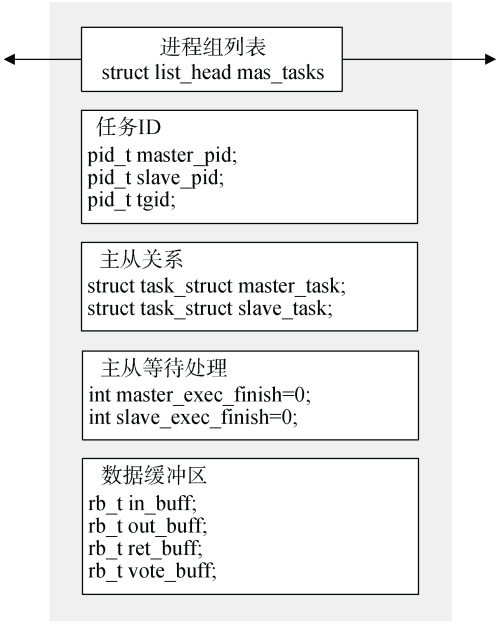

图 6 主从进程组的 mas_task_struct 数据结构

Figure 6 The mas_task_struct structure of the master-slave processes group

进程组列表将所有的 mas_task_struct 采用双向链表串起来进行管理, 每次 kmb_register 为进程组创建 mas_task_struct 时, 需要将其插入到进程组列表中; 进程组销毁时, 需要从进程组列表中删除节点。任务 ID 中记录主进程 pid、从进程 pid 和启动模块 pid。主从关系中的 master_task 指向主进程的 task_struct, slave_task 指向从进程的 task_struct。主从等待处理中的标志位 master_exec_finish 和 slave_exec_finish 用于主从进程间的“等待-唤醒”机制。数据缓冲区用于主从进程间表决数据的传递和非异构资源的共

享同步, 具有输入数据缓冲区 in_buff、输出数据缓冲区 out_buff、返回数据缓冲区 ret_buff、表决数据缓冲区 vote_buff 这四种类型的数据缓冲区。

3.3 如何实现冗余进程的内存地址空间异构

KMBox 基于 Linux 操作系统多种有关内存地址空间布局随机化的技术实现冗余进程异构, 可以利用 ASLR 机制, 根据等级设置实现栈基地址、堆基地址、共享库基地址、mmap 基地址的随机化, 同时还可以利用编译器的代码位置无关 PIE 选项, 在程序编译过程中为进程开启 PIE 功能, 使得冗余进程在两次加载可执行程序时会将代码段.text、初始化数据段.data、未初始化数据段.bss 的地址进行随机化, 除此之外, 编译器还可以开启栈保护机制 Canary, 表 2 展示了多种可利用的随机化方法。

综合以上随机化方法, KMBox 可以实现冗余进程的内存地址空间异构, 异构性集中表现在 task_struct 中用于管理进程整个虚拟地址空间的 mm_struct 数据结构, task_struct 的关键数据结构如图 8 所示。当然, 操作系统提供的随机化方法无论对于通过 KMBox 启动模块加载后生成的主从进程, 还是其它正常启动的进程, 作用都是同样的, 不会对未采用 KMBox 启动模块加载的进程产生影响。

3.4 如何共享和同步非异构资源

KMBox 实现了冗余进程的内存地址空间异构, 但为了保证冗余操作不影响程序的正确功能, 需要对冗余进程的非异构资源进行共享和同步, 如图 8 中 fs_struct 和 files_struct 数据结构需要设置共享和同步, 否则有关文件系统和文件的操作会被执行两次, 影响程序正确性。

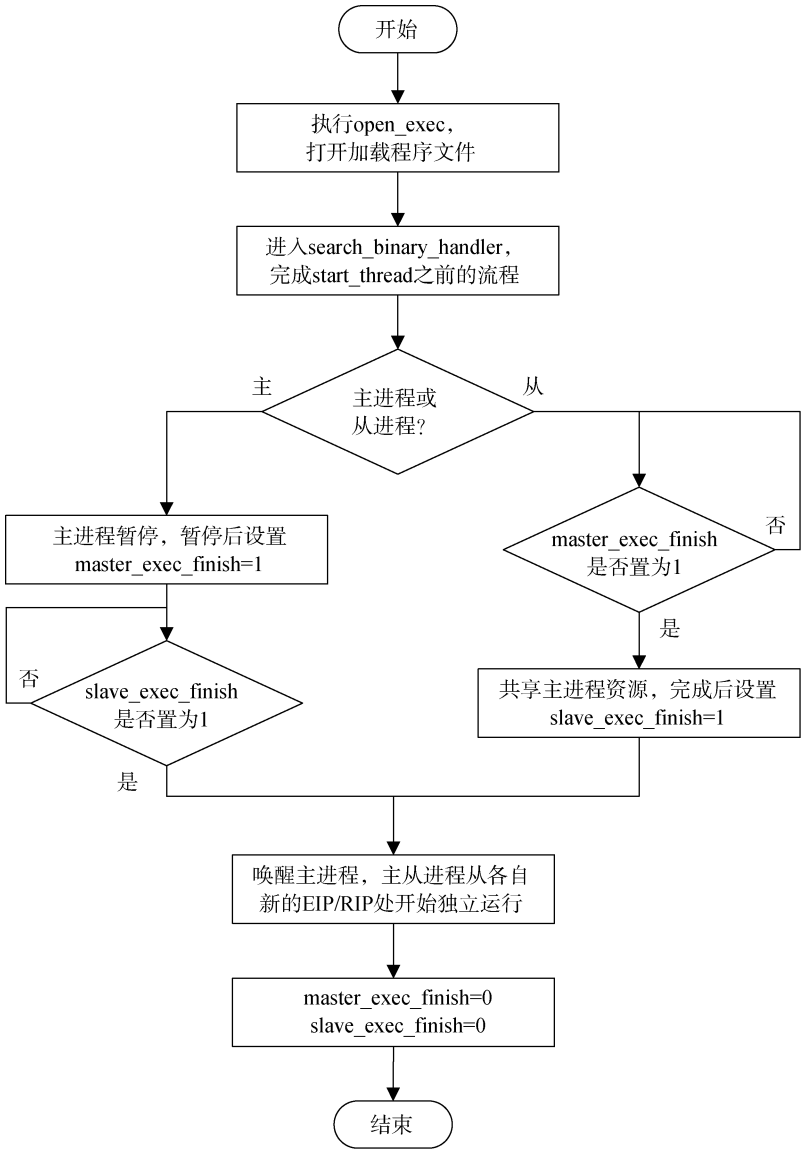

图 7 主从进程通过 kmb_execve 加载目标程序的流程

Figure 7 Process of the master-slave processes loading the target program through kmb_execve

表 2 Linux 操作系统的随机化方法

Table 2 Randomization methods of Linux OS

随机化方法	随机化对象	作用方式	作用时机
ASLR	等级 1: 栈基地址、共享库、mmap 基地址 等级 2: 在等级 1 的基础上, 增加堆基地址	操作系统设置	程序加载进入内存运行时
PIE	代码段.text、初始化数据段.data、未初始化数据段.bss	编译器指定编译指示	程序编译过程
Canary	高于局部变量, 低于 EBP/RBP	编译器指定编译指示	程序编译过程

表 3 部分与文件描述符表索引 fd 相关的系统调用

Table 3 Part of the system calls related to file descriptor fd

系统调用	功能说明	操作类型
open	打开文件, 产生文件描述符表索引 fd	设置/写类型
write	写入文件内容	设置/写类型且需要表决详见 3.5 节
read	读取文件内容	查询/读类型
close	关闭指定文件	设置/写类型
fsync	将文件数据由系统缓冲区写回磁盘	设置/写类型
flock	将文件进行锁定或解锁	设置/写类型
fcntl	按照指令操作文件的特性	设置/写类型
dup	复制 fd 所指的文件描述符表	设置/写类型

首先, 在 3.1 节中阐述了 kmb_execve 中增加主从进程间的“等待-唤醒”机制, 从进程复制主进程的相关数据结构 fs_struct 和 files_struct, 指向同一内存空间。其次, 对于共享资源的操作, 根据相关系统调用的功能进行选择性地改造: 对设置或写操作类型的系统调用进行改造, 首先判断当前进程是否为主从进程组成员, 若不是则按系统调用原操作执行, 不会影响未采用 KMBox 启动模块加载的进程; 若是则更进一步地判断是主进程还是从进程, 系统调用功能由主进程完成, 从进程复制主进程的返回结果, 由 mas_task_struct 中的数据缓冲区来传递, 源代码 1 和源代码 2 分别展示了 write 系统调用的改造前后代码, 不仅实现了主从进程的同步, 还增加了对参数的表决过程。查询或读操作类型的系统调用不需要区分进程类型, 由各自进程独立完成, 不需要对其进行改造, 同样也不会影响未采用 KMBox 启动模块加载的进程。表 3 展示了部分与 files_struct 数据结构中文件描述符表索引 fd 相关的系统调用。

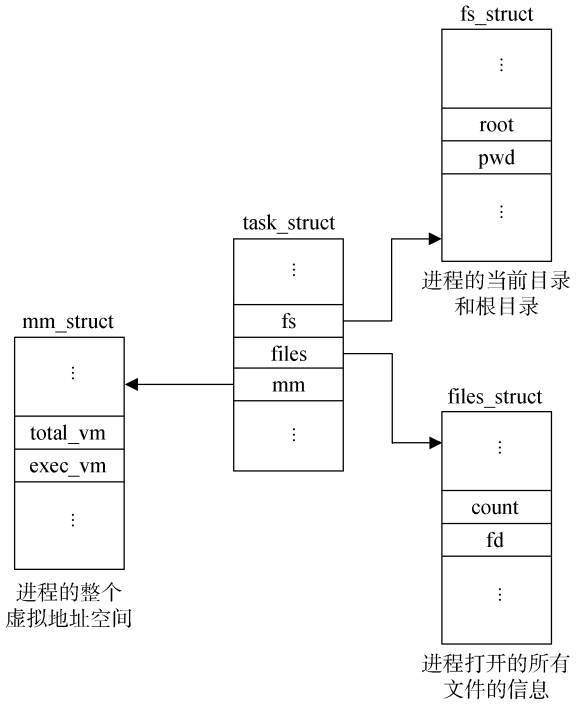

图 8 task_struct 的关键数据结构

Figure 8 Key data structure of task_struct

3.5 如何对系统调用表决

KMBox 的表决机制设置在虚拟内存空间的数据由内核空间向用户空间流出时, 总结得知攻击者在利用内存信息泄露实施攻击时, 都离不开对 write/read 系统调用的使用, 通过 write 来获取 gadget 地址, 然后构造有效的 payload, 进而使用 read 执行 payload 来劫持程序的控制流[15,34]。因此, 当冗余进程发生 write 系统调用, 在内核空间的执行过程中对参数 *buf* 进行表决, 当表决不一致时, KMBox 结束冗余进程, 阻止 gadget 地址泄露。

源代码 1 改造前的 write 系统调用定义

```
1)SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count)
2){
3)    return ksys_write(fd, buf, count);
4)}
```

内核文件 fs/read_write.c 中 write 系统调用的定义如源代码 1 所示, 为了对 write 系统调用进行表决, KMBox 重新定义了 write 系统调用的执行过程, 如源代码 2 所示: 首先判断当前进程是否为主从进程组成员, 若不是则按系统调用原操作执行, 不会影响未采用 KMBox 启动模块加载的进程, 若是则判断当

前进程为主进程还是从进程。

若是主进程, 则对 *buf* 进行表决, 表决过程中在表决数据缓冲区中读取从进程的 *buf* 数据, 与主进程的 *buf* 进行比对, 当表决不一致时结束当前进程, 当表决一致时进入 ksys_write 中执行, 同时将 ksys_ write 的返回结果存入返回数据缓冲区中, 用于主进程向从进程同步, 最后 write 返回 ksys_write 结果, 完成 write 系统调用; 若是从进程, 则向表决数据缓冲区中存入从进程的 *buf* 数据, 由于 write 属于设置或写操作类型的系统调用, 从进程不再执行 ksys_write, 读取返回数据缓冲区中主进程的返回结果, 将其返回并完成 write 系统调用。

源代码 2 改造后的 write 系统调用定义

```
1)SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count)
2){
3)    size_t ret;
4)
5)    if(! is_redundancy (current)){
6)          ret = ksys_write(fd, buf, count);
7)          return ret;
8)    } else if(is_master(current)){
9)          master_vote(buf, count, vote_buff);
10)         ret = ksys_write(fd, buf, count);
11)         write_buff(ret, ret_buff);
12)         return ret;
13)   } else if(is_slave(current)){
14)         write_buff(buf, vote_buff);
15)         return read_buff(ret_buff);
16)   }
17) }
```

3.6 如何结束主从关系的冗余进程

进程结束的时机具有三种情况, 一是进程本身的执行逻辑中正常结束进程; 二是在 write 系统调用表决过程中, 发现 *buf* 不一致时主进程主动结束具有主从关系的冗余进程; 三是注入错误的地址导致有进程崩溃, 使得冗余进程结束。

改造 exit 系统调用, 当冗余进程结束时: 首先判断当前进程是否为主从进程组成员, 若不是则直接按系统调用的原操作执行来结束进程, 同样不会影响未采用 KMBox 启动模块加载的进程; 若是就需要在用于维护主从进程组 mas_task_struct 构建的链表中注销使用的结构体, 然后继续系统调用的原操作, 结束进程。流程如图 9 所示。

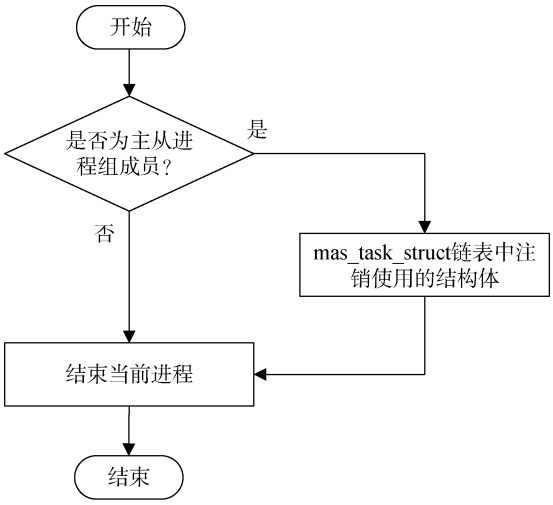

图 9 进程结束的流程

Figure 9 Process of ending processes

4 实验与分析

为验证本文设计的 KMBox 原型系统的有效性, 本节主要从防御有效性测试和性能测试对其进行实验与分析。

4.1 防御有效性测试

防御有效性测试选用具有代表性的 CTF 题目和 CVE 漏洞, 验证其利用方法在 KMBox 环境中是否有效。

PWN 题是 CTF 赛事中主流的题型之一, 通常给参赛选手已经编译好的有漏洞的二进制程序, 选手通过对二进制程序进行逆向分析和调试定位到可以利用的漏洞, 继而编写攻击脚本来远程获得目标系统的权限。CTF PWN 题在漏洞设计上针对强、覆盖广, 漏洞利用的原理具有代表性和普适性, 因此使用 CTF PWN 题来验证 KMBox 是否能够抵御进程控制流劫持攻击。

程序编译时开启 PIE 和 Canary, KMBox 部署所在的系统开启 ASLR, 将进程控制流劫持相关的 PWN 题编译后部署至 KMBox 环境中, 采用 KMBox 启动模块加载, 然后使用公开的脚本程序对漏洞进行利用, 实验结果如表 4 所示, PWN 题公开的脚本程序在 KMBox 环境中均失效, 漏洞无法被成功利用, 实验表明 KMBox 可以有效抵御进程控制流劫持相关的漏洞利用攻击。

接下来以“强网”拟态防御国际精英挑战赛 2020 中的 advance easy-stack 题目为例, 详细描述 KMBox 中由随机化方法构建冗余进程间的异构性, 对于防御作用的重要性。

advance easy-stack 题目包含了栈溢出漏洞和格式化字符串漏洞, 如源代码 3 所示, 进行三阶段的实验: 第一阶段, 程序编译时开启 PIE 和 Canary, 但未采用 KMBox 启动模块加载; 第二阶段, 程序编译时

表 4 Pwn 题实验结果

Table 4 Results of Pwn experiments

Pwn 题来源	利用方式	有效防御
Bamboofox ctf-ret2text	栈溢出	√
Bamboofox ctf-ret2libc	栈溢出	√
Bamboofox ctf-ret2syscall	栈溢出	√
CTF wiki	堆溢出	√
CTF wiki	堆溢出	√
CTF wiki	堆溢出	√
HITCON-training	堆溢出	√
Romanking98	堆溢出	√
2014 HITCON	堆溢出	√
XDCTF 2015	栈溢出	√
2016 ZCTF	堆溢出	√
2016 Seccon	堆溢出	√
2016 BCTF	堆溢出	√
东华杯 2016	劫持 vtables	√
强网拟态挑战赛 2020	栈溢出	√

关闭 PIE 和 Canary, 采用 KMBox 启动模块加载, KMBox 开启 ASLR; 第三阶段, 程序编译时开启 PIE 和 Canary, 采用 KMBox 启动模块加载, KMBox 开启 ASLR, 测试结果如表 5 所示。

源代码 3 强网拟态挑战赛 advance easy-stack 题目

```
1)unsigned __int64 dangerous_func( )
2){
3)    char format; // [rsp+0h] [rbp-90h]
4)    unsigned __int64 v2; // [rsp+88h] [rbp-8h]
5)
6)    v2 = __readfsqword(0x28u);
7)    gets(&format, 32LL, stdin);
8)    printf(&format);
9)    putchar(10);
10)   puts("now give you a gift");
11)   gets(&format, 160LL, stdin);
12)   puts("bye bye~");
13)   return __readfsqword(0x28u) ^ v2;
14) }
```

第一阶段实验首先利用格式化字符串漏洞泄露程序装载地址和 Canary 值; 再利用栈溢出漏洞并注入 Canary 值, 实施 ret2_csu_init 类似的 ROP 攻击来获得 libc 地址; 继而利用已经获得的程序装载地址和 libc 地址重写 got 表, 将 puts 函数地址改成 system 函数地址, 获得程序权限。

第二阶段实验由于 KMBox 开启了 ASLR, 冗余进程之间的 libc 地址不一致, 表决机制使得不能复现第一阶段实验结果, 无法通过格式化字符串和栈溢出漏洞来泄露内存信息。经绿盟科技 M01N 战队深入实验, 可利用程序装载地址固定和 libc 库中 syscall 函数相对于 write 函数的偏移地址固定的条件, 实施 ROP 攻击。首先通过溢出覆盖返回地址至_start; 再次溢出时将 /bin/sh 字符串写入至程序.data 段, 为后续做准备; 继续溢出覆盖 write_got 后一位地址使其变为 syscall 地址; 第四次溢出注入 pop %rax 的地址, 根据地址给 rax 赋值 0x3b, 将第二次溢出写入的.data 地址储存到 rdi 中, rsi、rdx 储存 0; 最后跳转至 syscall 执行 execve ("/bin/bash", NULL, NULL), 获得程序权限。

表 5 advance easy-stack 实验结果

Table 5 Results of advance easy-stack experiments

实验阶段	随机化方法	利用方式	有效防御
第一阶段	编译器: 关闭 PIE、Canary KMBox: 未部署	内存泄露 注入地址	×
第二阶段	编译器: 关闭 PIE、Canary KMBox: 开启 ASLR	注入地址	×
第三阶段	编译器: 开启 PIE、Canary KMBox: 开启 ASLR	/	√

第三阶段实验 KMBox 同时开启 ASLR、PIE 和 Canary。由于冗余进程的 Canary 值不同, 无法泄露 Canary 值, 同时冗余进程 pop %rax 的地址不同, 该阶段在注入 pop %rax 的地址时至少会造成其中一个进程崩溃, 导致冗余进程停止运行。因此在该阶段实验时攻击者无法获得程序权限。

CVE 漏洞实验中选取了真实软件 curl 和 Squid 的漏洞 CVE-2016-9586(影响版本 7.1.0~7.51.0)、CVE-2018-16890(影响版本 7.36.0~7.64.0)和 CVE-2019-18679 进行验证(影响版本≤4.8), 源代码在编译时均开启 PIE 和 Canary。

CVE-2016-9586 是 libcurl 中 printf 功能实现的浮点数溢出漏洞, lib/mprintf.c 中的 dprintf_formatf 函数在双精度浮点数向字符串转换输出时, FORMAT_DOUBLE 分支语句中未检查浮点数长度 *width* 是否小于字符数组的数组长度 *BUFFSIZE*, 攻击者可以构造长度超过 255 字节的特定浮点数在 sprintf 函数调用处引发缓冲区溢出, 进而实施控制流劫持攻击。利用具有浮点数溢出测试函数 test_float_formatting 的程序 tests/lib557.c 进行验证, 编译后使用 KMBox 启

动可执行程序, 冗余进程泄露的内存信息不一致, 导致无法通过 write 系统调用的表决机制, 阻断信息泄露。CVE-2018-16890 则是整型溢出漏洞, 同样在进行越界读取内存信息时无法通过表决机制, 信息泄露过程也被阻断。

CVE-2019-18679 是 Squid 信息泄露漏洞, 攻击者通过 Squid 代理任意访问网站可以在返回头的 nonce 字段中获得 base64 编码的内存地址。nonce 在初始化过程中函数 authDigestNonceEncode 调用函数 xstrdup 将 digest_nonce_data 拷贝给 nonce->*key*, 分析 digest_nonce_data 的结构体内有指向自身的指针, 如源代码 4 所示, 当 Squid 配置开启了认证功能, 制造状态码是 401 或 407 的返回消息即可触发该漏洞。实验中, 在 httpHeaderPutStrf 函数前将 nonce 字段内容输出在本地, 编译后使用 KMBox 启动 Squid, 由于冗余进程的 digest_nonce_data 地址不同, 向本地输出时 write 系统调用能够表决发现不一致, 从而抵御信息泄露避免被劫持控制流。

源代码 4 Squid 源程序中 digest_nonce_data 结构体

```
1)struct _digest_nonce_data
2){
3)    time_t creationtime;
4)    // in memory address of the nonce struct
5)    digest_nonce_h *self;
6)    long randomdata;
7) };
```

以上表明, KMBox 能够有效抵御以内存泄露和注入地址为基础的进程控制流劫持攻击, 并且在 KMBox 开启 ASLR、程序编译时开启 PIE 和 Canary 的情况下防御效果最佳。

4.2 性能测试

性能测试为了对比 KMBox 原型系统与基于 ptrace 的进程异构冗余执行系统 MimicBox, 采用一致的测试环境如表 6 所示, KMBox 基于 Linux 内核版本 3.10.0 设计实现。

表 6 性能测试环境

Table 6 Performance tests environment

配置	参数
CPU	Intel(R) Xeon® CPU E5-2603 v4@ 1.70 GHz 6 cores
内存	32GB
内核版本	3.10.0
操作系统	CentOS 7.3

选取 SPECint2006 中的样本, 分别在普通环境、KMBox 环境、MimicBox 环境中测试样本的运行时间, 以普通环境的运行时间为基准, KMBox 环境和 MimicBox 环境中样本运行时间的增幅比例如图 10 所示。ptrace 作为系统调试工具会对每个系统调用进行拦截, 频繁地在内核态和用户态之间切换, 造成性能开销, 因此基于内核改造的 KMBox 相较于基于 ptrace 实现的 MimicBox, 样本运行的时间增幅较少, 在控制性能损耗方面更有优势。

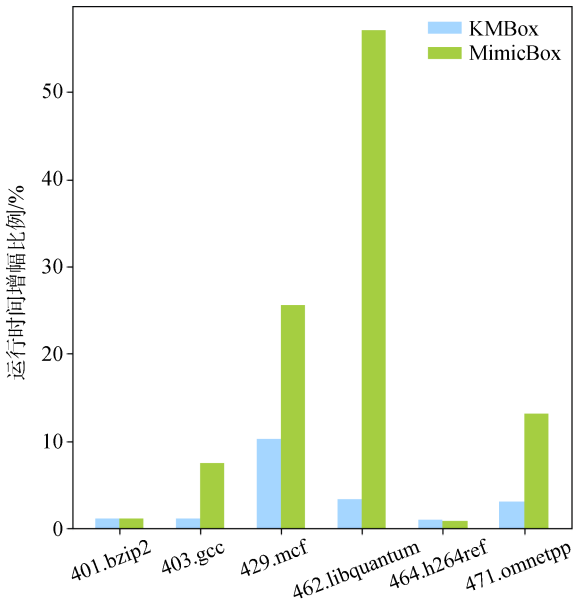

图 10 性能测试结果

Figure 10 Results of performance tests

5 局限性分析

KMBox 原型系统的局限性主要体现在以下 3 个方面:

1) KMBox 是面向进程的异构冗余执行系统, 无法适用于非确定性多线程应用程序, 首先是因为线程调度问题, 多线程间的系统调用顺序不同, 无法强制确保冗余线程间的等价关系, 导致无法构建表决机制; 其次多线程能够通过共享内存的方式进行通信, 无需使用系统调用, 因此以系统调用作为表决对象的 KMBox 就失去了作用。

2) 除了多线程调度产生的影响, 还有异步信号、时间和随机数等为表决引入了不确定性问题, 使 KMBox 失去作用。目前, 团队正基于编译器, 研究面向源代码消除在冗余执行时可能带来的不确定性问题的解决方案, 首先利用编译器对会引入不确定问题的变量和代码片段地址进行提取, 并插桩同步处理函数。然后在运行时利用内核模块实现的监视

器监视异构冗余进程的关键系统调用, 将执行结果由主进程拷贝至从进程。

3) 本文中 3.4 节已经描述了如何实现文件描述符和文件系统描述符的共享, 从而避免重复操作影响了程序的正确性。除此之外, 对任何设置/写类型的系统调用都需要改造, 重新设计执行过程, 尤其是具有输出功能的系统调用, 例如 send、sendto 等。未来 KMBox 原型系统若要适用更复杂的应用程序, 对内核的全面改造工作将是具有挑战性的, 因此综合以上因素, KMBox 当前更适用于专用系统或功能较固定的系统。

6 结束语

软件异构冗余执行系统是防御控制流劫持攻击的方法之一, 本文总结了基于 ptrace 和添加内核模块实现方式的不足。率先通过直接修改内核的方式设计出进程异构冗余执行系统 KMBox, 完全在内核中实现冗余进程的启动、主从关系注册、内存地址空间异构、非异构资源共享、系统调用表决及结束等关键环节。原型系统 KMBox 首先为进程在“启动—结束”的运行过程构建冗余执行架构, 其次通过内存地址空间随机化技术保证冗余进程的内存地址空间布局是异构的, 最后在与内存信息泄露相关的系统调用 write 处进行表决, 发现泄露信息不一致, 阻断进程控制流劫持攻击。即使攻击者能够跳过内存信息泄露进行漏洞利用, 异构内存空间布局也使得注入由 gadget 地址构造的 payload 无法同时在冗余的进程中有效, 阻断进程控制流劫持攻击。性能对比测试显示, KMBox 由于不再使用 ptrace, 不会对所有系统调用进行拦截, 性能要优于基于 ptrace 的进程异构冗余执行系统。防御有效性测试采用 CTF PWN 题和 CVE 漏洞, 验证了防御控制流劫持攻击的有效性。

参考文献

[1] van der Veen V, Dutt-Sharma N, Cavallaro L, et al. Memory Errors: The Past, the Present, and the Future[M]. Research in Attacks, Intrusions, and Defenses. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: 86-106.

[2] Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks[C]. *The 7th conference on USENIX Security Symposium - Volume 7*, 1998: 5.

[3] Shacham H, Page M, Pfaff B, et al. On the Effectiveness of Address-Space Randomization[C]. *The 11th ACM conference on Computer and communications security*, 2004: 298-307.

[4] He W J, Das S, Zhang W, et al. BBB-CFI: Lightweight CFI Approach Against Code-Reuse Attacks Using Basic Block Information[J]. *ACM Transactions on Embedded Computing Systems*, 2020, 19(1): 7.

[5] Schwartz E J, Cohen C F, Gennari J S, et al. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks[C]. *The 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020: 1789-1801.

[6] Tran M, Etheridge M, Bletsch T, et al. On the Expressiveness of Return-into-Libc Attacks[C]. *The 14th international conference on Recent Advances in Intrusion Detection*, 2011: 121-141.

[7] Shacham H. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)[C]. *The 14th ACM conference on Computer and communications security*, 2007: 552-561.

[8] Han H. *Detection of return-oriented programming attack and its variant*[D]. Nanjing: Nanjing University, 2011.
(韩浩. ROP 攻击及其变种的检测技术[D]. 南京: 南京大学, 2011.)

[9] Xing X, Chen P, Ding W B, et al. BIOP: Automatic Construction of Enhanced ROP Attack[J]. *Chinese Journal of Computers*, 2014, 37(5): 1111-1123.
(邢骁, 陈平, 丁文彪, 等. BIOP: 自动构造增强型 ROP 攻击[J]. *计算机学报*, 2014, 37(5): 1111-1123.)

[10] Checkoway S, Davi L, Dmitrienko A, et al. Return-Oriented Programming without Returns[C]. *The 17th ACM conference on Computer and communications security*, 2010: 559-572.

[11] Bletsch T, Jiang X X, Freeh V W, et al. Jump-Oriented Programming: A New Class of Code-Reuse Attack[C]. *The 6th ACM Symposium on Information, Computer and Communications Security*, 2011: 30-40.

[12] Yao F, Chen J, Venkataramani G, et al. JOP-alarm: Detecting jump-oriented programming-based anomalies in applications[C]. *2013 IEEE 31st International Conference on Computer Design*, 2013: 467-470.

[13] Graziano M, Balzarotti D, Zidouemba A. ROPMEMU: A framework for the analysis of complex code-reuse attacks[C]. *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016: 47-58.

[14] Onarlioglu K, Bilge L, Lanzi A, et al. G-Free: Defeating return-oriented programming through gadget-less binaries[C]. *The 26th Annual Computer Security Applications Conference on – ACSAC'10*, 2010: 49-58.

[15] Zhang C. *Research on Software Security Protection Technology Against Control Flow Hijacking Attack*[D]. Beijing: Peking University, 2013.
(张超. 针对控制流劫持攻击的软件安全防护技术研究[D]. 北京: 北京大学, 2013.)

[16] Wang F F, Zhang T, Xu W G, et al. Overview of Control-Flow Hijacking Attack and Defense Techniques for Process[J]. *Chinese Journal of Network and Information Security*, 2019, 5(6): 10-20.
(王丰峰, 张涛, 徐伟光, 等. 进程控制流劫持攻击与防御技术综述[J]. *网络与信息安全学报*, 2019, 5(6): 10-20.)

[17] Chen Y S, Chen P S, Processing C A. A software-based redundant execution programming model for transient fault detection and correction[C]. *2016 45th International Conference on Parallel*

Processing Workshops, 2016: 66-71.

[18] Franz M. E Unibus Pluram: Massive-Scale Software Diversity as a Defense Mechanism[C]. *The 2010 New Security Paradigms Workshop*, 2010: 7-16.

[19] Wu J X. *Network endogenous security-part two: Mimic defense and generalized robust control*[M]. Beijing: Science Press, 2020.
(邬江兴. 网络空间内生安全-下册: 拟态防御与广义鲁棒控制[M]. 北京: 科学出版社, 2020.)

[20] Wu J X. *Network endogenous security-part two: Mimic defense and generalized robust control*[M]. Beijing: Science Press, 2020.
(邬江兴. 网络空间内生安全-下册: 拟态防御与广义鲁棒控制[M]. 北京: 科学出版社, 2020.)

[21] Just J E, Cornwell M. Review and Analysis of Synthetic Diversity for Breaking Monocultures[C]. *The 2004 ACM workshop on Rapid malcode*, 2004: 23-32.

[22] Salamat B, Jackson T, Gal A, et al. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space[C]. *The 4th ACM European conference on Computer systems*, 2009: 33-46.

[23] Salamat B. Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks DISSERTATION[D]. *University of California, Irvine*, 2009.

[24] Jackson T, Salamat B, Wagner G, et al. On the Effectiveness of Multi-Variant Program Execution for Vulnerability Detection and Prevention[C]. *The 6th International Workshop on Security Measurements and Metrics*, 2010: 1-8.

[25] Volckaert S, De Sutter B, De Baets T, et al. GHUMVEE: Efficient, Effective, and Flexible Replication[C]. *The 5th international conference on Foundations and Practice of Security*, 2012: 261-277.

[26] Volckaert S, Coppens B, de Sutter B. Cloning your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution[J]. *IEEE Transactions on Dependable and Secure Computing*, 2016, 13(4): 437-450.

[27] Volckaert S, Coppens B, Voulimeneas A, et al. Secure and Efficient Application Monitoring and Replication[C]. *The 2016 USENIX Conference on Usenix Annual Technical Conference*, 2016: 167-179.

[28] Coppens B, De Sutter B, Volckaert S. Multi-variant execution environments[M]. *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018: 211-258.

[29] Wu J X. Research on Cyber Mimic Defense[J]. *Journal of Cyber Security*, 2016, 1(4): 1-10.
(邬江兴. 网络空间拟态防御研究[J]. *信息安全学报*, 2016, 1(4): 1-10.)

[30] Yao D, Zhang Z, Zhang G F, et al. MVX-CFI: A Practical Active Defense Framework for Software Security[J]. *Journal of Cyber Security*, 2020, 5(4): 44-54.
(姚东, 张铮, 张高斐, 等. MVX-CFI: 一种实用的软件安全主动防御架构[J]. *信息安全学报*, 2020, 5(4): 44-54.)

[31] Pan C X, Zhang Z, Ma B L, et al. Method Against Process Control-Flow Hijacking Based on Mimic Defense[J]. *Journal on Communications*, 2021, 42(1): 37-47.
(潘传幸, 张铮, 马博林, 等. 面向进程控制流劫持攻击的拟态防御方法[J]. *通信学报*, 2021, 42(1): 37-47.)

[32] Ma B L, Zhang Z, Liu H, et al. SQLMVED: SQL Injection Runtime Prevention System Based on Multi-Variant Execution[J]. *Journal on Communications*, 2021, 42(4): 127-138.
(马博林, 张铮, 刘浩, 等. SQLMVED: 基于多变体执行的 SQL 注入运行时防御系统[J]. *通信学报*, 2021, 42(4): 127-138.)

[33] Lu K. Securing software systems by preventing information leaks[D]. *Georgia Institute of Technology*, 2017.

[34] Bigelow D, Hobson T, Rudd R, et al. Timely Rerandomization for Mitigating Memory Disclosures[C]. *The 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015: 268-279.

马博林 于 2015 年在哈尔滨工业大学信息安全专业获得学士学位。现在信息工程大学网络空间安全专业攻读博士学位。研究领域为网络安全。Email: msd_mbl@qq.com

张铮 于 2006 年在信息工程大学计算机科学与技术专业获得博士学位。现任信息工程大学副教授。研究领域为网络安全、先进计算。Email: ponyzhang@126.com

邵昱文 于 2019 年在杭州电子科技大学计算机科学与技术专业获得学士学位。现在信息工程大学计算机技术专业攻读硕士学位。研究领域为网络安全。Email: 735011726@qq.com

李秉政 于 2019 年在信息工程大学计算机科学与技术专业获得学士学位。现在信息工程大学网络空间安全专业攻读博士学位。研究领域为网络安全。Email: francisleeha@163.com

潘传幸 于 2018 年在山东农业大学计算机科学与技术专业获得学士学位。现在信息工程大学计算机科学与技术专业攻读博士学位。研究领域为网络安全。Email: cxpan@mimicsecurity.cn

蒋鹏 现任网络通信与安全紫金山实验室研究员, 曾任华为 2012 实验室工程师。研究领域为网络安全。Email: jiangfly2011@163.com

邬江兴 现任信息工程大学教授, 博导, 中国工程院院士。研究领域为信息通信网络、网络安全。Email:ndscwjx@126.com