

通过 Glibc 堆信息提取检测 house of spirit 类型攻击

翟继强, 王家乾, 韩旭, 孙海旭

(哈尔滨理工大学 计算机科学与技术学院, 哈尔滨 150080)

摘要: 目前有关堆的取证研究主要是针对 Windows 系统的堆和 NT 堆, 然而怎样从转储文件中提取出 Linux 系统 Glibc 堆信息并没有得到充分的研究。为了重现 Linux 系统中 Glibc 堆的内部信息, 采用内存对象 vtype 描述信息中字段偏移定位结合在内存中 Glibc 堆实现的方法提取 Glibc 堆内部信息。并基于此方法研发了基于 Rekall 框架的 3 个堆信息提取插件。还研究了 house of spirit 类型的堆攻击, 建立了攻击模型并提取其攻击特征。基于提取的攻击特征设计出针对 house of spirit 攻击的检测算法。在堆信息提取插件的基础上研发了攻击检测插件。实验结果表明本方法可以有效地提取 Linux 系统进程中堆在内存中的信息, 并且基于这些信息结合攻击检测算法成功检测内存中的 house of spirit 类型攻击。

关键词: Glibc 堆; 信息提取; 堆攻击检测; Rekall 框架

DOI: 10.15938/j.jhust.2024.01.011

中图分类号: TP319 **文献标志码:** A **文章编号:** 1007-2683(2024)01-0096-11

Detecting House of Spirit Attacks by Glibc Heap Information Extraction

ZHAI Jiqiang, WANG Jiaqian, HAN Xu, SUN Haixu

(School of Computer Science and Technology, Harbin University of Science and Technology, Harbin 150080, China)

Abstract: The current forensic research on heaps mainly extracts information from the heap and the NT heap of Windows. However, the study of how to extract the information on the Glibc heap in the Linux from dump files is not sufficient. To reproduce the internal information on the Glibc heap, this paper proposes a method to extract the internal information of Glibc heap in the Linux according to the field offset in the vtype description information of memory object combined with Glibc heap implementation in memory. Based on this method, three heap information extraction plugins were developed on the Rekall framework. In addition, the House of Spirit heap attack is studied, established an attack model, and extracted its attack features. A detection algorithm for House of Spirit attack is designed based on the extracted attack features. The attack detection plug-in is developed based on the heap information extraction plugins. The experimental results show that this method can effectively extract the heap information in the memory of the Linux system process, and successfully detect the House of Spirit attack in the memory based on the information combined with the attack detection algorithm.

Keywords: Glibc heap; information extraction; heap attack detection; Rekall framework

0 引言

近年来, 黑客经常通过网络传播恶意程序, 当计

算机染上恶意程序时, 内存会留下痕迹, 通过内存取证技术就可以捕获到这些痕迹^[1]。堆中含有进程运行时的重要信息, 提取出堆中的信息可以了解进程的运行情况^[2], 因此对堆的内存取证研究在信息

收稿日期: 2022-10-20

基金项目: 国家自然科学基金(61403109); 国家自然科学基金(61402126); 国家自然科学基金(61602133)。

作者简介: 王家乾(1997—), 男, 硕士研究生;

韩旭(1997—), 男, 硕士研究生。

通信作者: 翟继强(1972—), 男, 博士, 教授, 硕士研究生导师, E-mail: zaijiqiang@163.com.

安全的防护领域意义重大。

到目前为止,对用户空间进程堆的分析还没有得到足够的重视^[3]。Urrea、Case et al. 和 Light et al. 主要研究涉及内核相关的工作^[4-5]。Leppert 和 Macht 研究的是分析应用程序和它们的堆数据^[6-7]。然而他们的分析主要集中在堆中包含的序列化 Java 对象上,而不是堆对象的管理方式。其他已经存在的插件诸如 cmdscan 和 bash^[8],它们分别从 Windows 的 cmd 和 Linux 的 bash shell 提取命令历史。然而,这些插件的工作原理只是将堆作为一个大的内聚内存区域来识别其中可能存在的信息^[9]。

此外,由 Light 等^[10]进行的记事本程序的堆分析是唯一 1 个使用了堆细节的研究,但它也仅仅与 Windows 系统有关。在内存取证范围之外,也存在关于堆的基础研究^[11],特别是关于 Linux 进程的堆以及如何管理堆的研究。Ferguson 的研究有助于深入理解 Glibc 的堆实现^[12]。然而这项研究更多地集中在利用堆的方式上,因此没有提供足够的信息来可靠地收集内存取证场景中堆的所有相关信息。

为了弥补这些缺陷,本文研究了多个版本的基于 Glibc 堆实现的 Linux 系统,并且基于内存中 Glibc 堆实现,结合字段信息偏移定位的方法,提取堆内部信息。经过测试,该方法能成功提取堆内部信息,并且基于这些信息成功设计出检测 house of spirit 堆溢出攻击的插件。本文研究的主要内容如下:

- 1) 研究了多个 Linux 版本的 Glibc 堆实现及其在内存中的结构及字段的作用;
- 2) 根据内存中 Glibc 堆实现,结合内存对象 vtype 描述信息中字段偏移定位实现提取 Glibc 堆内部信息的方法;
- 3) 分析了 house of spirit 类型的堆攻击,建立其攻击模型并提取其攻击特征;
- 4) 基于提取的攻击特性设计出针对 house of spirit 攻击的检测算法。在堆信息提取插件的基础上研发了攻击检测插件;
- 5) 使用本文研发出的插件检测 house of spirit 攻击。

1 Glibc 堆信息提取

1.1 内存中的 Glibc 堆实现分析

图 1 显示了正在运行进程的堆布局,重点是各个元素之间的引用。Chunk 包含实际的用户/进程

数据。通过 malloc 调用显式分配,这些块位于特定的内存区域中。

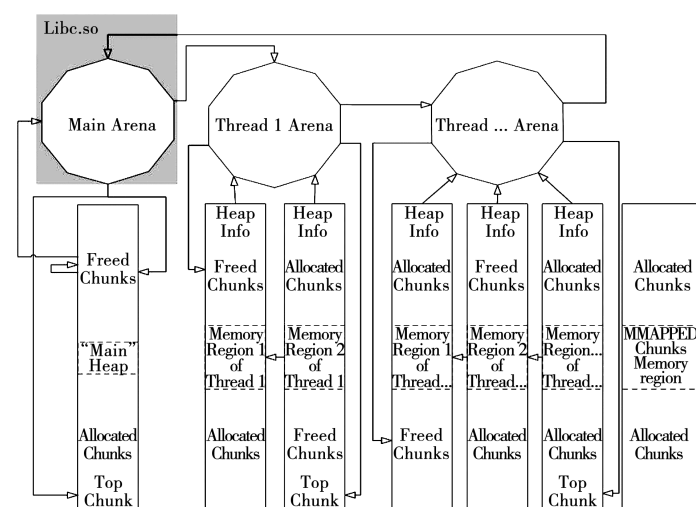


图 1 Glibc 堆布局

Fig. 1 Glibc HeapLayout

1.1.1 内存中的 Arena 和 heap_info 结构

主堆是一个连续的内存区域,被分割到 `vm_area_struct` 结构体描述的多个连续内存区域中。用于描述 Main Arena 的 `malloc_state` 数据结构存储于映射在内存中的 Libc 库中,在 Main Arena 中没有 `heap_info` 结构实例。

线程 Arena 的 `malloc_state` 数据结构与堆块一起存储在同一内存区域中。在属于线程 Arena 的每个映射内存区域的开始处都存在 1 个 `heap_info` 结构的实例。除此之外,在此之后的 `heap_info` 结构最终都位于同一映射内存区域中。

如图 2 所示,灰色区域是 `vm_area_struct` 数据结构所描述的内存区域,蓝色标记的结构和块属于同一 Arena,绿色标记的结构和块属于另一 Arena。`size` 成员存储的指针标记相对应 `heap_info` 结构所描述的内存区域的大小,`ar_ptr` 成员指向其关联的 Arena,`prev` 成员指向同一 Arena 中的上 1 个 `heap_info` 结构。

1.1.2 内存中的已分配堆块

内存中的已分配块如图 3 所示。用户数据开始于 `size` 成员之后,一直到下 1 个区块的 `size` 成员。`size` 成员后面的 `malloc_chunk` 结构的所有成员都会被用户数据覆盖。用户数据覆盖范围如图蓝色字段。

如果上一堆块为已释放堆块 `prev_size` 字段包含前一个区块的大小。否则 `prev_size` 成员为上 1 个已分配堆块用户数据的一部分。

1.1.3 内存中的已释放块和 bins

初始态时,所有 bins 和 fastbins 都是空的。对

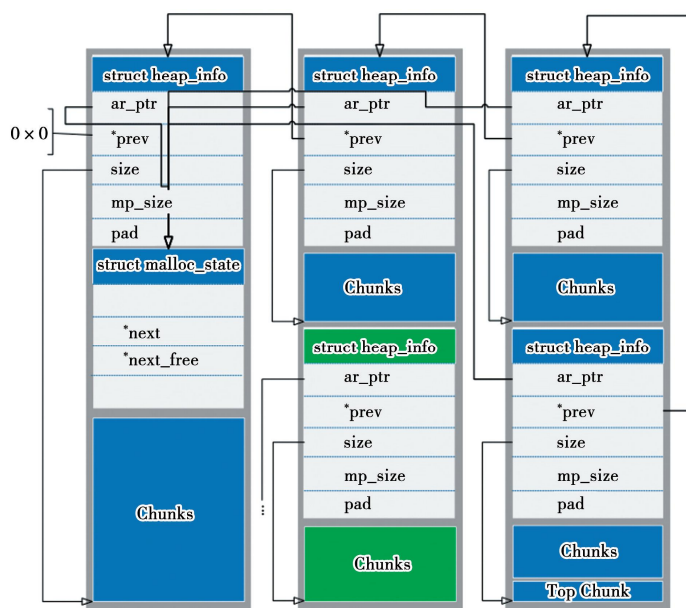


图2 内存中的 malloc_state 和 heap_info

Fig. 2 malloc_state and heap_info in memory

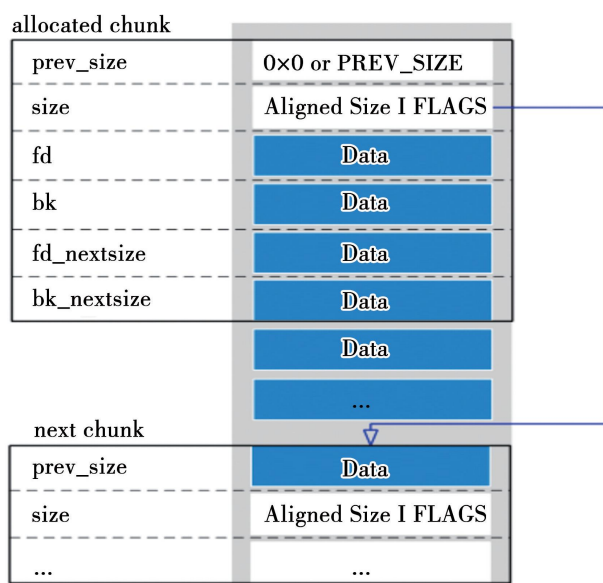


图3 内存中的已分配堆块

Fig. 3 Allocated heap chunk in memory

于 fastbins, 它们的指针都初始化为零, 而 bins 有 1 个指向它们自己的指针。每个 bin 都包含 1 个正向和反向链接。

图 4 显示了 bins 正好包含 1 个块的情况。可以看到, 实际释放的块引用 Arena 中的虚拟块, 该块的 fd 和 bk 指针与 bin 的指针相匹配。

图 5 为带有多个块的 bins。bin 的 bk 指针指向该 bin 的最后 1 个块, fd 指针指向第 1 个块。第 1 个块的 bk 指针和最后 1 个块的 fd 指针都引用 Arena 中的虚拟区块, 实现了 1 个循环双链表。

1.1.4 内存中的 MMAPPED 块

MMAPPED 块通常位于 vm_area_struct 数据结

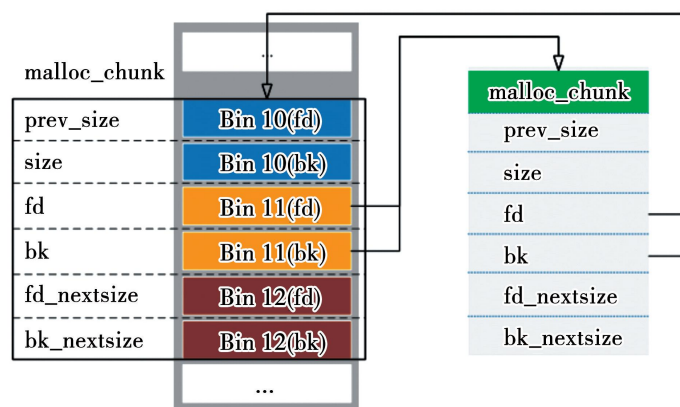


图4 内存中带有 1 个释放块的 bins

Fig. 4 Bins with a freed chunk in memory

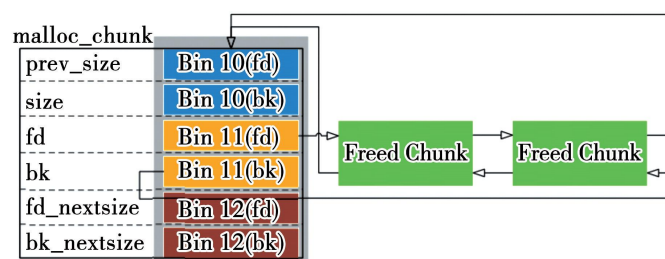


图5 内存中带有多个释放块的 bins

Fig. 5 Bins with multiple freed chunk in memory

构描述的专用内存区域中。没有来自 malloc_state 或 heap_info 等结构的指针引用内存区域或块本身。其大小为页面大小的倍数。此外, 由于内核可以放大一个区域, 多个 MMAPPED 块最终可能位于 1 个 vm_area_struct 数据结构描述的同—内存区域中。如果 1 个位于同一区域的两个或多个 MMAPPED 块之间的 MMAPPED 块被释放, 则连续内存区域将被拆分为两个单独的区域。

1.2 基于 Glibc 堆实现的堆信息提取方法

本节根据内存对象 vtype 描述信息中字段偏移定位结合在内存中 Glibc 的堆实现提出了一种提取 Glibc 堆内部信息的方法。

1.2.1 堆初始化信息提取方法

当识别出取证文件为 Linux 系统的转储文件时, 使用内存取证框架内置的扫描技术扫描内核空间并定位到 task 结构体, 通过 task 结构体遍历 VMA 结构, 获取 Glibc 库在内存中的位置, 提取 Glibc 库中 Main Arena 的位置。根据内存对象 vtype 描述信息中字段偏移定位 Glibc 堆的相关数据结构并提取堆的信息。堆初始化信息提取方法的流程图如图 6 所示。

该方法可以重现进程运行时, 堆的内部信息。该方法首先收集内存中所有的 Arena, 及其第 1 个堆块信息以及 MMAP 块的第 1 个堆块信息。通过收

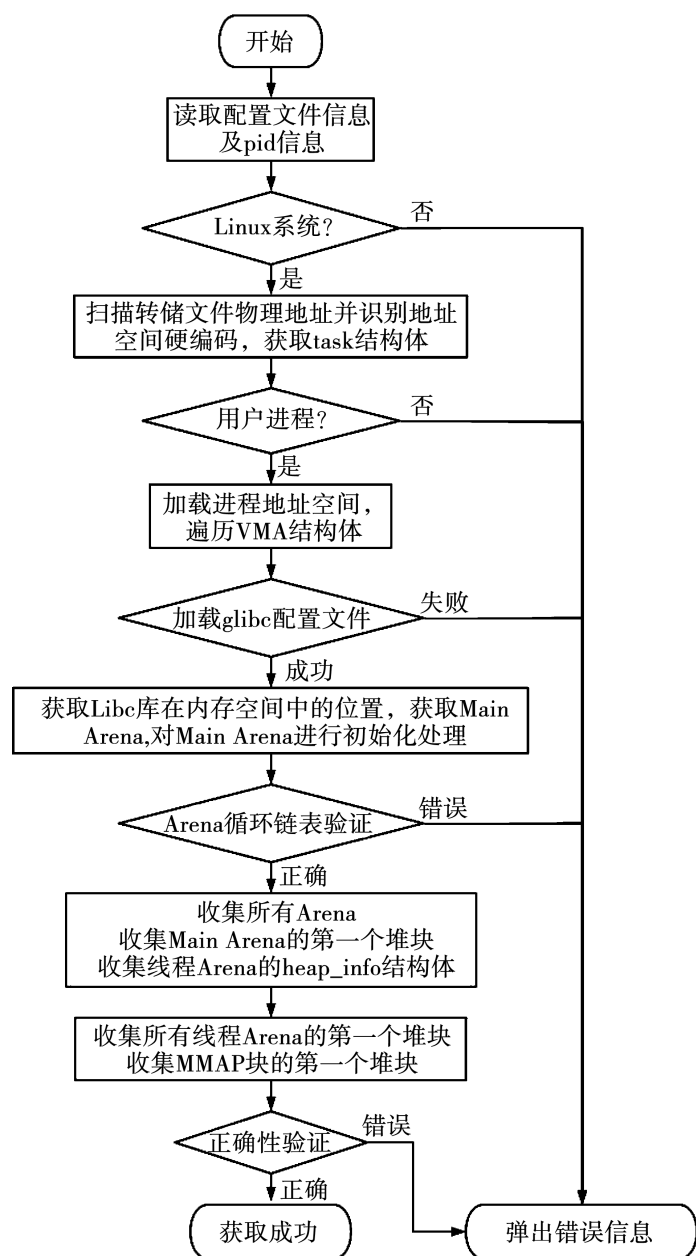


图6 堆信息提取方法

Fig. 6 Heap information extraction method

集到的初始信息,进一步收集内存中已分配及释放的堆块信息,通过 bins 列表收集 bins 表信息,从而实现对于堆的初始化信息提取。

1.2.2 堆块信息提取方法

当使用 Glibc 分配器分配堆块时,其堆块起始地址 add 加上 size 成员的值即下 1 个堆块的地址 next_add,堆块 size 位的标志位 PREV_INUSE 指示前 1 个堆块是否为已释放堆块。该方法基于堆初始化信息提取方法所收集到的信息,遍历堆空间,基于 malloc_chunk 数据结构追踪程序已分配/释放堆块信息。提取的信息有已分配/释放堆块的数量、及堆块内部信息等。该方法执行的步骤如下:

步骤 1:基于堆初始化信息提取方法所收集到的信息,定位 Main Arena.first_chunk 位置;

步骤 2:提取 first_chunk 地址,通过其地址及

malloc_chunk 数据结构追踪 next_chunk;

步骤 3:获取 next_chunk 相关数据结构的 size 位,识别标志位 PREV_INUSE。判定前一堆块为分配对堆块还是释放堆块,保存在相应堆块列表中,指针保存在相应指针列表中。

步骤 4:基于对初始化信息提取方法所收集到的信息,遍历 Thread Arena 列表,定位线程 heap_info 结构体位置,获取线程 malloc_state,紧接着定位到 Thread Arena.firstchunk,获取其地址。

步骤 5:重复步骤 2 和步骤 3。

步骤 6:统计已分配/释放堆块的数量。

2 house of spirit 攻击检测

2.1 house of spirit 攻击原理及特征

堆分配器具有广泛的攻击面。一系列攻击的重点是利用分配器算法本身^[13],例如特定的 fastbin 或 smallbin 实现。攻击者的目标是建立任意指针或任意代码执行^[14]。任意指针尤其危险,因为它们可以用来操纵目标程序的控制流^[15]。house of spirit 攻击便属于此类攻击。

house of spirit 攻击原理:①修改 1 个将被释放的堆指针,使其指向包含全局指针 m_ptr 的可控内存地址。②在可控内存区域中构建 1 个伪堆结构,绕过 Glibc 的检测机制。可控内存可以作为堆块释放到堆块的链表中。③在链表中的可控内存再次被应用后,可以实现 m_ptr 指针数据的可控性,并且可以写入任意地址。以实现任意地址可控。图 7 为 house of spirit 攻击过程的示意图。

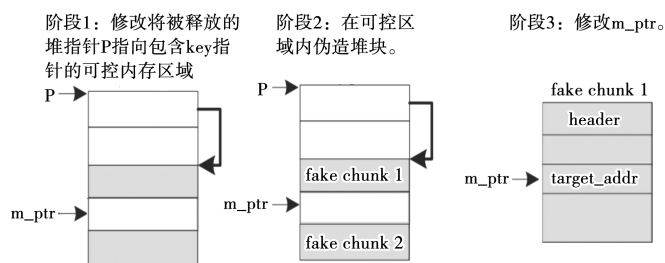


图7 house of spirit 攻击过程

Fig. 7 house of spirit attack process

因此,当程序中存在 1 个将要释放的可控堆指针时,意味着该程序满足 house of spirit 攻击的第 1 个攻击特征。

其次如果全局指针 m_ptr 附近存储空间为可控区域,并且在可控内存空间中存在伪造堆块。意味着该程序满足 house of spirit 攻击的第 2 个攻击特征。

最后释放受控内存区域,并将其再次分配为堆块。基于堆的特性, m_ptr 变为可控数据,意味着满足 house of spirit 攻击的第3个攻击特征。此时攻击者可以通过修改全局指针数据的值来实现任意地址写入的目的。

2.2 house of spirit 攻击分析及特征提取

为了实现对 house of spirit 攻击的自动检测,需要监控 house of spirit 攻击劫持控制流的过程,并在程序运行过程中收集程序控制流信息和堆块的操作信息。

为了消除保护机制对检测过程的影响,根据 house of spirit 的攻击过程设计了相应的数据体现过程,并建立了 house of spirit 攻击的攻击模型,如图8所示。

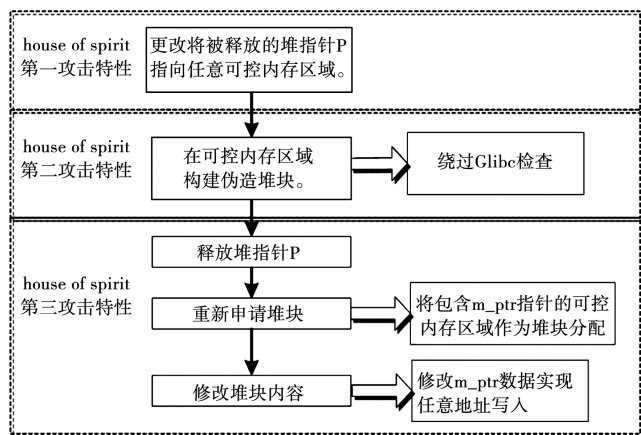


图8 house of spirit 攻击模型

Fig. 8 house of spirit attack model

如果堆指针 P 是可控堆指针,则表示该程序满足 house of spirit 攻击的第1攻击特征。如果 m_ptr 指针附近的存储区域为可控的,并且在可控内存空间中存在构建的伪造堆块结构。则表示该程序满足 house of spirit 攻击的第2攻击特征。释放指针 P ,将包含 m_ptr 的可控内存空间作为堆块释放到堆链表中,然后再申请,将数据写入新创建的堆块中,修改 m_ptr 的值,实现任意地址写入,则表示该程序满足 house of spirit 攻击的第3攻击特征。最终实现控制流劫持。

2.3 house of spirit 攻击检测算法

基于 Glibc 堆信息提取方法,动态追踪内存中程序用户空间进程堆信息。结合 house of spirit 攻击模型的3个攻击特征,设计针对 house of spirit 攻击的攻击检测模型,如图9所示。

针对第1攻击特性,本文设计了一种 house of spirit 第1攻击特征检测算法。基于堆信息提取所收集到的信息,获取已分配堆块列表 $Achunk$ 和已释放指针列表 $Fpointer$ 。当已释放指针列表新增释放指针 FP 时遍历 $Achunk$ 检测其是否属于之前的

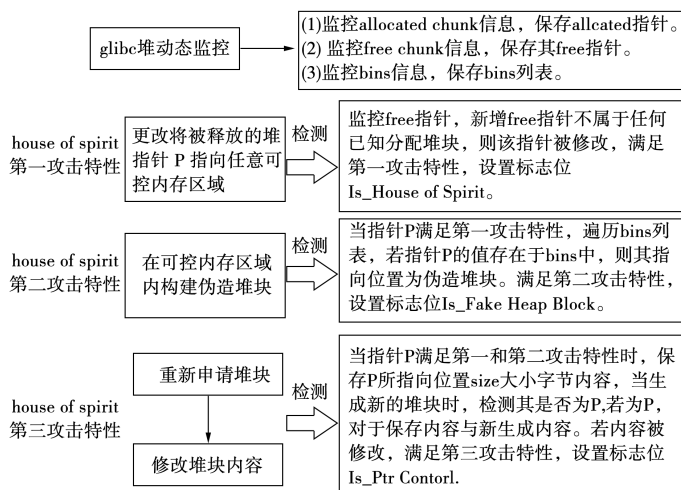


图9 house of spirit 攻击检测模型

Fig. 9 house of spirit attack detection model

已分配堆块,若其不属于之前的已分配堆块,则设置指针 FP 标志位 $Is_HouseOfSpirit$ 为 $true$ 。保存指针 FP 于特定列表。其所指向堆块保存在特定堆块列表中。

house of spirit 第1攻击特征检测算法如算法1所示。

算法1 house of spirit 第一攻击特征检测算法

输入:已分配堆块列表 $Achunk$;

已释放堆块指针列表 $Fpointer$;

将被释放的堆指针 FP ;

输出:house of spirit 第1攻击特征检测标志位

$Is_HouseOfSpirit$ 。

foreach (FP)

{

if (FP not in $Achunk$)

{

$FPchunk = Getchunk(FP)$;

$FPchunk \rightarrow Is_HouseOfSpirit = true$;

AddIs _ HouseOfSpiritChunkList ($FPchunk$);

AddIs_HouseOfSpiritPointerList(FP);

}

else

{

$FPchunk = Getchunk(FP)$;

AddFpointer(FP);

DelAchunk($FPchunk$);

$FPchunk \rightarrow Is_HouseOfSpirit = false$;

}

}

针对第2攻击特性,本文设计了一种 house of

spirit 第 2 攻击特征检测算法。基于堆信息提取所收集到的信息,定位 malloc_state 结构体,获取 bin 表信息。当检测到新增释放指针 P 时,检索 HouseOfSpiritPointer 指针列表,当指针 P 属于该列表时,获取其所指向堆块的相关数据结构,识别该堆块 size 位的值,确认其属于哪个 bin 表,遍历该 bin 表,若存在则设置标志位 Is_FakeHeapBlock。

house of spirit 第 2 攻击特征检测算法如算法 2 所示。

算法 2 house of spirit 第 2 攻击特征检测算法

输入:Main Arena 数据结构体 malloc_state;

第 1 攻击特征指针列表 HouseOfSpiritPointer;

新增释放堆指针 P;

输出:house of spirit 第 2 攻击特征检测标志位

Is_FakeHeapBlock。

foreach (P)

{

if(P in HouseOfSpiritPointer)

{

Pchunk = Getchunk(P);

size = Getsize(Pchunk);

bins = Getbins(malloc_state. size);

if(Pchunk in bins)

{

Pchunk -> Is_FakeHeapBlock = true;

Fakechunk = Heapchunkdump (P,

size);

}

else

{

Pchunk -> Is_FakeHeapBlock = false;

Del(HouseOfSpiritPointer(), P);

Del(HouseOfSpiritChunk(), Pchunk);

}

}

}

针对第 3 攻击特征,本文设计了一种 house of spirit 第 3 攻击特征检测算法。基于 Is_HouseOfSpirit 检测组件及其 Is_FakeHeapBlock 检测组件所提供信息,获取疑似攻击列表。当检测到新增分配指针 P 时,检索 HouseOfSpiritPointer 指针列表,获取其所指向的堆块,追踪新分配堆块,在新分配堆块内查找变化指针,其指向控制流,即 shellcode。

house of spirit 第 3 攻击特征检测算法如算法 3

所示。

算法 3 house of spirit 第 3 攻击特征检测算法

输入:第 1 攻击特征指针列表 HouseOfSpiritPointer;

新增分配堆指针 AP;

输出:house of spirit 第 3 攻击特征检测标志位

Is_PtrContorl。

foreach (AP)

{

if(AP in HouseOfSpiritPointer)

{

if(AP -> Is_HouseOfSpirit = true)

{

if(AP -> Is_FakeHeapBlock = true)

{

Allochunk = Getchunk(AP);

if(Fakechunk != Allochunk)

AP -> Is_PtrContorl = true;

}

}

}

}

基于上述 3 点,对 house of spirit 攻击进行识别检测。当程序满足上述 3 点,被认定为 house of spirit 攻击。

即:House_HiJack = Is_HouseOfSpirit && Is_FakeHeapBlock && Is_PtrContorl。

3 插件的设计与实现

3.1 内存取证框架

本文研发的功能插件,都是基于内存取证框架实现的^[16]。内存取证框架可以从转储文件和硬件磁盘镜像中解析休眠文件与页面文件信息,通过这两个文件信息的对比能获得隐藏进程的证据,而且还可以使用可执行页面检测算法遍历内存页并恢复可执行页面,帮助调查人员识别代码注入等^[17]。

Rekall 框架是 Google 开发的内存分析框架,是开源平台,可以提供可视化的系统运行状态,为相关的开发人员提供更多相关的数据和材料。

Rekall 中含有各个 Windows 10 系统版本的配置文件,使用时不需要进行额外的制作^[18]。然而 Linux 操作系统由于其内核版本、子内核版本和定制内核的数量众多,需要使用者自己进行制作^[19]。配置文件里面组合了许多 vtype 描述信息。在对内存

对象进行分析的时候,这些信息可以让 Rekall 框架对转储文件中的数据进行解析。

3.2 Heapinit 插件

Heapinit 插件是基于内存取证框架实现的,该插件提取所有的 Arena,及其第 1 个堆块信息以及 MMAP 块的第 1 个堆块信息。

Heapinit 插件基于堆初始化信息提取方法实现,相关伪代码如下所示:

if profile is Linux:

if task.mm.dereference is valid:

task ← Gettaskstruct

process_as ← Get_process_address_space()

vmas = Get_vmas_for_task(task)

if vmas:

Load_libc_profile()

libc_offset = Get_libc_offset(vmas)

if libc_offset:

Main_arena = libc_offset

+ libc_profile.main_arena_offset

Initialize_arenas(Main_arena)

Initialize_heap_first_chunks()

Initialize_mmapped_first_chunk()

check_and_report_size_inconsistencies()

3.3 Heapchunkinfo 插件

该插件输出的信息有已分配/释放堆块的数量、及堆块信息等,Heapchunkinfo 插件基于堆块信息提取方法实现,相关的伪代码如下:

Heapinit ← GetHeapinit()

while(Currentchunk.add < Topchunk.add):

KNextchunk ← Currentchunk.add + Currentchunk.

size

if(Nextchunk.PREV_INUSE):

Add(MainAllocatedChunk(), Currentchunk)

Add(MainAllocatedchunkPointer(), Currentchunk.add)

MainAllocatedChunkNum ++

else:

Add(MainFreedChunk(), Currentchunk)

Add(MainFreedChunkPointer(), Currentchunk.add)

MainFreedChunkNum ++

for arena in Theard_Arena:

heapinfo ← arena.heapinfo

Currentchunk ← heapinfo.firstchunk

while(Currentchunk.add < Topchunk.add):

Nextchunk ← Currentchunk.add + Currentchunk.

size

if(Nextchunk.PREV_INUSE):

Add(ThreadAllocatedChunk(), Currentchunk)

Add(ThreadAllocedchunkP(), Currentchunk.add)

ThreadAllocatedChunkNum ++

3.4 House_HiJack 插件

House_HiJack 插件用于检测 house of spirit 攻击。针对 house of spirit 攻击模型总结出其三种攻击特征。针对三种攻击特征设计针对性的检测组件。

3.4.1 Is_HouseOfSpirit 组件

基于 house of spirit 第 1 攻击特征检测算法,实现 Is_HouseOfSpirit 检测组件的相关伪代码如下:

Heapinit ← GetHeapinit()

Achunk ← MainAllocatedChunk()

Fpointer ← MainFreedChunkPointer()

FP ← GetAddMainFreedChunkPointer()

if FP in Achunk:

Del(MainAllocatedChunk(), Getchunk(FP))

Del(MainAllocatedChunkPointer(), FP)

else:

FP.Is_HouseOfSpirit = true

Add(HouseOfSpiritPointer(), FP)

Add(HouseOfSpiritChunk(), Getchunk(FP))

3.4.2 Is_FakeHeapBlock 组件

基于 house of spirit 第 2 攻击特征检测算法,实现 Is_FakeHeapBlock 检测组件的伪代码如下:

Heapinit ← GetHeapinit()

main_arena ← Heapinit.main_arena

bins ← GetBins()

P ← GetAddMainFreedChunkPointer()

if P in HouseOfSpiritPointer:

Pchunk ← Getchunk(P)

size ← Pchunk.size

if Pchunk in Bins(size):

P.Is_FakeHeapBlock = true

else:

Del(HouseOfSpiritPointer(), P)

Del(HouseOfSpiritChunk(), Pchunk)

3.4.3 Is_PtrContorl 组件

基于 house of spirit 第 3 攻击特征检测算法,实现 Is_PtrContorl 检测组件的伪代码如下:

```
Heapinit <—GetHeapinit()
P <—GetAddMainAllocatedChunkPointer()
PA <—GetMainAllocatedChunkPointer()
if P in HouseOfSpiritPointer&& P.Is_FakeHeapBlock:
    Pchunk <—GetHouseOfSpiritChunk(P)
    Heapchunkdump(Pchunk)
    PAchunk <—GetChunk(PA)
    Heapchunkdump(PAchunk)
    if Pchunk != PAchunk:
        P.Is_PtrContorl = true
        shellcode <—GetPointer(PAchunk)
    else:
        Del(HouseOfSpiritPointer(),P)
        Del(HouseOfSpiritChunk(),Pchunk)
```

4 测试与分析

测试分为堆信息提取测试和 house of spirit 攻击检测测试两部分,堆信息提取测试是为了验证插件能否在内存空间中有效提取堆的相关信息,house of spirit 攻击检测测试是为了验证插件是否能有效检测出 house of spirit 攻击。实验环境如下:

主机操作系统为 Linux 内核版本 4.9.5 – 5.7.17 64 位,CPU 为 2.20 GHz,内存大小 8 G,硬盘容量 2 T。

4.1 堆信息提取测试

选取 init 进程和 gnome-keyring-d 进程作为实验对象,在 4.9.5 版本、5.3.12 版本、5.7.17 内核版本的 Linux 系统中运行 gnome-keyring-d 程序,随后分别对系统内存进行转储生成转储文件,使用本文研发好的 3 个堆信息提取插件分别提取 init 进程和 gnome-keyring-d 进程中的堆信息。

以 init 进程 pid = 1 为例,图 10 显示了 Heapinit 插件的运行来获取 Main Arena 的位置及其定位的主堆第一个堆块位置及信息。

```
===== Arena and heap_info objects =====
Main_arena: [malloc_state next] @ 0x7F6B23641B00
0x00 mutex          [int:mutex]: 0x00000000
0x04 flags          [int:flags]: 0x00000000
0x08 fastbinsY      <Array 10 x Pointer @ 0x7F6B23641B08>
0x58 top            <malloc_chunk Pointer to [0x557470056e
0x60 last_remainder <malloc_chunk Pointer to [0x55746ffce5
0x68 bins           <Array 254 x Pointer @ 0x7F6B23641B68>
0x858 binmap        <Array 4 x unsigned int @ 0x7F6B23642
0x868 next          <malloc_state Pointer to [0x7f6b1c000
0x870 next_free     <malloc_state Pointer to [0x00000000]
0x878 attached_threads [long unsigned int:attached_threads]
0x880 system_mem    [long unsigned int:system_mem]: 0x00
0x888 max_system_mem [long unsigned int:max_system_mem]:
First chunk: [malloc_chunk malloc_chunk] @ 0x55746FF4B000
0x00 prev size     [long unsigned int:prev size]: 0x00000000
```

图 10 插件提取的 Main Arena 信息

Fig. 10 Main Arena information extracted by the plug-in

同样以 init 进程 pid = 1 为例,图 11 显示了 Heapchunkinfo 插件获取的进程的部分已分配堆块的位置信息。

```
[malloc_chunk malloc_chunk] @ 0x55746FF59AF0
0x00 prev_size     [long unsigned int:prev_size]: 0x00000000
0x08 size          [long unsigned int:size]: 0x00000021
0x10 fd            <malloc_chunk Pointer to [0x7f6b2300732d (n
0x18 bk            <malloc_chunk Pointer to [0x7f6b23641b58 (n
0x20 fd_nextsize   <malloc_chunk Pointer to [0x5574700067d0 (n
0x28 bk_nextsize   <malloc_chunk Pointer to [0x000000031] (bk_r

[malloc_chunk malloc_chunk] @ 0x55746FF59B10
0x00 prev_size     [long unsigned int:prev_size]: 0x557470000
0x08 size          [long unsigned int:size]: 0x00000031
0x10 fd            <malloc_chunk Pointer to [0x207375422d44]
0x18 bk            <malloc_chunk Pointer to [0x4d206d657473]
0x20 fd_nextsize   <malloc_chunk Pointer to [0x422065676173]
0x28 bk_nextsize   <malloc_chunk Pointer to [0x55746e44d100 (n

[malloc_chunk malloc_chunk] @ 0x55746FF59B40
0x00 prev_size     [long unsigned int:prev_size]: 0x00000090
```

图 11 提取的已分配堆块信息

Fig. 11 allocated chunk information extracted

表 1 显示对于不同进程,Heapchunkinfo 插件运行后收集到的堆块相关信息。

表 1 Heapchunkinfo 插件提取的数据
Tab. 1 The data extracted by Heapchunkinfo plug-in

进程名	版本	Arena	分配堆块	空闲堆块	分配堆块大小	分配堆块大小
init	4.9.5	2	4 076	74	771 776	83 536
	5.3.12	2	5 108	106	862 558	96 754
	5.7.17	2	4 095	89	812 384	91 646
gnome-keyring-d	4.9.5	1	141	37	19 648	13 216
	5.3.12	1	106	26	16 324	9 878
	5.7.17	1	122	41	18 258	12 468

从表1中可以看到该插件可以有效地提取出不同版本Linux系统不同进程用户空间进程堆中的堆块相关信息。

实验结果表明本文研发的插件能成功地提取出不同Linux版本的用户空间进程堆信息。随着Linux系统的更新,用户空间进程堆及其结构中字段位置发生了一些变化,当使用vtype描述信息解析堆时,根据信息名称就能进行解析,因此本文研发的插件具有较强的兼容性。

4.2 堆溢出攻击检测测试

House_HiJack使用堆信息提取插件所收集的信息。检测是否触发house of spirit攻击时存在的3个攻击特性:Is_HouseOfSpirit、Is_FakeHeapBlock和Is_PtrContorl。如果该程序只包含某些特征,则意味着该程序包含house of spirit攻击,但危害程度较低,或者该程序不包含house of spirit攻击。如果该程序包含所有三个攻击特征,则意味着该程序不仅包含house of spirit攻击,还包含危险程序。

4.2.1 house of spirit 攻击检测测试

以house_of_spirit_text程序的自动检测和追踪为例,演示House_HiJack的检测过程。图12显示了house_of_spirit_text测试程序的关键触发代码。

```
bin[0] = malloc(40);
read(handle,input,100);
//An overflow vulnerability exists.
//Modify the value of bin[0] through input
free(bin0);
bin1 = malloc(40);
```

图12 程序的关键触发代码息

Fig.12 The key trigger code of the program

如代码所示,将数据写入输入指针时存在溢出漏洞。可以溢出并修改bin[0]的值,以写入指向任何内存地址的输入指针。然后通过输入数据来构造1个伪堆块结构,当程序释放bin[0]时,会触发house of spirit攻击。此时,当程序应用相同大小的堆块时,可以将目标内存作为堆块结构应用,攻击者可以修改关键指针的数据,以达到任意地址写入的目的。

通过监视程序即将释放的堆指针。当堆指针即将释放时,Is_HouseOfSpirit组件触发第1攻击特性检测算法以检查指针变化。如果释放指针不属于已知已分配堆块,则触发house of spirit第1攻击特征。如图13所示。检测到了house of spirit第1攻击特征。

在检测到house of spirit第1攻击特征后,在程

序内存空间中搜索堆内存,若发现伪造堆块则检测到house of spirit第2攻击特征。识别的伪造堆块如图14所示。

```
taint-var introduce:range:[0,100]
54 [state 0] Found Modified Pointer at 0x804a038
The Pointer can be modified,and IS_HouseOfSpirit
= ture
```

图13 检测到house of spirit第1攻击特性

Fig.13 House of Spirit First attack feature detected

```
Fake Chunk 1
0x0804a038 0x0000000000000000 0x0000000000000030
0x0804a048 0x6161616161616161 0x0000000000000000
0x0804a058 0x6161616161616161 0x0000000000000030
0x0804a068 0x0000000000000000 0x0000000000001234

The Fake Chunk at 0x0804a038,and Is_FakeHeapBlock
= ture
```

图14 检测到house of spirit第2攻击特性

Fig.14 House of Spirit Second attack feature detected

在检测到house of spirit第2攻击特征后,将检查关键指针数据的可控性。如果指针数据是符号值,则表示关键指针数据发生了变化,追踪控制流。指针数据变化检查结果如图15所示。

```
Fake Pointer
0x804a058 0x00000018A00408 0x0000000000000000

Pointer 0x804a058 can be overwritten!and Is_PtrCo
ntorl = ture
```

图15 检测到house of spirit第3攻击特性

Fig.15 House of Spirit Third attack feature detected

4.2.2 house of spirit 攻击检测测试结果

本文从shellphish / how2heap测试集中选取5个具有堆溢出漏洞的实验程序进行验证。

为了更好地反映系统的效果,比较了插件与现有漏洞攻击检测技术的差异。在实验期间,每个测试程序都提交给House_HiJack和CRAX系统进行测试。

CRAX系统是SK. Huang等提出的一种基于动态符号执行和着色分析技术研发的house of spirit类型攻击检测工具^[20]。通过监控程序的动态运行过程来检测IP劫持的状态,使用动态符号执行和着色分析技术来监视程序运行过程中的程序控制流和数据流状态信息,对触发house of spirit漏洞触发点的符号数据的关键信息进行监控,构造触发house of spirit攻击的数据约束,并求解约束条件,可有效的对该类型攻击进行检测。

测试用例生成如表2所示。其中 t_1 表示系统

完成实验样本分析与检测所需的时间, t_2 表示系统完成控制流追踪所需的时间。

表 2 House_HiJack 和 CRAX 比较
Tab. 2 Comparison of House_HiJack and CRAX

测试程序	House_HiJack		CRAX	
	t_1/s	t_2/s	t_1/s	t_2/s
house_of_spirit	1.9	—	1	—
example	1.7	—	0.9	—
pwn200	5.7	6.9	3.4	—
shaxian	4.7	5.3	2.3	—
house_of_spirit_text	1.9	2.4	0.8	—

表 2 中的结果表明, CRAX 系统和本文研发的系统都可以成功检测全部 5 个 house of spirit 类型的堆溢出攻击。

通过比较 t_1 与 t_2 的数据可以看出。CRAX 系统对测试程序的分析时间短于本文提出的检测方法, 但 CRAX 系统无法为所有五个测试程序追踪控制流。House_HiJack 系统为其中 3 个程序成功追踪控制流代码。

这表明 CRAX 系统虽然能快速的检测其是否为 house of spirit 类型堆溢出攻击, 但无法定位其最终的自由指针变化过程, 无法追踪其 shellcode。而本文研发的系统可以成功定位该类型攻击的控制流, 通过控制流获取 shellcode。这表明本文提出的检测方法是有效的。

House_HiJack 插件对每个测试程序的检测过程进行详细记录和分析, 结果如表 3 所示。

表 3 House_HiJack 测试程序检测过程
Tab. 3 House_HiJack test procedure detection process

测试程序	Is_House	Is_Fake	Is_Ptr
	of Spirit	Heap Block	Contorl
house_of_spirit	1	1	0
example	1	1	0
pwn200	1	1	1
shaxian	1	1	1
house_of_spirit_text	1	1	1

从表 3 可以看出, 对于 pwn200、shaxian 和 house_of_spirit_text 3 个测试用例, 插件可以检测到 house of spirit 第 1 攻击特征、第 2 攻击特征和第 3 攻击特

征 3 个关键特征。因此, 可以在 3 个攻击特征检测的同时追踪构造相应的伪堆块和关键指针数据变化, 从而追踪控制流, 以获得最终的 shellcode 代码。house_of_spirit 和 example 测试用例只能检测 house_of_spirit 第 1 攻击特征和第 2 攻击特征, 但无法检测到第 3 攻击特征, 表明测试用例已经实现了对要释放的指针数据的修改, 并伪造了堆块结构, 但没有进一步的指针利用, 因此无法追踪控制流。

5 结 论

本文提出了一种在 Linux 下检测 house of spirit 类型攻击的技术。该技术利用动态追踪用户空间进程堆信息。为了提取出 Glibc 堆中的信息, 本文分析了 Glibc 在内存中的堆实现, 堆结构中含有的字段并结合内存取证框架和字段在结构信息中的偏移量, 设计出提取 Glibc 堆及其内部信息的功能插件, 这些插件可以解析 Linux 系统中用户空间进程堆内部含有的信息。

在此基础上, 提出了一种 house of spirit 型堆溢出漏洞自动检测模型。基于该模型, 实现了 house of spirit 型堆溢出漏洞的检测与控制流追踪。

实验结果表明本文提出的方法可以重现进程运行时堆及其内部组件在内存中的分配情况, 进而反映进程中堆内存信息, 通过利用这些信息成功的检测出 Linux 平台下 house of spirit 类型攻击的存在并追踪控制流。

参 考 文 献:

[1] MOSLI R, LI R, YUAN B, et al. Automated Malware Detection Using Artifacts in Forensic Memory Images [C]//2016 IEEE Symposium on Technologies for Homeland Security (HST). USA :IEEE, 2016:1.

[2] JEONG D, LEE S. Forensic Signature for Tracking Storage Devices: Analysis of UEFI Firmware Image, Disk Signature And Windows Artifacts[J]. Digital Investigation, 2019, 29:21.

[3] CASE A, RICHARD G. Memory forensics: The path forward[J]. Digital Investigation, 2017, 20:23.

[4] 张瑜, 刘庆中, 李涛, 等. 内存取证研究与进展[J]. 软件学报, 2015, 26(5):1151.

ZHANG Yu, LIU Qingzhong, LI Tao, et al. Research and Development of Memory Forensics [J]. Journal of Software, 2015, 26(5):1151.

[5] HEDBERG T D, KRIMA S, CAMELIO J A. Embedding

- X. 509 Digital Certificates in Three-Dimensional Models for Authentication, Authorization, and Traceability of Product Data [J]. Journal of Computing & Information Science in Engineering, 2017, 17(1): 011008.
- [6] KIM D, KWON B J, DUMITRA T. Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI [C]//2017 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2017: 1435.
- [7] UGARTE-PEDRERO X, GRAZIANO M, BALZAROTTI D. A Close Look at a Daily Dataset of Malware Samples [J]. ACM Transactions on Privacy and Security, 2019, 22(1): 1.
- [8] 翟继强, 陈攀, 徐晓, 杨海陆. 面向 Windows 10 系统段堆的内存取证研究 [J]. 西北工业大学学报, 2021, 39(5): 1139.
- ZHAI Jiqiang, CHEN Pan, XU Xiao, Yang Hailu. Research on Memory Forensics for Windows 10 System Segment Heap [J]. Journal of Northwestern Polytechnical University, 2021, 39(5): 1139.
- [9] Description of User Account Control and Remote Restrictions in Windows Vista [EB/OL]. (2020-9-18) [2020-12-09].
- [10] Sigcheck v2.81 [EB/OL]. (2020-9-17) [2020-12-09].
- [11] UROZ D, RICARDO J. On Challenges in Verifying Trusted Executable Files in Memory Forensics [J]. Digital Investigation, 2020, 32: 300917.
- [12] SYLVE J T, MARZIALE V, RICHARD G G. Pool Tag Quick Scanning for Windows Memory Analysis [J]. Digital Investigation, 2016, 16: 25.
- [13] 翟继强, 徐晓, 陈攀, 等. 基千元数据和指令流的 64 位 Windows 堆栈取证 [J]. 哈尔滨理工大学学报, 2021, 26(5): 51.
- ZHAI Jiqiang, XU Xiao, CHEN Pan, et al. Stack Forensics Based on Meta Data and Instruction Flow of 64-bit Windows [J]. Journal of Harbin University of Science and Technology, 2021, 26(5): 51.
- [14] ZHANG N, ZHANG R, SUN K, et al. Memory Forensic Challenges Under Misused Architectural Features [J]. IEEE Transactions on Information Forensics & Security, 2018, 13(9): 2345.
- [15] ZHU W T, LIN J. Generating Correlated Digital Certificates: Framework and Applications [J]. IEEE Transactions on Information Forensics & Security, 2016, 11(6): 1117.
- [16] PANDEY A K, TRIPATHI A K, KAPIL G, et al. Current Challenges of Digital Forensics in Cyber Security [J]. Critical Concepts, Standards, and Techniques in Cyber Forensics, 2020: 31.
- [17] PARIDA T, DAS S. PageDumper: a Mechanism to Collect Page Table Manipulation Information At Run-time [J]. International Journal of Information Security, 2021, 20(4): 603.
- [18] VOLATILITY: An Advanced Memory Forensics Framework [EB/OL]. (2020-11-11) [2021-6-18].
- [19] ZHAO X, HUANG S, PAN Z, et al. Buffer Overflow Vulnerability Detection Based on Unsafe Function Invocation [J]. Journal of Physics: Conference Series, 2020, 15(9): 1145.
- [20] 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]// In 2012 IEEE Sixth International Conference on Software Security and Reliability, IEEE, 2012: 78.

(编辑: 温泽宇)