Glibc 堆利用的若干方法

裴中煜, 张 超, 段海新

清华大学 网络科学与网络空间研究院, 北京 中国 100084

摘要 内存中的攻击与防御一直是系统安全领域的重要研究课题之一, 而堆内存破坏漏洞利用以及防护以其特殊的性质在这场宏大的攻防战中扮演着十分独特的角色。由于堆内存粒度细的缘故, 众多系统级的防御难以起到良好的效果。同时, 堆内存因受运行时输入影响, 行为表现难以预测, 也给破坏的侦测增加了难度。本文收集了近年来堆内存攻防领域的优秀博文与论文, 回顾了早期的 glibc 堆利用方法以及之后 glibc 与系统针对它们的修复方式。之后, 本文介绍了近年来流行的针对 glibc 堆的现代利用方法以及它们绕过系统保护的方式。最后, 本文根据所有的利用方法进行了堆利用特点的总结与梳理, 针对它们的特性提出相应的减缓措施, 并预测了未来堆攻防博弈发展可能的趋势。

关键词 运行时库; 堆; 利用; 分配算法
中图法分类号 TP301 DOI 号 10.19363/j.cnki.cn10-1380/tn.2018.01.001

Several Methods of Exploiting Glibc Heap

PEI Zhongyu, ZHANG Chao, DUAN Haixin

Institute for Network Science and Cyberspace, Tsinghua University, Beijing 100084, China

Abstract Attack and defense in memory has been an important research topic in the field of security systems for several decades. The heap memory corruption exploitation and protection play a very unique role in this grand arm race due to its special characteristic. Since the heap memory is of tiny granularity, many system-level defense methods are difficult to take good effect. Meanwhile, heap memory, affected by runtime user input, behaves unpredictably, which raises the difficulty of corruption detection. In this paper, we collect the outstanding blogs and thesis on heap memory arm race, and review the earlier exploit methods as well as the patch methods glibc has taken to defense them. Then, we introduce a few modern exploit methods against glibc and mitigations of system. At last, we conclude the key point of heap exploitation, put forward some mitigations against them, and predict the possible trend of heap exploitation in the future.

Key words Glibc; heap; exploitation; ptmalloc

1 引言

从 1996 年 Aleph1 发表的第一个被广泛应用的栈溢出文章[1]以来, 安全人员和黑客在内存中的较量就从未停止。从经典的栈溢出开始, 黑客们开发出众多漏洞利用的手段来对目标进行攻击, 随着对于二进制程序研究的深入, 利用的目标渐渐转向了另一块敏感的动态数据区域——堆。

与栈这种汇编级别的数据区域不同, 堆实现在系统层级, 每个系统都有自己的堆实现。我们研究的对象是 glibc 的堆实现, GNU Libc 是 Linux 操作系统的标准 C 语言库, 提供了系统调用的封装以及各种 I/O 的 API, 当然也包括动态数据——堆的实现。Glibc 作为程序运行所必须的底层运行时库被部署在所有的 Linux 发行版中, 其重要地位不言而喻。而正因为如此, glibc 堆也一直是这场攻防博弈的重灾区。

传统的栈溢出变得困难的原因, 在于多种系统级减缓措施(NX[2], ASLR[3], 等等)的开发[4], 尤其是 stack canary[5]的发明, 极大地提高了栈溢出漏洞利用的难度。而相对地, 关于堆溢出的利用则变得活跃, 因为堆数据的布局并不是编译时确定的, 而是根据运行时的状态动态确定的, 其分配和管理的难度都要比栈数据大。数十年来, 不断地有新的利用方式被提出, 同时 glibc 也在针对性地进行修补。

遗憾的是, 对于堆利用的研究新发现多见于个人博客以及商业、民间的研究机构, 且缺乏有效的总

通讯作者: 张超, 博士, 副教授, Email: chaoz@tsinghua.edu.cn。
本课题得到清华信息科学与技术国家实验室(筹)面上研究项目, 国家自然科学基金面上项目 61772308 资助。
收稿日期: 2017-07-18; 修改日期: 2017-10-03; 定稿日期: 2017-12-05

结和整理, 也没有能够很好地引起人们的重视。2015 年爆发的 Glibc 中的 GHOST 漏洞[6], 使得攻击者能够绕过所有现存的保护措施, 仅通过一封恶意邮件就能够攻陷远程的 Exim 邮件服务器, 造成这一结果的根源, 仅仅是在 gethostbyname 函数的实现中对于缓冲区长度的一个错误计算所致。由此可见, 我们需要全面地掌握当前 Glibc 堆上攻防双方的博弈形势, 才能够更好地作出针对性的防御措施, 降低被利用的可能性。

具体来说, 本文完成了以下几个方面的工作:

- 通过对流行的 glibc 2.19 版本的 malloc 源代码进行分析, 深入理解 Glibc 堆管理机制的实现特点以及可能发生的问题;
- 回顾了历史上经典的堆溢出利用方法及其局限性; 以及现代堆利用手段绕过 Linux 系统以及 glibc 检查等相关防护的方式;
- 总结了 glibc 堆利用的特点以及其共性条件, 并针对它们的本质特征提出进一步的防护方案。

文章结构安排如下, 第 2 章将根据源码简述流行的 glibc 2.19 版本堆管理机制; 第 3 章将回顾早期的堆利用手段; 第 4 章介绍当前操作系统中广泛使用的防护手段以及 glibc 新版本中采用的检查; 第 5 章介绍现代的堆利用方法; 第 6 章讨论当前防护手段的缺陷以及堆利用的共性特征; 第 7 章总结并展望未来堆攻防博弈的趋势。

2 Glibc Heap 管理机制

Glibc 的堆分配器是基于 Doug Lea 早期的 dlmalloc 分配器发展而来的 ptmalloc2 分配器, 具有速度快、碎片度低、线程安全的特点。本章以当前流行的 glibc 2.19 版本为例, 说明其堆的分配机制。

2.1 相关结构

2.1.1 堆块结构

堆块块(Chunk), 是 ptmalloc2 进行分配的基本单位, 其头部保存着自身的大小等信息, 在这之后即是用户数据。一共有 3 种类型, 它们都使用同一种数据结构(malloc_chunk)定义。

```
struct malloc_chunk {
  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T  prev_size;
  /* Size in bytes, including overhead. */
  INTERNAL_SIZE_T  size;
  /* double links -- used only if free. */
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  /* double links -- used only if free. */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

在 32 位系统中, 堆块的大小最小为 16 字节, 对齐 8 字节; 而在 64 位系统中, 堆块的大小最小为 32 字节, 对齐 16 字节。在本文之后的讨论中, 未做特别说明则默认为 64 位环境。

具体来说, 用于管理的堆块主要分为 3 种类型。

- Allocated Chunk: 已分配块, 如图 1 所示, 只使用 prev_size 与 size 这 2 个域, 用来记录上一块大小以及自身的大小, 剩余部分则为用户数据。

图 1 已分配块
Figure 1 Allocated Chunk

- Free Chunk: 被释放块, 如图 2 所示, 另外使用 fd、bk 2 个域。

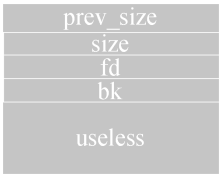

图 2 被释放块
Figure 2 Freed Chunk

- Top Chunk: 顶块, 位于所有块之后, 保存着未分配的所有内存, 与已分配块使用相同的域。

除此之外, 由于块的大小是对齐的, 使得低位字节不会使用到, 故 glibc 使用 size 域的最低 3 位来存储一些其它信息。相关的掩码信息定义如下:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
#define NON_MAIN_ARENA 0x4
```

从以上代码定义可以推断, size 域的最低位表示此块的上一块(表示连续内存中的上一块)是否在使用状态, 如果此位为 0 则表示上一块为被释放的块, 这个时候此块的 PREV_SIZE 域保存的是上一块的地址以便在 free 此块时能够找到上一块的地址并进行合并操作。第 2 位表示此块是否由 mmap 分配, 如果此位为 0 则此块是由 top chunk 分裂得来, 否则是由 mmap 单独分配而来。第 3 位表示此块是否不属于 main_arena, 在之后会提到 main_arena 是主线程用于

保存堆状态的结构, 如果此位为 0 则表示此块是在主线程中分配的。

2.1.2 Bins 结构

当一个分配块被执行 free 操作后, glibc 将其按照一定规则放入 bin 结构中, 以便下次分配时能够再次利用。bin 实际上即是链表结构, 利用 fd 与 bk 指针进行组织, 根据块的大小不同分为不同的 bin 结构。

● Fast Bins: chunk 的指针数组, 每个元素是一条单向链表的头部, 且同一条链表中块的大小相同。主要保存大小 32 至 128 字节的块, 特点是当 free 时不取消下一块的 PREV_INUSE 位, 也不检查是否能够进行合并操作, 主要目的是能够最快速地利用较小的内存块。由于是单向链表, 故 Fast bins 的取用机制是 LIFO (Last In First Out) , 即后释放的块将先被利用。

● Small Bins: chunk 的指针数组, 每个元素是一条双向循环链表的头部, 且同一条链表中块的大小相同。主要保存大小 32 至 1024 字节的块。由于是双向链表, Small Bins 的取用机制是 FIFO (First In First Out) , 即先释放的块会先被利用, 之后的 Large Bins 与 Unsorted Bins 也是同样的机制。

● Large Bins: chunk 的指针数组, 每个元素是一条双向循环链表的头部, 但同一条链表中块的大小不一定相同, 按照从大到小的顺序排列, 每个 bin 保存一定大小范围的块。主要保存大小 1024 字节以上的块。

● Unsorted Bins: 与 Small Bins 和 Large Bins 类似是双向循环链表, 只有一个 bin, 其中保存的块大小不定, 用于收集刚刚被 free 或从大的块中分裂剩下的块。

2.1.3 Arena 结构

当前主线程的堆分配状态是由 glibc 中的全局变量 main_arena 保存的, 这是一个 malloc_state 类型的结构体。而 malloc_state 结构体的部分定义如下:

```
struct malloc_state
{
  ...
  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];
  ...
};
```

我们关心的部分有以下几个域:

● fastbinsY: 保存 Fast Bins 的数组

● top: 保存 Top Chunk 的地址

● last_remainder: 保存上一次分裂的块

● bins: 其中下标为 1 的元素是 unsorted bin, 之后的 bins 从小到大对应 small bins 与 large bins, 下标为 0 的元素不用。

2.2 分配函数

Malloc 函数为 glibc 的主要分配接口, 给出需要分配的大小参数, 返回值为分配得到的用户数据指针。主要的功能由_int_malloc 函数实现。

在第一次执行 malloc 函数时, 系统会使用 brk 系统调用向操作系统扩展程序的数据区, 此时 glibc 将初始化 top chunk 与 main_arena, 取得 132KB 的空间。如果之后所有的 malloc 操作都可以满足, 即最后总是能在此 132KB 中找到合适的内存块返回, 则不再使用系统调用与内核交互。这段时间, 程序的堆内存由 glibc 管理。否则, 将使用 brk 或 mmap 系统调用来向内核申请更多空间。

当程序将需要的空间大小传入 malloc 时, glibc 首先将其加上 8 字节的额外开销(用于保存 size 域, 因为 prev_size 域实际占用的是上一块的空间故不算额外开销)然后对齐 16 字节, 如果不足 32 字节则分配 32 字节。接下来我们的叙述中, 请求大小皆指已经处理之后对齐 16 字节的大小。

2.2.1 检查 Fast bins

如果请求大小满足 Fast bins, 则在对应的 bin 中寻找是否有相同大小的块, 如果有则直接将其取出返回给程序, 同时更新 fast bins 中对应 bin 存储的链表头指针。

2.2.2 检查 Small bins

如果块大小符合 Small bins, 则在对应大小的 Small bin 中寻找是否有合适的块, 如果有则直接返回, 同时更新 Small bins 中对应 bin 的链表中该块的上一块和下一块的指针。

对于 Fast bins 和 Small bins 来说, 每个 bin 中的块大小都是相同的, 所以只要对应的 bin 中有块, 就能够直接返回恰好符合的块。

2.2.3 处理 Unsorted bin

如果之前没能返回恰好符合的块, 则开始处理 Unsorted bin 中的块(这里有一个例外, 如果 Unsorted bin 中只有一个块且这个块是 last_remainder, 而且大小足够, 则优先使用此块, 分裂后将前一块返回给用户, 剩下的一块作为新的 last_remainder 再次放入 Unsorted bin.[7])

具体来说, 处理循环如下:

a) 逐个迭代Unsorted bin中的块, 如果发现块的大小正好是需要的大小, 则迭代过程中止, 直接返回此块;否则将此块放入到对应的 Small bin 或者 large bin 中, 这也是整个 glibc 堆管理中唯一会将块放入 Small bins 与 large bins 中的代码。

b) 迭代过程直到 Unsorted bin 中没有块或超过最大迭代次数(10000)为止。

c) 随后开始在 Small bins 与 large bins 中寻找最合适的块(指大于请求大小的最小块), 如果能够找到, 则分裂后将前一块返回给用户, 剩下的块放入 Unsorted bin 中。

d) 如果没能找到, 则回到开头, 继续迭代过程, 直到 Unsorted bin 空为止

2.2.4 使用顶块

如果之前的操作都没能找到合适的块, 将分裂 Top chunk 返回给用户, 若 Top chunk 的大小仍然不足, 则再次执行 malloc_consolidate 函数清除 Fast bins, 若 Fast bins 已空, 只能使用 sysmalloc 函数借助系统调用拓展空间。

2.3 释放函数

Free 函数是 glibc 的释放接口, 将之前分配得到的用户内存指针作为参数, glibc 会释放这一块空间。主要的功能由_int_free 函数实现。

首先, 进行一系列的检查, 包括内存地址对齐, PREV_INUSE 位是否正确等等, 能够发现一些破坏与 double free 的问题。

如果块大小满足 Fast bins, 则不取消下一块的 PREV_INUSE 位, 也不设置下一块的 prev_size 域, 直接将该块放入对应的 Fast bin 中, 且不进行相邻块的合并操作。

检查被 free 内存块的前一块(这里的前一块指连续内存中的上一块, 通过 prev_size 域来寻找), 如果未使用, 则合并这 2 块, 将前一块从其 bin 中移除之后再检查其后一块, 如果发现是 Top chunk, 则最后将合并到 Top chunk 中, 不放入任何 bin; 如果不是 Top chunk 且未使用, 则再合并这 2 块, 将后一块从其 bin 中移除, 并且将合并过的大块放入 Unsorted bin 中。

2.4 重分配函数

realloc 是一项复合操作, 既要给出之前分配的用户内存指针, 又要传入需要的大小数值, 旨在重新分配这块空间以符合用户新的需求。在执行具体操作之前, 同样会先将用户传入的大小参数进行处理以对齐 16 字节。主要功能由_int_realloc 函数实现。

若发现之后需要的大小比之前的大小更小, 则直接对此块进行压缩操作, 分裂出的部分如果达到块的最小大小(32 字节), 则调用_int_free 函数释放此块。

若发现已分配的块后一块是 Top chunk(这里的后一块指的是连续内存中的下一块, 通过 size 域来寻找), 则直接向 Top chunk 中扩展一部分空间, 返回的指针与之前传入的指针相同。

若发现下一块已经被 Free, 且下一块的大小能够满足新的需求大小, 则向下一块中扩展, 使用 unlink 宏将下一块从对应的 bin 中移除, 扩展完成后再对剩下的块调用_int_free。返回的指针与之前传入的指针相同。

若无法向下一块扩展, 则直接调用_int_malloc 分配新的堆块, 然后把之前堆块中的用户数据复制到新的堆块中, 最后对之前的块调用_int_free 函数。

2.5 堆管理的特点

可以看到, glibc 在堆管理方面使用了很多技巧, 最终目的都是为了能够加快分配速度、降低碎片率、更好更合理的利用堆内存区域。而这样一来, 安全性便成为其最大的问题, 由于内存块的元数据与用户数据交错布置, 导致块的元数据很容易遭到破坏, 如果程序中有缓冲区溢出漏洞更是可以进一步的利用。

3 早期的利用技术

2002 年, glibc 还处于 2.3 版本[9], 其堆的实现缺乏安全性考虑, 一些安全检查没有被引入。并且很多系统级的保护措施还没有广泛应用, 导致早期的攻击者能够很容易地利用程序的漏洞实现任意命令执行。本章将以早期的 glibc 2.3 版本为例, 介绍堆利用时攻击者常用的攻击面, 以及经典的堆块溢出漏洞、UAF 漏洞的利用方式。最后介绍 Phantasmal Phantasmagoria 提出的利用堆管理自身的机制来欺骗 glibc 来进行巧妙利用的 5 种方法[12], 并总结早期攻击技术的特点。

3.1 利用目标

对堆利用来说, 不同于栈上的溢出能够直接覆盖函数的返回地址从而控制 EIP, 只能通过间接手段来劫持程序的控制流, 利用程序中的函数指针是一个常用的手段, 典型的方法有控制程序的 GOT 表, glibc 当中的钩子函数指针等。

3.2 利用解链操作

如果同一个堆指针被释放 2 次, 则可能造成两次释放(Double Free)漏洞。此方法的原理是利用 glibc 中的链表解链操作 unlink 宏来实现内存地址的修改。早期的 glibc 在实现 unlink 宏时没有进行任何检查。

具体代码如下:

```
#define unlink(P, BK, FD) {                                    \
    FD = P->fd;                                                \
    BK = P->bk;                                                \
    FD->bk = BK;                                               \
    BK->fd = FD;                                               \
}
```

根据 fd 和 bk 指针在 malloc_chunk 结构体中的位置, 该段代码的作用等同于:

$$*(fd+24)=bk$$

$$*(bk+16)=fd$$

因此, 如果堆中存在缓冲区溢出漏洞, 则可以通过修改块 P 的 fd 与 bk 指针来向指定内存中写入任意数据, 攻击过程如图 3-4 所示。

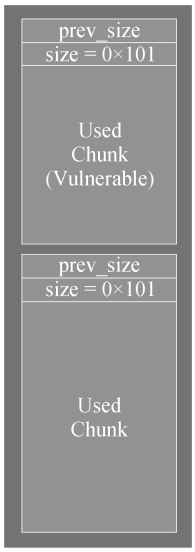

图 3　初始时, 堆中有 2 个已经分配的内存块, 大小都为 256 字节, 但第一块中存在溢出漏洞, 故可以覆盖下一块的元数据

Figure 3　Initial State

如此一来, 攻击者通过控制 fd 和 bk 的值, 可以将一个指针指向自己期望的地址。假设 *ptr* 的位置保存着攻击者的 shellcode(攻击者自己编写的汇编代码), 而 *victim* 的位置是攻击者希望控制的地址(例如 free 的 GOT 表项), 攻击者可以将 fd 与 bk 的值伪造成:

$$fd = victim - 24$$

$$bk = ptr$$

这样一来经过 unlink 宏的处理, 将会得到如下效果:

$$*(fd+24)=*(victim-24+24)=*victim=bk=ptr$$

$$*(bk+16)=*(ptr+16)=fd=victim-24$$

结果是 *victim* 处的指针被指向了 *ptr* 的位置, 虽然 *ptr+16* 的位置遭到了破坏, 不过攻击者可以通过

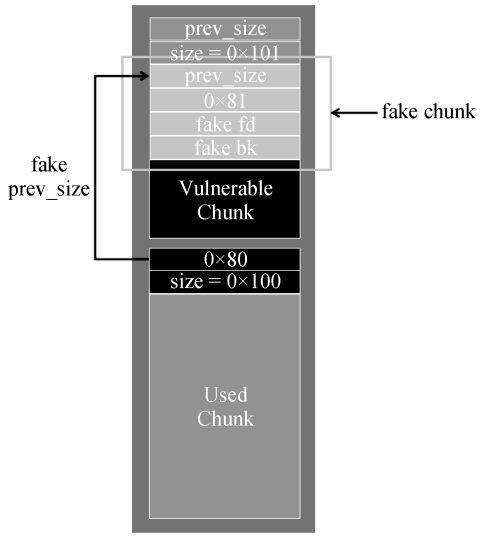

图 4　攻击者在上一块的内部伪造一个假的已经被释放的内存块, 同时将下一块的 prev_inuse 这个 flag 清 0, 并将其 prev_size 修改为指向自己伪造的被释放块。如此以来在对下一块执行 free 操作时, glibc 发现上一块已经被释放, 则会触发 unlink 宏来将伪造的块从对应的 bin 中解链

Figure 4　Unlink Attack

使用 jmp 指令跳转到其后执行即可。

3.3　释放后使用

由于内存块被释放后, 所在的内存仍是可以访问的, 如果程序编写时使用了已经被释放的内存, 就有可能导致释放后使用(*Use After Free*, 简称 *UAF*)的漏洞。

以下面的程序为例:

```
1  #include <stdio.h>
2
3  struct funcset {
4      void (*funca)();
5      void (*funcb)();
6      void (*funcc)();
7      void (*funcd)();
8  };
9
10 struct dataset {
11     size_t id;
12     char title[8];
13     char content[16];
14 };
15
16 int main(){
17     struct funcset *p =
           (*struct funcset)malloc(sizeof(struct funcset));
18     free(p);
19     struct *q = (*struct dataset)malloc(sizeof(struct dataset));
20     fgets(q -> content, 16, stdin);
21     p -> funcc();
22     return 0;
23 }
```

当第 8 行指针 p 被释放后, 在第 21 行又被使用,

在编译时不会发生错误, 但在使用时会造成严重问题。由于 funcset 结构体的大小为 32 字节, 对齐处理之后为 48 字节, 属于 Fast bin, 之后为指针 q 分配 32 字节时就会取得上次释放的 p 的地址, 而在第 20 行读入数据, 简介导致了用户可以控制第 21 行 funcc 函数调用的地址, 从而劫持程序的控制流。

3.4 The Malloc Maleficarum

2005 年, Phantasmal Phantasmagoria 发表了一篇文章 *The Malloc Maleficarum - Glibc Malloc Exploitation Techniques* [12], 介绍了利用 glibc 堆管理机制进行的一些漏洞利用的方法。其中介绍的 5 种利用 glibc 自身堆分配算法机制来进行利用的思路, 为之后攻击者开发新的利用方法以及 glibc 加固自身代码提供了参考。

3.4.1 The House of Prime

此方法是利用 glibc 在判断 Fast Bins 的下标时的一个疏忽来破坏 main_arena 中的数据, 在早期的 glibc 2.3.5 版本中, main_arena 结构与现在有所不同:

```
struct malloc_state {
  /* Serialize access.  */
  mutex_t mutex;
  // Should we have padding to move the mutex to its own cache line?
#if THREAD_STATS
  /* Statistics for locking.  Only used if THREAD_STATS is defined.  */
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
  /* The maximum chunk size to be eligible for fastbin */
  INTERNAL_SIZE_T  max_fast;  /* low 2 bits used as flags */
  /* Fastbins */
  mfastbinptr  fastbins[NFASTBINS];
...
```

而 glibc 在根据块大小取得 Fast bins 数组下标时使用的宏如下面所示:

```
#define fastbin_index(sz)  ((((unsigned int)(sz)) >> 3) - 2)
```

若此时的 sz 为 8, 则最后得到的下标为 (8 >> 3) – 2 = –1, 这就意味着若溢出时将下一块的 sz 修改为 8,则会造成 fastbins 数组的上一个域即 max_fast 变量遭到破坏。

但在 glibc 2.19 版本中, main_arena 的结构已经发生了变化, 虽然 *fastbin_index* 宏仍然存在漏洞, 但在 fastbins 数组的前面不再是 max_fast 变量, 此变量被移出结构体作为全局变量 global_max_fast 使用, 这样一来, 通过破坏前一个域来进行利用的方法已不可行。

3.4.2 The House of Mind

此方法利用块的 size 域中 NON_MAIN_ARENA 这个标识来进行利用, 若在溢出时修改下一块的 size 域并将其 NON_MAIN_ARENA 置为 1, 在 free 操作时可以触发如下宏的执行:

```
#define heap_for_ptr(ptr) \
 ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)
```

当 glibc 发现被释放块的 NON_MAIN_ARENA 设置为 1 时, 就会认为这个块属于某个子线程, 并使用 heap_for_ptr 宏来查找对应的 heap_info 结构, 如果能够控制 ptr 指针的地址并将 heap_for_ptr 宏的结果指向一块攻击者能够控制的区域, 攻击者就能够在那里伪造一个 heap_info 的结构, 其结构定义如下:

```
typedef struct _heap_info
{
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t mprotect_size; /* Size in bytes that has been mprotected
                           PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned, particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

可以看出, 此结构的第一个成员为 mstate 类型, 即某个子线程对应的 arena 结构, 此时攻击者通过伪造 heap_info, 亦可以进一步伪造出一个 arena 结构, 如此一来可以将其中的 Unsorted bin 指向任意的位置, 由于在 free 时块如果不是 Fast bins 在进行合并操作后会直接被放入 Unsorted bin, 如此一来便可以达到任意内存修改的目的。

此方法在当前流行的 glibc 中已不再可行, glibc 在 free 时都需要对 Unsorted bin 进行检查, 确认其中的第一个块的后向指针是否指向自身, 如此一来若攻击者将 Unsorted bin 指向希望控制的内存区域, 则会导致检查失败而无法利用。

3.4.3 The House of Force

此方法通过溢出 Top chunk 的 size 域来欺骗 glibc, 使得攻击者 malloc 一个很大的数目(负有符号整数)时能够使用 Top chunk 来进行分配, 此时 Top chunk 的位置会加上这个大数, 造成整数溢出, 结果是 Top chunk 能够被转移到堆之前的内存地址(如程序的.bss 段, .data 段, GOT 表等), 下次再执行 malloc 时, 就会返回转移之后的地址, 攻击者即能够控制其他的内存数据。

具体来说, 利用需要以下几个步骤(为了方便起

见, 这里以 32 位系统为例):

● 攻击者溢出 Top chunk 的上一块, 将 Top chunk 的大小修改为 0xffffffff。

● 攻击者需要能够控制下一次 malloc 时传入的大小, 此时传入 0xfffffec, 经过处理得到的大小是 0xfffffff0, glibc 的大小使用的是 size_t 类型存储, 这是一种与机器字长相等的无符号整形。故此时 glibc 认为 0xffffffff > 0xfffffff0, 即 Top chunk 的空间足够, 于是使用 Top chunk 进行分配。假设 Top chunk 之前的位置是 0x804a010 分配后, Top chunk 的地址将处于 0x804a010 + 0xfffffff0 = 0x804a000。

● 下次 malloc 时, 攻击者就能够取得 0x804a000 这块内存, 进一步修改就能够干扰程序的运行。

此方法的缺点在于, 会受到之后提到的 ASLR 技术的影响, 如果攻击者需要修改指定位置的内存, 他需要知道当前 Top chunk 的位置以构造合适的 malloc 大小来取得对应的内存。而 ASLR 开启时, glibc 的堆内存地址将会是随机的, 如果想要利用此项技术, 则需要进行信息泄露。

3.4.4 The House of Lore

此方法希望通过破坏已经放入 Small bins 中块的 bk 指针来达到取得任意地址的目的, 在早期的 glibc 2.3.5 版本的_int_malloc 中, 处理 Small Bins 的代码如下所示:

```
if (in_smallbin_range(nb)) {
  idx = smallbin_index(nb);
  bin = bin_at(av,idx);

  if ( (victim = last(bin)) != bin) {
     if (victim == 0) /* initialization check */
        malloc_consolidate(av);
     else {
        bck = victim->bk;
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;
        ...
        return chunk2mem(victim);
     }
  }
}
```

即如果请求的大小符合 Small Bins, 则在对应的 bin 中寻找是否有空闲的块, 如果有就直接从 bin 的链表中解链并返回给用户。

基于上面的机制, 当一个块存在于 Small Bin 的第一个块(这里的第一块指的是在取 Small Bin 块时选择的第一块, 即 Small Bin 的 bk 指针指向的那一块)时, 通过溢出修改其 bk 指针指向某个地址 ptr, 当下一次 malloc 对应大小的块时, 就有 bck = victim->bk = ptr, 且 bin->bk = bck = ptr, 这样以来就成功地将这个 bin 的第一块指向了 ptr, 下次再 malloc 对应大小就能够返回 ptr+16 的位置了, 这样攻击者再对取回的块进行写入就能控制 ptr+16 的内存内容。

尽管在 glibc 2.19 版本的代码中, 已经加入了对应的防御机制, 通过检查 bin 中第二块的 bk 指针是否指向第一块, 可以发现对 Small bins 的破坏。但攻击者如果同时伪造了 bin 中的前 2 块, 就可以绕过这个检查。故此种利用方式仍然可以进行。

3.4.5 The House of Spirit

此方法通过堆的 Fast bin 机制来辅助栈溢出的利用, 一般的栈溢出漏洞的利用都希望能够覆盖函数的返回地址以控制 EIP 来劫持控制流, 但若栈溢出的长度无法覆盖返回地址, 但是能够覆盖到栈上的一个即将被 free 的堆指针, 此时可以将这个指针改写为栈上的地址并在相应位置构造一个 Fast bin 块的元数据。接着在 free 操作时, 这个栈上的“堆块”即被投入 Fast Bin 中, 下一次 malloc 对应的大小时, 由于 Fast Bin 的机制为先进后出, 故上次 free 的栈上的“堆块”能够被优先返回给用户, 再次写入时就可能造成返回地址的改写, 亦可进行信息泄漏。

此方法的缺点与 *The House of Force* 方法类似, 需要对栈地址进行泄漏, 否则无法正确覆盖需要释放的堆指针, 且在构造数据时, 需要满足对齐的要求, 存在很多的限制。

3.5 小结

通过回顾早期的 glibc 堆利用技巧, 我们发现虽然同为缓冲区溢出漏洞, 堆上的缓冲区溢出利用思路与栈上的溢出有着很大区别。位于栈上的缓冲区溢出之后能够非常轻松地覆盖保存于栈上的函数返回地址, 进而劫持控制流。而堆上的缓冲区溢出则无法直接控制程序的执行, 仅能通过进一步的数据破坏来间接劫持程序控制流, 在利用的思路和复杂性上都高于栈上的缓冲区溢出利用。

而 *The Malloc Maleficarum* 一文则提供了从堆数据破坏开始, 将控制面逐步扩大直至破坏某一函数指针甚至获得任意内存写能力的思路。由于每个程序在对堆数据的使用上有着很大区别, 不同的程序可能在堆上存储不同的数据, 故通过破坏用户数据的方法来扩大控制面显得有所局限。而不依赖具体用户数据, 仅仅通过破坏 glibc 堆管理的元数据来完成利用的方式, 则更具有普遍性。尽管文中使用的 5 种利用方法条件苛刻, 且有些因为 glibc 的升级已经无法使用, 但为接下来攻击者开发新的利用方法以及 glibc 自身的加固提供了重要参考。

4 现代系统的防护措施

自2000年以来, 为了加固linux内核与系统安全的 PaX 研究组[14]开发了一些针对缓冲区溢出与shellcode 执行的系统级保护措施, 非常有效地降低了传统缓冲区溢出利用的成功率。堆上的漏洞利用受其影响, 诸多早期的利用手段已经失效。同时, glibc 自身也在不断加固, 到 2014 年 glibc 2.19 发布时, 其堆管理已经较为完善, 相比较早期版本加入了安全检查, 极大地增加了漏洞利用的难度。本章将介绍当前广泛使用的系统级保护措施, 以及 glibc 2.19 版本针对早期的利用方式进行的加固。

4.1 不可执行位 NX Bit

*NX(Non-eXecute)*位是一种针对 shellcode 执行攻击的保护措施, 意在更有效地识别数据区和代码区。该技术在 1997 年时被首次提出, 2004 年之后广泛应用于 Linux 与 Windows 操作系统上。众所周知, CPU 无法识别内存中的“数据”是指令还是用户数据, 如果 CPU 的 EIP 指向了攻击者控制的区域, 攻击者可以在此部署自己的 shellcode 从而控制程序的行为。而 *NX* 技术旨在更好地区分“数据”和“代码”, 通过在内存页的标识中增加“执行”位, 可以表示该内存页是否可以执行, 若程序代码的 EIP 执行至不可运行的内存页, 则 CPU 将直接拒绝执行“指令”造成程序崩溃。

在Linux中, 当装载器把程序装载进内存空间后, 将程序的.text 段标记为可执行, 而其余的数据段(.data, .bss 等)以及栈、堆均不可执行。如此一来, 当攻击者在堆栈上部署自己的 shellcode 并触发时, 只会直接造成程序的崩溃。因此, 传统利用方式中通过修改 GOT 来执行 shellcode 的方式已不再可行。

但是我们可以注意到, 尽管攻击者无法通过代码注入攻击进行利用, 其仍然无法阻止攻击者通过控制函数指针来劫持控制流的方法。而攻击者可以使用流行的代码重用攻击, 使用现有代码来进一步构造自身所需控制流, 进而完成利用。

4.2 地址空间布局随机化 ASLR

ASLR(*Address Space Layout Randomization*, 地址空间布局随机化)技术意在将程序的内存布局随机化, 使得攻击者不能轻易地得到数据区的地址来构造攻击载荷。由于程序的堆、栈分配与共享库的装载都是在运行时进行, 系统在程序每次执行时, 随机地分配程序堆栈的地址以及共享库装载的地址。尽管它们之间的相对位置没有改变, 但每次执行的差异仍然是页级的, 攻击者将无法预测自己写入的数据区的确切虚拟地址。

在 32 位系统中由于随机化的位数较少, 一个常用的绕过手段是通过枚举的方式, 猜测相关代码或者数据的位置, 并通过多次发送攻击载荷来不断触发程序运行。而在 64 位系统中, 因随机化位数较多, 使得通过枚举方式来猜测攻击的手段亦不可行。

但由于目前广泛应用在操作系统的地址随机化多为粗粒度的实现方式, 同一模块中的所有代码与数据的相对偏移固定。攻击者只需要通过信息泄露漏洞将某个模块中的任一代码指针或者数据指针泄露, 即可通过计算得出此模块中任意代码或者数据的地址。

4.3 位置无关可执行文件 PIE

PIE(*Position-Independent Executable*, 位置无关可执行文件)技术与 *ASLR* 技术类似, *ASLR* 将程序运行时的堆栈以及共享库的加载地址随机化, 而 *PIE* 技术则在编译时将程序编译为位置无关, 即程序运行时各个段加载的虚拟地址也是在装载时才确定。这就意味着, 在 *PIE* 和 *ASLR* 同时开启的情况下, 攻击者将对程序的内存布局一无所知, 传统的改写 GOT 表项的方法也难以进行, 因为攻击者不能获得程序的.got 段的虚地址。

使用 PIE 技术会很大程度上影响程序和系统的性能[16], 因此在 Linux 系统中, 除了关键的系统程序以及共享库使用了位置无关技术外, 大部分程序在编译时都直接确定各段加载的虚地址。

4.4 重定位只读 RELRO

RELRO(*RELocation Read-Only*, 重定位只读), 此项技术主要针对 GOT 改写的攻击方式。分为部分 *RELRO*(*Partial RELRO*)与完全 *RELRO*(*Full RELRO*)两种[17]。

- 部分 RELRO: 在程序装入后, 将其中一些段(如.dynamic)标记为只读, 防止程序的一些重定位信息被修改。
- 完全 RELRO: 在部分 RELRO 的基础上, 在程序装入时, 直接解析完所有符号并填入对应的值, 此时所有的 GOT 表项都已初始化, 且不装入 link_map 与_dl_runtime_resolve 的地址(二者都是程序动态装载的重要结构和函数)。

可以看到, 当程序启用完全 RELRO 时, 传统的 GOT 劫持的方式也不再可用。但完全 RELRO 对程序性能的影响也相对较大, 因为其相当于禁用了用于性能优化的动态装载机制, 将程序中可能不会用到的一些动态符号装入, 当程序导入的外部符号很多时, 将带来一定程度的额外开销。

对于攻击者来说, 劫持程序本身 GOT 的利用方式仅仅是破坏函数指针中的一种, 由于系统库大多没有应用重定位只读技术, 结合之前的信息泄露技术, 劫持其他模块中的 GOT 与其他函数指针也是攻击者的手段之一。

4.5 堆块元数据检查

该部分并非系统级别的通用防护措施, 而是 glibc 的堆管理措施中一些针对内存破坏进行的检查, 若 glibc 发现异常则会立即终止程序, 防止进一步的破坏和利用。

4.5.1 解链操作加固

针对传统的解链操作攻击, 对 unlink 宏进行了加固, 首先检查双向链表的完整性:

```
#define unlink(P, BK, FD) {                \
    FD = P->fd;                                    \
    BK = P->bk;                                    \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))          \
      malloc_printerr (check_action, "corrupted double-linked
list", P); \
    else {                                         \
        FD->bk = BK;                       \
        BK->fd = FD;
    ...
```

由于在解链操作之前, 针对堆块 P 自身的 fd 与 bk 检查了链表的完整性, 即判断堆块 P 的前一块 fd 的 bk 指针是否指向 P, 以及后一块 bk 的指针是否指向 P。而在 3.2 节中攻击者利用的传统解链攻击, 通常攻击者仅能控制堆块 P 的内容, 而无法控制其 fd 与 bk 指针指向的地址中的数据, 导致无法通过这一检查而失败。

针对完整性的检查大大减弱了解链攻击的利用能力, 但仍然可以造成一定的破坏, 甚至完成利用, 具体的利用方法将在 5.1 节讨论。

4.5.2 Fast bins 检查

由于 Fast bin 的组织方式为单向链表, 所以较为容易受到破坏, glibc 在对 Fast bin 操作时进行了一些额外的检查。

在执行分配操作时, 若块的大小符合 Fast bin, 则会在对应的 bin 中寻找是否有合适的块。此时 glibc 将根据候选块的 size 域计算出 fastbin 索引, 然后与对应 bin 在 fastbin 中的索引进行比较, 如果二者不匹配, 则说明块的 size 域遭到了破坏。

在执行释放操作时, 若块的大小符合 Fast bin, 则会检查对应的 bin 中的第一块是否为正在被释放的块, 这可以检测出连续两次释放同一块的问题。

在分配和释放操作的检测能够检测出一定的数据破坏, 不过仍然无法完全弥补 Fast bin 单向链表的脆弱性。Glibc 从性能考虑没有在释放时检查对应 bin 中的所有堆块是否为当前释放的块, 仅仅通过检查第一块来检测两次释放的问题。

4.6 小结

将上面的保护措施与第 3 章中的利用方法结合来看, 不难发现, 虽然大多数系统级的保护措施都是针对栈上的缓冲区溢出而开发, 但堆上的利用也受到了很大的影响。首先早期攻击者赖以执行任意代码的 shellcode 被 NX 技术完全封锁, 因为攻击者控制的可写区域绝大多数已经不可以执行; 其次, 受到 *ASLR* 技术的影响, 攻击者无法知道程序内部库函数的地址、程序堆栈的地址, 导致其想对相关结构进行破坏(例如, glibc 中的堆管理结构 main_arena)时, 由于不能得知它们在内存中的位置而无法成功。

而 glibc 在其堆管理中增加的安全检查, 也使得 3.2 节中的传统 unlink 方法不再可行, 因为攻击者想使得自己伪造的堆块的 FD 和 BK 能够通过检查较为困难, 特别是在 *ASLR* 开启的情况下堆地址未知, 无法构造合法的链表中的堆块。

尽管如此, Linux 的 glibc 对堆数据的保护仍然略显单薄。而相较而言, Windows 操作系统的堆防护措施则较为健全, 自从 Windows XP SP2 以及 Windows Server 2003 以来, 微软就采用了众多措施来防止堆内存的破坏以及堆利用[29]。包括堆完整性检查、移出异常处理块、禁止释放内部数据结构堆句柄、堆内部函数指针加密、防护页、分配顺序随机化等手段被用于 Windows 8 及之后的版本中, 极大地强化了攻击者针对 Windows 堆进行利用的难度[30]。

下一章将介绍在部署所有防御技术的情况下的现代堆漏洞利用方式, 而相关防护措施的局限性将在第 6 章中讨论。

5 现代利用手段

本章介绍针对当前流行的 glibc 2.19 版本进行漏洞利用的现代利用手段, 并阐述它们能够绕过的系统级保护措施以及绕过的条件。

5.1 现代解链操作

在 4.5.1 节提到, 当前的 glibc 对 unlink 宏进行了加固, 传统的 unlink 利用方法已不再可行。为了绕过当前的 glibc 对 unlink 做的检查, 需要在内存中找到指向构造的伪块的指针 P, 且需要知道 P 的地址即 &P[18]。如图 5 所示, 构造伪块时令

$$fd = \&P - 24$$

$$bk = \&P - 16$$

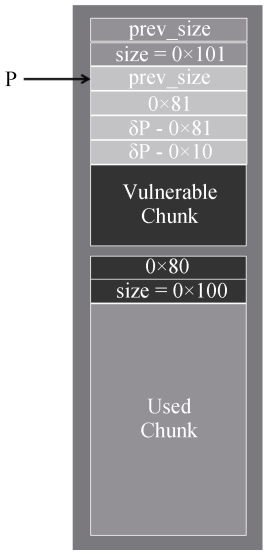

图 5　解链操作示意

Figure 5　Modern Unlink

根据堆块的结构以及 C 语言结构体的成员寻址方式, 程序由一个结构体指针取得成员的值时, 仅以其头指针的地址加成员的偏移来进行寻址。fd 的偏移为 16, bk 的偏移为 24, 故在进行 unlink 检查时将得到:

$$FD = P->fd = \&P-24$$

$$BK = P->bk = \&P-16$$

$$FD->bk = *(\&P-24+24) = P$$

$$BK->fd = *(\&P-16+16) = P$$

这样就通过了 unlink 的检查, 最终得到的效果

$$FD->bk = P = BK = \&P-16$$

$$BK->fd = P = FD = \&P-24$$

经过 unlink 后, 原本指向堆上伪块的指针 P 指向了自身地址减 24 的位置, 这就意味着如果程序功能允许对 P 进行写入, 就能改写 P 指针自身的地址, 从而造成任意内存写入。而若运行对 P 进行读取, 则会造成信息泄漏。

5.2　攻击 Fast bins

就像之前所说, Fast bin 的组织方式显得比较脆弱, 故针对 Fast bin 攻击比较容易, 不同的漏洞可以有不同的利用方式。

5.2.1　释放后使用漏洞的利用

若能够控制一个已经被释放的 Fast chunk 的内容, 可以改写其 fd 指针, 指向任意内存地址 P, 这样在下下次分配同样大小的块时, 就能取得对应的地址。但需要绕过 4.5.2 节中所说的检查机制, 即需要 P + 8(伪块的 size 域)的位置存有正确的 size 数据, 否则将会被 glibc 检测出来而终止程序。但这段代码没有检查内存地址的对齐, 使得通过内存地址不对齐来取得适当的 size 进行 Fast bin 攻击成为可能。

5.2.2　两次释放漏洞的利用

利用了 glibc 在检查 Fast bin 的 Double Free 时只检查第一块的机制的缺陷, 进行 Double Free 的利用[19], 如图 6-10 的例子所示。

初始的 Fast bin 结构如下, 其中含有 2 个大小为 32 字节的块:

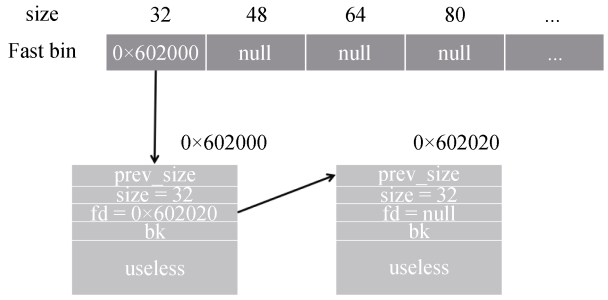

图 6　初始情形

Figure 6　Initial State

此时执行 free(0x602030), 由于与此块在 bin 中第 2 块的位置故不会被检测出来, 此时这个块被再次插入 Fast bin 中, 变成下面的情形

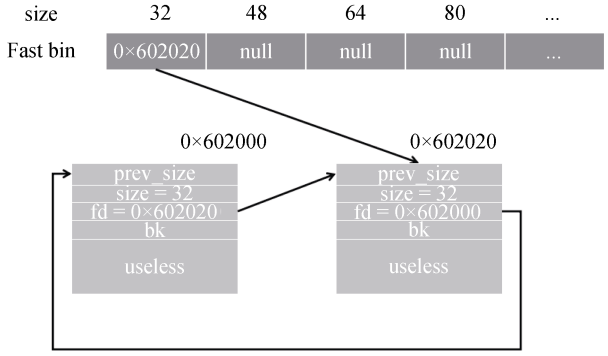

图 7　两次释放

Figure 7　Double Free

执行 malloc(24), 取得 0x602020 的块, 但这个块仍在 Fast bin 中, 此时可以控制其 fd

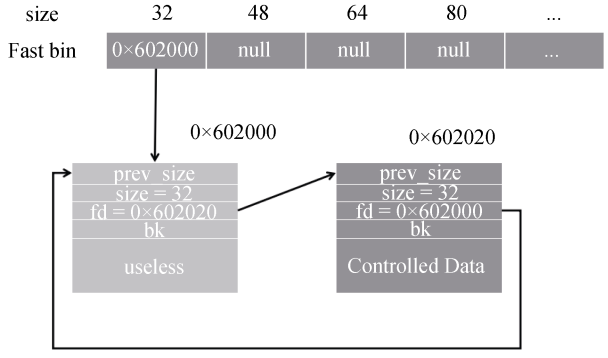

图 8　控制 fd

Figure 8　Control FD

再次执行 malloc(24), 取得 0x602000 的块, 此时 Fast bin 的头部指向 0x602020, 而攻击者可以更改其 fd 指针, 将其指向任意地址 ptr。

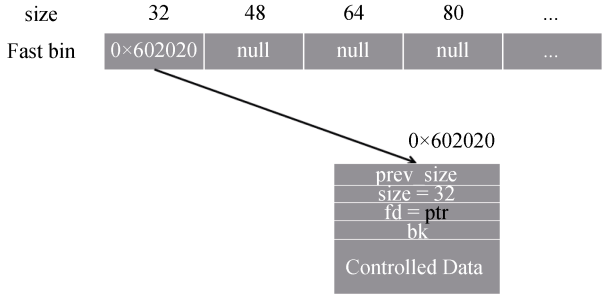

图 9 修改 fd 指针
Figure 9 Modify FD Pointer

第 3 次执行 malloc(24), 再次取得 0x602020 块, 当下一次 malloc(24)时, 将能够取得 ptr 的位置, 需要满足*(ptr + 8) = 32。

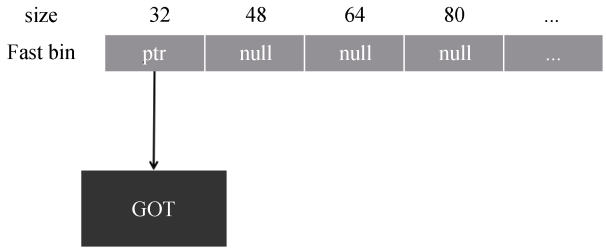

图 10 GOT 劫持
Figure 10 GOT Hijacking

5.3 改写 Realloc Hook

主要针对 4.4 节中提到的重定位只读的防御手段, 由于此时程序的 GOT 表已不可写, 无法利用 GOT 改写来劫持程序控制流。此时可以借助 glibc 中用于调试堆的 hook 机制[20], 修改存于 glibc .data 段的记录 hook 函数的指针变量__realloc_hook, 若 glibc 发现此变量的值不为 0, 则在进行 realloc 操作时会直接调用此变量中记录的函数地址, 从而达到劫持控制流的目的。

改写__free_hook 同样能够达到对应的效果, 但是在其附近没有用于进行 Fast Bin Attack 的 size 数值。而__realloc_hook 在程序启动时有一个非 0 的初始化值, 在 64 位系统中, 这个地址将以 7f 开头, 这就使得攻击者能够通过刻意的内存地址错位来进行 Fast Bin Attack, 从而达到改写__realloc_hook 的目的。

5.4 利用单字节溢出漏洞

在某些情况下, 程序的漏洞可能使攻击者无法进行任意长度的溢出, 此时能够覆盖和改写的部分就相对较少。但即使只能溢出一个字节(Off-by-one), 同样能够进行利用。通过溢出下一个块的 size 域, 攻击者能够在堆中创造出重叠的内存块, 从而达到改写其他数据的目的, 再结合其他的利用方式, 同样能够取得程序的控制权[22]。

5.4.1 扩展被释放块

若溢出块的下一块为被释放的块且处于 Unsorted Bin 中, 则可以通过溢出一个字节来将其大小扩大, 下次取得此块时就意味着其后的块将被覆盖而造成进一步的溢出。具体过程如图 11 扩大被释放块所示。

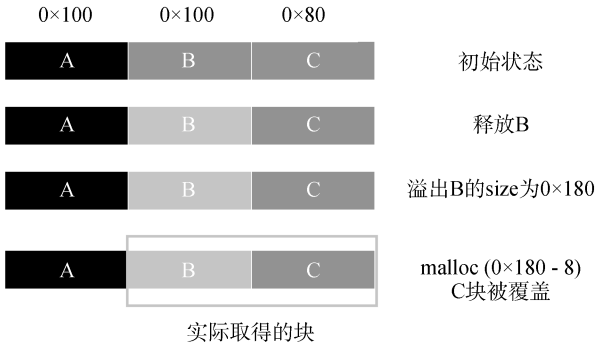

图 11 扩大被释放块
Figure 11 Extend Freed Chunk

5.4.2 扩展已分配块

若溢出块的下一块为使用中的块, 则需要合理控制溢出的字节, 使其被释放时的合并操作能够顺利进行, 例如直接加上下一块的大小使其完全被覆盖。下一次分配对应大小时, 即可取得已经被扩大的块, 并造成进一步溢出。具体过程如图 12 所示。

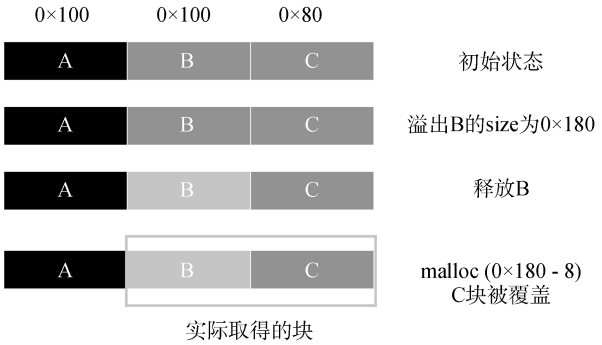

图 12 扩展已分配块
Figure 12 Extend Allocated Chunk

5.4.3 收缩被释放块

此情况针对溢出的字节只能为 0 的情况, 此时可以将下一个被释放块的大小缩小, 如此一来在之后分裂此块时将无法正确更新后一块的 prev_size 域,

导致释放时出现重叠的堆块。具体过程如图 13 所示。

5.4.4 The House of Einherjar

此方法也是针对溢出字节只能为 0 的情况, 利用的思路是不将溢出块下一块的大小缩小, 而是仅仅更新后一块的 prev_size 域, 使其在被释放时能够找到之前一个合法的被释放块并与其合并, 造成堆块的重叠。具体过程如图 14 所示。

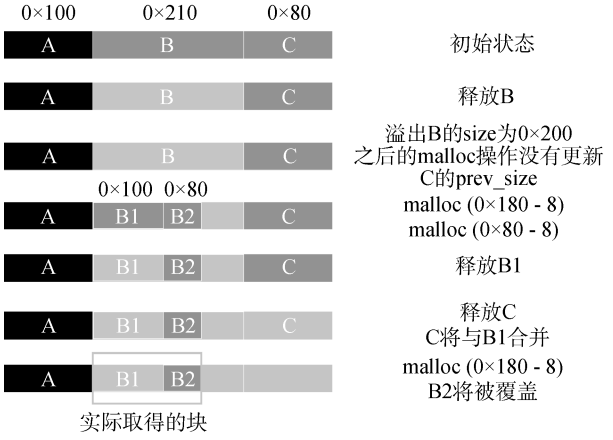

图 13 收缩被释放块

Figure 13 Shrink Freed Chunk

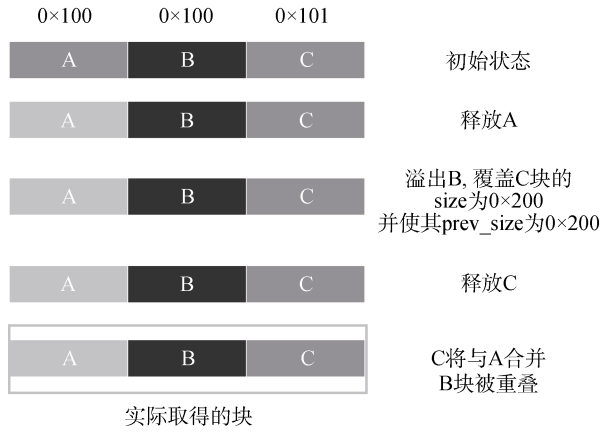

图 14 英灵(Einherjar)攻击示例

Figure 14 The House of Einherjar

5.5 攻击 Unsorted Bin

如 2.2.4 节中所说, 当 Unsorted Bin 中的块恰好为分配的块大小时, 则会直接从 Unsorted Bin 中移除返回给用户。若不是, 则将其投入对应的 Bin 之中。无论结果如何, 其最终都将从 Unsorted Bin 中移除, 而这里移除使用的方法并不是 2.3 节中释放操作使用的 unlink 宏, 而是如下代码:

```
bck = victim->bk;
...
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

不同于 unlink 宏, 这里对 Unsorted Bin 的链表完整性并没有做检查, 导致如果能够可以修改某一个处于 Unsorted Bin 中的块的 bk 指针, 则可以造成一个任意内存写入, 可以将 bk + 16 的位置改写为 Unsorted Bin 的地址, 但 Unsorted Bin 的 bk 指针将被破坏, 下一次再使用 Unsorted Bin 时, 将可能由于 bck 的位置不可写而造成程序崩溃。

5.6 重分配函数的利用

在对相关资料的研究中, 我发现攻击者利用的关注点都集中在 malloc 和 free 两个函数的机制上, 而对另一个频繁使用的分配库函数——realloc 的关注则不是很多。由于 realloc 函数的特性, 使其兼有 malloc 与 free 的双重功能, 且能够通过改写 __realloc_hook 的方式来绕过 *Full RELRO* 保护技术, 因而也是需要关注的一个函数。

5.7 小结与比较

本章介绍的所有利用技术都能够突破 glibc 堆管理的安全检查的封锁, 通过攻击程序的 GOT 表或.dtors 段, 以及改写__realloc_hook 或__free_hook 的方式, 可以绕过 *NX* 技术和 *Full RELRO* 的限制; 但在 *ASLR* 和 *PIE* 技术开启的情况下, 要找到这些相关结构的位置, 事先攻击者需要能够泄露出对应模块的地址(如 glibc 加载的地址、程序加载的地址、程序堆的地址等), 而泄露地址的过程并不总是非常容易。若程序是一个远程服务或某种解析程序(如图像、压缩解析), 则很难与攻击者进行进一步交互, 这也让利用变得十分困难。故更多时候, 攻击者会使用程序自身的一些实现特点和机制, 如程序内部实现的内存管理器、程序的扩展功能等, 甚至可以直接通过堆溢出覆盖堆块之后的函数指针, 达到执行敏感函数(如 system)的目的。

表 1 现代利用方式比较

Table 1 Comparison of Modern Exploitation Methods

利用方式	破坏堆块数据	破坏堆管理数据	触发函数	利用目标
现代解链	prev_size, size	无	free	破坏堆指针
Fast bins 攻击	fd	Fast bins	malloc	控制 malloc 返回的指针
Off-by-one	size	无	malloc/free	造成堆块重叠
Unsorted Bin 攻击	fd, bk	Unsorted Bin	malloc	任意地址写(内容不可控)
malloc_consolidate	prev_size,size	无	malloc/free	破坏堆指针

表 1 比较了本章介绍的各种利用方法的利用条件、破坏的数据结构以及最终造成的结果等特性。从中可以看出, 相关漏洞的初始能力都只能破坏堆块相关的结构, 无法威胁到其他敏感数据, 而攻击者需要通过伪造堆块数据来欺骗 glibc 以达到后续破坏的目的, 这将在下一章详细讨论。

6 讨论

6.1 系统防护措施的局限性

表 2 给出了之前介绍的防护措施的绕过方式, 可以看到, 在系统层面上的防护措施并不适用于堆利用的的防御, 因本身其在一般的内存破坏攻击的利用思路中就有绕过的手段。而针对堆块数据破坏的检查, 解链操作的加固, 也可以通过攻击者精心的构造或布局来绕过。

表 2 防护措施的绕过技术

Table 2 Bypassing Mitigation

防护措施	绕过方法
NX	代码重用攻击
ASLR	程序信息泄露
PIE	程序信息泄露
RELRO	改写 glibc 中其他函数指针
解链操作加固	构造符合条件的伪块
Fast bin 检查	多次释放不同块

从对之前防护措施的分析可以看出, 基于通用利用方法而开发的系统防护措施并不能有效地防止堆利用的成功进行。而由于 glibc 代码的开源性, 进行相应的元数据检查仍然可以通过一定的数据伪造来进行欺骗最终完成利用。故需要找到众多堆利用手段的共性特征并从其本质上阻止利用。

6.2 堆利用手段的共性特征

第 5 章中所提到的所有针对 glibc 的利用方法, 都没有借助堆数据中用户数据的内容, 实际环境中程序的堆内存中可能存在更加敏感的用户数据例如虚函数表、函数指针等, 攻击者若直接控制这些指针就能够不借助 glibc 自身功能完成利用。而本文中介绍的方法, 均是依靠堆内存上元数据的破坏, 之后利用 glibc 的 ptmalloc 分配器自身的代码, 使得攻击者具有改写堆内存之外数据的能力。具体来说, 堆利用的方法一般都具有以下的流程:

1. 使用堆数据溢出或者非法的分配释放操作, 让攻击者能够直接或者间接控制堆块自身的元数据, 甚至完全伪造若干个堆块的布局情况。

2. 通过分配与释放操作, 触发 glibc 的堆管理代码, 利用其功能进一步破坏 ARENA 结构中的数据, 从而让 glibc 误将某些敏感数据的地址当做堆块处理。

3. 攻击者进一步分配堆块, 并覆盖敏感数据最终达到控制流劫持的目的。

相较于栈溢出而言, 堆内存相关的漏洞利用具有以下特征:

- 无法直接控制 EIP: 与栈溢出不同的是, 堆利用无法通过溢出来直接控制 EIP 的值, 只能破坏堆块的元数据。
- 触发漏洞后数据的进一步破坏总是在分配和释放时产生。
- 漏洞触发后仍需要进行内存读写: 当溢出或 Use After Free 的漏洞被触发后, 攻击者仍然需要进行一定次数的内存读写才能进行进一步的破坏以劫持控制流。
- 依赖程序自身功能: 对于任意读写的操作, 需要利用程序自身的功能来实现。在程序提供的功能很少或保护很好的情况下, 堆漏洞仍然难以利用。
- 能够绕过现有保护机制: 一旦堆利用能够成功进行, 即使程序开启现有的全部保护机制*(NX, PIE, ASLR, Full RELRO*), 通过泄漏地址以及覆盖__ realloc_hook 等方式攻击者仍能完成利用。

6.3 堆利用可能的防护措施

通过前一节的分析, 我们可以就堆内存利用的一些共性特征提出针对性的防护措施。

首先, 堆内存的破坏大多由于堆块自身的元数据被攻击者伪造所致, 其根源在于 glibc 的分配器将堆块的元数据与用户数据混杂在一起, 导致程序在使用堆块时容易将堆块的元数据改写。而且, 由于在堆块释放时会将 glibc 数据区的 bins 相关的指针写入堆块, 这些指针同样成为攻击者进行信息泄露的理想目标。解决这一问题的办法就是将元数据从堆块中移除并安排在其它位置, 例如 glibc 自身的数据区中, 使得攻击者在程序存在溢出等内存破坏漏洞时, 难以通过改写堆块的元数据来欺骗 glibc。

而释放后使用等类似的时序错误, 以及非法分配、释放操作引发的数据破坏, 通常都是在调用分配释放函数时发生。对此, glibc 仍然需要使用一定的数据来记录堆块的使用情况, 或在堆块释放时对其中的用户数据进行一定的混淆或者破坏, 避免攻击者利用其中未清除的数据进行信息泄露或控制流劫持。

7 结论

7.1 堆利用与防护的现状

对于栈利用而言, 多数加固措施都着眼于系统

层面, 如 *NX*, *Stack Canary* 等都是工作在系统层级的通用防护措施。而在堆内存中, 攻击者大多寻找 glibc 堆管理代码的缺陷, 结合漏洞的特点进行利用。随着对 glibc 的 ptmalloc2 源代码研究的深入, 越来越多的攻击方式被开发出来, 其中涉及各种各样的数据结构。面对攻击者巧妙的数据伪造, 粗粒度的系统级加固措施将失去作用。

然而由于堆漏洞的利用过于依赖特定的漏洞类型以及程序功能, 同一种利用的流程往往只能适用于一款特定的应用, 利用过程较难实现自动化。故无法针对同类漏洞进行大规模利用, 这也从某种意义上降低了堆漏洞的影响, 然而由于堆利用具有能够绕过当前所有系统级保护的特点, 一旦针对某个应用完成利用过程, 则防御方在规避时就需要修改应用的源代码来直接修补漏洞。

而对于防御方来说, 存在同样的问题, 由于现有的利用方式往往只限定与某一特定的漏洞, 而且修复代码涉及众多结构, 可能会让代码的整体架构产生变化, 开发代价与运行时的开销都将大大增加。且对于通常只有几十个字节的堆块而言, 很难部署细粒度的通用通用防御措施。虽然已有研究实现了基于 Canary 的防御系统[25], 但由于在每个堆块上都部署 Canary 将会造成较大的空间开销, 故 glibc 当前仍未采用。

与操作系统部署的防护措施不同, 由于 glibc 与程序的代码一样工作在用户态, 无法比用户代码拥有更多的特权, 且难以做到在数据破坏发生时进行快速检测。目前 glibc 的安全检查仍然只是在分配和释放的函数中进行。

虽然近年来, glibc 的 ptmalloc2 一直在针对新的攻击方式在进行修复, 但在程序中存在漏洞, 且功能较多的情况下, 攻击者仍然能够通过伪造数据等方式造成内存破坏, 最终进行控制流劫持。

7.2 堆利用与防护的未来发展

就像前一节中所说, 尽管近期软件在堆使用上的可利用漏洞层出不穷, 但堆溢出的利用方式却各不相同, 难以部署有效的通用防御措施。在实际软件安全领域的堆漏洞利用与防护, 还将持续“发现——修补——再发现——再修补”的循环。

而在安全研究领域, 可以预见, 在 glibc 的堆上进行的攻防博弈, 在一段时间内仍然会围绕 ptmalloc2 的管理代码来进行。攻击者尽力找到能够利用的代码来进行内存破坏, 而防御者则将加入更多检查以发现元数据的损坏, 以及时终止程序, 同时需要权衡额外开销和安全性。从这点上来说, 防御的难度要比攻击大得多, 这也是 glibc 在修复上一直缓慢的原因。

而就长远来看, 为了能够尽可能减小堆数据破坏给分配器带来的影响, 进行有效的数据隔离, 使得堆分配器的元数据不受攻击者的破坏仍然是值得尝试的做法。而如何在 glibc 这一用户态运行库上实现数据保护也将是未来需要研究的课题。

致谢 感谢网络与信息安全实验室的杨坤博士提供的相关资料以及对于论文撰写提出的建设性意见。

参考文献

[1] One A. Smashing the stack for fun and profit. Phrack magazine, 1996, (14-16)

[2] Ven A. New security enhancements in red hat enterprise linux v. 3, update 3. Raleigh, North Carolina, USA: Red Hat, 2004.

[3] Spengler B. Pax: The guaranteed end of arbitrary code execution. G-Con2: Mexico City, Mexico, 2003. Slides 22 through 35

[4] Cowan C, Wagle P, Pu C, et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings, volume 2. IEEE, 2000. 119–129

[5] Cowan C, Pu C, Maier D, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. Usenix Security, volume 98, 1998. 63–78

[6] Advisory Q S. CVE-2015-0235, Ghost. https://www.qualys.com/2015/01/27/cve-2015-0235/GHOST-CVE-2015-0235.txt

[7] Sploitfun. Understanding glibc malloc. https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/comment-page-1/

[8] Glibc 2.19 malloc source code. http://osxr.org:8080/glibc/source/malloc/malloc.c

[9] Wiki G. The gnu c library release timeline. https://sourceware.org/glibc/wiki/Glibc%20Timeline

[10] ELF Application Binary Interface Supplement. Dynamic Linking. http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0 zSeries/x2251.html

[11] Rivas J M B. Overwriting the.dtors section. http://www.bright-shadows.net/tutorials/dtors.txt, 2000

[12] Phantasmagoria P. The Malloc Maleficarum: Glibc Malloc Exploitation Techniques. https://packetstormsecurity.com/files/40638/MallocMaleficarum.txt, 2005

[13] Glibc 2.19 arena source code. http://osxr.org:8080/glibc/source/malloc/arena.c

[14] PaX. Homepage of the pax team. https://pax.grsecurity.net/

[15] Levine J R. Linkers and loaders. Morgen Kaufmann, 1999. 170–171. ISBN 1-55860-496-0

[16] Payermathias M. Too much PIE is bad for performance. https://nebelwelt.net/publications/files/12TRpie.pdf, 2012

[17] Federico A, Cama A, Shoshitaishvili Y, et al. How the elf ruined christmas. USENIX Security, 2015

[18] AngelBoy. Heap exploitation. http://www.slideshare.net/AngelBoy1/heap-exploitation-51891400, 2015

[19] Ferguson J N. Understanding the heap by breaking it. Black Hat USA, 2007. 1–39

[20] Memory Allocation Hooks. http://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html

[21] AngelBoy. Advanced heap exploitation. http://www.slideshare.net/AngelBoy1/advanced-heap-exploitaion, 2015

[22] Goichon F. Glibc Adventures: The Forgotten Chunks. http://www.contextis.com/documents/120/Glibc Adventures-The Forgotten Chunks.pdf, Jan, 2015

[23] Zero G P. The poisoned NUL Byte. http://googleprojectzero.blogspot.sg/2014/08/the-poisoned-nul-byte-2014-edition.html, Aug, 2014

[24] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity. Proceedings of the 12th ACM conference on Computer and communications security. ACM, 2005. 340–353

[25] Robertson W K, Kruegel C, Mutz D, et al. Run-time detection of heap-based overflows. LISA, volume 3, 2003. 51–60

[26] Kaempf M M. Vudo - an object superstitiously believed to embody magical powers. Phrack magazine, 2001.

[27] Anonymous. Once upon a free(). Phrack magazine, 2001.

[28] acez. Hitcon ctf 2014 stkof or the ”unexploitable” heap overflow ? http://acez.re/ctf-writeup-hitcon-ctf-2014-stkof-or-modern-heap-overflow/

[29] Windows ISV Software Security Defenses https://msdn.microsoft.com/en-us/library/bb430720.aspx

[30] Software Defense: mitigating heap corruption vulnerabilities https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/

裴中煜 于 2016 年在清华大学计算机科学与技术专业获得学士学位。现在清华大学计算机科学与技术专业攻读硕士学位。研究领域为系统安全方向，研究兴趣包括：二进制漏洞的挖掘与利用，二进制程序的自动分析等。Email: peizy16@mails.tsinghua.edu.cn

段海新 清华大学网络科学与网络空间研究院教授，加州大学伯克利分校访问学者，主要研究方向为网络安全，包括协议安全、入侵检测以及地下产业检测等。

张超 博士，清华大学副教授，研究方向为软件和系统安全。