Low level attacks
Shellcode (part 1)

Mario Alviano

University of Calabria, Italy

A.Y. 2019/2020

Spawn a shell in C

- Try `shell.c` (on a old machine)

Spawn a shell in C

- Try `shell.c` (on a old machine)
- Now modify the owner to root, and set the SUID bit

```
$ sudo chown root a.out
$ sudo chmod +s a.out
```

Spawn a shell in C

- Try `shell.c` (on a old machine)
- Now modify the owner to root, and set the SUID bit

```
$ sudo chown root a.out
$ sudo chmod +s a.out
```

- Execute the binary again... and you are root!

Spawn a shell in C

- Try `shell.c` (on a old machine)
- Now modify the owner to root, and set the SUID bit

```
$ sudo chown root a.out
$ sudo chmod +s a.out
```

- Execute the binary again... and you are root!

What all this means

- Processes are associated with two user ids
  - Real UID: who started the process
  - Effective UID: for who the process acts
- Similarly, there are real and effective group ids

Spawn a shell in C

- Try `shell.c` (on a old machine)
- Now modify the owner to root, and set the SUID bit

```
$ sudo chown root a.out
$ sudo chmod +s a.out
```

- Execute the binary again... and you are root!

What all this means

- Processes are associated with two user ids
  - Real UID: who started the process
  - Effective UID: for who the process acts
- Similarly, there are real and effective group ids
- If SUID is set, effective UID is set to the user owning the file
- If SGID is set, effective GID is set to the group owning the file
- `exec*` functions start new processes... acting for the effective user and group!

Our first shellcode

- A shellcode is a set of machine instructions
- Essentially, instructions spawning a shell
- Try `shellcode.c`

Our first shellcode

- A shellcode is a set of machine instructions
- Essentially, instructions spawning a shell
- Try `shellcode.c`

Problems we have to face

- Inject our shellcode in a vulnerable buffer
- Jump to the first instruction of our shellcode

Shellcode injection

- Try `victim.c`
- How to inject our shellcode?

Shellcode injection

- Try `victim.c`
- How to inject our shellcode?

The NOP Method

```
<NOPs (0x90)> <shellcode> <padding> <saved return address>
```

- We will jump in the NOP sled
- The more NOPs, the more likely the injection
- Follow the instructions in `attack-victim.txt`

Shellcode creation

- Let's create a simple shellcode

Shellcode creation

- Let's create a simple shellcode
- Essentially, the syscall `exit(0)`
- Code it in assembly (see `exit.asm`)

Shellcode creation

- Let's create a simple shellcode
- Essentially, the syscall `exit(0)`
- Code it in assembly (see `exit.asm`)
- Check the machine code with `objdump`

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

Shellcode creation

- Let's create a simple shellcode
- Essentially, the syscall `exit(0)`
- Code it in assembly (see `exit.asm`)
- Check the machine code with `objdump`

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Now try `exit_shellcode.c`

Injectable shellcode

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Can we remove zeros from our shellcode?

Injectable shellcode

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Can we remove zeros from our shellcode?
- Two possibilities:
  1. Replace assembly instructions
  2. Add zeros at runtime

Injectable shellcode

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Can we remove zeros from our shellcode?
- Two possibilities:
  1. Replace assembly instructions
  2. Add zeros at runtime

Replace assembly instruction

- The first instruction can be replaced by
  `xor ebx, ebx`

Injectable shellcode

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Can we remove zeros from our shellcode?
- Two possibilities:
  1. Replace assembly instructions
  2. Add zeros at runtime

Replace assembly instruction

- The first instruction can be replaced by

```
xor ebx, ebx
```

- The second instruction can be replaced by

```
xor eax, eax
mov al, 1
```

Injectable shellcode

```
08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    ebx,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
```

- Can we remove zeros from our shellcode?
- Two possibilities:
  1. Replace assembly instructions
  2. Add zeros at runtime

Replace assembly instruction

- The first instruction can be replaced by
  `xor ebx, ebx`
- The second instruction can be replaced by
  `xor eax, eax`
  `mov al, 1`
- Try `exit2.asm` and `exit2_shellcode.c`

Create a real shellcode (1)

- Let's look again `shell.c`
- Possible implementation in assembly: `shell.asm`

Create a real shellcode (1)

- Let's look again `shell.c`
- Possible implementation in assembly: `shell.asm`

Problems to face

- We cannot use zeros
- We should use relative addressing as much as possible
- Can we get the address of `filename`?

Create a real shellcode (2)

Try `shellcode.asm`

Make text segment writable

Run `ld` with option `-N`

Create a real shellcode (2)

Try `shellcode.asm`

Make text segment writable

Run `ld` with option `-N`

- The `call` instruction pushes the address of `filename`
- It is popped and stored into a register
- All instructions can use relative addressing
- Now get the machine code and try it with `shellcode2.c`

Create a real shellcode (2)

Try `shellcode.asm`

Make text segment writable

Run `ld` with option `-N`

- The `call` instruction pushes the address of `filename`
- It is popped and stored into a register
- All instructions can use relative addressing
- Now get the machine code and try it with `shellcode2.c`

Better to extract the shellcode automatically

```
objdump -D -M intel shellcode.o | grep -P ":\t" |
   sed 's/.*:\t//' | sed 's/\s*\t.*$//' |
   sed 's/ /\\x/g' | sed 's/\(.*\)/"\x1"/'
```

Create a real shellcode (3)

Now check `shellcode-with-p.asm`

Exercise

Extract the shellcode and inject in `shellcode2.c`

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Example

- Replace the return address with the address of `system()`
- Leave 4 bytes (it is the return address of `system()`)
- Write the address of the string to execute
- Follow the instructions in `ret2libc.txt`
- Also check `bypass-suid-drop-policy.txt`

Return to libc (ret2libc)

- Alternative to code injection
- Just inject return addresses (and arguments)

Example

- Replace the return address with the address of `system()`
- Leave 4 bytes (it is the return address of `system()`)
- Write the address of the string to execute
- Follow the instructions in `ret2libc.txt`
- Also check `bypass-suid-drop-policy.txt`

Return Oriented Programming (ROP)

Chain several calls to small instruction sets terminated by `ret`

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection

Protection mechanisms

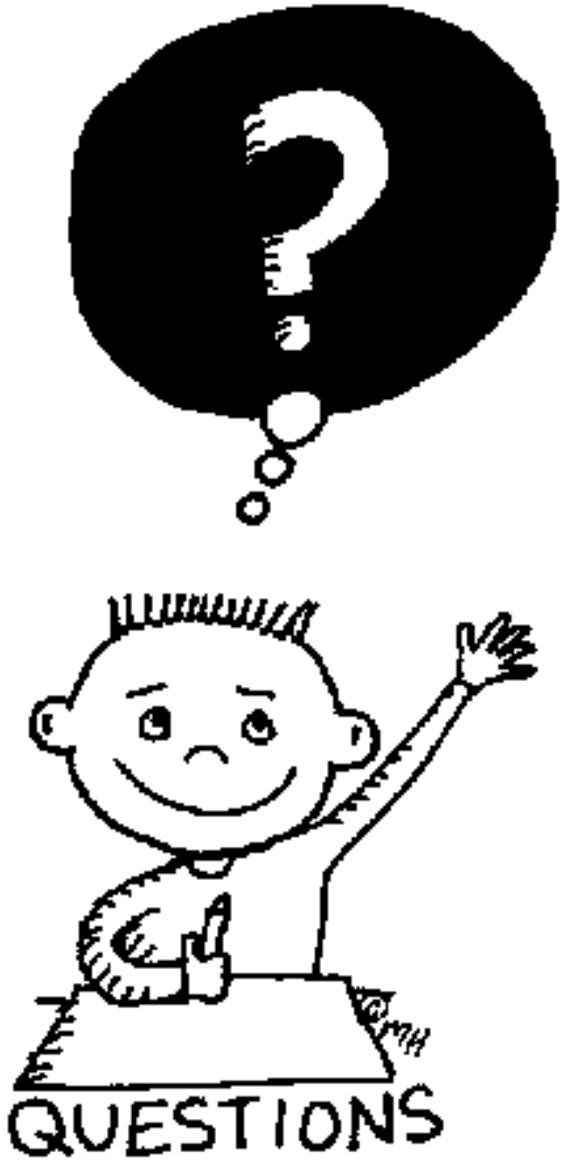
- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict
- AAAS: ASCII Armored Address Space
  - Start addresses of subroutines with \x00
  - Limit calls in case of overflows

Protection mechanisms

- NX bit: mark each memory segment as writable xor executable
  - Protect against code injection
- Canaries: memory after buffers store special values
  - Protect against buffer overflows
  - Usually randomized, and difficult to predict
- AAAS: ASCII Armored Address Space
  - Start addresses of subroutines with \x00
  - Limit calls in case of overflows
- ASLR: Address Space Layout Randomization
  - Randomly change addresses at each execution



END OF THE LECTURE