

Hacking in C

Attacks 3 and memory safety
Thom Wiggers



Table of Contents

Recap

printf is Turing complete

Defeating W \oplus X

ASLR

Memory safety

Some announcements



Recap of last week

- Overwriting buffers to take over control flow
 - Overwriting local variables
 - Overwrite the return address
- Shell code: bytecode to spawn a shell
 - Using tricks to stay clear of **NULL** bytes.
- Mitigations
 - Less code
 - Safer languages
 - Dynamic analysis
 - Static analysis
 - Stack canaries



The general plan of attack

1. Identify vulnerabilities
 - Format strings: `%p` leads something else than `%p` being printed
 - Buffer overflows: `gets`, `strcpy`, `segmentation error`
2. Identify how you can figure out what's going on at the other side
 - Local: use `gdb`
 - Remote: `%p%p%p`
3. Determine for a buffer overflow when it crashes: is there maybe a return address or frame pointer there?
4. Figure out how you're going to reach your goals
 - Take over return address to execute other function
 - a. Find other function's address
 - b. Overwrite return address
 - Inject your own code (shellcode)
 - a. Figure out where to put shellcode
 - b. Overwrite return address



Table of Contents

Recap

`printf` is Turing complete

Defeating $W \oplus X$

ASLR

Memory safety

Some announcements



Tic-Tac-Toe

```
char* fmt = O(10,39)N(40)N(41)N(42)N(43)N(66)N(69)N(24)O(22,65)O(5,70)O(8,44)N(
45)N(46)N(47)N(48)N(49)N(50)N(51)N(52)N(53)O(28,
54)O(5,55)O(2,56)O(3,57)O(4,58)O(13,73)O(4,
71)N(72)O(20,59)N(60)N(61)N(62)N(63)N(64)R R
E(1,2,3,13)E(4,5,6,13)E(7,8,9,13)E(1,4,7,13)E
(2,5,8,13)E(3,6,9,13)E(1,5,9,13)E(3,5,7,13
)E(14,15,16,23)E(17,18,19,23)E(20,21,22,23)E
(14,17,20,23)E(15,18,21,23)E(16,19,22,23)E(14,18,
22,23)E(16,18,20,23)R U O(255,38)R G(38)O(255,36)
R H(13,23)O(255,11)R H(11,36)O(254,36)R G(36)O(
255,36)R S(1,14)S(2,15)S(3,16)S(4,17)S(5,18)S(6,
19)S(7,20)S(8,21)S(9,22)H(13,23)H(36,67)N(11)R
G(11)""O(255,25)R s(C(G(11)))n(G(11))G(
11)N(54)R C("aa")s(A(G(25)))T(G(25))N(69)R o
(14,1,26)o(15,2,27)o(16,3,28)o(17,4,29)o(18
,5,30)o(19,6,31)o(20,7,32)o(21,8,33)o(22,9,
34)n(C(U))N(68)R H(36,13)G(23)N(11)R C(D(G(11)))
D(G(11))G(68)N(68)R G(68)O(49,35)R H(13,23)G(67)N(11)R C(H(11,11)G(
11))A(G(11))C(H(36,36)G(36))s(G(36))O(32,58)R C(D(G(36)))A(G(36))SS
```

Figure: tic-tac-toe in a format string



Table of Contents

Recap

printf is Turing complete

Defeating $W \oplus X$

ASLR

Memory safety

Some announcements



W⊕X

- Write XOR eXecute
- Mark “data” pages as writable, “code” pages as executable, never both.
- We had to turn this off for our shellcode-based attacks!
- Means we can only jump to code already present in the program.
- Is W⊕X the end of attacks on programs that do not contain a function `give_me_shell_pls()`?



Looking for code

- There is a lot more code present than just what's in `program.c`
- Whole of `libc` usually loaded in most programs.
- Does `libc` contain `give_me_shell_pls()`?
- Answer: Kinda.
- `system`



system

```
int system(const char* command);
```

“The `system()` library function uses `fork()` to create a child process that executes the shell command specified in `command`...”



Return to libc

If we can somehow prepare the argument for `system()`, we can overwrite the return address with the address of `system()` and start the shell. . .



Back in the days of yore

Plan of attack in *Ye olden days* (x86) when arguments were passed via the stack

1. Get address of libc and offset of `system()`
 - `/proc/$PID/maps | grep libc`
 - `nm -D /lib/libc.so.6 | grep system`
2. Write address of `/bin/sh` to the stack in place of argument
3. Overwrite return address in stack frame with address of `system()`
4. Optional: set up return address to normally terminate program
 - Alternatively, set up return address to address of `exit()`



Nowadays: AMD64

- Arguments passed through registers
- How do we load the address of `/bin/sh` into `%rdi`?
- We're looking for code that does

```
pop %rdi
retq
```
- There probably isn't any function that just does that...
- **But we don't have to jump to the start of any function!**
- We can jump to *any* place in `libc`
- We can probably find `pop %rdi;retq` *somewhere* in `libc`.
- We call such snippets **gadgets**



Plan of attack

1. Overwrite return address with address of gadget
2. Put address of `/bin/sh` behind gadget
3. Put address of `system()` behind

What will happen?

1. Function returns: pops return address from stack
2. Returns to gadget: pops address of `/bin/sh`
3. Gadget returns: pops address of `system()` and jumps to it

Note that we write multiple return addresses, which means we need to write **NULL** bytes on AMD64!



Countermeasures

- Can make sure a `0x00` is in the address of `libc`
 - Will stop string functions from reading past it
 - Mainly helps on x86, AMD64 addresses already contain `0x00` bytes
 - Only complicates string-based attacks
- Remove functionality from `libc`
 - What is necessary, and what is not though?
 - Compatibility issues?
 - What code exactly can cause problems?



Return-oriented programming

- As seen, we are not restricted to the functions in `libc`
 - We can use any gadget at any arbitrary address
- We can chain many such gadgets, if each ends in `return` (or `jump`).
- Use these gadgets to construct any code we want
- This is called **return-oriented programming**
- ROP enables *malicious computation* without `malicious code`
- Introduced in 2007 by Shacham, won ACM CCS 2007 Test of Time award.
- `libc` contains enough gadgets to allow ROP to be **Turing-complete**
- There are tools to automate the search for gadgets and ROP chains.



ROP: Example

vulnfunc()

```
...  
retq
```

0xfeedface

```
...  
xor %rax, %rax  
retq
```

0xcafebabe

```
...  
pop %rdi  
retq
```

0xdeadbeef

```
...  
mov %rdx, %rax  
pop %rsi  
retq
```

(corrupted) stack

0x7f1229d0f4a0 (execlp)
0x7f1229dd9f20 (“/bin/sh”)
0xdeadbeef
0xfeedface
0x7f1229dd9f20 (“/bin/sh”)
0xcafebabe
0x414141414141414141414141

registers

rax	unknown0x0
rdx	unknown0x0
rdi	unknown 0x7f1229dd9f20
rsi	unknown0x7f1229dd9f20

Will now jump to execlp with arguments in rdi, rsi, rdx
i.e. execlp(“/bin/sh”, “/bin/sh”, NULL);



Table of Contents

Recap

printf is Turing complete

Defeating $W \oplus X$

ASLR

Memory safety

Some announcements



Static addresses

- Both ROP and our shellcode-based attacks required us to know addresses
- Especially ROP requires exact addresses
 - Shellcode could maybe work around randomisation it with a large NOP sled and some brute force
- We have been switching off address randomization throughout our exercises because it makes life hard
 - `setarch -R bash`



ASLR

- Invented by the PaX project (publish patches for hardening Linux) in 2001
- First enabled by default in OpenBSD in 2003, Linux 2005
- Windows gained support in Vista (2007), only for enabled executables
 - It seems Windows 10 randomizes more executables
 - It also seems addresses are only rerandomized each reboot
- MacOS randomizes system libraries since October 2007 (OS X 10.5 Leopard)
 - All applications since 2011 (10.7 Lion), kernel since 2012.
- Android requires all code to support ASLR (PIE) since Android 5.0.



Implementing ASLR

- Move around locations of executable base address, libraries, stack and heap.
 - Stack, heap are easy to do: just change stack pointer and heap allocator managed by OS.
- Shared libraries have to be compiled with ASLR support: use relative instead of absolute addressing
 - “Position-independent code” (PIC) (compile with `-fPIC`)
- Executables can also be enabled for ASLR using `-fPIE`.
 - “Position-independent executable”
- Depending on your Linux distribution, these may be turned on by default.



Defeating ASLR

- Everything is loaded at an *offset*
- If the offset leaks we may compute the addresses
 - `printf`
 - Memory dumps
 - ...
- If only one library is not randomized, we can still ROP
- Side-channels sometimes leak the randomization
 - most famous are Spectre, Meltdown
- Guessing



Entropy

Problems on 32-bit machines: not enough room for randomness

- Cannot randomize lower 12 bits of address
 - Would break page alignment
- Cannot randomize upper 4 bits (breaks large memory mappings)
- Result: $32 - 12 - 4 = 16$ bits of entropy
- Only 65536 possibilities

Largely solved on 64-bit machines



Table of Contents

Recap

printf is Turing complete

Defeating W⊕X

ASLR

Memory safety

Some announcements



Why do we have all these problems

In C and C++,

- there is no information at run-time to check if we're within buffers
- It's not possible to reliably tell the size of a buffer given as an argument, leading to unsafe designs (`memcpy`, `gets`, `strcpy`)
- The compiler allows definition of variables without initializer.
- Pointers are completely managed by the programmer
- Heap especially is a complete headache: when to free, etc.
- Many of these problems are amplified when references are shared between threads
 - Data races
 - Complicated locking mechanisms
 - Which of the two threads needs to free, ...



Java's solution

- All code is compiled to special bytecode
- Bytecode runs in virtual machine (JVM)
- Heap is managed by JVM and garbage collector
 - Keeps track of all references and cleans up things that went out-of-scope
- Check all memory accesses if they're within scope
- Garbage collector frequently suspends threads to do cleanup



Python's solution

- Interpreted code, interpreter does all sorts of checks
- No fixed-size array type: all types resize themselves when necessary
- Also garbage-collected



Rust's solution

Observations

- Fixing bugs takes longer than spending more time on compile-time checks
- You can generate a lot of code with checks and rely on the compiler (LLVM) to optimize any unnecessary bits out.



Arrays

- Fixed-size arrays contain the size in the type of the function
`let array: [u8; 10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];`
- Silently generate functions for `array[0] . . . array[9]`.
- This means that the compiler can turn `array[10]` into a compiler error when it won't find such a function.
- Of course, for `array[var]` you will simply need to check if you're within bounds.
- Buffers for which the size is not known at compile time can only be put in resizable vectors.



Handling the heap

- Rust is designed to be compiled to machine code: no runtime environment
- That means no garbage collector, so heap needs to be managed otherwise
- Yet you do not want to burden the programmer with calling `free`...



Ownership

- Rust uses the concept of **ownership** to establish the *lifetime* of a variable
- Each variable has exactly one owner, although ownership may be passed on
- When ownership is transferred to another object, it is *moved*¹.
- The original function can no longer access it!

```
let value = Foo(); // create value
func(value);      // move value into func
value             // <-- compiler error
```

¹ C++11 also has move semantics, but they are optional, which means you need a lot of discipline



References

- To keep ownership, you can also pass on a reference (“borrow it”)
- You can have one or more read-only references OR one mutable reference
- This makes sure that there are no data races when accessing the variable concurrently.
- Checked by the compiler at compile-time



Lifetimes

- If you have a reference to a variable, how do you make sure it doesn't get deleted?
- Rust attaches a **lifetime** to the type of borrowed variables
- If the reference will outlive the owned variable, the *compiler* won't let it be borrowed!
- This also solves the "return a pointer to a stack variable" problem

```
{ let r;  
  { let x = 5; r = &x; }  
  println!("r: {}", r); }
```

```
error[E0597]: `x` does not live long enough
```

```
--> src/main.rs:3:26
```

```
3 | { let x = 5; r = &x; }  
  |                ^^ - `x` dropped here  
  |                |  
  | borrowed value does not live long enough
```



What has Rust learnt

- C specifies undefined behaviour and forces the programmer to avoid it
 - Admittedly, it's much simpler to write a C compiler
- In Rust, the much more advanced type system won't allow undefined behaviour
- Rust shows that you don't need a runtime environment to generate fast code.
- If you want to learn more about Rust
 - <https://rust-lang.org>
 - The Rust book <https://doc.rust-lang.org/book/>
- About type systems and compiler design
 - Compiler Construction (NWI-IMC004)
 - ▶ Master's course



Table of Contents

Recap

printf is Turing complete

Defeating W \oplus X

ASLR

Memory safety

Some announcements



Exercise 4

A write-up for exercise 4 is available on my website.



Exercise 5

Solutions to exercise 5 will be presented tomorrow, by me, in the tutorial.



Q&A

After the presentation of the solutions, I will have time to answer questions.



Exam

I will also talk a bit more about the exam tomorrow.

The deadline for the exam is on the last day of the exam period, so

Friday 3 July.

The exam will be a set of assignments. They will be in varying levels of difficulty.

You will be graded mainly on the write-up that you produce, much less so on if you manage to complete them all. We will be looking for you demonstrating a systematic approach, your analysis of what you see happening, and your understanding of the course material.



Exam (cont.)

The exam assignments will be **individual**. You can use any normal resource (books, internet); try to include what you use in your write-up and explain why any such thing applies. You are not supposed to work with other people or course participants on these assignments, however.

