Lesson 1: Buffer Overflows

Getting your feet wet, by pwning your first binary.

Leonardo Galli

flagbot (CTF@VIS)

November 2, 2021

Table of Contents

- Introduction
- Intermezzo: Game Hacking
- Intermezzo 2: Easy PicoCTF
- Setting up the Environment
- Buffer Overflow
  - The Stack
  - Overflow
  - ROP
- Exercises
- Further Readings

Introduction

Information

- Slides and additional material on our website flagbot.ch (under materials)
- Read the slides for Lesson 0, if you are new
- Subscribe to the mailinglist for receiving information: lists.vis.ethz.ch (CTF, not CTF-announce)

Web Security Workshop

- Collaboration between FAUST, polygl0ts (EPFL) and flagbot
- Workshop aimed at beginners to jump start them on web challenges
- Takes place on 9th and 10th October
- Registration deadline on Thursday, 7th October at midnight
- More information and registration at flagbot.ch/events

Disclaimer

Ethical Hacking

In these lessons you will gain firsthand experience with methods used to exploit all kinds of systems. Our purpose is mostly to help you get better at solving CTF challenges, so we strongly urge you to only practice ethical hacking.

We do not condone trying to gain access to any system you are not specifically authorized to do so. If you do find any vulnerabilities in software, always report it through the proper channels!

Disclaimer

Ethical Hacking

In these lessons you will gain firsthand experience with methods used to exploit all kinds of systems. Our purpose is mostly to help you get better at solving CTF challenges, so we strongly urge you to only practice ethical hacking.

We do not condone trying to gain access to any system you are not specifically authorized to do so. If you do find any vulnerabilities in software, always report it through the proper channels!

Intermezzo: Game Hacking

Showcase of Tasteless Shores

- Game challenges at this weekend's CTF
- Uses godot engine
- Quick showcase, no in depth slides (yet)
- More resources found at the end of the slides

Intermezzo 2: Easy PicoCTF

Discussion

What challenges did you try?

Setting up the Environment

How to not Infect Yourself

- Most of the time, challenges are geared towards Linux
- Thus, you will probably need to setup a virtual machine
- Even if you run linux natively, setting up a vritual machine has a lot of benefits:
  - No risk when running random binaries on your computer
  - Tooling is setup and ready to go immediately
  - If you fuck something up, just run `vagrant destroy && vagrant up` and you are good to go
  - Different VMs for different libc versions

How to not Infect Yourself

- Most of the time, challenges are geared towards Linux
- Thus, you will probably need to setup a virtual machine
- Even if you run linux natively, setting up a vritual machine has a lot of benefits:
  - No risk when running random binaries on your computer
  - Tooling is setup and ready to go immediately
  - If you fuck something up, just run `vagrant destroy && vagrant up` and you are good to go
  - Different VMs for different libc versions

Kali et al.

We strongly advise against using distros built for ”hacking”. While the tools they provide by default can be nice in some scenarios, CTFs often require quite different tools and challenges often work best on standard distributions such as ubuntu.

Making your Life Easy

- We have prepared tooling for you to setup a VM that is geared towards CTFs
- For detailed config instructions, read the config.rb file found once you download the repo
- Start downloading and setting everything up now, so that you can participate in the challenge later :)

Instructions For Now

1. Download and Install VirtualBox (Click on your OS here)
2. Download and Install Vagrant (here)
3. Clone our git repository from here:
   `git clone https://gitlab.ethz.ch/vis/ctf/ctf-vm.git`
4. Change config.rb to your liking (most likely you only need to change your shared folders and the ssh key)
5. Run `vagrant up 27` to start the download and installation of the VM.

```
:shared_folders => {
    "/path/to/my/projects" => "/home/vagrant/CTF"
},
# ...
:ssh_key => "/path/to/my/user/.ssh/id_rsa.pub",
```

Buffer Overflow

Buffer Overflow

The Stack

The Stack

- Before we can start overflowing buffers, we need to learn about "the stack" (AKA the call stack)
- stack data structure, that stores information about the active functions (or subroutines) of a computer program
- Every function has a so called "stack frame" - on the stack - where information is stored, such as
  - local variables (numbers, strings, arrays)
  - saved instruction pointer - the return address (used to know where to return back to, when the function is done)
  - additional arguments (not really relevant yet)
- Local variables allocated in the stack go from low to high addresses, i.e. we may find the first character of `"Hello World"` at `0x7fa0`, while the last character would be at `0x7fab`
- However, the stack grows downwards, i.e. if we push items a, b on the stack (in that order), we may find a at `0x7698`, while b would be at `0x7690`

Example

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4; ⇐
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```

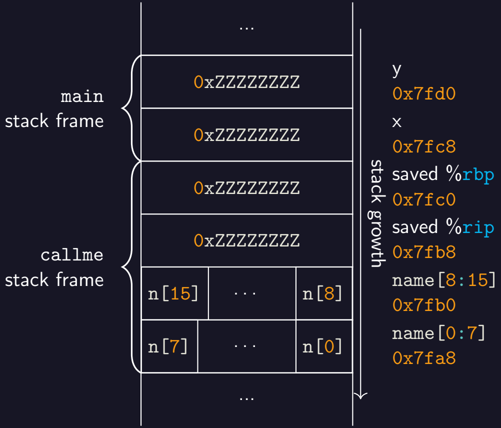
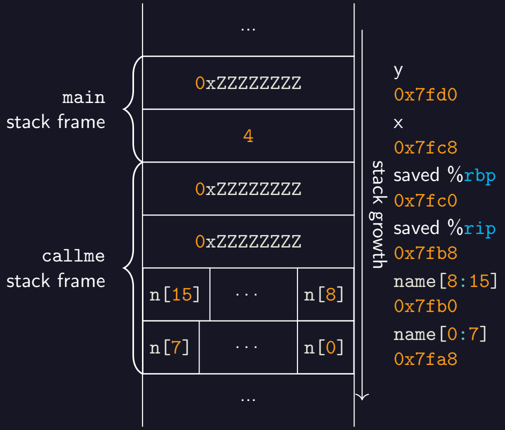


Figure: The Stack

Example

```c
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme(); ⇐
    printf("x + y = %d", x + y);
    return 0;
}
```



...

main
stack frame

0xZZZZZZZZ — y 0x7fd0

4 — x 0x7fc8

callme
stack frame

0xZZZZZZZZ — saved %rbp 0x7fc0

0xZZZZZZZZ — saved %rip 0x7fb8

n[15] · · · n[8] — name[8:15] 0x7fb0

n[7] · · · n[0] — name[0:7] 0x7fa8

...

stack growth

Figure: The Stack

Example

```c
int callme() {
    char name[16];
    gets(name); ⇐
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```

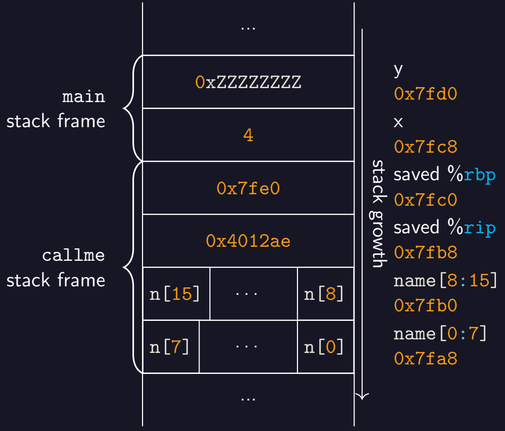
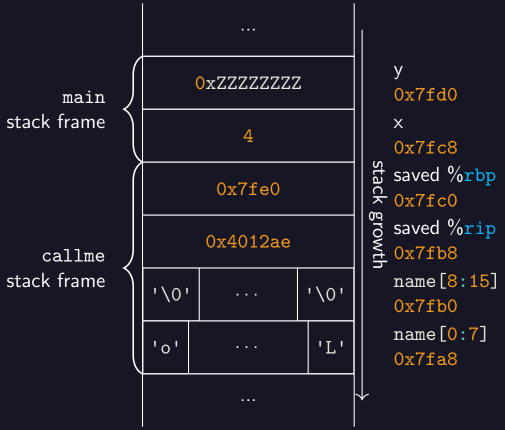


Figure: The Stack

Example

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name); ⇐
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```



...

main stack frame

callme stack frame

0xZZZZZZZZ — y 0x7fd0

4 — x 0x7fc8

0x7fe0 — saved %rbp 0x7fc0

0x4012ae — saved %rip 0x7fb8

'\0' ... '\0' — name[8:15] 0x7fb0

'o' ... 'L' — name[0:7] 0x7fa8

...

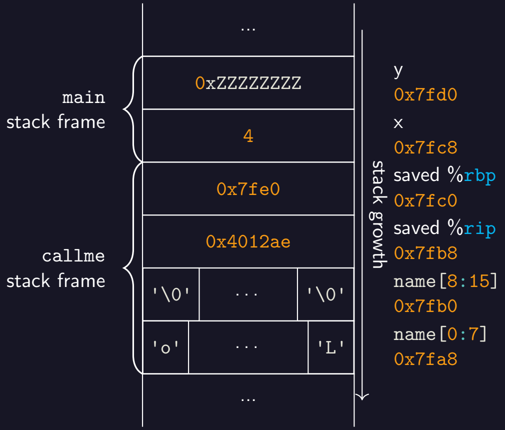
stack growth

Figure: The Stack

Example

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2; ⇐
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```



...

main stack frame

0xZZZZZZZZ y 0x7fd0

4 x 0x7fc8

callme stack frame

0x7fe0 saved %rbp 0x7fc0

0x4012ae saved %rip 0x7fb8

'\0' ... '\0' name[8:15] 0x7fb0

'o' ... 'L' name[0:7] 0x7fa8

...

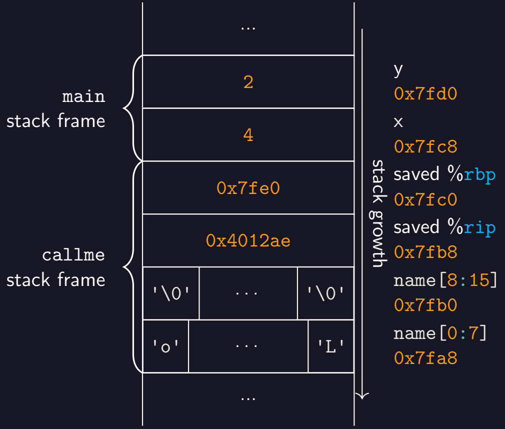
stack growth

Figure: The Stack

Example

```c
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y); ⇐
    return 0;
}
```



...

main
stack frame

2 — y 0x7fd0

4 — x 0x7fc8

callme
stack frame

0x7fe0 — saved %rbp 0x7fc0

0x4012ae — saved %rip 0x7fb8

'\0' ... '\0' — name[8:15] 0x7fb0

'o' ... 'L' — name[0:7] 0x7fa8

...

stack growth

Figure: The Stack

Example

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0; ⇐
}
```

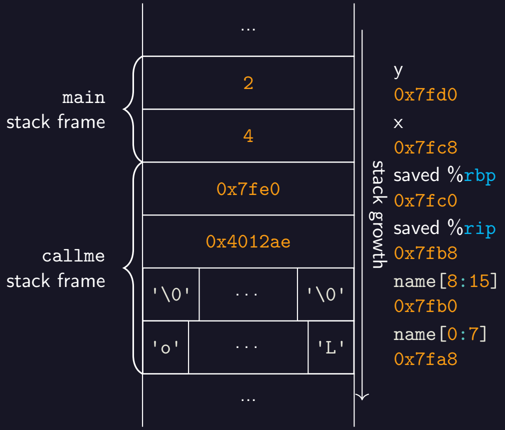


Figure: The Stack

Buffer Overflow

Overflow

C is hard

- C is a very low level language
- Basically no checks in place!
  - No type checks, no checks on array sizes, etc.!
- At every step the programmer needs to make sure, user input is correct

Programming Mistakes

- Common mistakes are using functions that do not check sizes of strings, arrays, etc.
  - Examples are: `gets, strcpy, sprintf, strcat, ...`
- If you see them used in a program, immediately assume there is an attack vector here
- How can we exploit this?

Exploit?

Almost always, the ”holy grail” is getting code execution when trying to exploit the attack vector. Because these programs are either run on a server or - more general - a remote computer, this allows us to execute code on it.
However, often this might entail using multiple different attack vectors. Furthermore, sometimes reading arbitrary memory can already be enough.

Buffer Overflow
ROP

Getting Code Execution

- If the programmer fails to check the length of our input, we can overwrite stuff on the stack
- What of interest is stored on the stack?

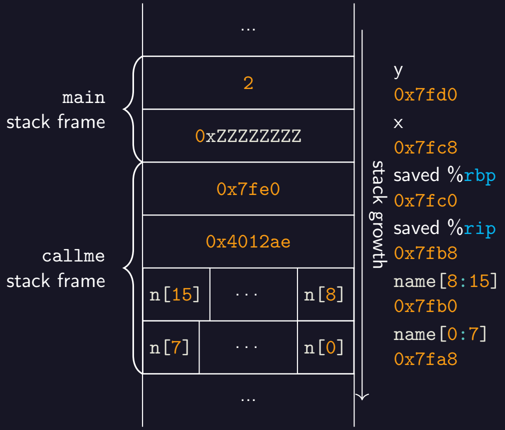
Getting Code Execution

- If the programmer fails to check the length of our input, we can overwrite stuff on the stack
- What of interest is stored on the stack?
- If we overwrite the saved instruction pointer, we can control where the function returns to
- We have effectively gained code execution!
- Let's give the example program 32 characters of A (`0x41` in hex)

Example 2.0: Smashing the Stack for Fun and Profit

```
int callme() {
    char name[16];
    gets(name); ⇐
    printf("Hello %s", name);
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```



...

main
stack frame

2 — y 0x7fd0

0xZZZZZZZZ — x 0x7fc8

0x7fe0 — saved %rbp 0x7fc0

callme
stack frame

0x4012ae — saved %rip 0x7fb8

n[15] ... n[8] — name[8:15] 0x7fb0

n[7] ... n[0] — name[0:7] 0x7fa8

...

stack growth

Figure: The Stack: Smashed

Example 2.0: Smashing the Stack for Fun and Profit

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name); ⇐
    return 2;
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```

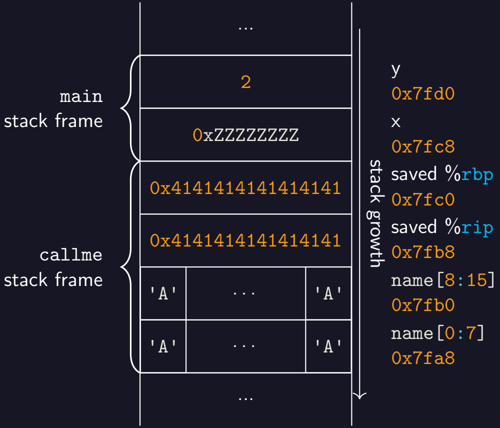
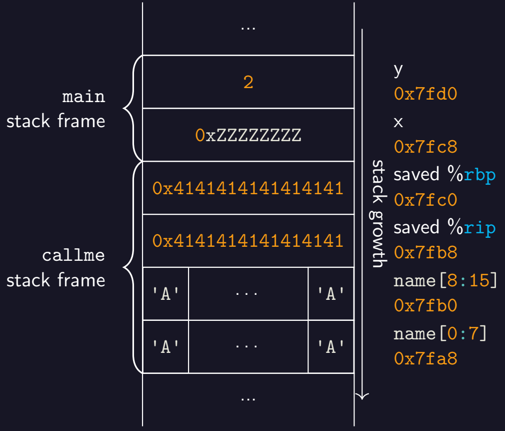


Figure: The Stack: Smashed

Example 2.0: Smashing the Stack for Fun and Profit

```
int callme() {
    char name[16];
    gets(name);
    printf("Hello %s", name);
    return 2; ⇐
}

int main(int argc, char* argv[]) {
    long x = 4;
    long y = callme();
    printf("x + y = %d", x + y);
    return 0;
}
```



...

main stack frame

2 — y 0x7fd0

0xZZZZZZZZ — x 0x7fc8

callme stack frame

0x4141414141414141 — saved %rbp 0x7fc0

0x4141414141414141 — saved %rip 0x7fb8

'A' ... 'A' — name[8:15] 0x7fb0

'A' ... 'A' — name[0:7] 0x7fa8

...

stack growth

Figure: The Stack: Smashed

Example 2.0: Smashing the Stack for Fun and Profit

- What happens now?

Example 2.0: Smashing the Stack for Fun and Profit

- What happens now?

Segmentation Fault

```
Program received signal SIGSEGV, Segmentation fault.
```

If we would look at the program in a debugger, we would see that it tried executing code at address `0x4141414141414141`! Mission accomplished!

Return Oriented Programming

- While this is already cool, we need to have a useful location to jump to, otherwise this alone is useless
- Enter Return Oriented Programming (ROP)
- By finding small snippets of useful assembly code in the binary (or others loaded by it), that we can chain together (ROP chain), to achieve a goal
  - useful usually means: does something we want (e.g. setting a register to a certain value, setting register value from stack, etc.) and also contains another return instruction!
  - Otherwise, we could not continue our chain!
  - Goal is usually to call `system("/bin/sh")`, which gives us a remote terminal
  - Achieved by setting the registers %`rdi` = address of string `"/bin/sh"`, %`rsi` = 0, %`rdx` = 0, then jumping to address of `system`

Finding Gadgets and Addresses

- For now you can use `objdump -d program` to get assembly and location for the functions
- Looking through the different functions you can find gadgets by hand
- Often the values you want to have on the stack are not nice characters (for example 0x80)
- Use `echo -ne '\x80\x40\x00\x7f' | ./program` to pass 0x7f004080 to the program

Exercises

PicoCTF

- `clutter-overflow`
  - Easy buffer overflow
  - No need for ROP
- `Guessing Game 1`
  - A lot harder
  - Needs ROP
  - Might want to hold off on this for now :)

flagbot Challenge

babybof

This is a simple buffer overflow challenge. Both binary and source code are provided on our website under materials. Once you have a working exploit, you can run it against the server.

There are two flags for this challenge, one is easier to get, while the other is in a file called flag2. First one to claim a flag, gets the right to present their exploit ;)

Hints: For flag1 you just need to ”call” the win function. For flag2 you need to get a shell.

Files: babybof.zip

Server: `spclr.ch 1337`

Author: Robin Jadoul

Further Readings

Game Hacking

- Pwn Adventure 3: Pwnie Island Walkthrough by Live Overflow on YouTube
- CheatEngine: useful tool for memory editing
- ILSpy: .NET decompiler, useful for C# based games, e.g. Unity
- gdsdecomp: Godot reverse engineering tools
- BasteG0d69 on YouTube: Diverse hacking videos, sometimes game related :)

Assembly, Disassemblers and Decompilers

- x86 Assembly
  - ”Machine Language” of modern Intel/AMD processors and so most binary challenges use it
  - x86 Assembly Guide
  - CS:APP Chapter 3: Machine-Level Representation of Programs
- Disassemblers
  - Take a binary and display the x86 assembly nicely, cross reference stuff, etc.
  - `objdump -D binary` is the simplest version
  - Cutter (powered by radare2)
- Decompilers
  - Take the output of a dissassembler and try to create C code that does the same thing as the assembly
  - Usually Programms do both
  - Ghidra is pretty good and available for free at ghidra-sre.org
  - Introduction to Ghidra

Advanced ROP Techniques and Protection Measures

- ROPing all the things!
  - ROP on a voting machine
  - ROP on an Adobe Reader PDF
- Must-have tools for ROP:
  - ROPgadget
  - ropper
- Cool ROP Techniques:
  - (basic) ret2libc
  - (hard) sigreturn oriented programming
- Defending against ROPs:
  - ASLR
  - G-free
  - PAC: Pointer Authenication (use crypto to secure return pointers)