Intro to Binary Exploitation

Contacts:

Telegram & Discord: @MatteB_01

OUR TELEGRAM CHANNEL AND DISCORD SERVER

>>>>>>> m0leCon_Beginner <<<<<<<<

Useful Resources

- https://pwn.college/
- http://pwnable.kr/
- https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/
- https://github.com/Tzaoh/pwning
- https://beginners.re/

What is binary exploitation?

The art of exploiting vulnerabilities in a program to achieve something not intended.

- This can be done through **ANY** input

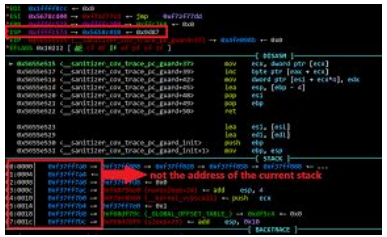
What does PWNING mean?

> **Verb** [edit]
>
> **pwn** (*third-person singular simple present* **pwns**, *present participle* **pwning**, *simple past and past participle* **pwned** *or* **pwnd** *or* **pwnt**)
>
> 1. (*Internet slang*, *online gaming*, *originally leet*, *transitive*, *intransitive*) To own, to defeat or dominate (someone or something, especially a game or someone playing a game).

What you will need

Low-level

- How a program works
- Disassemblers
- Decompilers
- Assembly (x86, ARM, MIPS ...)



High-level

- C (reversing)
- Python (exploiting)

The thought process of a pwner

- what does a program/service do
- what’s stinky
- what could I investigate further
- i found a misbehaviour, what caused it, can i escalate it further?

static and dynamic analysis of a program

STATIC ANALYSIS

- Done by analyzing the source code (or what has been decompiled)
- Requires decompilers (Ghidra, IDA, Binary Ninja, etc.)

DYNAMIC ANALYSIS

- Done by analyzing the flow of the program while it is running
- Requires debuggers (gdb, pwndbg) and/or tracers (ltrace, strace)

Dynamic Analysis

gdb and pwndbg ➜ “The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes. “

“Pwndbg is a Python module which is loaded directly into GDB, and provides a suite of utilities and crutches to hack around all of the cruft that is GDB and smooth out the rough edges.”

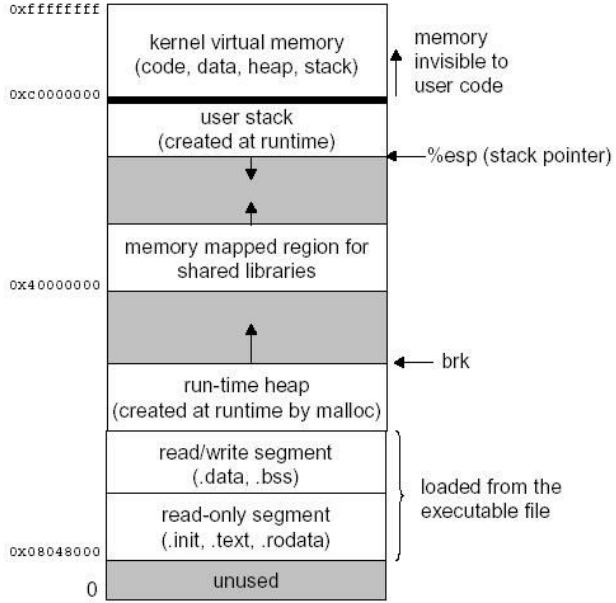
ltrace ➜ “It intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process. It can also intercept and print the system calls executed by the program.”

strace ➜ “runs the specified command until it exits. It intercepts and records the system calls which are called by a process and the signals which are received by a process”

BUFFER OVERFLOWS

Process memory and the stack

- what's the stack
- what's a stack frame
- calling a function



The stack

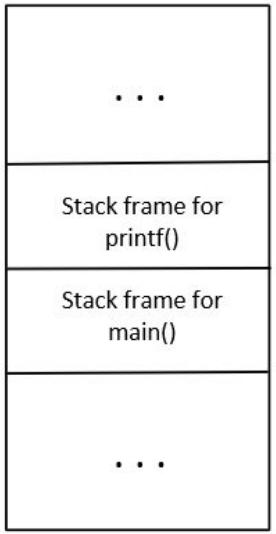
The stack is the area inside a process memory designed to store user data (variables, function pointers, etc.).

The whole stack can be divided into smaller pieces called **frames**. Each frame will store the function's own data (like local variables, but more on that later).

In assembly, there's a specific registry (ESP/RSP) that keeps track of where the stack is and that is used to access local variables inside the frame.

Function calls

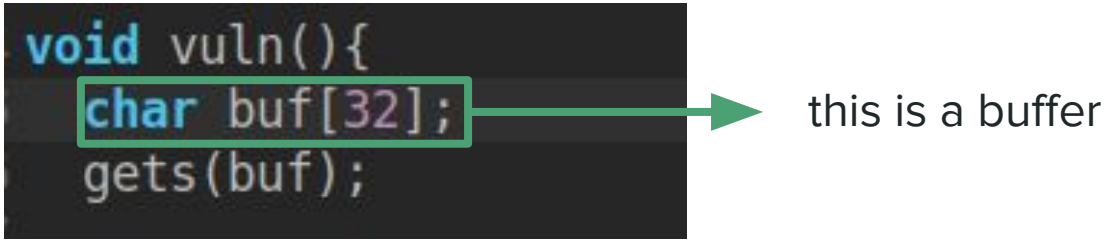
When a function gets called, a stack frame is created for that function. Suppose we have a program in which the main() function calls printf(); the situation will look something like this:



The frames are limited by **return pointers**, which are pointers used by the code to know where the execution needs to go after the called function is done executing.
As per our example, saying that printf() is called by main() on line 5, the return pointer for the frame will point to line 6 of main().

What is a buffer?

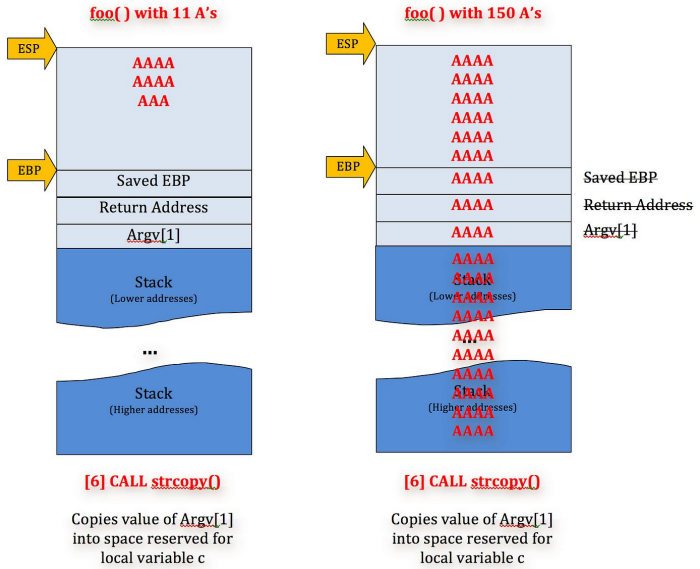
A buffer is simply a sequential section of memory allocated to contain anything.



Buffer overflows

A buffer overflow (in short BOF), occurs when **more data** is put into a fixed-length buffer than the buffer can handle. The extra information may overflow into adjacent **memory space**, corrupting or **overwriting** the data held there.

This can allow the attacker to **hijack** the normal flow of the process in various ways



real world BOF vulnerabilities

- **CVE-2022-47949** → buffer overflow in Animal Crossing
- **CVE-2018-6242** → buffer overflow allowed arbitrary code execution via USB payloads

CVE-2019-8050 → buffer overflow while processing malformed PDF

CVE-2023-5474 → buffer overflow via corrupted PDF

spotting a BOF vulnerability

There are many ways in which a code can be vulnerable to buffer overflows, for example many C functions **do not have** any check for the buffer size, the most common ones are:

- **scanf();**
- **strcpy();**
- **gets();**
- **read();**

shellcodes

A **shellcode** is a small piece of executable code used as a payload, built to exploit vulnerabilities in a system or carry out malicious commands. This can easily be chained with other type of vulnerabilities that make the execution flow jump to it in order to make it achieve its tasks.

```
; execve("/bin/sh", ["/bin/sh"], NULL)

section .text
        global _start

_start:
        xor     rdx, rdx
        mov     qword rbx, '//bin/sh'
        shr     rbx, 0x8
        push    rbx
        mov     rdi, rsp
        push    rax
        push    rdi
        mov     rsi, rsp
        mov     al, 0x3b
        syscall
```

The snippet of code on the left makes the process spawn an interactive shell for the attacker to use

Mitigations on a program

- **Canaries:** values stored before each return pointer.
- **NX:** Non-eXecutable, code written on the stack can’t be executed.
- **PIE:** Position Independent Execution, the code will be at a different address each run.

BOF CHALLENGES

FORMAT STRING VULNERABILITIES

Format strings

```
printf("My age is %d",23);
```

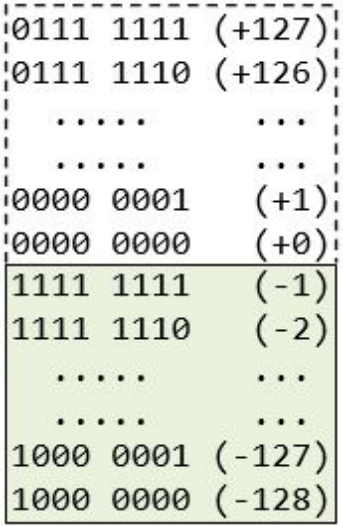
The printf() function normally expects the first argument to be a string containing format specifiers (%s, %d, %x, ...). However, if a bad programmer makes a call to printf() and feeds it user-controlled input as a first argument, things can get dangerous.

printf() will treat the user input as a format string; if the latter contains format specifiers, printf() will gently replace those with data from the stack and cause a **memory leak**, which an attacker might easily escalate

INTEGER ISSUES

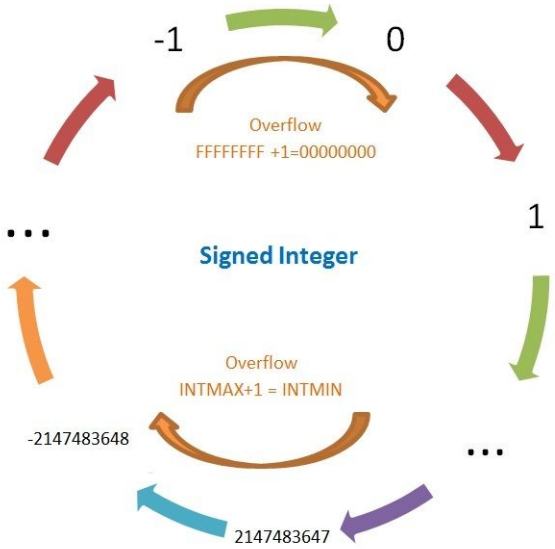
Integer issues

In most programming languages, integer have a default size of 32 bits, the most significant one of which is used as a **“sign” identifier** (32th bit set to 1 means the number is negative). On an 8-bit representation (where the 8th is the sign bit), integers look like this:



Exploiting integer issues

Suppose we have a program that keeps incrementing an integer on user's request without any checks, what might happen is that **the integer gets so big** (all the bits are set to 1 except the 32th) that incrementing it again will cause it to become negative (setting the 32th bit to 1). This might often allow an attacker to **bypass checks** or perform generally malicious activities.



In last year’s workshop


```
mov     rax, 0x3
cvtsi2ss xmm1, rax
mov     rax, 0x2710
cvtsi2ss xmm0, rax
mov     rax, qword [rbp-0x38 {var_40}]
cvtsi2ss xmm2, dword [rax+0x38]
divss   xmm2, xmm0
movss   dword [rbp-0x24 {var_2c_1}], xmm2
cvtsi2ss xmm0, dword [rbp-0x1c {var_24}]
movss   xmm2, dword [rbp-0x24 {var_2c_1}]
movss   dword [rbp-0x3c {var_44_1}], xmm0
movaps  xmm0, xmm2
call    powf
// vita come valore signed
mov     ecx, 10000
mov     rax, 0x4
cvtsi2ss xmm1, rax
movss   xmm2, dword [rbp-0x3c {var_44_1}]
mulss   xmm2, xmm0
mulss   xmm2, xmm1
cvttss2si edx, xmm2
mov     dword [rbp-0x28 {var_30_1}], edx
mov     rax, qword [rbp-0x38 {var_40}]
sub     ecx, dword [rax+0x38]
mov     dword [rbp-0x2c {var_34_1}], ecx
mov     ecx, dword [rbp-0x28 {var_30_1}]
cmp     ecx, dword [rbp-0x2c {var_34_1}]
// jump below or equal =  controllo per unsigned int
jbe     0x13afeb
```

THANKS FOR YOUR ATTENTION