Security Workshop

HTS

LSE Team

EPITA 2018

February 3rd, 2016

Introduction

What is this talk about?

- Presentation of some basic memory corruption bugs
- Presentation of some simple protections
- Writing some (really basic) exploits

Notes:

- Sources of the exercises/examples at https://www.lse.epita.fr/data/workshop-secu.tar.gz
- Lots of them come from http://www.exploit-exercises.com

Plan

- Exploitation 101
    - ASM for exploitation
    - Shellcodes
- Buffer Overflows
    - Stack overflow
    - Heap overflow
- Format string
- Preventions
    - DEP
    - ASLR
    - PIE
- ROP
- Going further

Exploitation 101

- ASM for exploitation
  - `%eip`: Program counter: pointer to instruction to be executed
  - `%esp`: Stack pointer

Exploitation 101 - Push

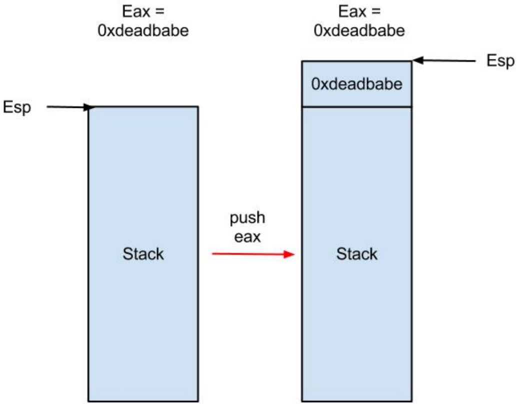


Figure: Push

Exploitation 101 - Pop

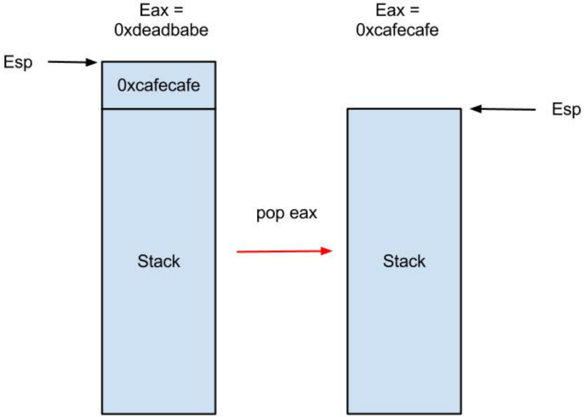


Figure: Pop

Exploitation 101 - jmp, call, ret & int 0x80

- jmp XXX: %eip = XXX
- call XXX:
  - push %eip
  - jmp XXX
- ret: pop %eip
- int $0x80: syscall

Shellcodes

- ”A shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability.” (Wikipedia)
- Called shellcode because the usual goal is to get a shell.
- In general it is the final step of exploitation.
- Triggering the vulnerability allows you to ”jump” on your shellcode.

Shellcode is an art

There is lots of methods used when writing shellcodes:

- Nop sled
- Null-free (or any other kind of restriction)
- multi-staged shellcode
- self-deciphering shellcode
- …

Exercise steps (shellcode)

```
int main()
{
    char input[4096];
    open("flag", O_RDONLY);
    read(0, input, 4096);
    ((func)&input)();
    return 0;
}
```

- Find a place to write the data
- Read the content of the open file into a buffer
- Write the content of the buffer to STDOUT
- No null bytes (shellcode1)

Buffer overflows: Generality

- Really simple principle
- Possibility of writing past the bounds of a buffer (wherever it is)
- When?
    - Some functions trigger a BOF in ”almost” every case (gets, strcpy, . . .)
    - Just bad code. . .
- Two major categories (stack and heap), but virtually anywhere

Stack overflows: The stack ?

- A memory zone (like the heap)
- Used for:
  - Passing arguments (calling convention dependant, but for x86_32 it's generally on the stack)
  - Local variables
  - *Return address of the functions*

Stack overflows: The stack ?

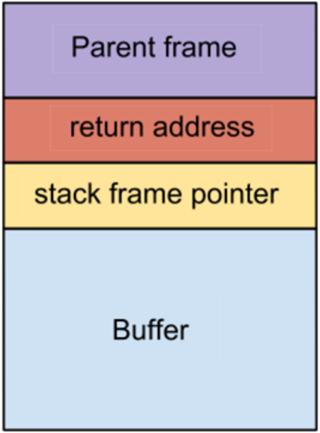


Figure: The stack

Exercise: overflow0

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    volatile int modified;
    char buffer [64];
    modified = 0;
    gets(buffer);
    if (modified != 0)
        printf("you have changed %d\n", modified);
    else
        printf("Try again?\n");
}
```

Exercise: overflow1

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
// gcc -m32 -o main main.c

void win()
{
    printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
    char buffer[64];
    gets(buffer);
}
```

Stack overflows prevention: Canary

```c
#include <stdio.h>
// gcc -fstack-protector-all example.c
int example(void)
{
    return getchar();
}

int main(void)
{
    return example();
}
```

Stack overflows prevention: Canary

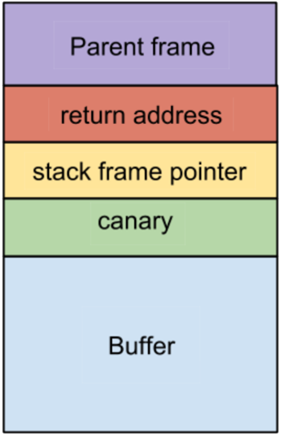


Figure: The stack with the canary

Stack overflows prevention: Canary

```
push    %rbp
mov     %rsp,%rbp

sub     $0x10,%rsp                     ; Allocate slot
mov     %fs:0x28,%rax                  ; Get canary
mov     %rax,-0x8(%rbp)                ; "Push" it.
xor     %eax,%eax
callq   400460 <getchar@plt>
mov     -0x8(%rbp),%rdx                ; Take it back
xor     %fs:0x28,%rdx                  ; xor with original

je      400596 <example+0x30>
callq   400450 <__stack_chk_fail@plt> ; If not equal, abort
leaveq
retq
```

Exercise steps (overflow2)

- Overflow the buffer byte by byte to leak the canary
- Overwrite it and the return address
- Take control of the instruction flow

Heap overflow

- The heap is just another memory zone
- Used when calling malloc, new…
- No return address to overwrite on the heap :(
- The goal is to rewrite:
  - a pointer (hopefully leading to a write-what-where)
  - a function pointer (hopefully called later)
  - some metadata (like the malloc metadata)
  - a vtable pointer (C++ code only)

Heap overflow - Example

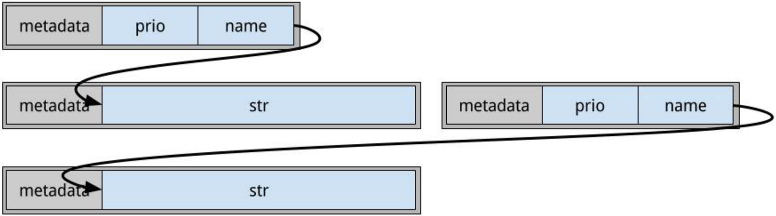


Figure: Clean heap

strcpy(i1->name, argv(1));

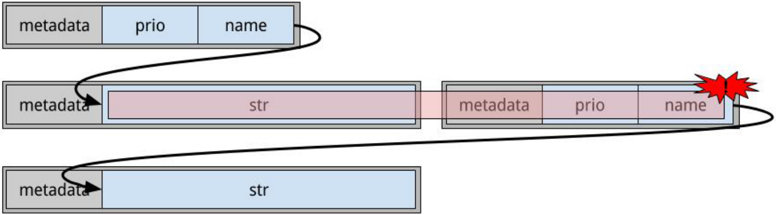
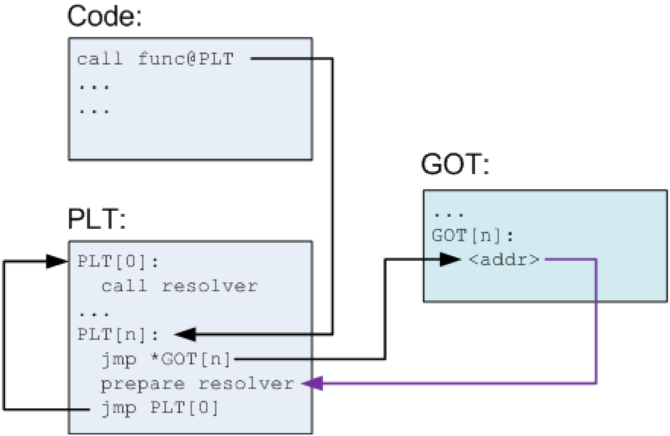


Figure: Overflow

Write-What-Where

- write-what-where: we can write what we want where we want (almost always equivalent to success)
- what can we possibly want to rewrite?
  - A stack return? Good if we don't have ASLR, but what if we have some?
  - A function pointer? Sure if we have one, and know where it is
  - The GOT? Almost always one, not affected by ASLR (but if we have PIE we are doomed)

PLT & GOT



Code:

```
call  func@PLT
...
...
```

GOT:

```
...
GOT[n]:
  <addr>
```

PLT:

```
PLT[0]:
  call  resolver
...
PLT[n]:
  jmp *GOT[n]
  prepare resolver
  jmp PLT[0]
```

Figure: PLT/GOT

PLT & GOT

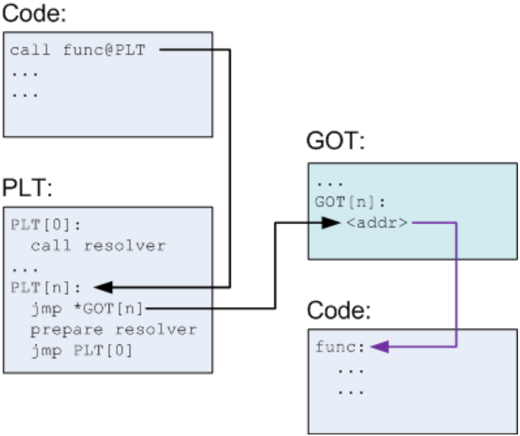


Figure: PLT/GOT

Exercise step (heap0)

- Find `exit`'s PLT entry
- Rewrite the second allocation metadata
- Write over `exit`'s PLT entry with `winner` address

Format string

- Remember that `va_arg` doesn't know the function's *arity*? That's quite sad…
- What can we do with it?
  - We can obviously leak data…
  - What about `%n`? The number of characters written so far is stored into the integer specified by the `int*` (or variant) pointer argument. No argument is converted.

%n

```
int i;
printf("%s%n", "Hello", &i);
```

- i == 5
- We can write to the address of a given argument (&i) the *number we want*. We just need to control the address to have a write-what-where.
- At one point the arguments are taken from the stack (`va_arg`). If the buffer was once on the stack: we can take the content of the buffer as argument.
- Lets get the address we want in the buffer and take this one as argument for the `%n`: we have a write-what-where!

Some tips for the format

- %(num)$(option) takes the arg. num for option
  - `%2$x`: draws the hex value of the second arg.
- %.(num)(option) draws at least num byte(s) (actually depends on the given option)
  - `%.200x`: draws the first arg with at least 200 chars
- `%n` writes an int at the address
- `%hn` writes a short at the address
- `%hhn` writes a byte at the address

Example

```c
int main()
{
        // 0xcafe = 51966
        long i = 0x22222222;
        printf("\i = 0x%x\n", i);
        printf("%.51966x%1$hn", &i);
        printf("\ni = 0x%x\n", i);
}
```

```
$ ./format
i = 0x22222222
...
i = 0x2222cafe
```

Exercise steps (format0)

- Find the return address on the stack
- Overwrite it with `hello`'s address

Preventions: DEP

- Data Execution Prevention (NX, W^X, …)
- Basic idea is `Write XOR Execute`
- You can't execute code on your stack, heap…

Preventions: ASLR

- Address space layout randomization
- If not enabled, everything is always at the same address (the stack, the heap, the libraries…)
- When enabled the base address of the stack, the heap and the libraries are randomized.
- But the address of the loaded binary is not.
- `echo 1 > /proc/sys/kernel/randomize_va_space`

Preventions: PIE

- Position-independent executable
- Like ASLR but with the base of the binary randomized.
- `echo 2 > /proc/sys/kernel/randomize_va_space`
- `-fpie` for gcc, `-pie` for ld

Bypassing ASLR & PIE

- We need an `address leak`
- Then, we can calculate the position of the base address
  - `offset = ref_leaked_addr - ref_base_address`
  - `base = leaked_addr - offset`
- Once we have the base address, ASLR is down. Really simple *once we have a leak*…

ROP

- Return Oriented Programming.
- The idea: Rewrite the whole stack and use returns to call the parts of code we want.
- With DEP, we can't inject our own code and get it executed. So we just reuse code from the binary.
- No good technique (yet) to prevent ROP, but quite painful to write.
- In x86, ROP is simpler because the calling convention uses the stack while it uses registers in x64.

Gadgets

- To ROP, we need sequences of instructions ending with ret (or something like `call *%eax`).
- This kind of sequence is called a gadget
- A typical gadget, is something like `pop [REG]; ret`
- Some tools for finding gadgets already exist:
  - ropmount by Hakril
  - ROPgadget by JonathanSalwan
  - rp by 0vercl0ck

Exercise steps (rop)

- 1 exercises, 3 versions
- Modification of a DEFCON Quals 2013 exercise.
- The goal is to call system from libc…
- Probably an infinity of solutions…
- Easy: x86 - no stack protection, no ASLR
- Half: x64 - stack-protection, no ASLR
- Hard: x64 - stack-protection, PIE
- Toggling ASLR/PIE:
  - x=0: no ASLR - noPIE
  - x=1: ASLR - no PIE
  - x=2: ASLR - PIE
- `echo $x > /proc/sys/kernel/randomize_va_space`

Going further

- Other vulnerability types (use-after-free, off-by-one, heap spraying)
- Vulnerability discovery (fuzzer, static analysis. . .)
- Metadata corruptions
- Sandbox escape
- Kernel exploitation
- Windows exploitation

Links

- ctf.lse.epita.fr
- exploit-exercises.com/
- https://www.root-me.org
- crackme.de
- reddit.com/r/netsec
- oss-sec
- bugtrack
- fulldisclosure
- phrack.org