Sylvain Kerkour

Home Follow About My Book

Software development and security tips from the field. Mostly Rust and Go. Creator of Bloom and author of Black Hat Rust.

How to execute shellcodes from memory in Rust

Tue, Mar 23, 2021

Executing code from memory in Rust is very dependant of the platform as all modern Operating Systems implement security measures to avoid it. The following applies to Linux.

There are at least 3 ways to execute raw instructions from memory:

- By embedding the shellcode in the `.text` section of our program by using a special attribute.
- By using the mmap crate and setting a memory-mapped area as `executable`.
- A third alternative not covered in this post is to use Linux's mprotect function.

Our shellcode

We will use the following shellcode:

```
488d35140000006a01586a0c5a4889c70f056a3c5831ff0f05ebfe68656c6c6f20776f726c640a
```

Which is a basic hello-world for `x86_64`.

We can use the following command to write it to a binary file:

```
$ echo '488d35140000006a01586a0c5a4889c70f056a3c5831ff0f05ebfe68656c6c6f20776f726c64
```

```
$ objdump -D -b binary -mi386 -Mx86-64 -Mintel shellcode.bin
```

```
shellcode.bin:     file format binary



Disassembly of section .data:


00000000 <.data>:
   0:   48 8d 35 14 00 00 00    lea    rsi,[rip+0x14]        # 0x1b
   7:   6a 01                   push   0x1
   9:   58                      pop    rax
   a:   6a 0c                   push   0xc
   c:   5a                      pop    rdx
   d:   48 89 c7                mov    rdi,rax
  10:   0f 05                   syscall           // <- write(1, "hello world\n", 12
  12:   6a 3c                   push   0x3c
  14:   58                      pop    rax
  15:   31 ff                   xor    edi,edi
  17:   0f 05                   syscall           // <- exit
  19:   eb fe                   jmp    0x19
  1b:   68 65 6c 6c 6f          push   0x6f6c6c65 // <- hello world\n
  20:   20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  23:   72 6c                   jb     0x91
  25:   64                      fs
  26:   0a                      .byte 0xa
```

Embedding a shellcode in the `.text` section

Embedding a shellcode in our program is easy thanks to the `include_bytes!` macro, but adding it to the `.text` section is a little bit tricky as by default **only the reference to the buffer** will be added to the `.text` section, and not the buffer itself.

It can be achieved as follow:

main.rs

```
use std::mem;
```

```
// we do this trick because otherwise only the reference of the buffer is in the .te
// and not the buffer itself
const SHELLCODE_BYTES: &[u8] = include_bytes!("../shellcode.bin");
const SHELLCODE_LENGTH: usize = SHELLCODE_BYTES.len();

#[no_mangle]
#[link_section = ".text"]
static SHELLCODE: [u8; SHELLCODE_LENGTH] = *include_bytes!("../shellcode.bin");

fn main() {
    let exec_shellcode: extern "C" fn() -> ! =
        unsafe { mem::transmute(&SHELLCODE as *const _ as *const ()) };
    exec_shellcode();
}
```

```
$ cargo run
hello world
```

Subscribe

I'll write you once a week about software development, hacking, and entrepreneurship.

You can subscribe by Email, Matrix or RSS.

Setting a memory-mapped area as executable

Cargo.toml

```
[dependencies]
mmap = "0.1"
```

main.rs

```
use mmap::{
    MapOption::{MapExecutable, MapReadable, MapWritable},
    MemoryMap,
};
use std::mem;

// as the shellcode is not in the `.text` section, we can't execute it as it
const SHELLCODE: &[u8] = include_bytes!("../shellcode.bin");

fn main() {
    let map = MemoryMap::new(SHELLCODE.len(), &[MapReadable, MapWritable, MapExecutal

    unsafe {
        // copy the shellcode to the memory map
        std::ptr::copy(SHELLCODE.as_ptr(), map.data(), SHELLCODE.len());
        let exec_shellcode: extern "C" fn() -> ! = mem::transmute(map.data());
        exec_shellcode();
    }
}
```

The code is on GitHub

As usual, you can find the code on GitHub: github.com/skerkour/kerkour.com (please don't forget to star the repo 🙏).

Want to learn Rust and offensive security? Take a look at my book **Black Hat Rust**.
All early-access supporters get a special discount and awesome bonuses:
https://academy.kerkour.com/black-hat-rust?coupon=BLOG.
Warning: this offer is limited in time!

Tags: rust, programming, tutorial, hacking, security, shellcode

Related posts

- How to send emails with Rust
- The biggest threat to Rust's sustainability
- Rust for web development: 2 years later
- The unusual way I'm funding my open source work
- Bloom December update: Native applications are coming