

SafeLLVM: LLVM Without The ROP Gadgets!

Federico Cassano

Northeastern University

Charles Bershatsky

Northeastern University

Jacob Ginesin

Northeastern University

Sasha Bashenko

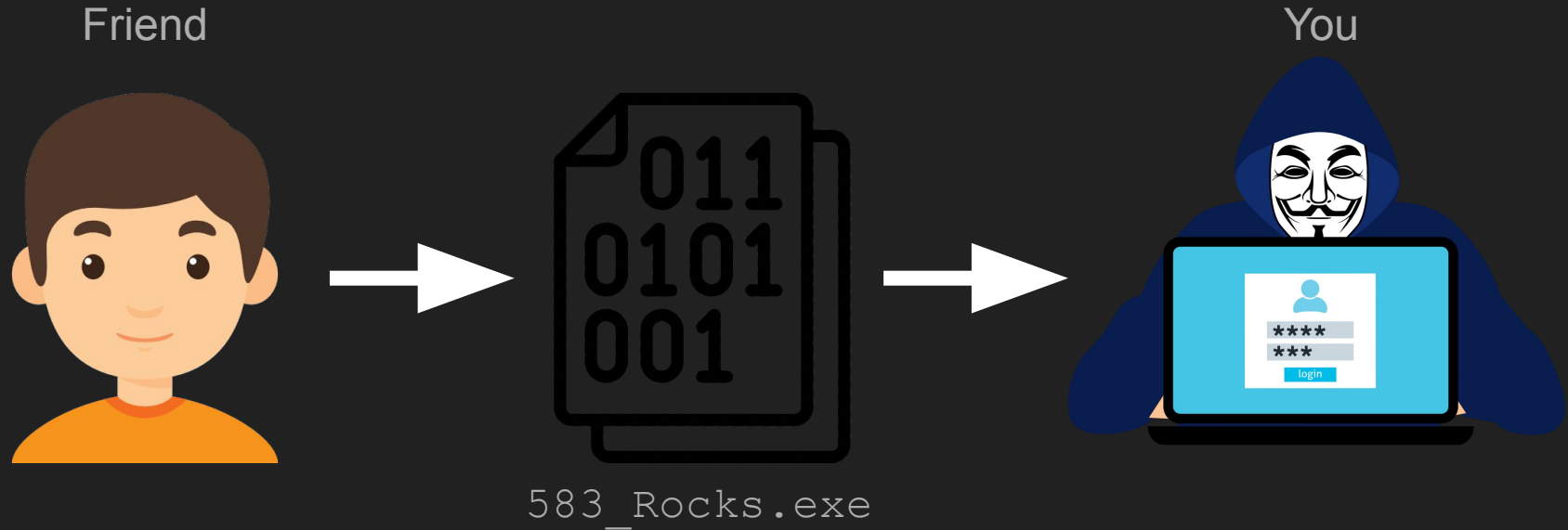
North Broward Preparatory School

Group 19

Aidan Delwiche, Ben Schwartz, John Kim, Nina Moyski, Sydney Zhong

Scenario

- You've intercepted a binary from a friend, and you want to hijack it



Step 1: Inspection

- Don't run it yet!

```
$ file 583_Rocks.exe
583_Rocks.exe: ELF 64-bit LSB pie executable, x86-64,
...
```

- Hmm... seems to be an x86_64 executable...
- Let's use a reverse engineering tool made by NSA!

Step 2: Decompile

```
Decompile: main - (583_Rocks.exe)
1
2 undefined8 main(int param_1,undefined8 *param_2)
3
4 {
5     if (param_1 < 2) {
6         printf("Usage: %s <your_name>\n",*param_2);
7     }
8     else {
9         safeFunction(param_2[1]);
10    }
11    return 0;
12 }
13
```

```
Decompile: safeFunction - (583_Rocks.exe)
1
2 void safeFunction(char *param_1)
3
4 {
5     char local_12 [10];
6
7     strcpy(local_12,param_1);
8     printf("Hey %s, 583 Rocks!\n",local_12);
9     return;
10 }
11
```

Step 3: Looks safe, let's run it! (Don't actually run an untrusted executable, ever)

```
$ ./583_Rocks.exe
Usage: ./583_Rocks.exe <your_name>

$ ./583_Rocks.exe Ben
Hey Ben, 583 Rocks!

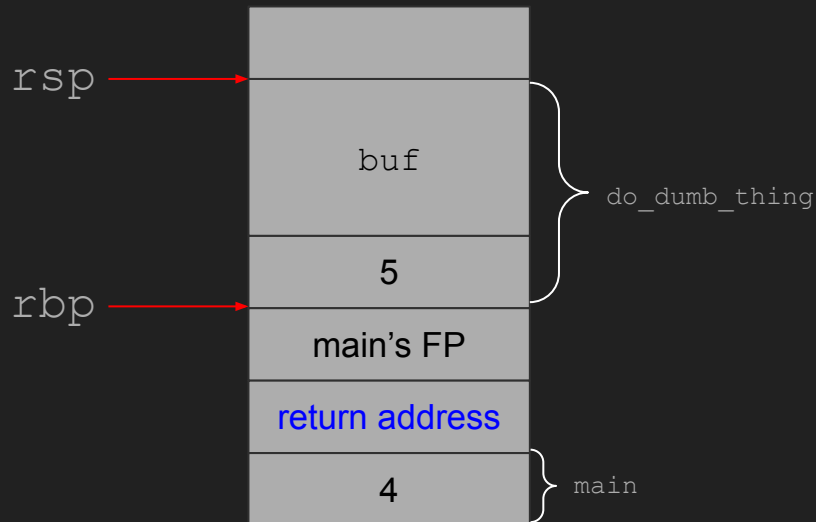
$ ./583_Rocks.exe BenHasALongNameYESAAAAAAA
Hey BenHasALongNameYESAAAAAAA, 583 Rocks!
Segmentation fault (core dumped)
```

Huh, segfault?

Refresh: x86_64 Calling Convention

```
void do_dumb_thing() {  
    int b = 5;  
    char buf[10];  
    printf("Hello, world!\n");  
}  
  
void main() {  
    int a = 4;  
    do_dumb_thing();  
    printf("Goodbye, world!\n");  
}
```

Diagram illustrating the x86_64 calling convention. The `rip` register points to the `printf` call in `do_dumb_thing()`. The `main` function calls `do_dumb_thing()`, and the `printf` call in `main` is also shown.

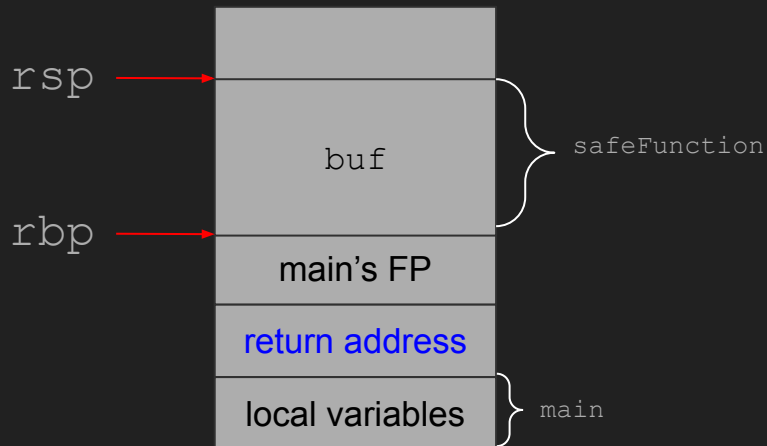


Revealing the source code

```
$ ./583_Rocks.exe BenHasALongNameYESAAAAAAA
```

```
rip → void safeFunction(char *str) {  
        char buffer[10];  
        strcpy(buffer, str);  
        printf("Hey %s, 583 Rocks!\n", buffer);  
    }  
  
    int main(int argc, char **argv) {  
        if (argc > 1) {  
            safeFunction(argv[1]);  
        } else {  
            printf("Usage: %s <your_name>\n", argv[0]);  
        }  
        → return 0;  
    }
```

583_Rocks.c

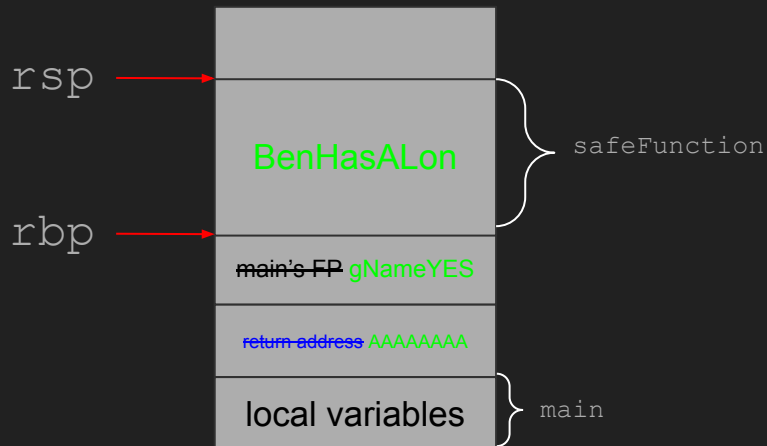


Revealing the source code

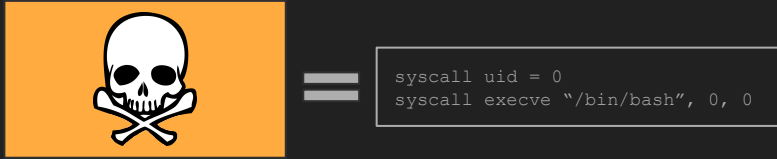
```
$ ./583_Rocks.exe BenHasALongNameYESAAAAAAA
```

```
rip → void safeFunction(char *str) {  
        char buffer[10];  
        strcpy(buffer, str);  
        printf("Hey %s, 583 Rocks!\n", buffer);  
    }  
  
    int main(int argc, char **argv) {  
        if (argc > 1) {  
            safeFunction(argv[1]);  
        } else {  
            printf("Usage: %s <your_name>\n", argv[0]);  
        }  
        → return 0;  
    }
```

583_Rocks.c



So why does this matter?



code injection



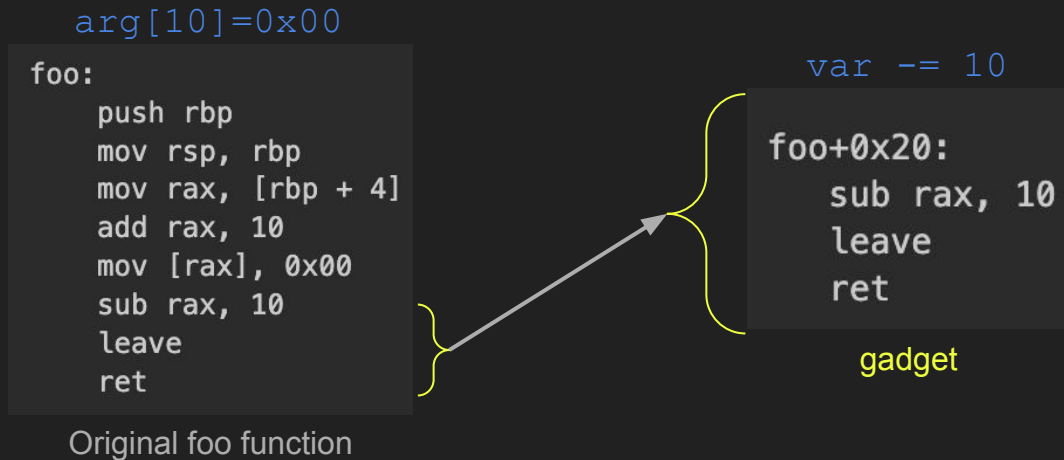
return-to-lib-c

Defenses

- Data Execution Prevention (DEP)
 - Prevents anything on the stack from being executed as code
 - Defends against **code injection**
- Address Space Layout Randomization (ASLR)
 - Randomizes locations of key data areas in a binary
 - Makes it difficult to predict target addresses
 - Defends against **code injection** and **return-to-lib-c**

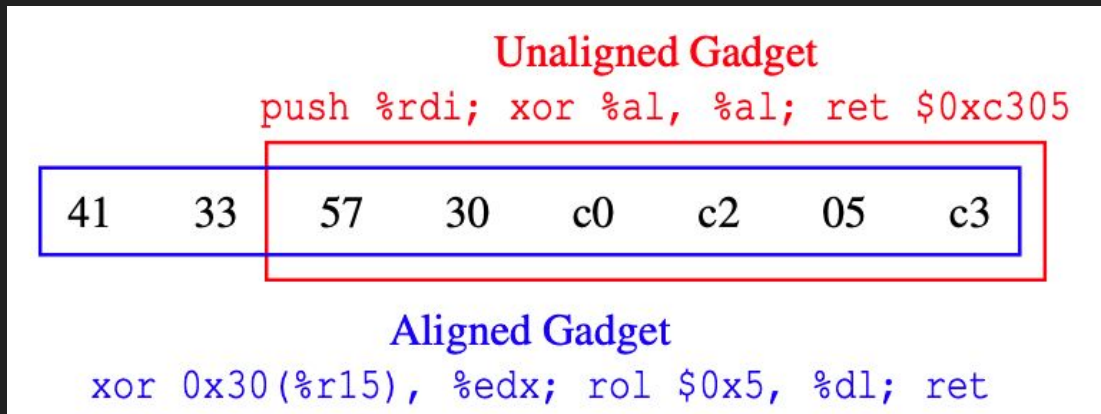
Return Oriented Programming

- What if DEP is enabled, and there are no functions that can open a shell?
 - Let's use instructions that are still there!
- **return-to-lib-c** without calling entire functions
 - All the instructions of a function that opens a root shell are still probably somewhere in memory, just not sequential
 - These individual pieces/snippets of instructions are called **gadgets**



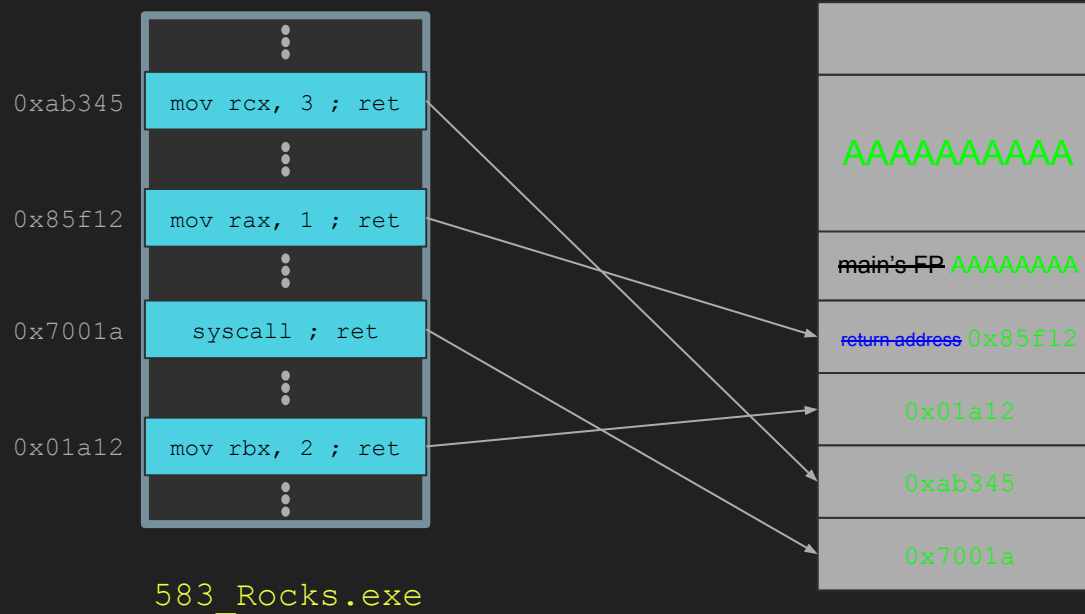
Gadgets

- Can come from anywhere in the binary
- Must end in a free-branch instruction (`ret`, `jmp %reg`)
 - This allows gadgets to be run sequentially, called ROP chains
- Don't even have to be instructions from the program's normal execution!



ROP Attack

- For simplicity, assume all you have to do to open a root shell is:
 - Perform a `syscall` with `rax == 1 && rbx == 2 && rcx == 3`



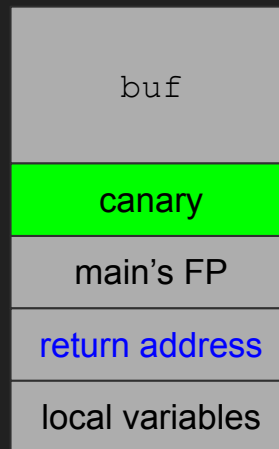
Defending Against ROP

- Stack Canary

- Place a small integer before the return address
- Detect if it is overwritten

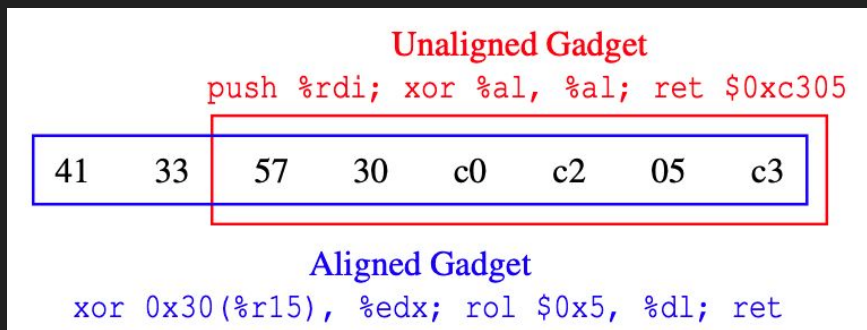
- G-Free

- Technique that attempts to remove all gadgets from a program's memory
 - Works by replacing all gadgets with semantically equivalent code that does not end in a free-branch instruction
- Aligned gadgets must be treated differently by G-Free, as their removal would change the program's semantics



on return:

if canary != expected:
goto stack_chk_fail
return



Protecting Aligned Gadgets

- Encrypt return address of function in stack every time function is entered, decrypt on exit
 - Encrypt with stack canary value
- If an attacker jumps into a function at an arbitrary position, the decryption routine processes the attacker's unencrypted return address and computes an invalid value

```
00000000000001000 <add>:
    ;; encrypt return address
1000: mov     %fs:0x28, %r11
1009: xor     %r11, (%rsp)
    ;; start of the function
100d: push    %rbp
100e: mov     %rsp, %rbp
1011: mov     %edi, -0x4(%rbp)
1014: mov     %esi, -0x8(%rbp)
1017: mov     -0x4(%rbp), %eax
101a: add     -0x8(%rbp), %eax
101d: pop     %rbp
    ;; decrypt return address
101e: mov     %fs:0x28, %r11
1027: xor     %r11, (%rsp)
    ;; return to caller
102b: ret
```

XOR return
addr with
canary
secret

XOR again
with canary
secret to
undo

SafeLLVM does this with
SafeReturnMachinePass

Done before emitting machine code for a function
(X86PassConfig::addPreEmitPass)

Removing Unaligned Gadgets

- Statically remove by substituting immediates with semantically equivalent instructions that **do not** contain any free branches

- `ret (0xc3), ret imm16 (0xc2), retf (0xcb), retf imm16 (0xca), iret (0xcf)`

```
mov $0xc3, %rax
```

(a) Instruction before the transformation. 0xc3 is the opcode for `ret`, and it is being moved into the `%rax` register.

```
mov $0x62, %r11
add $0x61, %r11
mov %r11, %rax
```

(b) Sequence of instructions after the transformation. 0xc3 is divided into 0x62 and 0x61.

SafeLLVM does this with
`ImmediateReencodingMachinePass`

Done before register allocation
(`X86PassConfig::addPreRegAlloc`
)

Removing Unaligned Gadgets

- Restore Alignment with NO-OPs
 - Prepend aligned free branch byte with `nop` sled

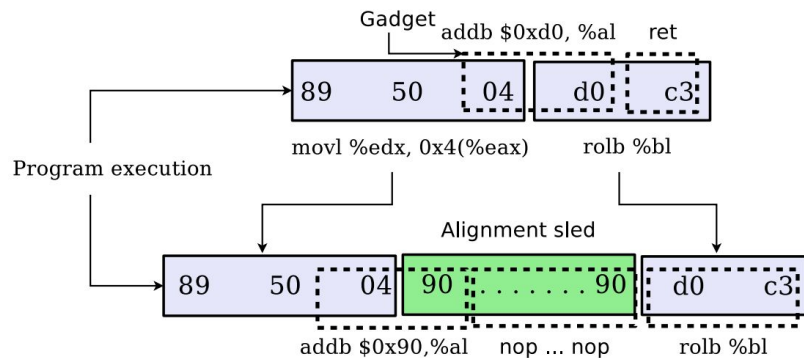


Figure 2: Application of an alignment sled to prevent executing an unaligned `ret` (0xc3) instruction

SafeLLVM does this with
`SafeReturnMachinePass`

Done before emitting machine code for a function
(`X86PassConfig::addPreEmitPass`)

Results

- Reduced ROP Gadgets

Toolchain	LLVM		SafeLLVM	
	Gadgets	ROP Chain	Gadgets	ROP Chain
zlib	1169	yes	194	no
cJSON	525	no	64	no
mimalloc	2014	yes	377	no
curl	1268	yes	166	no
SURF	343	no	105	no
ST	999	no	306	no
Doom	7735	yes	1528	no
LittleFS	414	no	60	no

- Compiled Binary Performance

Toolchain	LLVM		SafeLLVM	
	Tests	Time (ms)	Tests	Time (ms)
cJSON	19/19	40	19/19	40
mimalloc	3/3	4,706	3/3	1,395
LittleFS	817/817	6,420	817/817	6,505

Limitations

- Code that depends on the return address
 - May lead to potential crash
- Stack Canary Leaks
 - The technique utilizes the stack canary as a source of randomness
- Jump-Oriented Programming (What we are doing for final project!)

Commentary

- Most of the computer security vulnerabilities are memory issues.
- Effective while not over complicated.

Q&A