return-to-csu: A New Method to Bypass 64-bit Linux ASLR

Hector Marco, Ismael Ripoll
About us:

- 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
Motivation

- 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
1. Brief of the Linux ASLR

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 us per boot, per exec, etc.
1. Brief of the Linux ASLR

**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 side** 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
7ffe4a530000-7ffe4a740000 rw-p [stack]
7ffe4ad2000-7ffe4ad5000 r--p [vvar]
7ffe4ad5000-7ffe4ad7000 r-xp [vdso]
```

- Linux loads the executable and the dynamic loader/linker (**ld.so**)
- The **libc.so** library is later loaded.
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  /lib/x86_64-linux-gnu/libc-2.26.so
7f2ce47b2000-7f2ce47d9000  r-xp  /lib/x86_64-linux-gnu/ld-2.26.so
7f2ce49b8000-7f2ce49ba000  rw-p  /lib/x86_64-linux-gnu/ld-2.26.so
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
7ffe fa453000-7ffe fa474000  rw-p  [stack]
7ffe fa4d2000-7ffe fa4d5000  r--p  [vvar]
7ffe fa4d5000-7ffe fa4d7000  r-xp  [vdso]
```

- Libraries are loaded side by side there is no “more” randomization.
- There is not actual randomization from userland.
## 1. Brief of the Linux ASLR

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

- The ASLR in 64-bit systems is not only better but faster.
- It is not only a matter of entropy, the x64 ABI introduces a challenge.
Why Offset2lib attacks are “partially” possible?

- Offset2lib is a practical ASLR bypass for 64-bit systems
- It was a generic method valid for multiple attack scenarios
- No longer possible in modern Linux (fixed in 2015)
- Part of the attack method still valid to de-randomize the executable
- But exec-libc offset is not longer a constant value

We need to find an 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

```c
int main(int argc, const char *argv[]) {
    return 0;
}
```
2. The real battlefield: The Attached code

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

```bash
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
0000000000000520 t deregister_tm_clones
00000000000005b0 t __do_global_dtors_aux
0000000000020df8 t __do_global_dtors_aux_fini_array_entry
0000000000000684 T _fini
0000000000000684 t .fini
0000000000020df8 t .fini_array
00000000000005f0 t frame_dummy
0000000000020df0 t __frame_dummy_init_array_entry
00000000000004b8 T _init
00000000000004b8 t .init
0000000000020df0 t .init_array
0000000000020df8 t __init_array_end
0000000000020df0 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

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?

---

**empty.c**

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

---

```
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
```

```plaintext
0000000000000520 t deregister_tm_clones
00000000000005b0 t __do_global_dtors_aux
000000000020df8 t __do_global_dtors_aux_fini_array_entry
000000000000684 T _fini
000000000000684 t .fini
000000000020df8 t .fini_array
0000000000005f0 t frame_dummy
000000000020df0 t __frame_dummy_init_array_entry
000000000004b8 T .init
000000000004b8 t .init
000000000020df0 t .init_array
000000000020df8 t __init_array_end
000000000020df0 t __init_array_start
00000000000680 T __libc_csu_fini
00000000000610 T __libc_csu_init
000000000005fa T main
000000000004d0 t .plt
000000000004e0 t .plt.got
00000000000560 t register_tm_clones
000000000004f0 T .start
000000000004f0 t .text
```
2. The real battlefield: The Attached code

$ objdump -d empty

disassembly of section .init:

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

Disassembly of section .init:

00000000000004b8 <.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:

00000000000004d0 <.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:

00000000000004e0 <__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

Disassembly of section .text:
00000000000004f0 <_start>:
 4f0:  31 ed        xor %ebp,%ebp
 4f2:  49 89 d1     mov %rdx,%r9
 4f5:  5e           pop %rsi
 4f6:  48 89 e2     mov %rsp,%rdx
 4f9:  48 83 e4 f0  and %0xfffffffffffffff0,%rsp
 4fd:  50           push %rax
 4fe:  54           push %rsp
 4ff:  4c 8d 05 7a 01 00 00    lea 0x17a(%rip),%r8       # 680 <__libc_csu_fini>
 506:  48 8d 03 01 00 00 00    lea 0x103(%rip),%rcx      # 610 <__libc_csu_init>
 50d:  48 8d 3d e6 00 00 00 00    lea 0xe6(%rip),%rdi     # 5fa <main>
 514:  ff 15 c6 0a 20 00      callq *0x200ac6(%rip)      # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 51a:  c4 8d 05 9a 0a 20 00    lea 0x200a9a(%rip),%rax   # 610 <__lib_init>
 51e:  48 85 e5        mov %rsp,%rbp
 521:  74 19           je 550 <deregister_tm_clones+0x30>
 524:  48 8b 05 9a 0a 20 00    lea 0x200a9a(%rip),%rax   # 200f8 <__ITM_deregisterTMCloneTable>
 527:  5d           pop %rbp
 528:  48 8d 05 e1 0a 20 00    lea 0x200ae1(%rip),%rax     # 2010d <__TMC_END__>
 52f:  48 39 f8        cmp %rdi,%rax
 532:  48 89 e5        mov %rsp,%rbp
 535:  74 0d           je 550 <deregister_tm_clones+0x30>
 538:  5d           pop %rbp
 539:  c3           retq
 540:  00 00 00 00 00 00 00 00    lea 0x200000(%rip),%rbp  
 547:  66 66 0f 80 84 00 00 00    lea 0xc0(%rip),%rsi     # 780 <__libc_csu_fini>
 54d:  5d           pop %rbp
 550:  5d           pop %rbp
 551:  c3           retq
 552:  0f 1f 40 00    nop 0x0(%rax)
 555:  66 66 0f 80 84 00 00 00    lea 0xc0(%rip),%rsi     # 780 <__libc_csu_fini>
 55d:  00 00 00 00 00 00 00 00    lea 0x200000(%rip),%rbp  

2. The real battlefield: The Attached code

```
$ objdump -d empty
```

Disassembly of section .text:

```
00000000000004f0 <_start>:
  4f0:   31 ed xor %ebp,%ebp
  4f2:   49 89 d1 mov %rdx,%r9
  4f5:   5e pop %rsi
  4f6:   48 89 e2 mov %rsp,rdx
  4f9:   48 83 e4 f0 and $0xfffffffffffffff0,%rsp
  4fd:   50 push %rax
  4fe:   54 push %rsp
  4ff:   4c 8d 05 7a 01 00 00 lea 0x17a(%rip),%r8  # 680 <__libc_csu_fini>
  506:   48 8d 0d 03 01 00 00 lea 0x103(%rip),%rcx  # 610 <__libc_csu_init>
  50d:   48 8d 3d e6 00 00 00 lea 0xe6(%rip),%rdi  # 5fa <main>
  514: ff 15 c6 0a 20 00 callq *0x200ac6(%rip)  # 200fe0 <__libc_start_main@GLIBC_2.2.5>
  51a:   f4 hlt
  51b:   0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)
  520:   48 8d 3d a9 0a 20 00 lea 0x200aa9(%rip),%rdi  # 201010 <__TMC_END__>
  527:   55 push %rbp
  528:   48 8d 05 e1 0a 20 00 lea 0x200ae1(%rip),%rax  # 201010 <__TMC_END__>
  52f:   48 39 f8 cmp %rdi,%rax
  532:   48 89 e5 mov %rsp,rbp
  535:   74 19 je 550 <deregister_tm_clones+0x30>
  537:   48 8b 05 9a 0a 20 00 mov 0x200a9a(%rip),%rax  # 200fd8 <_ITM_deregisterTMCloneTable>
  53e:   48 85 c0 test %rax,%rax
  541:   74 0d je 550 <deregister_tm_clones+0x30>
  543:   5d pop %rbp
  544: ff e0 jmpq *%rax
  546:   66 2e 0f 1f 84 00 00 nopw 0x0(%rax,%rax,1)
  550:   5d pop %rbp
  551:   c3 retq
  552:   0f 1f 40 00 nopl 0x0(%rax)
  556:   66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
  55d:   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 <__TM_CEND__>
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 %rbx,%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 $0x1,0x200a1(%rip)   # 201010 <__TM_CEND__>
5df:  5d  pop %rbp
5e0:  c3   retq
5e1:  0f 1f 80 00 00 00 00 00  nopl 0x0(%rax)
5e8:  e3 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  imov 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:

00000000000000610 <_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 0x20007ce(%rip),%r12 # 200d60 <__frame_dummy_init_array_entry>
   622: 55      push %rbp
   623: 48 8d 2d ce 07 20 00 lea 0x20007ce(%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: a8 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+0x60>
   666: 48 83 c4 08 add $0x8,%rsp
   668: 4b 75 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:

Disassembly of section .fini:

Disassembly of section .fini:
2. The real battlefield: The Attached code

Application source code

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

Application compiled code
2. The real battlefield: source ≠ compiled

Application source code

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

Application compiled code

We have named it Attached code

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?
2. The real battlefield: Who is attaching it?

The minimum static linked code in dynamic linked applications

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>FILE PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>/home/blackHat2018/empty</td>
</tr>
<tr>
<td>deregister_tm_clones()</td>
<td></td>
</tr>
<tr>
<td>register_tm_clones()</td>
<td></td>
</tr>
<tr>
<td>__do_global_dtors_aux()</td>
<td>/usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o</td>
</tr>
<tr>
<td>frame_dummy()</td>
<td></td>
</tr>
<tr>
<td>__libc_csu_fini()</td>
<td>/usr/lib/x86_64-linux-gnu/libc_nonshared.a</td>
</tr>
<tr>
<td>__libc_csu_init()</td>
<td></td>
</tr>
<tr>
<td>_start()</td>
<td>/usr/lib/x86_64-linux-gnu/Scrt1.o</td>
</tr>
<tr>
<td>_init()</td>
<td>/usr/lib/x86_64-linux-gnu/crti.o</td>
</tr>
<tr>
<td>_fini()</td>
<td></td>
</tr>
</tbody>
</table>
## 2. The real battlefield: Who is attaching it?

The minimum static linked code in dynamic linked applications

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

Diagram showing VM space with stack, heap, exec, and mmap_base.
2. The real battlefield: What is it used for?

Simplified `exec()` syscall flow. The Linux Kernel:
- Loads the executable and dynamic loader
- Jumps to `main()` in the dynamic loader (`ld.so`)
#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");
}

$ gcc consdest.c -o consdest
./consdest
Before main()
main()
myfunctAtExit()
After main()
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
...  ...  ...
```
2. The real battlefield: How protected it is?

How protected is that attached code?

```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>\|>:"

---

**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 \( \mu \text{ROP} \) to control the execution flow: Controlling up to 3 arguments

4) Info leak with the \( \mu \text{ROP} \): 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
3. Return-to-csu: 64-bit ASLR bypass

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

**ropper result**

Found gadgets to fill rdi and rsi
But for arbitrary execution it still needs:
- 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**

- Found gadgets to fill rdi and rsi
- But for arbitrary execution it still needs:
  - 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**

   - ropper result
     - Found gadgets to fill rdi and rsi
     - But for arbitrary execution it still needs:
       - 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**

- **ropper result**
  - Found gadgets to fill rdi and rsi
  - But for arbitrary execution it still needs:
    - 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**

**ropper result**

- Found gadgets to fill rdi and rsi
- But for arbitrary execution it still needs:
  - write-what-where (params)
  - rdx control (third argument)
  - syscall/int 0x80 control

**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()`

```
0000000000000610 <__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()`

```asm
0000000000000610 <__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: 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()

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

Gadget 2:
- arguments + call
  - \texttt{edi} from \texttt{r13}
  - \texttt{rsi} from \texttt{r14}
  - \texttt{rdx} from \texttt{r15}
- To control the destination we need \texttt{rbx} and \texttt{r12}
2) Manual analysis of the “attached code” for fun and profit
• We found something interesting in \texttt{__libc_csu_init()}

\begin{verbatim}
0000000000000610 <__libc_csu_init>: ...
650: 4c 89 ea
653: 4c 89 f6
656: 44 89 ff
659: 41 ff 14 dc
65d: 48 83 c3 01
661: 48 39 dd
664: 75 ea
666: 48 83 c4 08
66a: 5b
66b: 5d
66c: 41 5c
66e: 41 5d
670: 41 5e
672: 41 5f
674: c3
\end{verbatim}

\textbf{Gadget 2: arguments + call}
\begin{itemize}
\item \texttt{edi} from \texttt{r13}
\item \texttt{rsi} from \texttt{r14}
\item \texttt{rdx} from \texttt{r15}
\end{itemize}

To control the destination we need \texttt{rbx} and \texttt{r12}

\textbf{Gadget 1: not bad, we control:}
\begin{itemize}
\item \texttt{rbx}, \texttt{rbp}, \texttt{r12}, \texttt{r13}, \texttt{r14}, \texttt{r15}
\end{itemize}

The interesting ones are:
\begin{itemize}
\item \texttt{rdi}: First argument
\item \texttt{rsi}: Second argument
\item \texttt{rdx}: Third argument
\end{itemize}
3. Return-to-csu: A controlled call

3) Universal µROP to control the execution flow from __libc_csu_init()
3. Return-to-csu: A controlled call

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

C code

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

funcPtr = addr;
(*funcPtr)(arg1, arg2, arg3);
```
3. Return-to-csu: A controlled call

3) Universal µROP 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

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

A controlled call where:
- addr = r12 + (rbx * 8)
- funcPtr = addr;
- arg1 = edi
- arg2 = rsi
- arg3 = rdx

We can jump where we want and control up to 3 arguments. edi only the 32 lowest bits
3. Return-to-csu: A controlled call

3) Universal µROP 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

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

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

A controlled call where:

- `addr = r12 + (rbx * 8)`
- `funcPtr = addr;`
- `arg1 = edi` (only the 32 lowest bits)
- `arg2 = rsi`
- `arg3 = rdx`

We can jump where we want and control up to 3 arguments. edi only the 32 lowest bits
3. Return-to-csu: A controlled call

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

Considering only the Attached Code we have:

- A µROP 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}`
  - Half `rdi`: we have edi 😊

- 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) Universal µROP to control the execution flow from \_\_libc\_csu\_init() 

Considering only the Attached Code we have:

- A µROP 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}
  - Half rdi: we have edi
- 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 µROP: Analyzing the PLTs/GOTs

- Let’s review again the “attached code”
4) Info leak with a \( \mu \)ROP : Analyzing the PLTs/GOTs

- Let’s review again the “attached code”

```bash
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
```

PLTs are good candidates:

- They are part of the
- We can call any \( @\text{PLT} \)
- Basic interaction of any program
  - read()/write() or send()/recv()
4) Info leak with a µROP : Reusing the connection

$ 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>
3. Return-to-csu: looking for a destination

4) Info leak with a \( \mu \text{ROP} \) : Reusing the connection

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>

Attached Code

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

1st arg: file descriptor (\( fd \) 😊)
2nd arg: buffer to write (\( *\text{buff} \) 😊)
3rd arg: Bytes to write (\( \text{count} \) 😊)

Basic sever calling read()/write() only

Attached Code

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

1st arg: file descriptor (\( fd \) 😊)
2nd arg: buffer to write (\( *\text{buff} \) 😊)
3rd arg: Bytes to write (\( \text{count} \) 😊)
4) Info leak with a µROP: Reusing the connection

```c
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 µROP: Reusing the connection

- 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 array containing lib addresses!

Attached Code

```c
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 µROP: Reusing the connection

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 array containing lib addresses!

**Attached Code**

3. Return-to-csu: looking for a destination

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

1st arg: file descriptor (`fd`)  
2nd arg: buffer to write (`*buff`)  
3rd arg: Bytes to write (`count`)  

Bytes to be written:
- Unsigned integer that we fully control
4) Info leak with a µROP: De-randomizing libraries
- Direct libc de-randomization

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

Assuming that `accept()` returned 4
- We just need to set `fd` to 4
4) Info leak with a µROP: De-randomizing libraries

- Direct libc de-randomization

Leaking write() address example
write@plt(4, \&GOT_TABLE[1], 8);

Assuming that accept() returned 4
- We just need to set fd to 4

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 \(\mu ROP\): De-randomizing libraries

- Direct \libc\ de-randomization

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

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

Leaking `write()` address example

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

```
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 µROP: De-randomizing libraries
   • Direct libc de-randomization

Assuming that `accept()` returned 4
   • We just need to set `fd` to 4

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

Bytes to be written/leaked:
   • Addresses in 64 bits = 8 bytes
4) Info leak with a μROP : De-randomizing libraries

- Direct libc de-randomization

Leaking `write()` address example

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

Server is sending us where `write()` is loaded

```
libc de-randomized!!!!
```

Attached Code

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

Assuming that `accept()` returned 4

- We just need to set `fd` to 4

```
fd = 4
```

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 attack in two stages:

**Stage 1:** Create a µROP-chain payload to leak a libc address
- Attackers will receive where the libc is in memory

**Stage 2:** Create a second payload using the input of the stage 1
- This ROP-chain uses all libc
3. Return-to-csu: Building the final attack

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`

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

- **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>`

Payload input

Libc Code

- **libc Gadget 1**
  - `pop %rdi`  
  - `pop %rsi`  
  - `retq`

- **libc Gadget 2**
  - `pop %rdx`  
  - `retq`

- **libc Gadget n**
  - `syscall`

Attached Code

- write() addr
- remote shell/arbitrary exec
3. Return-to-csu: When can we use return-to-csu

- **Forking Server**
  - PIE Executable
  - Non-PIE Executable
    - Brute force (offset2lib attack)

- **Inetd Server**
  - PIE Executable
  - Non-PIE Executable
    - ?

**Note:** Per boot-ASLRs == Forking Server

**return-to-csu attack**
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

When advanced ROP tools say “there are not enough gadgets” it is **not always** true. A manual inspection can reveal valid gadgets.
4. Making return-to-csu attack profitable

We have modified **ropper** to support **return-to-csu**

- New support for `dup2()` rop chain generation
- New support for `execve()` with `({"bash", "i", NULL}, NULL)` as 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 /home/BH/server --ret2csu "fd=0x4"
./Ropper.py --file /bin/ls /lib/libc.so.6 --console
...
```
To show a more realistic PoC:

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comment</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>App. relocatable</td>
<td>Yes</td>
<td>-fpie -pie</td>
</tr>
<tr>
<td>Lib. relocatable</td>
<td>Yes</td>
<td>-Fpic</td>
</tr>
<tr>
<td>ASLR config.</td>
<td>Enabled</td>
<td>randomize_va_space = 2</td>
</tr>
<tr>
<td>SSP</td>
<td>Enabled</td>
<td>-fstack-protector-all</td>
</tr>
<tr>
<td>Arch.</td>
<td>64 bits</td>
<td>x86_64 GNU/Linux</td>
</tr>
<tr>
<td>NX</td>
<td>Enabled</td>
<td>PAE or x64</td>
</tr>
<tr>
<td>RELRO</td>
<td>Full</td>
<td>-Wl,-z,relro,-z,now</td>
</tr>
<tr>
<td>FORTIFY</td>
<td>Yes</td>
<td>-D_FORTIFY_SOURCE=2</td>
</tr>
<tr>
<td>Optimization</td>
<td>Yes</td>
<td>-O2</td>
</tr>
</tbody>
</table>
5. DEMO: return-to-csu
Mitigation 1: Move some of the gadgets to `libc`

- The attack needs the 3 gadgets otherwise it 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 `libc` 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)`

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

```
mov %r13,%rdx
mov %r14,%rsi
mov %r15d,%edi
callq *(%rcx,%rax,8)
```

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

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

Without the control of the `callq` the Stage 1 ROP-chain will fail
Mitigation 3: Patching the current applications

- 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 desired solution is to move all code to \texttt{libc (ld.so)}

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

This solution is hardly applicable in real life

- Backward compatibility: Executables with the attached code will execute it twice (\texttt{libc} and executable). New \texttt{libc} call to avoid this.
- All sections can not be moved: \texttt{.plt .got}
  - Lazy binding requires use of the \texttt{.plt}
  - Eliminating \texttt{.plt} stubs require \texttt{.got} loads
  - Global variables from shared libraries (\texttt{R_386_GLOB_DAT}) need \texttt{.got}
6. Mitigations and solutions

How did we find it?
6. Conclusions and Black Hat Sound Bytes

• **return-to-csu** is a method to automate the construction of exploits to bypass the ASLR in 64-bit systems.

• 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 how to use a **µROP** 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