

# 整数溢出

## 测试平台

系统: CentOS release 6.10 (Final)、32 位  
内核版本: Linux 2.6.32-754.10.1.el6.i686 i686 i386 GNU/Linux  
gcc 版本: 4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)  
gdb 版本: GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)  
libc 版本: libc-2.12.so  
glibc 版本: 2.12-1.212

## 整数溢出漏洞原理介绍

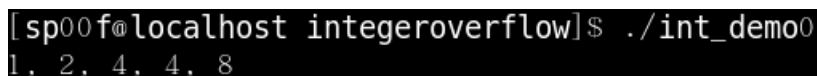
### 有符号数和无符号数

为什么会出现整数溢出漏洞呢? 这样从计算机怎么表示整数开始说起; 人类通常把数字分为正数和负数(0 除外), 如 1 是正数, -1 是负数, 对于人类而言这很好理解, 然而这对于计算机来说就显得不那么好理解了; 计算机是二进制的, 它只理解 0 和 1, 标识负数符号 -, 它不认识, 因此为了让计算机能够表达正数和负数, 制作计算机的科学家们想出了一个办法, 他们把数字分为有符号数和无符号数, 而区别于有符号数和无符号数的关键就在于数字的高位是 0 还是 1, 0 为正数, 1 为负数。举例说明:

如果我们想用计算机标识一个无符号数如 0xf (十进制数为 15), 它对应的二进制数是 0x1111; 但是现在我们让它变成有符号数, 那首先我们看这个数的高位(0x1111) 为 1, 因此它表示 -0x111 对应十进制数为 -7; 我们以 c 语言为例:

```
int main() {  
  
    printf("%d, %d, %d, %d, %d\n", sizeof(char), sizeof(short),  
        sizeof(int), sizeof(long), sizeof(long long));  
    return 0;  
}
```

运行结果如图 6-1



```
[sp00f@localhost integeroverflow]$ ./int_demo0  
1, 2, 4, 4, 8
```

图 6-1 32 位 linux 系统下, 各类型所占 byte 数

c 语言中定义了如下类型的整数 (32 位 Linux 操作系统, 64 位 long 是 8 byte): char (1 byte)、short (2 byte)、int (4 byte)、long (4 byte)、long long (8 byte)。1 byte = 8 bit, char 类型表示的无符号数的范围是 0x0 - 0x11111111, 即十进制 0-255, 它表示的有符号数范围是 (去掉高位符号位): -128 - +127, c 语言各种类型对应符号和无符号数范围如下表:

sp00f | 版权属于我个人所有, 你可以用于学习, 但不可以用于商业目的

类型	所占字节	表示范围
char	1 byte	0 - 255
unsigned char	1 byte	-128 - 127
short	2 byte	-32768 - 32767
unsigned short	2 byte	0 - 65535
int	4 byte	-2147483648 - -2147483647
unsigned int	4 byte	0 - 4294967295
long	4 byte	-2147483648 - -2147483647
unsigned long	4 byte	0 - 4294967295
long	8 byte	-9223372036854775808 - 9223372036854775807
unsigned long long	8 byte	0 - 18446744073709551615

表 6-1 c 语言定义整数类型所占字节和取值范围

## 溢出漏洞原理

当一个数据超过其取值范围时就会发生溢出，溢出一般多发生在类型转换上或者作为函数参数传递时忽略形参和实参的取值范围，整数类型的溢出就称做整数溢出。  
类型转换包含两种：一种不同类型转换，一种相同类型有符号和无符号数转换。

举例说明：

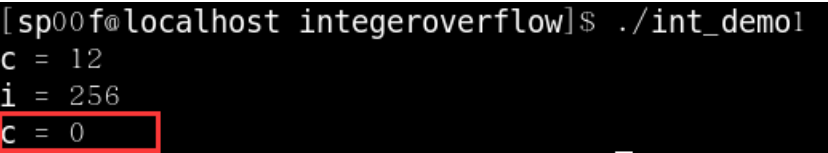
(1) 不同类型转换

```
char c = 12;
printf("c = %d\n", c);

int i = 256;
printf("i = %d\n", i);

c = i; // 把 signed int 型转换为 signed char 型
printf("c = %d\n", c);
```

运行结果如图 6-2，从运行结果可以看到变量 i 的高位被忽略了，其值从 256 变成了 0：



```
[sp00f@localhost integeroverflow]$ ./int_demo1
c = 12
i = 256
c = 0
```

图 6-2 把 signed int 型转换为 signed char 型整数溢出

(2) 同类型有符号和无符号数转换

```
int c = -1;
printf("c = %d\n", c);

unsigned int i = 256;
printf("i = %d\n", i);

i = c; // 把 signed int 型转换为 unsigned int 型
printf("i = %u\n", c);
```

运行结果如图 6-3，从运行结果可以看到本想让 i 等于 -1，结果它变成了无符号 int 型整数表示的最大值：

```
c = -1
i = 256
i = 4294967295
```

图 6-2 把 signed int 型转换为 unsigned int 型整数溢出

(3) 参数传递中不同类型转换引发的整数溢出

```
unsigned int sum(int a, int b) {
    unsigned int x = a;
    unsigned int y = b;

    return (unsigned int)(x + y);
}

int main() {
    int a = -1, b = -1;
    printf("a + b = %u\n", sum(a, b));
    return 0;
}
```

运行结果如图 6-3，从运行结果可以看到本想计算 $-1 + -1 = -2$ ，结果溢出为错误值：

```
[sp00f@localhost integeroverflow]$ ./int_demo3
a + b = 4294967294
```

图 6-3 参数传递过程中类型转换错误引起的整数溢出

## 整数溢出产生的逻辑漏洞

此类漏洞中的整数不参与堆栈中相关变量的操作，它们一般出现在条件判断中，该整数通过函数传递，并且该整数可控，如下面的程序：

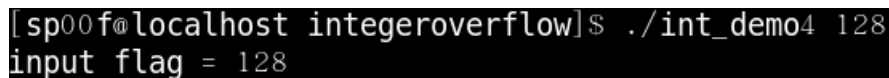
```
5 #define MAXFLAG 255
6 void exec() {
7     // do logic
8     printf("do a lot logic, but dangerous, we don't expect it to execute!\n");
9 }
10
11 int main(int argc, char** argv) {
12     int a = 0;
13     if (argc > 1) {
14         a = (int)atoi(argv[1]);
15     } else {
16         printf("Bad input!\n");
17         exit(-1);
18     }
19
20     if(a > 254) {
21         printf("Bad input!\n");
22         exit(-1);
23     }
```

```

24
25     printf("input flag = %d\n", a);
26
27     if( (char)(MAXFLAG - a) == 0) {
28         exec();
29     }
30     return 0;
31 }

```

从源码中我们可以看到，在程序的第 27 行处存在整数溢出；程序似乎假设获得的输入整数都为正数且输入的数字小于 255，但是它忽略了输入的整数可能是负数的情况，一点输入的整数为负数，第 27 行处就发生了溢出，最终高位 (高 8bit) 溢出被忽略，低位 (低 8bit) 永远为 0，导致条件成立，执行了本不该执行的逻辑。运行结果如下面两幅图所示：

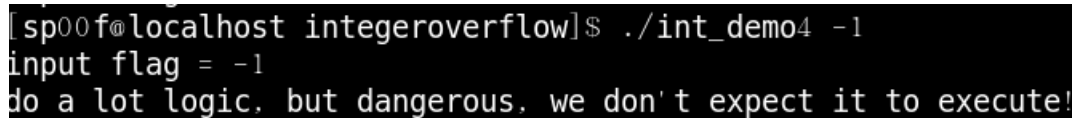


```

[sp00f@localhost integeroverflow]$ ./int_demo4 128
input flag = 128

```

图 6-4 程序执行正常的情况



```

[sp00f@localhost integeroverflow]$ ./int_demo4 -1
input flag = -1
do a lot logic. but dangerous. we don't expect it to execute!

```

图 6-5 程序执行产生溢出的情况

系统和应用软件中存在很多这样的例子，大多情况下程序能按照预期执行，但是这种问题一旦被攻击者发现，他们就会通过控制这些整数让程序按照程序本不该执行的逻辑执行程序。

## 整数溢出之栈溢出

此类溢出的整数作为一个控制栈变量的参数存在并且该整数可控；这种漏洞可以转换为栈溢出漏洞，通过栈溢出漏洞我们可以覆盖返回地址控制程序执行流程 (见第二章)，也可以通过溢出实现任意地址读或者信息泄露 (多见于数组下标)；下面我们来分配举例说明：

## 把整数溢出转换为栈溢出

```

4 #define MAX_BUF_LEN 256
5 int main(int argc, char** argv) {
6     unsigned char buflen = 0;
7     char* arg = NULL;
8
9     char buf[MAX_BUF_LEN];
10    if (argc > 1) {
11        arg = argv[1];
12        buflen = (unsigned char)strlen(arg);
13    } else {
14        printf("Bad input!\n");
15        exit(-1);
16    }

```

程序在第 9 行定义了一个 buf 数组，数组最大长度为 256，对应下标 0-255，该取值范围恰好是无符号 char 类型对应整数的取值范围；程序第 18 行对取得的字符串长度和数组最大长度做对比(对比前通过 strlen 获取字符串长度，strlen 函数定义是 size\_t strlen ( const char \* str );它返回一个 size\_t 类型的整数，32 位系统下它对应 unsigned int)；很明显程序的第 12 行处存在一个整数溢出漏洞(如果输入字符串长度在 0-255 之间时程序不会出现问题，但是当字符串长度超过 255 字节后它就会产生整数溢出，溢出位被忽略，这将导致第 18 行的判断永远成立)，溢出的字符串将会覆盖程序返回地址。程序运行结果如下面两幅图：

图 6-6 程序正常执行

图 6-7 程序产生溢出的情况，由于屏幕关系，截取的字符串 A 并不全

图 6-8 程序产生溢出的情况，程序崩溃在 0x41414141 (A 的 ascii 码是 0x41)

## 通过整数溢出来实现信息泄露

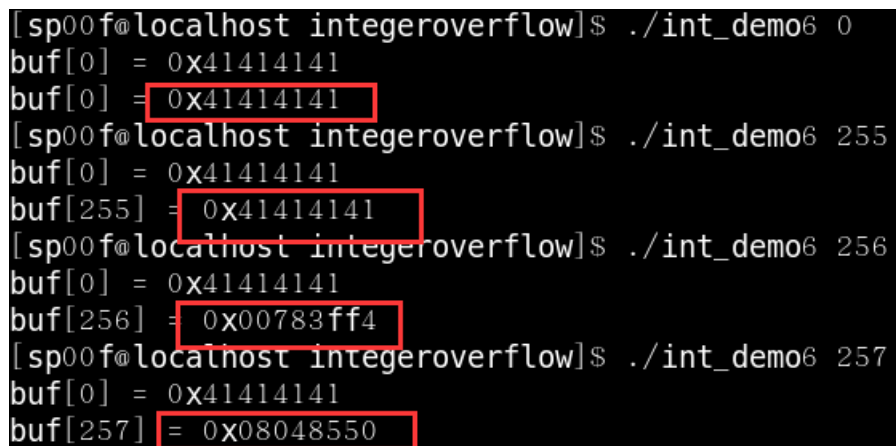
SP00F1 版权属于我个人所有，你可以用于学习，但不可以用于商业目的

```

4 #define MAX_BUF_LEN 256
5 int main(int argc, char** argv) {
6     unsigned char buflen = 0;
7
8     int* buf[MAX_BUF_LEN];
9     if (argc > 1) {
10         buflen = (unsigned char)atoi(argv[1]);
11     } else {
12         printf("Bad input!\n");
13         exit(-1);
14     }
15
16     if(buflen > MAX_BUF_LEN) {
17         printf("Bad input!\n");
18         exit(-1);
19     }
20
21     memset(buf, 'A', MAX_BUF_LEN * sizeof(int));
22     printf("buf[0] = 0x%.8x\n", buf[0]);
23     printf("buf[%d] = 0x%.8x\n", atoi(argv[1]), (buf[atoi(argv[1])]));
24
25     return 0;
26 }

```

上面的程序是在之前的程序的基础上修改的，程序大体相同，不过这个程序定义的数组是 int 型指针数组，大小同样为 256 (无符号 char 最大的取值范围)，程序在 10 行存在溢出，溢出的结果是第 16 行条件判断永远不成立，而在第 23 行我们就可以通过控制输入让程序泄露任意我们想要的地址的数据内容。运行结果如以下图所示：



```

[sp00f@localhost integeroverflow]$ ./int_demo6 0
buf[0] = 0x41414141
buf[0] = 0x41414141
[sp00f@localhost integeroverflow]$ ./int_demo6 255
buf[0] = 0x41414141
buf[255] = 0x41414141
[sp00f@localhost integeroverflow]$ ./int_demo6 256
buf[0] = 0x41414141
buf[256] = 0x00783ff4
[sp00f@localhost integeroverflow]$ ./int_demo6 257
buf[0] = 0x41414141
buf[257] = 0x08048550

```

图 6-9 程序运行结果，实现任意地址读

从图 6-9 我们可以看到第一次和第二次程序运行是在数组合理范围内读取数据的运行结果，第三次和第四次是超出程序数组的运行结果，不难发现我们实现了任意地址读，结合任意地址读我们可以轻易绕过安全机制的保护如 ASLR、PIE。

## 整数溢出之堆溢出

和整数溢出栈溢出不同，整数溢出堆溢出是整数成为了堆内存的读写变量而不是栈。同样它也有两种存在形式或者说两种转换技巧：一、转换为堆溢出漏洞；二、转换为任意地址读漏洞。

## 把整数溢出转换为堆溢出漏洞

```
4 #define MAX_BUF_LEN 256
5 int main(int argc, char** argv) {
6     unsigned char buflen = 0;
7     char* arg = NULL;
8
9     char* buf = (char*)malloc(MAX_BUF_LEN);
10    char* second = (char*) malloc(0x10);
11
12    if (argc > 1) {
13        arg = argv[1];
14        buflen = (unsigned char)strlen(arg);
15    } else {
16        printf("Bad input!\n");
17        exit(-1);
18    }
19
20    if(buflen > MAX_BUF_LEN || arg == NULL) {
21        printf("Bad input!\n");
22        exit(-1);
23    }
24
25    printf("input buf len = %d\n", buflen);
26    memset(buf, 0, MAX_BUF_LEN);
27    strcpy(second, "hello world");
28
29    printf("second = %s\n", second);
30    strncpy(buf, arg, strlen(arg));
31    printf("second = %s\n", second);
32
33    return 0;
34 }
```

该程序和前面整数溢出之栈溢出存在很多相似之处(为方便讲解，该程序是在之前程序的基础上进行修改的)，整数溢出产生的方式完全相同，程序在第 9 行通过 malloc 申请了一个 buf 数组，数组最大长度为 256，该取值恰好是无符号 char 类型对应整数的最大取值；程序第 10 行再次通过 malloc 申请了 16 字节大小的字符数组 second；程序第 14 行

sp00f | 版权属于我个人所有，你可以用于学习，但不可以用于商业目的





```

24     memset(second, 'B', 0x10);
25     printf("buf[0] = 0x%.8x\n", buf[0]);
26     printf("buf[%d] = 0x%.8x\n", atoi(argv[1]), (buf[atoi(argv[1])]));
27
28     return 0;
29 }

```

该程序和前面整数溢出之栈溢出存在很多相似之处(为方便讲解, 该程序是在之前程序的基础上进行修改的), 整数溢出产生的方式完全相同, 程序在第 8 行通过 malloc 分配了一个 int 型指针数组, 大小同样为 256(无符号 char 最大的取值范围), 在第 9 行申请了一个字符指针数组, 程序在 12 行存在整数溢出, 溢出的结果是第 18 行条件判断永远不成立, 而在第 26 行我们就可以通过控制输入让程序泄露任意我们想要的地址的数据内容。程序运行结果如下图:

```

[sp00f@localhost integeroverflow]$ ./int_demo8 1
buf[0] = 0x41414141
buf[1] = 0x41414141
[sp00f@localhost integeroverflow]$ ./int_demo8 256
buf[0] = 0x41414141
buf[256] = 0x00000000
[sp00f@localhost integeroverflow]$ ./int_demo8 257
buf[0] = 0x41414141
buf[257] = 0x00000019
[sp00f@localhost integeroverflow]$ ./int_demo8 258
buf[0] = 0x41414141
buf[258] = 0x42424242

```

图 6-12 整数溢出转换为堆溢出的程序溢出的运行结果, 实现任意地址读

图 6-12 是程序运行四次的结果, 第一次我们对数组下标设置了合理取值范围(无符号 char 类型整数的合理取值范围)输出正常; 从第二次开始我们对数组下标设置了超出合理取值范围的数, 我们可以看到第二次打印出了内存 second 对应的 chunk 的元数据 mchunk\_prev\_size, 第三次同样打印出了其元数据 mchunk\_size, 第四次打印出了 second 对应内存的内容 0x42424242 对应字符 BBBB。通过把整数溢出转换为任意地址读我们可以获取任何内存, 可以绕过系统安全机制, 如 ASLR(堆地址随机化), 这大大得降低了我们的攻击难度, 有关堆漏洞原理和细节详细请看第 5 章。

## 如何发现整数溢出漏洞

我们已经知道了整数溢出的原理, 并且也了解了整数溢出漏洞转换为其他类型漏洞的方式, 想必如果我们发现了整数溢出漏洞应该会加以利用实现漏洞攻击; 但目前还遗留一个问题, 我们该如何发现整数溢出漏洞呢?

- (1) 源代码审计, 如果我们有源码那最好不过了, 我们可以对源码进行白盒审计, 按照前面介绍的原理相信在发现整数溢出漏洞不难, 难在把审计自动化。
- (2) 很多情况下我们是没有源代码的, 在这种情况下我们需要通过逆向了解程序逻辑, 同时收集整数作为参数出现的所有位置, 收集哪些作为参数的整数是可控的, 一点一点进行摸索测试; 如果你完全了解了程序执行逻辑, 以及程序中依赖的各种数据结构, 你可以按照自己的想法编写 fuzzer, 通过 fuzzer 测试, 检测崩溃点和预期成功样本之外的点, 来实现半自动化挖掘。

## 不同类型的整数在汇编中有和不同

### 整数通过栈存储

```
11      char a = 0x10;
12      unsigned char b = 0x11;
13      short c = 0x100;
14      unsigned short d = 0x101;
15      int e = 0x1000;
16      unsigned int f = 0x1001;
17      long g = 0x10000;
18      unsigned long h = 0x10001;
19      long long i = 0x100000;
20      unsigned long long j = 0x100001;
```

在上面的 c 代码中我们分别定义了不同类型的整数，现在让我们来看看它们在汇编中的表现形式：

```
8048408: c6 44 24 5a 10      movb    $0x10,0x5a(%esp)
804840d: c6 44 24 5b 11      movb    $0x11,0x5b(%esp)
8048412: 66 c7 44 24 5c 00 01  movw    $0x100,0x5c(%esp)
8048419: 66 c7 44 24 5e 01 01  movw    $0x101,0x5e(%esp)
8048420: c7 44 24 60 00 10 00 00  movl    $0x1000,0x60(%esp)
8048428: c7 44 24 64 01 10 00 00  movl    $0x1001,0x64(%esp)
8048430: c7 44 24 68 00 00 01 00  movl    $0x10000,0x68(%esp)
8048438: c7 44 24 6c 01 00 01 00  movl    $0x10001,0x6c(%esp)
8048440: c7 44 24 70 00 00 10 00  movl    $0x100000,0x70(%esp)
8048448: c7 44 24 74 00 00 00 00  movl    $0x0,0x74(%esp)
8048450: c7 44 24 78 01 00 10 00  movl    $0x100001,0x78(%esp)
8048458: c7 44 24 7c 00 00 00 00  movl    $0x0,0x7c(%esp)
```

上面的汇编代码(32 位操作系统) 清晰地向我们展示了不同类型的整数在汇编中的表现形式，char 类型是通过 movb 指令定义的(单字)，short 类型是通过 movw 定义的(双字)，其他类型均采用 movl 定义的(四字)；但 long long 型占用两个四字的栈空间；

### 整数通过寄存器存储

上面通过栈存储不同类型整数的形式，如果用寄存器存储不同整数是什么样子呢？

```
movb    $0x10,%al
movb    $0x11,%dl
movw    $0x100,%ax
movw    $0x101,%dx
movl    $0x1000,%eax
movl    $0x1001,%edx
movl    $0x10000,%eax
```

```
movl    $0x10001,%edx
movl    $0x100000,%ebx
movl    $0x0,%esi
movl    $0x100001,%ebx
movl    $0x0,%esi
```

我们看到 char 类型的整数可以通过寄存器 al、bl 这样的 8 位寄存器来进行存储，short 类型的整数可以通过 ax、bx 这样的 16 位寄存器来进行存储，int、long 类型的整数可以通过 eax、ebx 这样的 32 位寄存器来进行存储，long long 类型的整数在 32 位处理器下需要两个寄存器来存储，一个寄存器表示该整数的低 32 位，另一个寄存器用于存储高 32 位(64 操作系统，同时处理器支持 64 位架构，它可以用类似 rax 这样的 64 位寄存器存储占 8 字节的整数)。

## 不同类型的整数转换

下面这段 c 代码是相同类型的整数在有符号和无符号之间的转换。

```
23      char a = 0x10;
24      unsigned char a1 = (unsigned char)a;
25      unsigned char b = 0x11;
26      char b1 = (char) b;
27      short c = 0x100;
28      unsigned short c1 = (unsigned short) c;
29      unsigned short d = 0x101;
30      short d1 = (short) d;
31      int e = 0x1000;
32      unsigned int e1 = (unsigned int) e;
33      unsigned int f = 0x1001;
34      int f1 = (int) f;
35      long g = 0x10000;
36      unsigned long g1 = (unsigned long) g;
37      unsigned long h = 0x10001;
38      long h1 = (long) h;
39      long long i = 0x100000;
40      unsigned long long i1 = (unsigned long long) i;
41      unsigned long long j = 0x100001;
42      long long j1 = (long long) j;
43
44      unsigned int k = (unsigned int) a;
45      k = (unsigned short)c;
46      int l = (int) a;
47      l = c;
48      a = (char) e;
49      a = (char) f;
50      c = (short) e;
51      c = (short) f;
```

```

52         i = (long long) e;
53         i = (long long) f;

```

对应的编译后的汇编代码如下：

8048438: c6 44 24 54 10	movb \$0x10,0x54(%esp)
804843d: 0f b6 44 24 54	movzbl 0x54(%esp),%eax
8048442: 88 44 24 55	mov %al,0x55(%esp)
8048446: c6 44 24 56 11	movb \$0x11,0x56(%esp)
804844b: 0f b6 44 24 56	movzbl 0x56(%esp),%eax
8048450: 88 44 24 57	mov %al,0x57(%esp)
8048454: 66 c7 44 24 58 00 01	movw \$0x100,0x58(%esp)
804845b: 0f b7 44 24 58	movzwl 0x58(%esp),%eax
8048460: 66 89 44 24 5a	mov %ax,0x5a(%esp)
8048465: 66 c7 44 24 5c 01 01	movw \$0x101,0x5c(%esp)
804846c: 0f b7 44 24 5c	movzwl 0x5c(%esp),%eax
8048471: 66 89 44 24 5e	mov %ax,0x5e(%esp)
8048476: c7 44 24 60 00 10 00 00	movl \$0x1000,0x60(%esp)
804847e: 8b 44 24 60	mov 0x60(%esp),%eax
8048482: 89 44 24 64	mov %eax,0x64(%esp)
8048486: c7 44 24 68 01 10 00 00	movl \$0x1001,0x68(%esp)
804848e: 8b 44 24 68	mov 0x68(%esp),%eax
8048492: 89 44 24 6c	mov %eax,0x6c(%esp)
8048496: c7 44 24 70 00 00 01 00	movl \$0x10000,0x70(%esp)
804849e: 8b 44 24 70	mov 0x70(%esp),%eax
80484a2: 89 44 24 74	mov %eax,0x74(%esp)
80484a6: c7 44 24 78 01 00 01 00	movl \$0x10001,0x78(%esp)
80484ae: 8b 44 24 78	mov 0x78(%esp),%eax
80484b2: 89 44 24 7c	mov %eax,0x7c(%esp)
80484b6: c7 84 24 80 00 00 00 00 10 00	movl \$0x100000,0x80(%esp)
80484c1: c7 84 24 84 00 00 00 00 00 00	movl \$0x0,0x84(%esp)
80484cc: 8b 84 24 80 00 00 00	mov 0x80(%esp),%eax
80484d3: 8b 94 24 84 00 00 00	mov 0x84(%esp),%edx
80484da: 89 84 24 88 00 00 00	mov %eax,0x88(%esp)
80484e1: 89 94 24 8c 00 00 00	mov %edx,0x8c(%esp)
80484e8: c7 84 24 90 00 00 00 01 00 10 00	movl \$0x100001,0x90(%esp)
80484f3: c7 84 24 94 00 00 00 00 00 00 00	movl \$0x0,0x94(%esp)
80484fe: 8b 84 24 90 00 00 00	mov 0x90(%esp),%eax
8048505: 8b 94 24 94 00 00 00	mov 0x94(%esp),%edx
804850c: 89 84 24 98 00 00 00	mov %eax,0x98(%esp)
8048513: 89 94 24 9c 00 00 00	mov %edx,0x9c(%esp)
.....	
8048523: 0f be 44 24 5c	movsbl 0x5c(%esp),%eax
8048528: 89 84 24 a8 00 00 00	mov %eax,0xa8(%esp)
804852f: 0f b7 44 24 60	movzwl 0x60(%esp),%eax
8048534: 0f b7 c0	movzwl %ax,%eax

```

8048537: 89 84 24 a8 00 00 00      mov    %eax,0xa8(%esp)
804853e: 0f be 44 24 5c           movsbl 0x5c(%esp),%eax
8048543: 89 84 24 ac 00 00 00      mov    %eax,0xac(%esp)
804854a: 0f bf 44 24 60           movswl 0x60(%esp),%eax
804854f: 89 84 24 ac 00 00 00      mov    %eax,0xac(%esp)
.....
804857c: 89 c2                   mov    %eax,%edx
804857e: c1 fa 1f               sar    $0x1f,%edx
8048581: 89 84 24 88 00 00 00      mov    %eax,0x88(%esp)

```

上面的整数是通过栈进行存储的，我们从汇编中不难发现这里面多出了几个 `movxxx` 指令和 `sar` 指令，`movxxx` 主要有两大类 `movzxxx`、`movsxxx`；`MOVS` 和 `MOVZ` 指令类都是将一个较小的数复制到一个较大的数据位置，高位用符号扩展 (`MOVS`) 或者零扩展 (`MOVZ`) 进行填充；如 `movzbl` 将做了 0 扩展的字节传送到双字，`movswl` 将进行了符号扩展的字传送到双字等；下面我们就代码中的部分片段说明一下，如下面代码：

```

char a = 0x10;
unsigned char a1 = (unsigned char)a;

```

上面两个代码对应下面几行汇编，相同类型整数从有符号转换为无符号它会首先进行 0 扩展，扩展后在取该寄存器低 8 位数据。

```

movb    $0x10,0x54(%esp)
movzbl  0x54(%esp),%eax
mov     %al,0x55(%esp)

```

相反如果相同类型整数从无符号转换为有符号类型直接从对应位数的寄存器获取数据，不需要转换，如下代码：

```

unsigned char b = 0x11;
char b1 = (char) b;

```

对应汇编如下：

```

movb    $0x11,0x56(%esp)
movzbl  0x56(%esp),%eax
mov     %al,0x57(%esp)

```

因此我们就明白了，如果同类型有符号数转换为无符号数首先需要经过 0 扩展，相同类型无符号数转换为有符号数不需要处理；那不同类型的整数是怎么转换的呢？

```

44      unsigned int k = (unsigned int) a;
45      k = (unsigned short)c;
46      int l = (int) a;

```

代码第 44 行是把一个有符号 `char` 型整数转换为无符号 `int` 型整数 (存放 `int` 的寄存器足

```

movsbl 0x5c(%esp),%eax
mov    %eax,0xa8(%esp)

```

够容纳 `char` 类型整数)，我们通过下面汇编代码可以看到该有符号数先进行了符号位 (`MOVS`) 扩展然后把对应寄存器的数据存储到对应位置 (因为这里最终数据占据 4 字节，数据对应 32 位，因此第二条指令把 `eax` 对应的内容全部放入栈中存储了，否则之取对应位数的寄存器的内容，如数据被转换成了 `signed short`，对应汇编就会变成 `mov %ax, 0xa8(%esp)`)；如果 `int` 转换为 `char` (目标寄存器不足以存储原寄存器内存) 寄存器的高位被忽略，低位保存；这里还有一处用到了 `sar` 指令 (算术右移指令，高位什么左边扩展什么)，这个指令用在了代码 `i = (long long) f;` 处，`long long` 型占用八个字节，而

高 32 位是通过对有符号 int 进行算术右移得到的。

出于篇幅的考虑我们就不一一列举各种情况了 (本书不是一本讲授汇编和计算机信息处理的书籍, 如果由对此感兴趣的同学可以参考由机械工业出版社出版翻译的深入理解计算机系统那本书), 但所谓工欲善其事必先利其器, 相信了解了这些底层细节对你进行漏洞挖掘会有非常大的帮助的。

## 以 int\_demo1.c 为例尝试手动挖掘

int\_demo1.c 源码如下:

```
char c = 12;

printf("c = %d\n", c);

int i = 256;

printf("i = %d\n", i);

c = i; // 把 signed int 型转换为 signed char 型

printf("c = %d\n", c);
```

有源码就一目了然了, 很明显 `c = i` 这行会溢出, 高位被忽略了; 我们来看看汇编:

```
80483cd: c6 44 24 1b 0c      movb    $0xc,0x1b(%esp)
80483d2: 0f be 54 24 1b      movsbl  0x1b(%esp),%edx
.....
80483e8: c7 44 24 1c 00 01 00 00      movl    $0x100,0x1c(%esp)
.....
8048405: 8b 44 24 1c          mov     0x1c(%esp),%eax
8048409: 88 44 24 1b          mov     %al,0x1b(%esp)
```

我从这段汇编中去除了调用 `printf` 的逻辑, 上面汇编主要是定义整数和类型转换; 第 80483cd 行用到了 `movb` 指令, 那很明显在栈 `esp + 0x1b` 的位置上存储了 `char` 型整数 (我们单单看这一行是搞不清楚它是 `signed char` 还是 `unsigned char` 的), 紧接着 80483d2 行调用了 `movsbl` 对栈上整数 `esb + 0x1b` 进行了扩展 (`MOVS`, 符号扩展), 那现在我们知道了它是个 `signed char` 类型整数; 第 80483e8 行用到了 `movl` 指令, 它定义了一个 `int` 或者 `long` 型整数 (我们暂且忽略它是有符号的还是无符号的), 接着第 8048405 行汇编把刚刚定义的整数存入寄存器 `eax`, 最后 8048409 行取 `eax` 寄存器的低 8 位寄存器 `al` 存储的值, 把它赋值给最开始定义的 `char` 型整数 (`esp + 0x1b` 的位置); 我们来通过寄存器视角看看最终栈上 `esp + 0x1b` 的位置存储了什么。

31	16	0
	ah1	al00000000

表 6-2 `eax` 寄存器

十六进制数 `0x100` 所占字节超过 8 个, 其中低八位为 `00000000` (8 个 0), 因此 `mov %al` 或者的值 0, 最终 `esp + 0x1b` 的位置上存储了 0; 本来我们的目的是打印 `0x100` (十进制数 256), 结果却变成了 0, 那明显这里存在一个整数溢出的它 bug, 至于是否是漏洞, 那要看它是否能够被转换为可利用的攻击。