

C语言的整型溢出问题

7 回复 25 查看



(<https://www.shiyanlou.com/user/8490>) 实验楼管理员 (<https://www.shiyanlou.com/vip>)

2小时前

技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>)

整型溢出有点老生常谈了，bla, bla, bla...但似乎没有引起多少人的重视。整型溢出会有可能导致缓冲区溢出，缓冲区溢出会导致各种黑客攻击，比如OpenSSL的heartbleed事件，就是一个buffer overread的事件。在这里写下这篇文章，希望大家都了解一下整型溢出，编译器的行为，以及如何防范，以写出更安全的代码。

分享到微博

全部回答



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

一、什么是整型溢出

C语言的整型问题相信大家并不陌生了。对于整型溢出，分为无符号整型溢出和有符号整型溢出。

对于unsigned整型溢出，C的规范是有定义的——“溢出后的数会以 $2^{(8*\text{sizeof}(\text{type}))}$ 作模运算”，也就是说，如果一个 unsigned char（1字符，8bits）溢出了，会把溢出的值与256求模。例如：

```
unsigned char x = 0xff;
printf("%d\n", ++x);
```

上面的代码会输出：0（因为 $0xff + 1$ 是256，与 2^8 求模后就是0）

对于signed整型的溢出，C的规范定义是“undefined behavior”，也就是说，编译器爱怎么实现就怎么实现。对于大多数编译器来说，算得啥就是啥。比如：

```
signed char x = 0x7f; //注：0xff就是-1了，因为最高位是1也就是负数了
printf("%d\n", ++x);
```

上面的代码会输出：-128，因为 $0x7f + 0x01$ 得到 $0x80$ ，也就是二进制的1000 0000，符号位为1，负数，后面为全0，就是负的最小值，即-128。

另外，千万别以为signed整型溢出就是负数，这个是不定的。比如：

```
signed char x = 0x7f;
signed char y = 0x05;
signed char r = x * y;
printf("%d\n", r);
```

上面的代码会输出：123

相信对于这些大家不会陌生了。

二、整型溢出的危害

下面说一下，整型溢出的危害。

示例一：整型溢出导致死循环

```

... ..
... ..
short len = 0;
... ..
while(len< MAX_LEN) {
    len += readFromInput(fd, buf);
    buf += len;
}

```

上面这段代码可能是很多程序员都喜欢写的代码（我在很多代码里看到过多次），其中的 `MAX_LEN` 可能会是个比较大的整型，比如32767，我们知道short是16bits，取值范围是-32768 到 32767 之间。但是，上面的while循环代码有可能会造成整型溢出，而len又是个有符号的整型，所以可能会成负数，导致不断地死循环。

示例二：整形转型时的溢出

```

int copy_something(char *buf, int len)
{
    #define MAX_LEN 256
    char mybuf[MAX_LEN];
    ... ..
    ... ..

    if(len > MAX_LEN){ // <---- [1]
        return -1;
    }

    return memcpy(mybuf, buf, len);
}

```

上面这个例子中，还是[1]处的if语句，看上去没有问题，但是len是个signed int，而memcpy则需一个 `size_t` 的len，也就是一个unsigned 类型。于是，len会被提升为unsigned，此时，如果我们给len传一个负数，会通过了if的检查，但在memcpy里会被提升为一个正数，于是我们的mybuf就是overflow了。这个会导致mybuf缓冲区后面的数据被重写。

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) 💖 (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

示例三：分配内存

关于整数溢出导致堆溢出的很典型的例子是，OpenSSH Challenge-Response SKEY/BSD_AUTH 远程缓冲区溢出漏洞。下面这段有问题的代码摘自OpenSSH的代码中的 `auth2-chall.c` 中的 `input_userauth_info_response()` 函数：

```

nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}

```

上面这个代码中，nresp是 `size_t` 类型（`size_t` 一般就是 `unsigned int/long int`），这个示例是一个解数据包的示例，一般来说，数据包中都会有一个len，然后后面是data。如果我们精心准备一个len，比如：1073741825（在32位系统上，指针占4个字节，unsigned int的最大值是 `0xffffffff`，我们只要提供 `0xffffffff/4` 的值——`0x40000000`，这里我们设置了 `0x40000000 + 1`），nresp就会读到这个值，然后 `nresp*sizeof(char*)` 就成了 `1073741825 * 4`，于是溢出，结果成为了 `0x100000004`，然后求模，得到4。于是，`malloc(4)`，于是后面的for循环1073741825 次，就可以干坏事了（经过 `0x40000001` 的循环,用户的数据早已覆盖了xmalloc原先分配的4字节的空间以及后面的数据，包括程序代码，函数指针，于是就可以改写程序逻辑。关于更多的东西，你可以看一下这篇文章《Survey of Protections from Buffer-Overflow Attacks (<http://engj.org/index.php/ej/article/view/112/167>)》）。

示例四：缓冲区溢出导致安全问题

```
int func(char *buf1, unsigned int len1,
        char *buf2, unsigned int len2 )
{
    char mybuf[256];

    if((len1 + len2) > 256){    //<--- [1]
        return -1;
    }

    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

上面这个例子本来是想把buf1和buf2的内容copy到mybuf里，其中怕 len1 + len2 超过256 还做了判断，但是，如果 len1+len2 溢出了，根据unsigned的特性，其会与 2^{32} 求模，所以，基本上来说，上面代码中的[1]处有可能为假的。（注：通常来说，在这种情况下，如果你开启-O代码优化选项，那个if语句块就全部被和谐掉了——被编译器给删除了）比如，你可以测试一下 len1=0x104, len2 = 0xffffffffc 的情况。

示例五：size_t 的溢出

```
for (int i= strlen(s)-1; i>=0; i--) { ... }
```

```
for (int i=v.size()-1; i>=0; i--) { ... }
```

上面这两个示例是我们经常用的从尾部遍历一个数组的for循环。第一个是字符串，第二个是C++中的vector容器。strlen() 和 vector::size() 返回的都是 size_t，size_t 在32位系统下就是一个unsigned int。你想想，如果 strlen(s) 和 v.size() 都是0呢？这个循环会成为个什么情况？于是 strlen(s) - 1 和 v.size() - 1 都不会成为 -1，而是成为了 (unsigned int) (-1)，一个正的最大数。导致你的程序越界访问。

这样的例子有很多很多，这些整型溢出的问题如果在关键的地方，尤其是在搭配有用户输入的地方，如果被黑客利用了，就会导致很严重的安全问题。

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

三、关于编译器的行为

在谈一下如何正确的检查整型溢出之前，我们还要来学习一下编译器的一些东西。请别怪我罗嗦。

1、编译器优化

如何检查整型溢出或是整型变量是否合法有时候是一件很麻烦的事情，就像上面的第四个例子一样，编译的优化参数 -O/-O2/-O3 基本上会假设你的程序不会有整形溢出。会把你的代码中检查溢出的代码给优化掉。

关于编译器的优化，在这里再举个例子，假设我们有下面的代码（又是一个相当相当常见的代码）：

```
int len;
char* data;

if (data + len < data){
    printf("invalid len\n");
    exit(-1);
}
```

上面这段代码中，len 和 data 配套使用，我们害怕len的值是非法的，或是len溢出了，于是我们写下了if语句来检查。这段代码在-O的参数下正常。但是在-O2的编译选项下，整个if语句块被优化掉了。

你可以写个小程序，在gcc下编译（我的版本是4.4.7，记得加上-O2和-g参数），然后用gdb调试时，用disass /m命信输出汇编，你会看到下面的结果（你可以看到整个if语句块没有任何的汇编代码——直接被编译器和谐掉了）：

```
7      int len = 10;
8      char* data = (char *)malloc(len);
0x00000000004004d4 <+4>:      mov     $0xa,%edi
0x00000000004004d9 <+9>:      callq  0x4003b8 <malloc@plt>

9
10     if (data + len < data){
11         printf("invalid len\n");
12         exit(-1);
13     }
14
15 }
```

```
0x00000000004004de <+14>:      add     $0x8,%rsp
0x00000000004004e2 <+18>:      retq
```

对此，你需要把上面 char* 转型成 uintptr_t 或是 size_t，说白了也就是把 char* 转成unsigned的数据结构，if语句块就无法被优化了。如下所示：

```
if ((uintptr_t)data + len < (uintptr_t)data){
    ...
}
```

关于这个事，你可以看一下C99的规范说明《ISO/IEC 9899:1999 C specification (<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>)》第 §6.5.6 页，第8点，我截个图如下：（这段话的意思是定义了指针+/-一个整型的行为，如果越界了，则行为是undefined）

8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression *P* points to the *i*-th element of an array object, the expressions *(P) + N* (equivalently, *N + (P)*) and *(P) - N* (where *N* has the value *x*) point to, respectively, the *i+N*-th and *i-N*-th elements of the array object, provided they exist. Moreover, if the expression *P* points to the last element of an array object, the expression *(P) + 1* points one past the last element of the array object, and if the expression *Q* points one past the last element of an array object, the expression *(Q) - 1* points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.

注意上面标红线的地方，说如果指针指在数组范围内没事，如果越界了就是undefined，也就是说这事交给编译器实现了，编译器想咋干咋干，那怕你想把其优化掉也可以。在这里要重点说一下，**C语言中的一个大恶魔——Undefined! 这里都是“野兽出没”的地方，你一定要小心小心再小心。**

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) 💎 (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

2、花絮：编译器的彩蛋

上面说了所谓的undefined行为就全权交给编译器实现，gcc在1.17版本下对于undefined的行为还玩了个彩蛋（参看Wikipedia (http://en.wikipedia.org/wiki/Undefined_behavior#Compiler_easter_eggs））。

下面gcc 1.17版本下的遭遇undefined行为时，gcc在unix发行版下玩的彩蛋的源代码。我们可以看到，它会去尝试去执行一些游戏NetHack (<http://en.wikipedia.org/wiki/NetHack>), Rogue ([http://en.wikipedia.org/wiki/Rogue_\(computer_game\)](http://en.wikipedia.org/wiki/Rogue_(computer_game))) 或是 Emacs的 Towers of Hanoi (http://en.wikipedia.org/wiki/Tower_of_Hanoi#Applications), 如果找不到，就输出一条NB的报错。

```
execl("/usr/games/hack", "#pragma", 0); // try to run the game NetHack
execl("/usr/games/rogue", "#pragma", 0); // try to run the game Rogue
// try to run the Tower's of Hanoi simulation in Emacs.
execl("/usr/new/emacs", "-f", "hanoi", "9", "-kill", 0);
execl("/usr/local/emacs", "-f", "hanoi", "9", "-kill", 0); // same as above
fatal("You are in a maze of twisty compiler features, all different");
```

四、正确检测整型溢出

在看过编译器的这些行为后，你应该会明白——“在整型溢出之前，一定要做检查，不然，就太晚了”。

我们来看一段代码：

```
void foo(int m, int n)
{
    size_t s = m + n;
    .....
}
```

上面这段代码有两个风险：**1) 有符号转无符号**，**2) 整型溢出**。这两个情况在前面的那些示例中你都应该看到了。**所以，你千万不要把任何检查的代码写在 `s = m + n` 这条语句后面，不然就太晚了**。undefined行为就会出现——用句纯正的英文表达就是——“Dragon is here”——你什么也控制不住了。（注意：有些初学者也许会以为 `size_t` 是无符号的，而根据优先级 `m` 和 `n` 会被提升到 `unsigned int`。其实不是这样的，`m` 和 `n` 还是 `signed int`，`m + n` 的结果也是 `signed int`，然后再把这个结果转成 `unsigned int` 赋值给 `s`）

比如，下面的代码是错的：

```
void foo(int m, int n)
{
    size_t s = m + n;
    if ( m>0 && n>0 && (SIZE_MAX - m < n) ){
        //error handling...
    }
}
```

上面的代码中，大家要注意 **`(SIZE_MAX - m < n)`** 这个判断，为什么不用 `m + n > SIZE_MAX` 呢？因为，如果 `m + n` 溢出后，就被截断了，所以表达式恒真，也就检测不出来了。另外，这个表达式中，`m`和`n`分别会被提升为`unsigned`。但是上面的代码是错的，因为：

- 1) 检查的太晚了，`if`之前编译器的undefined行为就已经出来了（你不知道什么会发生）。
- 2) 就像前面说的一样，`(SIZE_MAX - m < n)` 可能会被编译器优化掉。
- 3) 另外，`SIZE_MAX` 是 `size_t` 的最大值，`size_t` 在64位系统是64位的，严谨点应该用 `INT_MAX` 或是 `UINT_MAX`

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>) 💎 (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

所以，正确的代码应该是下面这样：

```
void foo(int m, int n)
{
    size_t s = 0;
    if ( m>0 && n>0 && ( UINT_MAX - m < n ) ){
        //error handling...
        return;
    }
    s = (size_t)m + (size_t)n;
}
```

在《苹果安全编码规范

(<https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf>)》(PDF) 中，第28页的代码中：

```
if ( n > 0 && m > 0 && SIZE_MAX/n >= m ) {
    size_t bytes = n * m;
    ... /* allocate "bytes" space */
}
```

如果`n`和`m`都是 `signed int`，那么这段代码是错的。正确的应该像上面的那个例子一样，至少要在 `n*m` 时要把 `n` 和 `m` 给 `cast` 成 `size_t`。因为，`n*m`可能已经溢出了，已经undefined了，undefined的代码转成 `size_t` 已经没什么意义了。（如果`m`和`n`是`unsigned int`，也会溢出），上面的代码仅在`m`和`n`是 `size_t` 的时候才有效。

不管怎么说，《苹果安全编码规范

(<https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf>)》绝对值得你去读一读。

1、二分取中搜索算法中的溢出

我们再来看一个二分取中搜索算法（binary search），大多数人都会写成下面这个样子：

```
int binary_search(int a[], int len, int key)
{
    int low = 0;
    int high = len - 1;

    while ( low<=high ) {
        int mid = (low + high)/2;
        if (a[mid] == key) {
            return mid;
        }
        if (key < a[mid]) {
            high = mid - 1;
        }else{
            low = mid + 1;
        }
    }
    return -1;
}
```


上面这个代码中，你可能会这样的想法：

1) 我们应该用 `size_t` 来做`len`, `low`, `high`, `mid`这些变量的类型。没错，应该是这样的。但是如果这样，你要小心第四行 `int high = len - 1;` 如果`len`为0，那么就“high大发了”。2) 无论你用不用 `size_t`。我们在计算 `mid = (low+high)/2;` 的时候，`(low + high)` 都可以溢出。正确的写法应该是：

```
int mid = low + (high - low)/2;
```

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

2、上溢出和下溢出的检查

前面的代码只判断了正数的上溢出`overflow`，没有判断负数的下溢出`underflow`。让我们看看怎么判断：对于加法，还好。

```
#include <limits.h>

void f(signed int si_a, signed int si_b) {
    signed int sum;
    if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
        ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
        /* Handle error */
        return;
    }
    sum = si_a + si_b;
}
```

对于乘法，就会很复杂（下面的代码太夸张了）：

```

void func(signed int si_a, signed int si_b)
{
    signed int result;
    if (si_a > 0) { /* si_a is positive */
        if (si_b > 0) { /* si_a and si_b are positive */
            if (si_a > (INT_MAX / si_b)) {
                /* Handle error */
            }
        } else { /* si_a positive, si_b nonpositive */
            if (si_b < (INT_MIN / si_a)) {
                /* Handle error */
            }
        } /* si_a positive, si_b nonpositive */
    } else { /* si_a is nonpositive */
        if (si_b > 0) { /* si_a is nonpositive, si_b is positive */
            if (si_a < (INT_MIN / si_b)) {
                /* Handle error */
            }
        } else { /* si_a and si_b are nonpositive */
            if ( (si_a != 0) && (si_b < (INT_MAX / si_a)) ) {
                /* Handle error */
            }
        } /* End if si_a and si_b are nonpositive */
    } /* End if si_a is nonpositive */


    result = si_a * si_b;
}

```

更多的防止在操作中整型溢出的安全代码可以参看《INT32-C. Ensure that operations on signed integers do not result in overflow (<https://www.securecoding.cert.org/confluence/display/seccode/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>)》

2小时前



实验楼管理员 (<https://www.shiyanlou.com/user/8490>)  (<https://www.shiyanlou.com/vip>)

(<https://www.shiyanlou.com/user/8490>)

五、其它

对于C++来说，你应该使用STL中的 `numeric_limits::max()` 来检查溢出。

另外，微软的SafeInt类是一个可以帮你远理上面这些很tricky的类，下载地址：<http://safeint.codeplex.com/> (<http://safeint.codeplex.com/>)

对于Java 来说，一种是用JDK 1.7中Math库下的safe打头的函数，如 `safeAdd()` 和 `safeMultiply()`，另一种用更大尺寸的数据类型，最大可以到BigInteger。

可见，写一个安全的代码并不容易，尤其对于C/C++来说。对于黑客来说，他们只需要搜一下开源软件中代码有 `memcpy/strcpy` 之类的地方，然后看一看其周边的代码，是否可以通过用户的输入来影响，如果有的话，你就惨了。

参考：

- Basic Integer Overflow (<http://phrack.org/issues/60/10.html>)
- OWASP: Integer overflow (https://www.owasp.org/index.php/Integer_overflow)
- C compilers may silently discard some wraparound checks (<https://www.kb.cert.org/vuls/id/162289>)
- Apple Secure Coding Guide (<https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf>)
- Wikipedia: Undefined Behavior (http://en.wikipedia.org/wiki/Undefined_behavior)
- INT32-C. Ensure that operations on signed integers do not result in overflow (<https://www.securecoding.cert.org/confluence/display/seccode/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>)

最后，不好意思，这篇文章可能罗嗦了一些，大家见谅。

(全文完)

文章来源：酷壳 – CoolShell.cn

文章地址：<http://coolshell.cn/articles/11466.html> (<http://coolshell.cn/articles/11466.html>)

作者：陈皓

2小时前

登录后才能回答问题哟~

我要提问

标签

Linux (<https://www.shiyanlou.com/questions/?tag=Linux>) Python (<https://www.shiyanlou.com/questions/?tag=Python>)

C/C++ (<https://www.shiyanlou.com/questions/?tag=C/C++>) 实验环境 (<https://www.shiyanlou.com/questions/?tag=实验环境>)

技术分享 (<https://www.shiyanlou.com/questions/?tag=技术分享>) 功能建议 (<https://www.shiyanlou.com/questions/?tag=功能建议>)

课程需求 (<https://www.shiyanlou.com/questions/?tag=课程需求>) Java (<https://www.shiyanlou.com/questions/?tag=Java>)

其他 (<https://www.shiyanlou.com/questions/?tag=其他>) Hadoop (<https://www.shiyanlou.com/questions/?tag=Hadoop>)

NodeJS (<https://www.shiyanlou.com/questions/?tag=NodeJS>) SQL (<https://www.shiyanlou.com/questions/?tag=SQL>)

Web (<https://www.shiyanlou.com/questions/?tag=Web>) PHP (<https://www.shiyanlou.com/questions/?tag=PHP>)

常见问题 (<https://www.shiyanlou.com/questions/?tag=常见问题>) Shell (<https://www.shiyanlou.com/questions/?tag=Shell>)

Git (<https://www.shiyanlou.com/questions/?tag=Git>) HTML (<https://www.shiyanlou.com/questions/?tag=HTML>)

网络 (<https://www.shiyanlou.com/questions/?tag=网络>) 信息安全 (<https://www.shiyanlou.com/questions/?tag=信息安全>)

HTML5 (<https://www.shiyanlou.com/questions/?tag=HTML5>) NoSQL (<https://www.shiyanlou.com/questions/?tag=NoSQL>)

GO (<https://www.shiyanlou.com/questions/?tag=GO>) Android (<https://www.shiyanlou.com/questions/?tag=Android>)

训练营 (<https://www.shiyanlou.com/questions/?tag=训练营>) Ruby (<https://www.shiyanlou.com/questions/?tag=Ruby>)

Perl (<https://www.shiyanlou.com/questions/?tag=Perl>)

相关问题

5个你可能会忽略的有用命令行工具 (<https://www.shiyanlou.com/questions/3554>)

一些 Linux 桌面小技巧 (<https://www.shiyanlou.com/questions/3526>)

13款实用的JS插件&前端资源 (<https://www.shiyanlou.com/questions/3524>)

一些不起眼但非常有用的 Vim 命令 (<https://www.shiyanlou.com/questions/3514>)

Scala入门笔记 (<https://www.shiyanlou.com/questions/3510>)

动手做实验，轻松学IT。

实验楼-通过动手实践的方式学会IT技术。

公司简介 (<https://www.shiyanlou.com/aboutus>) 联系我们 (<https://www.shiyanlou.com/contact>) 常见问题 (<https://www.shiyanlou.com/faq#howtostart>)
我要开课 (<https://www.shiyanlou.com/labs>) 隐私协议 (<https://www.shiyanlou.com/privacy>) 会员条款 (<https://www.shiyanlou.com/terms>)
友情链接 (<https://www.shiyanlou.com/friends>)
站长统计 (http://www.cnzz.com/stat/website.php?web_id=5902315) 蜀ICP备13019762号 (<http://www.miibeian.gov.cn/>)



QQ群



微信



微博
(<http://weibo.com/shiyanlou2013>)