Computer System Security **CS3312**

计算机系统安全

2024年 春季学期

主讲教师：张媛媛 副教授

上海交通大学 计算机科学与技术系

第八章

软件安全：整型溢出

Software Security: Integer Overflow

引例

```
const long MAX_LEN = 0x00007fff;

short len = strlen(input);

if (len < MAX_LEN)  // compare a signed 16-bit with
                    // a signed 32-bit, lead to
                    // an upcasting to 32-bit
    //do A
else
   // do B
```

逻辑错误!
无论如何都无法执行到B分支

整型溢出高危操作

- Casting operations 类型转换 （upcast / downcast）
- Binary operations 二进制操作
- Compiler optimizations 编译器优化
- Arithmetic operations 算术运算
  - 加减乘除模运算

```
unsigned 8-bit integers: 255+1=0
unsigned 8-bit integers: 2-3=255
Signed 8-bit integer: 127+1=-128
```

```
const long MAX_LEN = 0x00007fff;

short len = strlen(input);

if (len < MAX_LEN)  // compare a signed 16-bit with
                    // a signed 32-bit, lead to
                    // an upcasting to 32-bit
    //do A
else
   // do B
```

参考short->long的upcast：

```
len = 0x0100;
(long)len = 0x00000100;
```

```
len = 0xffff;  //-1
(long)len = 0xffffffff;
```

程序本意：如果第2行代码的 strlen(**input**) 如果是一个比0x00007fff大的正数，按程序本意应执行到B分支。这样一个**input**至少大于等于 0x00007fff + 1 = 0x00008000，此例中input取这个值。

问题解析：
line2处，strlen(**input**) 返回类型为long(4B)，却被强制转化为short(2B) 发生downcast，变为0x8000，有符号数的负数
line3处，len(signed short (2B)) 和MAX_LEN(signed long (4B)) 做比较，发生upcast，位数较少的被 upcast 到32位(4B)，len仍然保持为负数（upcast操作不会改变正负性），则程序走向A。与程序本意相违背。

二进制操作和编译器优化

```
int flags = 0x7f;
char LowByte = 0x80;

if(flags ^ LowByte == 0xff)
        return ItWorked;
```

- **想象的状况:**

int flags = 0111 1111;
char lowbyte = 1000 0000
flags ^ lowbyte = 1111 1111

- **实际状况:**

if语句中, flags 和 lowbyte 类型均转换为int,
flags = 0x0000 007f, lowbyte = 0xffff ff80 (compiler认为它原来是个负数)
因此 flag ^ lowbyte = 0xffff, 而ffff 是个负数

强制转化规则总结

- **Signed int to Larger signed int**
  - The smaller value is sign-extended; for example, (char)0x7f cast to an int becomes 0x0000007f, but (char)0x80 becomes 0xffffff80.
- **Signed int to Same-Size unsigned int**
  - The bit pattern is preserved, though the value will change if the input is negative. So (char)0xff (–1) remains 0xff when cast to an unsigned char, but –1 clearly has a different meaning than 255. Casts between signed and unsigned integers are always danger signs to watch out for.
- **Signed int to Larger unsigned int**
  - This combines the two behaviors: The value is first sign-extended to a larger signed integer and then cast to preserve the bit pattern. This means that positive numbers behave as you'd expect, but negative numbers might yield unexpected results. For example, (char)–1 (0xff) becomes 4,294,967,295 (0xffffffff) when cast to an unsigned long.
- **Unsigned int to Larger unsigned int**

safe

  This is the best case: the new number is zero-extended, which is generally what you expect. Thus (unsigned char)0xff becomes 0x000000ff when cast to an unsigned long.
- **Unsigned int to Same-Size signed int**
  - As with the cast from signed to unsigned, the bit pattern is preserved, and the meaning of the value may change, depending on whether the uppermost (sign) bit is a 1 or 0.
- **Unsigned int to Larger signed int**

safe

  This behaves very much the same as casting from an unsigned int to a larger unsigned int. The value first zero-extends to an unsigned int the same size as the larger value and then is cast to the signed type. The value of the number is maintained and won't usually cause programmer astonishment.