



## 第5章 整数安全

---

北京邮电大学 徐国胜  
[guoshengxu@bupt.edu.cn](mailto:guoshengxu@bupt.edu.cn)



# 整数

---

- 整数已日益成为C和C++程序中漏洞的源泉
- 在大多数C和C++软件系统的开发中都没有系统地进行整数范围检查
- 因整数导致的安全缺陷问题肯定存在
  - 其中一些缺陷很可能会成为漏洞。
- 当程序对一个整数求出了一个非期望中的值，软件漏洞就出现了。



# 整数安全例子

---

```
1.  int main(int argc, char *argv[]) {  
2.      unsigned short int total;  
3.      total = strlen(argv[1])+  
4.          strlen(argv[2])+1;  
5.      char *buff = (char *)malloc(total);  
6.      strcpy(buff, argv[1]);  
7.      strcat(buff, argv[2]);  
8.  }
```



# 整数表示方法

---

- 原码表示法
- 反码表示法
- 补码表示法
- 对整数表示法而言，需要考虑的问题主要就是负数的表示

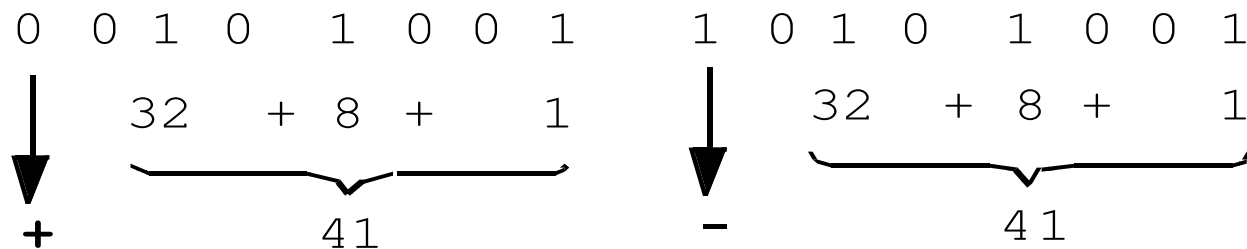
# 原码值的表示方法

- 利用最高位表示数值的符号：

- 0 正

- 1 负

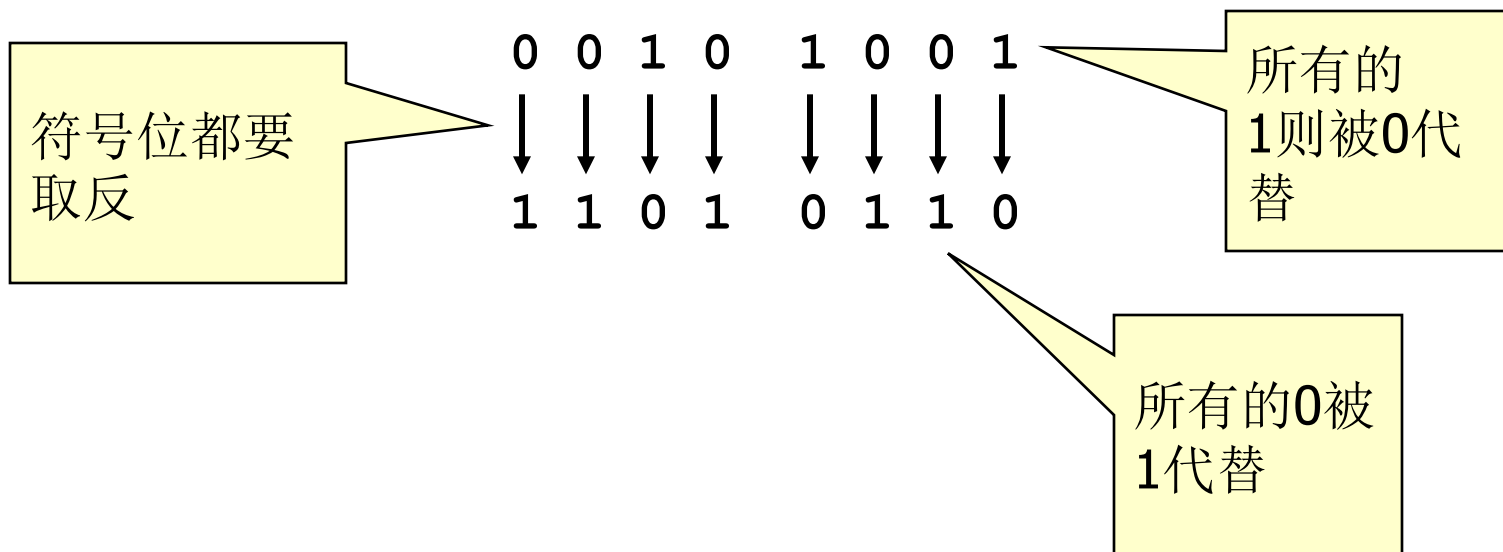
- 余下的所有低位表示值的大小



- 如果最高位未置位，表示**+41**，如果最高位被置位，则表示**-41**。

# 反码表示法

- 由于实现原码表示法所需的电路过于复杂，因此人们后来采用反码表示法取而代之。
- 将一个整数值的每一位取反，就得到于其对应的负数。





# 补码表示法

- 补码表示法的负数是在反码表示法的结果末位加 1 而得

$$\begin{array}{cccccccc} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array} + 1 = \begin{array}{cccccccc} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}$$

- 补码表示法对0只有+0一种表示。
- 最高位仍然是符号位。
- 正数的补码表示法则与原码表示法相同。



# 带符号和无符号类型

---

- C和C++ 中的整数分为带符号和无符号两种。
- 每一种带符号类型都有对应的无符号类型。



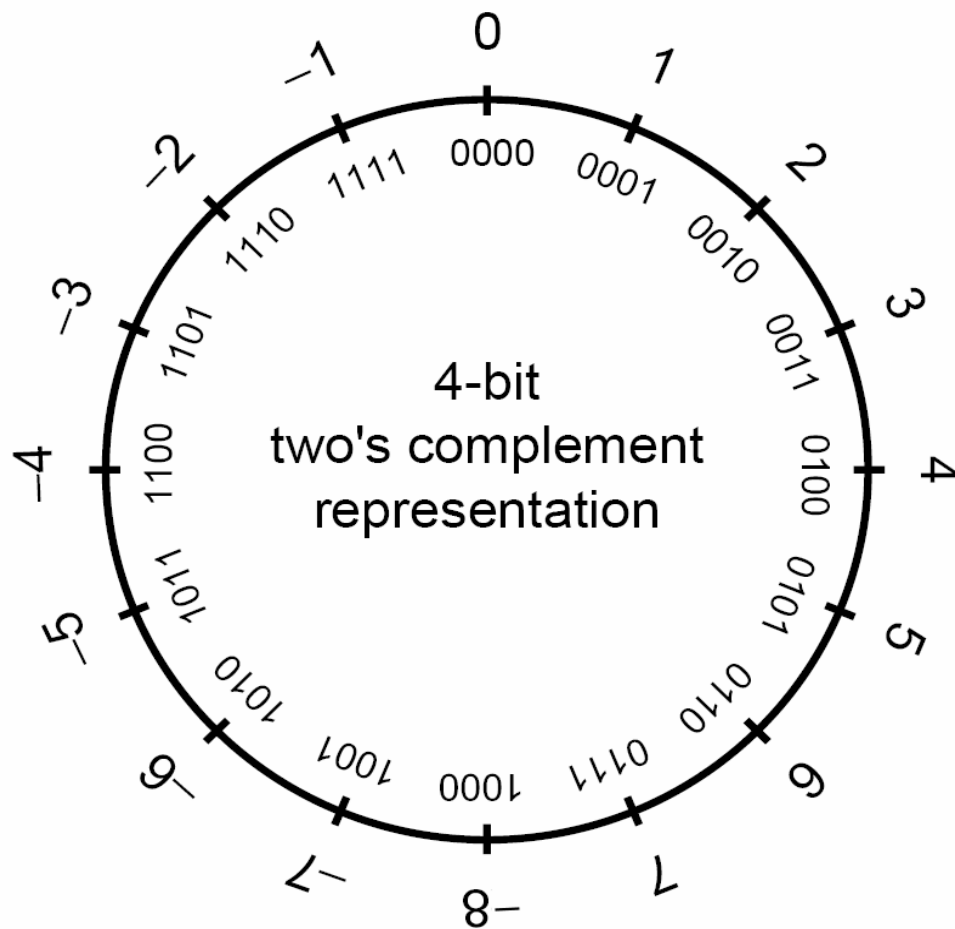


# 带符号整数

---

- 带符号整型用来表示正值和负值
- 在一个使用补码表示法的计算机上，带符号整数的取值范围是  $-2^{n-1}$  到  $2^{n-1}-1$ 。

# 帶符号的整数表示法





# 无符号整数

---

- 无符号整数的取值范围：最小值为0，最大值依赖于该类型所占位数的大小
- 如果该类型所占位数的大小为  $n$ ，那么最大值即为  $2^{n-1}$

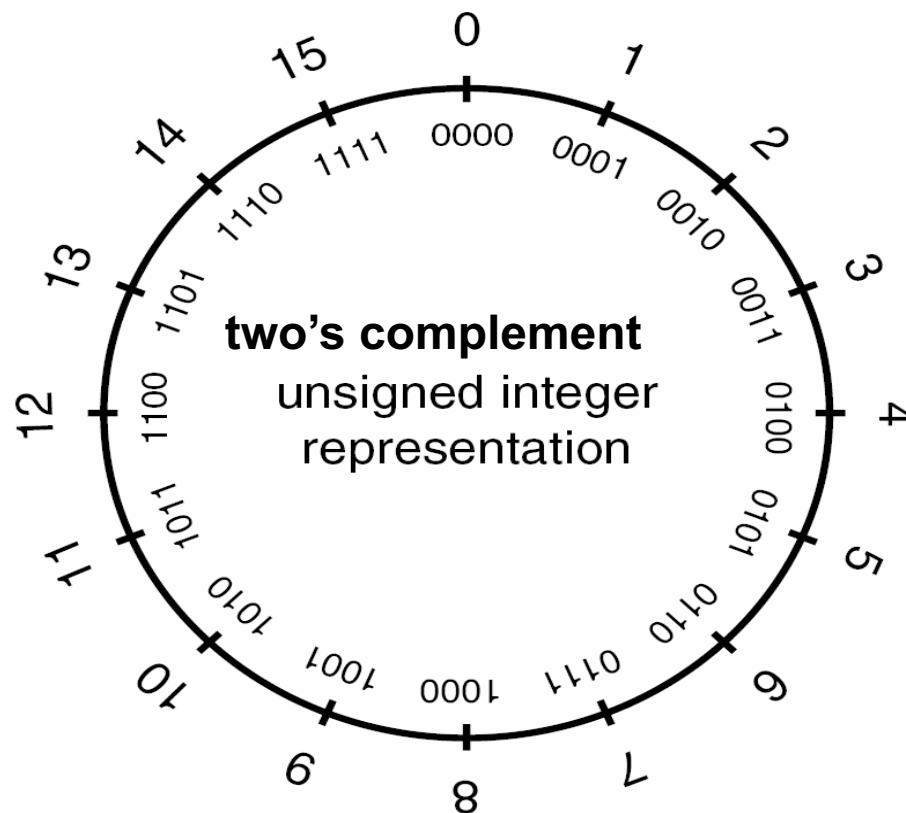


# 无符号整数

---

- 无符号整数的取值范围：最小值为0，最大值依赖于该类型所占位数的大小
- 如果该类型所占位数的大小为  $n$ ，那么最大值即为  $2^n - 1$
- 任一种带符号整型都有对应的无符号整型。

# 无符号整数表示法





# 整数类型

---

- 整数类型可以分为两大类：标准整型和扩展整型。
  - 标准整型包括所有已经广为人知的那些整型
  - 扩展整型则是指**C99**标准中定义的带有定长约束的整型。



# 标准类型

---

- 标准整型包含的类型如下（以长度非递减的顺序排列）：
  - `signed char`
  - `short int`
  - `int`
  - `long int`
  - `long long int`



# 扩展整型

---

- 扩展整型由具体实现定义，包含如下类型
  - `int#_t`, `uint#_t` 其中，`#` 表示该类型的精确宽度
  - `int_least#_t`, `uint_least#_t` 其中 `#` 表示该类型的最小宽度
  - `int_fast#_t`, `uint_fast#_t` 其中 `#` 表示在保证指定的最小宽度的前提下具有最快处理速度的宽度值
  - `intptr_t`, `uintptr_t` 表示其宽度足够保存对象指针类型的整型。
  - `intmax_t`, `uintmax_t` 表示最宽的整型。





# 平台相关的整型

---

- 厂商通常会提供一些平台相关的整数类型
- 例如微软Windows API就定义了大量的整型
  - `__int8, __int16, __int32, __int64`
  - `ATOM/typedef WORD ATOM`
  - `BOOLEAN, BOOL`
  - `BYTE`
  - `CHAR`
  - `DWORD, DWORDLONG, DWORD32, DWORD64`
  - `WORD`
  - `INT, INT32, INT64`
  - `LONG, LONGLONG, LONG32, LONG64`
  - `Etc.`



# 整数取值范围

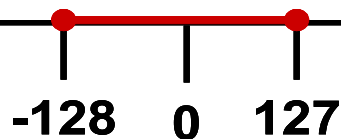
---

- 一个整数类型的最大值和最小值取决于
  - 该类型的表示法
  - 是否带符号
  - 分配的内存位数大小
- **C99**标准规定了这些值的最小范围。



# 整数取值范围的例子

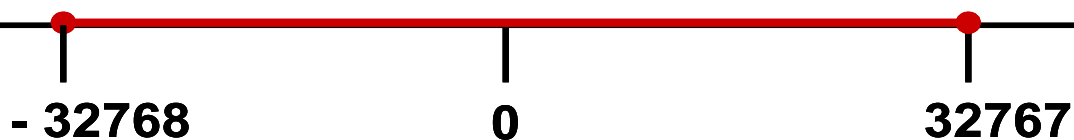
**signed char**



**unsigned char**



**short**



**unsigned short**





# 整型转换

---

- 在C和C++中，类型转换既可能作为转型（**cast**）操作的结果显式发生，也可能因为某个操作的需要而隐式发生。
- 整型转换可能会导致数据丢失或错误的表示。
- 隐式转换是C语言可以对混合数据类型执行操作能力的结果。
- C99标准规定了C编译器应该如何处理转换操作
  - 整型提升
  - 整型转换级别
  - 普通算术转换



# 整型提升

---

- 在比**int**小的整型进行操作时，它们会被提升。
- 如果原始类型的所有值都可以用**int** 表示，
  - 较小的类型会被转换成一个**int**
  - 否则被转换成一个**unsigned int**
- 整型提升被作为普通算术转换的一个组成部分
  - 某些自变量表达式
  - 一元的+、 - 、 和 ~操作符
  - 移位操作



# 整型提升例子

---

- 由于整型提升，因此这里**c1**和**c2** 都要被提升到**int**类型的大小
- ```
char c1, c2;  
c1 = c1 + c2;
```
- 然后两个**int**类型的数据相加，求和结果被截断以适应**char**类型的大小。
- 整型提升主要是为了防止运算过程中中间结果发生溢出而导致算术错误

# 隐式转换

- 1. `char cr` c1、c2的和超过了signed char类型的最大值
- 2. `c1 = 100;`
- 3. `c2 = 90;`
- 4. `c3 = -120;` 然而，由于发生了整型提升，c1、c2和c3都被转换为整型，因此整个表达式的结果能够被成功地计算出来。
- 5. `cresult = c1 + c2 + c3;`

该结果随后被截断，并存储在 `cresult` 中，没有数据丢失

c1与c2相加



# 整数转换级别

---

- 每一种整数类型都有一个相应的整型转换级别（**integer conversion rank**），决定了转换操作将会如何执行。





# 整数转换级别的规则

---

- 没有任何两种不同的带符号整型具有相同的级别，即使它们的（内存）表示法相同。
- 低精度的带符号整型的级别比高精度的带符号整型类型的级别低。
- `long long int`类型的级别比`long int`高，`long int`的级别比`int`高，`int`的级别比`short int`高，`short int`的级别比`signed char`高。
- 无符号整型的级别与对应的带符号整型的级别相同。



# 从无符号整型转换

---

- 从较小的无符号整型转换到较大的无符号整型
  - 总是安全的
  - 通常通过对其值进行零扩展而完成
- 当一个较大的无符号整型被转换到一个较小的无符号整型的时候
  - 较大的值将会被截断
  - 低位数据被保留



## 从无符号整型转换2

---

- 当无符号整型转换到其对应的带符号整型的时候
  - 位模式（即所有的位数据）将会被保留，因此没有数据会因此丢失
  - 最高位数据变成了符号位
- 如果该符号位被置位，该值的符号和大小都会发生改变。

| 源：无符号        | 目标类型                  | 转换方法           |
|--------------|-----------------------|----------------|
| <b>char</b>  | <b>char</b>           | 保留原位表示，最高位变符号位 |
| <b>char</b>  | <b>short</b>          | 零扩展            |
| <b>char</b>  | <b>long</b>           | 零扩展            |
| <b>char</b>  | <b>unsigned short</b> | 零扩展            |
| <b>char</b>  | <b>unsigned long</b>  | 零扩展            |
| <b>short</b> | <b>char</b>           | 保留低位字节         |
| <b>short</b> | <b>short</b>          | 保留原位表示，最高位变符号位 |
| <b>short</b> | <b>long</b>           | 零扩展            |
| <b>short</b> | <b>unsigned char</b>  | 保留低位字节         |
| <b>long</b>  | <b>char</b>           | 保留低位字节         |
| <b>long</b>  | <b>short</b>          | 保留低位字          |
| <b>long</b>  | <b>long</b>           | 保留原位表示，最高位变符号位 |
| <b>long</b>  | <b>unsigned char</b>  | 保留低位字节         |
| <b>long</b>  | <b>unsigned short</b> | 保留低位字          |

关键：

丢失数据

数据曲解



# 带符号整型转换

---

- 当一个非负的带符号整数被转换为一个相同大小或更大的无符号整型的时候
  - 值不会发生变化
  - 带符号整数需作符号扩展
- 当一个带符号整数被转换为一个较短的带符号整型的时候，则是通过截断高位完成的



## 带符号整型转换2

---

- 当带符号整数转换到无符号整数
  - 其位模式被保留，故不会有数据的丢失
  - 高位失去了符号位的功能
- 如果带符号整型的值非负的话，那它的值不会发生改变
- 如果其值是负数的话，得到的无符号结果将被求值为一个非常大的带符号整数

| 源类型   | 目标类型           | 转换方法                                |
|-------|----------------|-------------------------------------|
| char  | short          | 符号扩展                                |
| char  | long           | 符号扩展                                |
| char  | unsigned char  | 保留原位表示，最高位失去符号位功能                   |
| char  | unsigned short | 符号扩展到short; short转换到unsigned short  |
| char  | unsigned long  | 符号扩展到 long; long 转换到 unsigned long  |
| short | char           | 保留低位字节                              |
| short | long           | 符号扩展                                |
| short | unsigned char  | 保留低位字节                              |
| short | unsigned short | 保留原位表示，最高位失去符号位功能                   |
| short | unsigned long  | 符号扩展 到 long; long 转换到 unsigned long |
| long  | char           | 保留低位字节                              |
| long  | short          | 保留低位字                               |
| long  | unsigned char  | 保留低位字节                              |
| long  | unsigned short | 保留低位字                               |
| long  | unsigned long  | 保留原位表示，最高位失去符号位功能                   |

关键：

丢失数据

曲解数据

# 带符号整型转换例子

```
1. unsigned int l = ULONG_MAX;
```

```
2. char c = -1;
```

```
3. if (c == l) {
```

```
4.   printf("-1 = 4,294,967,295?\n");
```

```
5. }
```

c与1 进行相等性  
比较

由于整型提升规则发挥作用，导致c被转换成为一个无符号的整数，其值为**0xFFFFFFFF** 或  
4,294,967,295





# 带符号或无符号字符

---

- **char**类型既可以是带符号的，也可以是无符号
- 当一个符号位被置位的**signed char**类型的数据被当作整型保存时，其结果就是一个负数。
- 当处理值可能会大于**127 ( 0x7F)** 的字符数据时，对于涉及的字符缓冲区、指针及转型，最好用**unsigned char**代替**char**或**signed char**。



# 普通算术转换

---

1. 如果两个操作数具有同样的类型，则不需要进一步的转换。
2. 如果两个操作数拥有同样的整型（带符号或无符号），具有较低整数转换级别的类型的操作数会被转换到拥有较高级别的操作数的类型。
3. 如果具有无符号整型操作数的级别大于或等于另一个操作数类型的级别，则带符号整型操作数将被转换为无符号整型操作数的类型。
4. 如果带符号整型操作数类型能够表示无符号整型操作数类型的所有可能值，则无符号整型操作数将被转换为带符号整型操作数的类型。
5. 否则，两个操作数都被转换为与带符号整型操作数类型相对应的无符号整型。



# 整数错误情形1

---

- 整型操作下面情况下会导致非预期的结果
  - 溢出
  - 符号错误
  - 截断错误



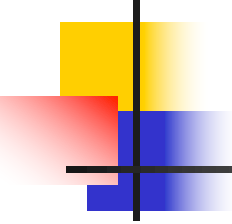
# 整数溢出

---

- 当一个整数被增加超过其最大值或被减小小于其最小值时即会发生整数溢出
- 带符号和无符号的数都有可能发生溢出

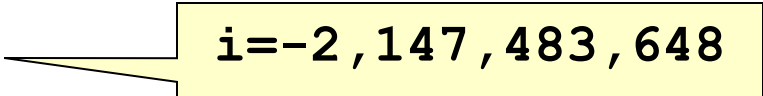
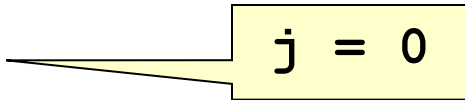
带符号溢出发生于对符号位执行进位时

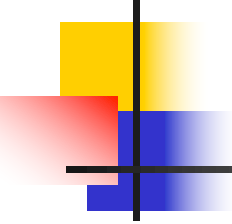
无符号溢出则发生于当底层表示不再能够表示一个值时。



# 整数溢出例子S 1

---

- 1. `int i;`
- 2. `unsigned int j;`
- 3. `i = INT_MAX; // 2,147,483,647`
- 4. `i++;`   
`i=-2,147,483,648`
- 5. `printf("i = %d\n", i);`
- 6. `j = UINT_MAX; // 4,294,967,295;`
- 7. `j++;`
- 8. `printf("j = %u\n", j);`   
`j = 0`



## 整数溢出例子S 2

■ 9. `i = INT_MIN; // -2,147,483,648.`

`i=2,147,483,647`

■ 10. `i--;`

■ 11. `printf("i = %d\n", i);`

■ 12. `j = 0;`

■ 13. `j--;`

`j = 4,294,967,295`

■ 14. `printf("j = %u\n", j);`



# 截断错误

---

- 截断错误发生于
  - 将一个较大整型的数转换到较小的整型
  - 该数的原值超出较小类型的表示范围
- 原值的低位被保留下来而高位则被丢弃。

# 截断错误例子

- `1. char cresult, c1, c2, c3;`
- `2. c1 = 100;`
- `3. c2 = 90;`
- `4. cresult = c1 + c2;`

c1加 c2 超过了signed char (+127) 的最大值

当该值赋给一个太小的数据类型的时候，而无法表示结果值的时候。会发生截断错误。

在操作前，把比**int** 小的整型提升为 **int** 或**unsigned int**





# 符号错误

---

- 从无符号整型转换到带符号整型
  - 相同大小-位模式保留不变；最高位变成符号位
  - 更大- 进行符号扩展，然后才执行转换
  - 更小-保留低位
- 如果无符号整数的最高位
  - 没被设置- 值不变
  - 被设置 - 变成负值



# 符号错误

---

- 带符号整型转换到无符号整型
  - 相同大小-位模式保留不变; 最高位有效值位
  - 更大- 进行符号扩展, 然后才执行转换
  - 更小-保留低位
- 如果带符号整数的值是
  - 非负的- 值不变
  - 负的 -结果通常是一个很大的正值

# 符号错误例子

1. `int i = -3;`
2. `unsigned short u;`
3. `u = i;`
4. `printf("u = %hu\n", u);`

隐式转换为较小的无符号整数

有足够的位来表示值，所以没有发生截断。但是补码表示法被解释为一个大的符号值，所以 **u = 65533, FFFD**



## 5.4 整数操作

---

- 整数操作会导致错误和非预期的值
- 非预期的整数值可能会导致
  - 非预期的程序行为
  - 安全漏洞
- 大部分整数操作会导致异常条件



# 整数的重要使用场景

---

- 作为数组索引
- 在任何指针的算术运算中
- 作为一个对象的长度或大小
- 作为一个数组边界（例如，一个循环计数器）
- 作为内存分配函数的参数
- 在对安全要求很关键的代码中

# 异常情况

| 运算符             | 异常情况       | 运算符 | 异常情况            | 运算符  | 异常情况           | 运算符 | 异常情况 |
|-----------------|------------|-----|-----------------|------|----------------|-----|------|
| +               | 溢出, 回绕     | -=  | 溢出, 回绕, 截断      | <<   | 溢出, 回绕         | <   | 无    |
| -               | 溢出, 回绕     | *=  | 溢出, 回绕, 截断      | >>   | 无 <sup>a</sup> | >   | 无    |
| *               | 溢出, 回绕     | /=  | 溢出, 截断          | &    | 无              | >=  | 无    |
| %               | 溢出         | <<= | 溢出, 回绕, 截断      | ^    | 无              | ==  | 无    |
| ++              | 溢出, 回绕     | >>= | 截断 <sup>a</sup> | ~    | 无              | !=  | 无    |
| --              | 溢出, 回绕     | &=  | 截断              | !    | 无              | &&  | 无    |
| =               | 截断         | =   | 截断              | 一元 + | 无              |     | 无    |
| += <sup>a</sup> | 溢出, 回绕, 截断 | ^=  | 截断              | 一元 - | 溢出, 回绕         | ?:  | 无    |

- 不包含算术转换产生的错误
- 整数错误能够被侦测, 有符号溢出和无符号回绕都被描述为实当的先验条件测试和后验条件测试



# 整数加法

---

- 加法用来将两个算术操作数或者一个整数与一个指针相加
- 如果两个操作数都是算术类型，那么将会对它们执行普通算术转换
- 如果结果整型占用的位数不足以表示其结果，那么就会导致溢出



# IA-32 加法指令

---

- IA-32 加法指令
  - add destination, source
  - 将第一操作数（目的）与第二操作数相加
  - 并将结果存放到目的操作数
  - 目的操作数可以是一个寄存器或者内存位置
  - 源操作数可以是一个立即数、寄存器或者内存位置
  - 侦测和报告带符号和无符号的整数溢出条件





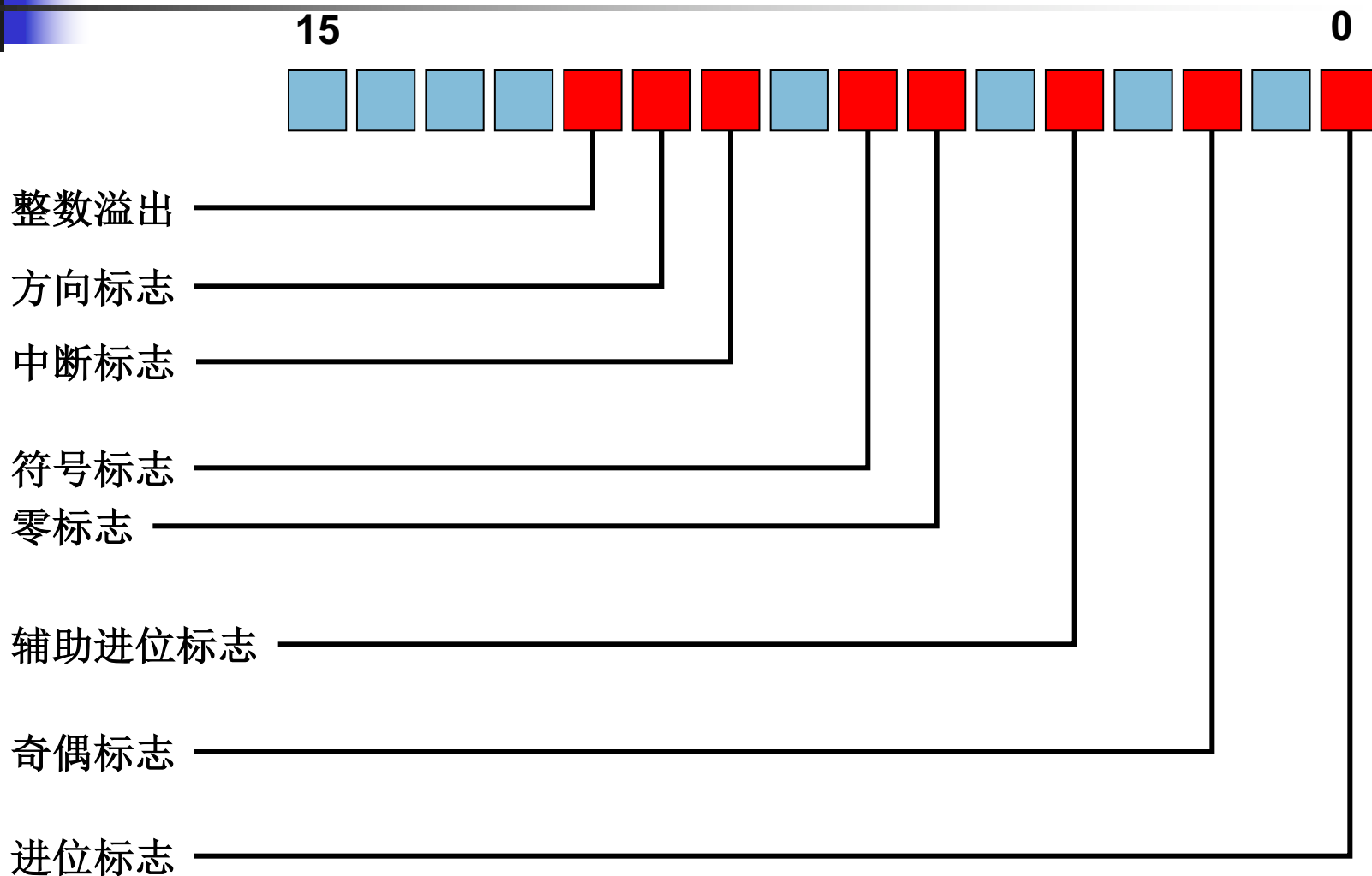
# IA-32 加法指令

---

## ■ **add ax, bx**

- 将16位寄存器**bx**和16位寄存器**ax**相加
- 并将结果存储到寄存器**ax**中
- 加法指令在标志寄存器中设置标志
  - 一个用于指示带符号算术溢出的溢出标志。
  - 一个用于指示无符号算术溢出的进位标志。

# 标志寄存器的布局





# 解释标志

---

- 在机器水平上带符号和无符号的整数是没有区别的。
- 整数溢出和进位标志必须要根据实际情况来解释



# 加法 带符号/无符号的char

当把两个**signed char**相加的时候，它们的值会发生符号扩展

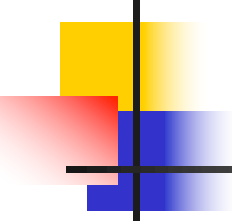
**sc1 + sc2**

```
1. movsx          eax, byte ptr [sc1]
2. movsx          ecx, byte ptr [sc2]
3. add            eax, ecx
```

当把两个**unsigned char**相加的时候，它们的值会发生零扩展来避免改变大小

**uc1 + uc2**

```
4. movzx          eax, byte ptr [uc1]
5. movzx          ecx, byte ptr [uc2]
6. add            eax, ecx
```



## 加法 带符号/无符号的int

---

把两个**unsigned int** 相加

**ui1 + ui2**

```
7. mov          eax, dword ptr [ui1]
```

```
8. add          eax, dword ptr [ui2]
```

为**signed int** 的值生成相同的代码

# 加法 带符号的 long long int

加法指令 使得低32 bits相加

■ **s111 + s112**

```
■ 9.  mov          eax, dword ptr [s111]
■ 10. add          eax, dword ptr [s112]
■ 11.  mov          ecx, dword ptr [ebp-98h]
■ 12.  adc          ecx, dword ptr [ebp-0A8h]
```

**adc** 指令把高位上的32 位和进位标志位的值相加



# 无符号整数溢出检测

---

- **进位标志**意味着无符号算术溢出
- 无符号整数溢出通过使用下列方法检测：
  - **jc** 指令 (如果有进位标志, 则转跳)
  - **jnc**指令(如果没有进位标志, 则转跳)
- 这些跳转指令放在下列条件后
  - 做**32**位运算时, 放在指令**add**之后
  - 做**64**位运算时, 则放在**adc**之后



# 带符号整数溢出检测

---

- **进位标志**意味着带符号算术溢出
- 带符号整数溢出通过使用下列方法检测
  - **j<sub>o</sub>**指令 (如果有溢出, 则转跳)
  - **j<sub>no</sub>**指令(如果没有溢出, 则转跳)
- 这些跳转指令放在下列条件后
  - 做**32**位运算肘, 放在指令**add**之后
  - 做**64**位运算肘, 则放在**adc**之后





# 先验条件

---

带符号整数的相加可能会导致整数溢出，如果加法操作的左操作数(left-hand side, LHS)和右操作数（ tight-band side, RHS ）的和大于  
UINT\_MAX （对于int相加而言）

或大于

UL LONG\_MAX （对于unsigned long long相加而言）的话。



# 先验条件例子

---

当A 和 B 是无符号的，并且满足下列条件时，会发生整数溢出

$$\mathbf{A + B > UINT\_MAX}$$

整数溢出防范检测代码：

$$\mathbf{A > UINT\_MAX - B}$$

当A 和 B 是long long int，并且满足下列条件时，会发生整数溢出

$$\mathbf{A + B > ULLONG\_MAX}$$



# Int类型的带符号整数相加

- 带符号整数相加更加复杂

| LHS | RHS | 异常情况                                                |
|-----|-----|-----------------------------------------------------|
| 正数  | 正数  | 整数溢出 if $\text{INT\_MAX} - \text{LHS} < \text{RHS}$ |
| 正数  | 负数  | 不可能溢出                                               |
| 负数  | 正数  | 不可能溢出                                               |
| 负数  | 负数  | 整数溢出 if $\text{LHS} < \text{INT\_MIN} - \text{RHS}$ |



# 后验条件

---

- 是先执行加法，然后对结果进行评估。
- 例如：令 **sum = lhs + rhs**.
  - 如果 **lhs** 非负且 **sum < rhs**，表明发生了溢出
  - 如果 **lhs** 为负且 **sum > rhs**，表明发生了溢出
  - 所有其他的情况则表明加注操作成功完成而无溢出。
  - 对于无符号整数来说，如果 **sum** 比任意一个操作数小，就表明发生了溢出



# 整数减法

---

- IA-32指令集包含
  - **sub** (减法)
  - **sbb** (带借位减法)
- **sub**和**sbb**指令可以对溢出和进位标志置位，以表示带符号或无符号结果的溢出



## sub 指令

---

- **sub**指令从第一个操作数（目的操作数）中减去第二个操作数（源操作数），并将结果存储在目的操作数中
- 目的操作数可以是一个
  - 寄存器
  - 内存位置
- 源操作数可以是一个
  - 立即数
  - 寄存器
  - 内存位置



## sbb 指令

---

- 指令**sbb**通常被用在多字节或多字减法场合
- **sbb**指令将源操作数（第二个操作数）和进位标志相加，并从目的操作数（第一操作数）中减去刚才所得的结果。
- 减法操作的结果被存储在目的操作数中
- 进位标志表示上一个减法操作中是否有借位出现



# 带符号long long整型 Sub

**s111 - s112**

1. `mov eax, dword ptr [s111]`
2. `sub eax, dword ptr [s112]`
3. `mov ecx, dword ptr [ebp-0E0h]`
4. `sbb ecx, dword ptr [ebp-0F0h]`

**sub** 指令减去低位的32 位

**sbb** 指令减去低位的32 bits





# 先验条件

---

要测试无符号整数减能是否溢出，只需检验是否  $LHS < RHS$ 。对于具有相同符号的精符号整散，不会发生异常情况

- 对于混合符号的带符号整数情形，应用下列规则：
  - 如果LHS为负，而RHS为正，对signed int类型检查  $lhs < INT\_MIN + rhs$
  - 如果LHS非负，且RHS为负，检查  $lhs > INT\_MAX + rhs$
- 例如，  $0 - INT\_MIN$  会导致溢出情况，因为该操作的结果比能表示的最大值还要大1.



# 后验条件

---

- 如果要测试带符号整数的溢出，设  $\text{difference} = \text{lhs} - \text{rhs}$ ，并应用如下规则
  - 如果  $\text{rhs}$  非负，并且  $\text{difference} > \text{lhs}$ ，则发生了溢出
  - 如果  $\text{rhs}$  为负，并且  $\text{difference} < \text{lhs}$ ，则发生了溢出
  - 其他所有情况，没有溢出发生
- 对于无符号整数而言，如果  $\text{difference} > \text{lhs}$ ，则发生溢出



# 整数乘法

---

- 乘法操作容易引起溢出错误，因为当进行乘法运算时，即使是较小的操作数也可能导致给定的整数类型溢出
- 其中一个解决方案是，为积分配两倍于“两个操作数中类型较大者的”的存储空间



# 乘法指令

---

IA-32指令集包含有一个

**mul** (无符号乘法) 指令

**imul** (带符号乘法)指令

**mul**指令

用于将第一个操作数（目的操作数）和第二个操作数（源操作数）相乘，并将结果存储在目的操作数中

# 无符号数乘法伪代码

```
1. if (OperandSize == 8) {  
2.     AX = AL * SRC;  
3. else {  
4.     if (OperandSize == 16)  
5.         DX:AX = AX * SRC;  
6.     }  
7.     else { // OperandSize == 32  
8.         EDX:EAX = EAX * SRC;  
9.     }  
10. }
```

8位操作数，存储在16位的目的寄存器中

16位操作数存储在32位目的寄存器中

32位操作数存储在64位目的寄存器中



# 进位标志和整数溢出标志

---

- 如果需要高位来表示两个操作数的积，则进位标志和溢出标志都被置位
- 如果不需要高位（也就是说它们全为0），那么进位标志和溢出标志都被消除



# 带符号和无符号字符乘法(Visual C++)

---

```
sc_product = sc1 * sc2;
```

```
1. movsx      eax, byte ptr [sc1]
2. movsx      ecx, byte ptr [sc2]
3. imul       eax, ecx
4. mov        byte ptr [sc_product], al
```

```
uc_product = uc1 * uc2;
```

```
5. movzx      eax, byte ptr [uc1]
6. movzx      ecx, byte ptr [uc2]
7. imul       eax, ecx
8. mov        byte ptr [uc_product], al
```



# 带符号和无符号整数乘法(Visual C++)

---

```
si_product = si1 * si2;
```

```
ui_product = ui1 * ui2;
```

```
9.  mov          eax, dword ptr [ui1]
10.  imul         eax, dword ptr [ui2]
11.  mov          dword ptr [ui_product],
    eax
```





## 带符号和无符号字符乘法(g++)

- 不管char是否带符号， g++对char类型的整数都使用mul 指令的字节形式

```
sc_product = sc1 * sc2;
```

```
uc_product = uc1 * uc2;
```

```
1. movb -10(%ebp), %a1
```

```
2. mulb -9(%ebp)
```

```
3. movb %a1, -11(%ebp)
```



## 带符号和无符号整数乘法(g++)

---

- g++对单字长度的整型，则采用**imul** 指令，不管该类型是否带符号

```
si_product = si1 * si2;
```

```
ui_product = ui1 * ui2;
```

```
4. movl    -20(%ebp), %eax
```

```
5. imull   -24(%ebp), %eax
```

```
6. movl    %eax, -28(%ebp)
```



# 先验条件

---

为了防止无符号整数相乘时发生溢出，可以检验

$$A * B > \text{MAX\_INT}$$

也就是  $A > \text{MAX\_INT} / B$

但是除法的开销更大



# 后验条件

---

- 后验条件同样可以用来检测乘法溢出，不过由于结果需要“两倍于较大操作数的大小”的位数进行表示，因此与加法相比这种情形要复杂一些



## 后验条件

---

- 将两个操作数放到下一个更大的数据类型上，然后相乘。
- 对于无符号整数
  - 检查下一个大整数的高阶位，如果被设置了，抛出错误。
- 对带符号整数，如果结果的高半部分及低半部分的符号位全为0或1，则没有发生整数溢出。



## 后验条件

---

- 对于**16位**（一个字长）带符号整数，可以通过这种方式简化对溢出的检测：将**LHS**和**RHS**两个操作数都转型成**32位**值，并将乘积结果存储到**32位**的目的域中。如果结果积右移**16位**和右移**15位**所得结果不一致，则说明发生了溢出
- 对于正的结果，这种方法可以检测结果值是否溢出到低**16位**中的符号位，对于负的结果，这种方法可以检测结果值是否溢出到高半部分的位中。



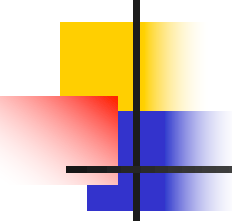
# Upcast例子

---

```
void* AllocBlocks(size_t cBlocks) {  
    //分配没有 blocks是一个错误  
    if (cBlocks == 0) return NULL;  
    // 分配足够的内存  
    // 把结果提升到一个64-bit的整数  
    // 检查32-bit UINT_MAX  
    // 确保没有整数溢出  
    ULONGLONG alloc = cBlocks * 16;  
    return (alloc < UINT_MAX)  
        ? malloc(cBlocks * 16) : NULL;  
}
```



你能找到错误吗?



# 结果总是 > UINT\_MAX

```
void* AllocBlocks(size_t cBlocks) {
```

```
//分配没有 blocks 是一个结果是32-bit 值的32-bit 操作。结果被赋值到  
if (cBlocks == 0) return NULL; // 到一个ULONGLONG，但是计算中可能已经发生了的  
// 溢出。
```

```
// 分配足够的内存  
// 把结果提升到一个64-bit 数  
// 检查32-bit UINT_MAX  
// 确保没有整数溢出
```

```
ULONGLONG alloc = cBlocks * 16;
```

```
return (alloc < UINT_MAX)
```

```
    ? malloc(cBlocks * 16) : NULL;
```

```
}
```





# 修正的提升的例子

---

```
void* AllocBlocks(size_t cBlocks) {  
    //分配没有 blocks是一个错误  
    if (cBlocks == 0) return NULL;  
    // 分配足够的内存  
    // 把结果提升到一个64-bit的整数  
    // 检查32-bit UINT_MAX  
    // 确保没有整数溢出  
    ULONGLONG alloc = (ULONGLONG) cBlocks*16;  
    return (alloc < UINT_MAX)  
        ? malloc(cBlocks * 16)  
        : NULL;  
}
```



# 整数除法

---

- 如果32位或64位的带符号整数的MIN\_INT值除以-1，那么将会发生溢出。
  - 在32位情况下， $-2,147,483,648/-1$ 的结果应该等于2,147,483,648。
  - 由于2,147,483,648无法用带符号的32位整数表示，所以结果出错。
- 如果参与除法操作的整数的符号和类型不同，那么也容易出问题。

**- 2,147,483,648 / -1 = - 2,147,483,648**



# 错误检测

---

IA-32指令集包含如下除法指令

**div, divpd, divps, divsd, divss**

**fdiv, fdivp, fdiv, idiv**

**div** 指令

用源操作数（除数）除存储于**ax**、**dx:ax**或**edx: eax**寄存器中的（无符号）整数（被除数），并将结果存储于**ax**（**ah:al**）、**dx :ax**或**edx:eax**寄存器中。

■ **idiv**指令则对（带符号）值执行同样的操作



# 带符号整数除法

---

```
mov  eax, dword ptr [si_dividend]
```

```
cdq
```

```
idiv eax, dword ptr [si_divisor]
```

```
    si_quotient = si_dividend / si_divisor;
```

```
mov  dword ptr [si_quotient], eax
```



# 无符号整数除法

---

```
ui_quotient = ui1_dividend / ui_divisor;
```

```
mov eax, dword ptr [ui_dividend]
```

```
xor edx, edx
```

```
div eax, dword ptr [ui_divisor]
```

```
mov dword ptr [ui_quotient], eax
```



# 先验条件

---

- 可以通过检查分子是否为整型的最小值以及检查分母是否为一1来防止整型除法溢出的发生。
- 当然，只要确保除数不为0，就可以保证不发生除零错误



# 错误检测

---

- Intel 除法指令 **div** 和 **idiv** 没有设置整数溢出标志
- 下列情况会产生除法错误
  - 源操作数（除数）为0
  - 对于目的寄存器而言结果商值太大
- 除法错误会导致一个中断标志向量0.
- 错误报道时，处理器恢复错误指令开始执行时的机器状态。



# Microsoft Visual Studio

---

**C++**异常处理机制并不允许应用程序从下列异常恢复

- 从一个硬件异常
- 诸如下列异常
  - 存取违例
  - 除零错误

**Visual Studio** 提供

- 结构化异常处理的设施来处理这类硬件和其他异常情况。
- 为C语言提供了一套扩展，从而使C程序可以处理Win32结构化异常。

结构化异常处理是操作系统提供的一项设施，它不同于C++的异常处理机制





# C++ 异常处理机制

```
Sint operator / (signed int divisor) {  
    __try {  
        return si / divisor;  
    }  
    __except (EXCEPTION_EXECUTE_HANDLER) {  
        throw SintException (ARITHMETIC 整数溢出);  
    }  
}
```

除法嵌入到一个 **\_\_try** 块中

如果除法发生错误, **\_\_except** 逻辑块被执行



# C++异常处理

---

```
Sint operator / (unsigned int divisor) {  
    try {  
        return ui / divisor;  
    }  
    catch (...) {  
        throw  
            SintException (ARITHMETIC_integer);  
    }  
}
```

由于在Visual C++中C++异常是用结构化异常机制实现的，因此这种做法是可以的



# Linux 错误处理1

---

在Linux环境中，类似于除零错误这样的硬件异常是使用信号（**signal**）机制进行处理的。

如果源操作数（除数）为0，或者对于目的寄存器而言结果商值太大，系统将会产生一个**SIGFPE** (floating point exception，浮点异常)

为了防止程序在这种情况下非正常终止，可以像下面这样利用**signal()**函数安装一个信号处理器

```
signal(SIGFPE, Sint::divide_error)
```



# Linux 错误处理2

---

**signal()** 调用接受两个参数

- 信号值
- 信号处理器的地址

■ 由于除零错误是当作故障处理，因此，其返回地址指向发生故障的指令。如果该信号处理器只是简单地返回。那么引起故障的指令以及该信号处理器就会披无限循环地交替调用了。

■ 为了解决这个问题，信号处理器抛出了一个能够被调用函数捕获的C++ 异常



# 信号处理程序

---

```
static void divide_error(int val) {  
    throw SintException(ARITHMETIC_integer);  
}
```



## 5.5 漏洞

---

- 漏洞就是一系列允许违反显式或隐式的安全策略的情形。安全缺陷可能是由于硬件层的整数错误或者是跟整数有关的不完善逻辑所造成的
- 当这些安全缺陷与其他情形结合起来时，就可能产生漏洞



## (1) JPEG例子

---

- 基于在处理JPEG文件注释域（comment field）时存在的实际漏洞
  - JPEG文件的注释域中包含一个长为两个字节的长度域，后者用来指示注释域的长度（也包括该两个字节的长度域本身在内）
  - 为了确定注释字符串的单独长度（以便进行内存分配），函数会读取长度域的值并将其减2。
  - 后函数根据注释的长度加上用于表示终结null字符的1个字节所得的总长度来分配内存空间。

# Integer 整数溢出例子

```
void getComment(unsigned int len, char *src) {  
    unsigned int size;  
    size = len - 2;  
    char *comment = (char *)malloc(size + 1);  
    memcpy(comment, src, size);  
    return;  
}
```

分配到0字节内存可以成功  
执行

一个极大的正整数0xffffffff

```
int _tmain(int argc, _TCHAR* argv[]) {  
    getComment(1, "Comment ");  
    return 0;  
}
```

当图像注释的长度域的数值为1时可能会产生溢出





## (2) 内存分配例子

---

- 整数溢出例子也会发生于内存分配**calloc()**时，当计算一块内存区域的大小并调用**calloc()**或其他内存分配函数来分配内存时可能会引起整数溢出
- 可能会返回一个小于需求大小的缓存，从而可能会导致后面的缓冲区溢出
- 以下代码片断可能会产生漏洞

C: **p = calloc(sizeof(element\_t), count);**

C++: **p = new ElementType[count];**



# 内存分配

---

库函数**calloc**（）接受两个参数

- 存储元素类型所需要的空间
- 元素的个数

对于C++的**new**操作符的情形，则不需要显式指定元素类型大小

为了计算所需内存的大小，使用元素个数乘以该元素类型所需的单位空间来计算



# 整数溢出条件

---

如果计算所得结果无法用带符号整数表示，那么，尽管分配程序看上去能够成功地执行，但实际上它只会分配非常小的内存空间。

应用程序对分配的缓冲区的写操作可能会越界，从而导致基于堆的缓冲区溢出。

### (3) 符号错误例子 1

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]) {
    int len;
    char buf[BUFF_SIZE];
    len = atoi(argv[1]);
    if (len < BUFF_SIZE) {
        memcpy(buf, argv[2], len);
    }
}
```

程序接受两个参数：将要被复制的字符串和它的长度

**len**被声明为带符号整型

**argv[1]** 也可以是一个负值

一个负值能够绕过该检测

值被认为是无符号的**size\_t**类型



## 符号错误例子 2

---

负的长度值被解释为一个很大的正整数，从而导致缓冲区溢出，这个漏洞可以通过限制整数`len`为一个有效值来避免

- 假设一个更有效的范围校验以保证`len`的值在0到`BUFF_SIZE`之间
- 声明为无符号整型 **declare as an unsigned integer**
  - 消除在调用函数`memcpy()`时从带符号整型到无符号整型的转换
  - 防止发生符号错误



## (4) 截断:漏洞执行

```
bool func(char *name, long cbBuf) {  
    unsigned short bufSize = cbBuf;  
    char *buf = (char *)malloc(bufSize);  
    if (buf) {  
        memcpy(buf, name, cbBuf);  
        if (buf) free(buf);  
        return true;  
    }  
    return false;  
}
```

**cbBuf** 用户初始化用于分配**buf**内存的 **bufSize**

**cbBuf**被声明为**long**用于设定**memcpy()**操作中的大小参



# 截断漏洞

---

- **cbBuf**被临时保存在了一个unsigned short bufsize中
- 不论是GCC还是Visual C++，在基于IA-32的编译器上，unsigned short的最大值都是65 535。而同一平台上signed long的最大值是2 147 483 647。
- 任何值位于65 535和2 147 483 647之间的cbBuf在进行赋值时都会发生截断错误



## 截断漏洞

---

- 当bufSize同时用于调用malloc ()和memcpy()的时候，只会发生错误并不会产生漏洞
- 由于bufSize被用于分配缓冲区的大小，而cbBuf则是用来在调用函数memcpy()时指定大小，因此，任何在1 到 2,147,418,112 (2,147,483,647 - 65,535)字节之间的buf值都会引起溢出





## (5) 非异常的整数逻辑错误

---

许多可利用的软件缺陷并不完全需要一个异常条件(比如整数溢出)。



# 负数索引

```
int *table = NULL;

int insert_in_table(int pos, int value){
    if (!table) {
        table = (int *)malloc(sizeof(int) * 100);
    }
    if (pos > 99) {
        return -1;
    }
    table[pos] = value;
    return 0;
}
```

从堆中分配存储空间给数组

**pos**小于 99

**value**被插入数组的指定位置



# 漏洞

---

- 对插入位置`pos`缺乏必要的范围检查，因此将会导致一个漏洞
  - 因为`pos`开始时被声明为带符号整数，即传递到函数中的值可正可负
  - 可以捕获下标越界的正值，但是负值却不会被捕获



## (6) 其他 C99 整数类型

---

- 下面的类型有特定的用处：
  - **ptrdiff\_t** 为表示两指针相减的结果的带符号整型
  - **size\_t** 是表示**sizeof**操作符结果的无符号整型。
  - **wchar\_t** 的取值范围可以表示所支持的现场（**locales**）中最大扩展字符集中的所有字符代码。



## 介绍案例

接受两个字符串类型的参数并且计算它们的总长度（加上结尾空字符占用的1个额外的字节）

```
int main(int argc, char *const *argv) {  
    unsigned short int total;  
    total = strlen(argv[1]) +  
            strlen(argv[2]) + 1;  
    char *buff = (char *) malloc(total);  
    strcpy(buff, argv[1]);  
    strcat(buff, argv[2]);  
}
```

分配足够的内存来存储两个字符

首先将第一个参数复制到缓冲区中，然后将第二个参数连接在其尾部



# 漏洞

---

攻击者可能会提供两个总长度无法用 `unsigned short` 整数 `total` 表示的字符串做参数

`strlen()` 函数返回一个 `size_t` 类型，一个 IA-32 上的 **`unsigned long int`**

- 有因此，`lengths+1` 的和是一个 **`unsigned long int`**.
  - 分配给 **`unsigned short int total`** 时，值必须被截断
- 一旦长度的总和大于结果类型的表示范围，它将会被截



# NetBSD例子1

---

- NetBSD 1.4.2及之前的版本中都使用了以下形式的整数范围检查：

```
if (off > len - sizeof(type-name))  
    goto error;
```

- 这里的off和len都是带符号整型

1. NetBSD 安全报告2000-002.



# 漏洞

---

- `sizeof`操作符返回的是一个无符号整型(`size_t`)
- 因此整数提升规则要求

`len - sizeof(类型名)`

应该按照无符号整型计算

- 当`len`小于`sizeof`的返回值时
  - 减法操作造成下溢并产生一个很大的正值
  - 整数范围检查逻辑被绕过





# 利用

---

- 一种能够消除此类问题的替代形式的整数范围检查如下所示：

```
■ if ((off + sizeof(type-name)) > len)  
    goto error;
```

- 程序员仍然必须保证**off**的值在一个定义的范围之内，以确保加法操作不会导致溢出

# 缓解策略:

## (1) 范围检查

- 如果适当地运用类型范围检查，就够消除所有的整型漏洞。诸如Pascal或者Ada这些语言。允许对任何标量类型应用范围限制，以形成子类型。

以Ada为例，允许使用range关键字来声明对派生类型的范围约束

```
type day is new INTEGER range 1..31;
```

- 范围约束会被语言运行时强制执行
- C和C++在强制类型安全性方面并不擅长



# 范围检查例子

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]) {
    unsigned int len;
    char buf[BUFF_SIZE];
    len = atoi(argv[1]);
    if ((0<len) && (len<BUFF_SIZE) ) {
        memcpy(buf, argv[2], len);
    }
    else
        printf("too much data\n");
}
```

隐式类型检查是将**len**声明为无符号整数而实现的

显式范围检查数组的上下界



# 范围检查的解释

---

仅仅将`len`声明为无符号整数并不足以完成范围约束，因为它只能约束从0到`MAX_INT`的范围。

检查上下界，确保不会将越界的值传递给  
`memcpy()`

同时使用隐式和显式检查看上去有些累赘，但是我们推荐这种“健康的偏执式的编程实践”



# 范围检查

---

- 所有外部输入的数据都要进行上下界检查
  - 应该通过接口来强制执行对它们的限制
  - 任何能够限制过大或过小输入的措施，都有助于防止溢出和其他类型范围错误
- 够限制过大或过小输入
- 排版约定
  - 区分代码中的常量和变量
  - 区分受外部影响的变量和拥有良好定义范围的局部变量



# 强类型

---

- 提供更好的类型检查的方式之一是提供更好的类型定义
- 将一个变量声明为无符号的类型就能够保证该变量不会包含负值
- 这种解决方案不能阻止溢出
- 使用强类型机制（**strong typing**），有助于编译器更有效地识别与范围相关的问题



# 强类型例子

---

- 要想定义一个整数存储水处于液态的华氏温度，可以如下方式声明一个变量：

**unsigned char waterTemperature;**

- 使用无符号8位数来描述waterTemperature是足够的，因此它的值的范围是1 ~ 255
  - 水温范围是从华氏32度（冰点）到华氏212度（沸点）。然而，使用这个类型既无法阻止溢出，也允许了无效数值（即1~31和213-255）的使用



# 抽象数据类型

---

- 解决方案之一是创建一个包含私有数据成员 `waterTemperature` 的抽象类型，用户不能直接访问该数据成员
- 这些方法中必须提供类型安全机制，以保证 `waterTemperature` 的值在有效范围之内
- 如果正确地做到了这一点，就不可能再发生整型范围错误了





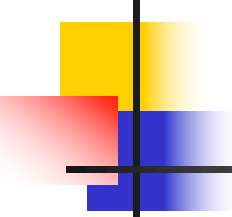
# Visual C++编译器检查

当一个整数值被赋给较小的整型时， Visual C++  
NET 2003编译器会生成一个警告（C4244）

- 在警告级别1，如果类型为\_int64的值被赋值给unsigned int类型的话将会生成一个警告
- 在警告级别3和4，如果一个整型被转换给一个较小的整型将会生成“可能会丢失数据”警告

在警告级别4下，以下例子中的赋值就会生成一个C4244警告

```
int main() {  
    int b = 0, c = 0;  
    short a = b + c;    // C4244  
}
```



# Visual C++ 运行时检查

---

- 当一个整数值被赋给较小的整型时， Visual C++ NET 2003 编译器会生成一个警告
- `/RTCc`提供了与C4244警告类似的功能，以报告当一个整数被赋值给较小的整型时所导致的数据丢失。
- Visual C++中还包含一个`runtime_checks pragma`， 用于禁用或启用 `/RTC`设置，但它并不包括用于捕获其他运行时错误（例如溢出）的标志。



# GCC运行时检查

---

- gcc和g++编译器都包含一个-ftrapv的编译选项，该选项对检测运行时整数异常提供了有限的支持
- 按照gcc用户手册的说明，这个选项“为加、减、集运算的带符号溢出产生陷阱”
- gcc编译器产生对已有库函数的调用



# 加带符号的整数

gcc运行时系统的一个函数，用于检测带符号**16**位整数加操作导致的溢出。

```
Wtype __addvsi3 (Wtype a, Wtype b) {  
    const Wtype w = a + b;  
    if (b >= 0 ? w < a : w > a)  
        abort ();  
    return w;  
}
```

加操作被执行，结果与操作数相比较以判断是否发生了溢出。

**abort()** 被调用，如果

- **b**非负且**w < a**
- **b**为负数并且**w > a**



## 安全的整数操作

---

- 整数操作可能会导致溢出和数据丢失
- 防止整数漏洞的第一条防线就是进行范围检查
  - 显式
  - 隐式-通过使用强类型达
- 很难保证多个输入变量不被恶意用户操纵，从而导致程序中的某些操作出错



# 安全的整数操作

---

- 另一种可选的或者说是辅助的方式是将每一个操作保护起来
- 这属于一种劳动密集型的方式，实现起来代价很大
- 让输入可能被不确定来源所影响的所有整数操作都使用一个安全整数库



# 安全的整数操作

---

- C 语言兼容库
  - 由Michael Howard 编写
  - 使用IA-32特定机制，侦测整数溢出条件



# 无符号的加函数

---

```
in bool UAdd(size_t a, size_t b, size_t *r) {  
    asm {  
        mov eax, dword ptr [a]  
        add eax, dword ptr [b]  
        mov ecx, dword ptr [r]  
        mov dword ptr [ecx], eax  
        jc  short j1  
        mov al, 1 // 1 is success  
        jmp short j2  
    j1:  
        xor al, al // 0 is failure  
    j2:  
};  
}
```





# 无符号的加函数例子

```
int main(int argc, char *const *argv) {
    unsigned int total;
    if (UAdd(strlen(argv[1]), 1, &total) &&
        Uadd(total, strlen(argv[2]), &total)) {
        char *buff = (char *)malloc(total);
        strcpy(buff, argv[1]);
        strcat(buff, argv[2]);
    } else {
        abort();
    }
}
```

调用函数UAdd（）来计算两个字符串长度的总和时，使用了适当的错误检查机制



# SafeInt类

---

**SafeInt**是一个由David LeBlanc编写的C++ 模板类。

在执行操作之前对操作数的值进行测试，以决定是否会导致错误

由于这个类被声明为模板类，因此可以用于任何整数类型

重截了几乎每一个有关的操作符（{}下标索引操作符除外）

变量s1和s2分别被声明为  
**SafeInt**类型

## SafeInt例子

```
int main(int argc, char *const *argv) {  
    try{  
        SafeInt<unsigned long> s1(strlen(argv[1]));  
        SafeInt<unsigned long> s2(strlen(argv[2]));  
        char *buff = (char *) malloc(s1 + s2 + 1);  
        strcpy(buff, argv[1]);  
        strcat(buff, argv[2]);  
    }  
    catch(SafeIntException err) {  
        abort();  
    }  
}
```

调用+操作符时，使用的是**SafeInt**类提供的  
安全版本的+操作符。这个安全版本的操作  
符保证，倘若结果无效则抛出一个异常



# 整数安全解决方案对比

---

与Howard方法相比， **SafeInt**库有好几个优点

- 比依赖于汇编语言指令实现务全算术操作的Howard方法移植性更好
- 更好的可用性
  - 算术操作符可以用于常规的内联表达式
  - **SafeInt**使用C++异常处理机制代替C风格的返回代码检查
- 更好的性能表现(对于启用优化编译选项的应用程序而言)



# 什么时候使用整数安全库

让输入可能被不确定来源所影响的所有整数操作都使用一个安全整数，比如

- 结构大小
- 分配的结构个数

函数有两个参数，一个指定了给定结构的大小，一个指定了由非可信来源所能操纵的、应分配的结构个数。这两个值的乘积决定了所分配的内存大小

```
void* CreateStructs(int StructSize, int HowMany) {  
    SafeInt<unsigned long> s(StructSize);  
  
    s *= HowMany;  
    return malloc(s.Value());  
}
```

乘法操作很容易引起整型变量的溢出，同时也为缓冲区溢出提供了机会



# 什么时候不使用整数安全库

---

- 不需要使用安全整数进行操作

- 被紧密控制的循环
- 变量不受外部影响

```
void foo() {  
    char a[INT_MAX];  
    int i;  
  
    for (i = 0; i < INT_MAX; i++)  
        a[i] = '\\0';  
}
```



# GNU多精度算术库(GMP)

---

- **GMP**是一个用**C**编写的可移植的库，用于对整数、有理数以及浮点数进行任意精度的算术运算
- 需要比**C**基本类型所直接支持的精度更高的精度的应用程序提供尽可能快的算术运算
- **GMP**对速度的强调高于简单性和优雅性
- 它使用了复杂的算法，全字（**full words**）作为基本的算术类型，以及精心优化的汇编代码



## (2) 测试

---

对输入的整数值进行检查仅仅是一个良好的开端，但这并不能保证对这些整数后继的操作中不会引起溢出或其他错误。

测试也不能提供任何保证

- 除非程序很简单，要覆盖所有可能的输入是不可能的
- 如果运用得当的话，测试能够增加代码安全的信心





# 测试

---

- 整数漏洞测试应该涵盖所有整型变量的边界条件。
  - 如果类型范围检查被内嵌到代码中，即可测试它们针对整盘上下界是否可以正确地发挥功能。
  - 如果没有包含边界测试，就检测每一个整型所能使用的最大值和最小值。
- 可以用自盒测试决定应该使用什么类型的变量
- 如果得不到源代码，可以用各种类型的最大值和最小值进行测试



# 源代码审计

---

- 必须对源代码进行审查，以发现可能存在的整数范围错误
- 以下是应该审查的项目：
  - 对整数类型范围进行彻底地检查
  - 根据输入值将被使用的情况检查它们是否被约束在有效范围之内
- 值不能为负的整数（例如作为索引、大小以及循环计数器的整数）应该被声明为无符号型并对上下界进行适当的范围检查
- 对所有来自不确定性来源的整数操作，都使用安全整数库完成



# 著名的漏洞

---

## XDR库中的整数溢出

- SunRPC xdr\_array缓冲区溢出
- [http://www.iss.net/security\\_center/static/9170.php](http://www.iss.net/security_center/static/9170.php)

## Windows DirectX MIDI 库

- eEye 数字安全公告AD20030723
- <http://www.eeye.com/html/Research/Advisories/AD20030723.html>

## Bash

- CERT Advisory CA-1996-22
- <http://www.cert.org/advisories/CA-1996-22.html>



# 著名漏洞- 1

---

## ■ XDR库中的整数溢出

- 由Sun发布的XDR (external data representation) 库中的xdr\_array ( ) 函数包含一个整型溢出
- 该漏洞已经导致多个应用程序出现可被远程利用的缓冲区溢出，从而可以执行任意的代码
- 很多产商都在自己的实现中包含了那些有漏洞的代码



## 著名漏洞- 2

- XDR库为从一个系统进程发送数据到另一系统进程（通常是通过网络连接发送的）提供了平台无关的方法
- 这样的函数通常被用在远程过程调用(RPC)的实现中，为需要使用公共接口与许多不同种类的系统进行交互的应用层程序员提供了透明性
- 由Sun提供的XDR库中的xdr\_array（）函数中包含有一个整型溢出，可能会造成大小错误的动态内存分配，从而可能会导教诸如缓冲区溢出等相应问题



# Windows DirectX MIDI 库- 1

---

- `quartz.dll`，中包含一个整数溢出漏洞，可以让攻击者执行任意的代码，或使任何使用了这个库的应用程序崩溃，从而导致拒绝服务
- Windows操作系统包含了名为DirectX 和DirectShow的多媒体技术
- DirectX由一组为Windows程序提供多媒体支持的底层应用编程接口(APIs)组成
- 在DirectX中，DirectShow技术执行客户端音频 / 视频的收集、操纵和呈现



# Windows DirectX MIDI 库- 2

---

- DirectShow对MIDI文件的支持由一个名为quartz. dll 的库实现。
- 由于该库没有充分验证MIDI文件的MThd区中的音轨（tracks）值的有效性，一个精心构造的MIDI文件就可以导致整数溢出，进而造成堆内存破坏
- 任何使用了DirectX或DirectShow来处理MIDI文件的应用都可能会受到此漏洞的影响
- 尤其值得注意的是，IE在处理HTML文档内嵌的MIDI文件时会载入这些有漏洞的库



# Windows DirectX MIDI 库 - 3

---

- 攻击者只需要说服受害人浏览一个嵌入了有害MIDI文件的HTML文档，就能对受害人实施攻击
- 大量的应用（例如， Outlook, Outlook Express, Eudora, AOL, Lotus Notes, Adobe PhotoDeluxe）都使用了IE的HTML呈现引擎(WebBrowser ActiveX control) 来解释HTML文档





# Bash – 1

---

- GNU项目 Bourne Again Shell (bash) 是 Unix Bourne Shell (/bin/sh ) 的一个替代品。
- 它和标准Shell拥有相同的语法，但提供了如作业控制、命令行编辑和历史记录等额外的功能
- 它最流行的使用是在Linux上
- bash 1.14.6以及更早的版本中存在一个漏洞，该漏洞会导致bash被欺骗以执行任意的命令



## Bash – 2

---

- bash源代码的parse.y模块中的yy\_string\_get()函数内有一个变量声明错误.
- 该函数负责将用户输入的命令行解析为单拙的记号(tokens)
- 这个错误源于一个名为string的变量，它被声明为char \*类型
- string变量用于遍历包含将被解析的命令行字符串



## Bash – 3

---

- 从该指针取回来的字符被存储到一个**int**型的变量中
- 对那些默认**char**为**signed char**的编译器，当将其值赋给**int**变量时，会进行符号扩展
- 对十进制代码为**255**的字符（补码形式的**-1**），符号扩展会导政**int**变量被赋值为**-1**.
- 解析器的其他部分又使用**- 1**作为一个命令的结束标志



## Bash – 4

---

- 十进制的**255**（八进制的**377**）就被作为一个通过 `-c` 选项提供给 **bash** 的非意料的命令分隔符

- 例如

- `bash -c 'ls\377who'`

- （其中 `\377` 表示一个具有十进制值**255**的单个字符）将执行两个命令：`ls`和`who`



# 小结

---

- 整数漏洞自数据丢失或者错误的表示所产生
- 当整数操作产生了一个超过其特定整型范围的值的时候，就会发生整数溢出
- 当一个值存储在一个太小的类型里，以至于无法表示其结果的时候，就会发生截断。
- 标志错误源于符号位的误解,但不会导致数据的丢失



# 小结

---

- 防止这些整数漏洞发生的关键在于理解数字系统中这些整数行为的细微差异
- 限制整数的输入使其处于一个有效的范围内，可以阻止那些可能引起整数类型溢出的非常大或非常小的数据进入系统
- 很多整数输入都定义有明确的范围，其他的整数则拥有一个合理的上下限



## 小结 3

---

- 确保对整数的操作不会造成整数错误需要周详的考虑
  - 一如既往地利用现有的工具、过程和技术来发现和防止整数漏洞是非常有意义的
  - 静态分析和代码审核对于发现错误非常有效
  - 源代码审核还为开发者提供了一种论坛，用来讨论什么会、什么不会造成安全缺陷，并考虑可能的解决方案

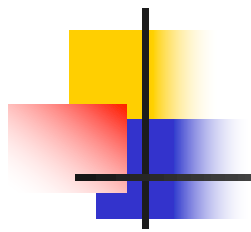


## 小结 4

---

- 动态分析工具与测试相结合使用，可以用作质量保证过程的一部分，尤其是当边界条件可被正确地评估出来的时候
- 如果正确地应用了整数类型范围检查，并且对某些可能超出范围的值（尤其是因为来自外部的操作）采用安全整数操作，完全有可能避免由整数范围错误所导致的漏洞





谢谢大家!