CIS 4360
Secure Computer Systems

Integer-related Attacks & Format String Attacks

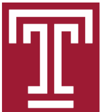
Professor Qiang Zeng

Spring 2017

Some slides are courtesy of Dr. Robert Seacord

Outline

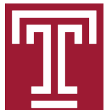
- Integer-related Attacks
- Format String Attacks



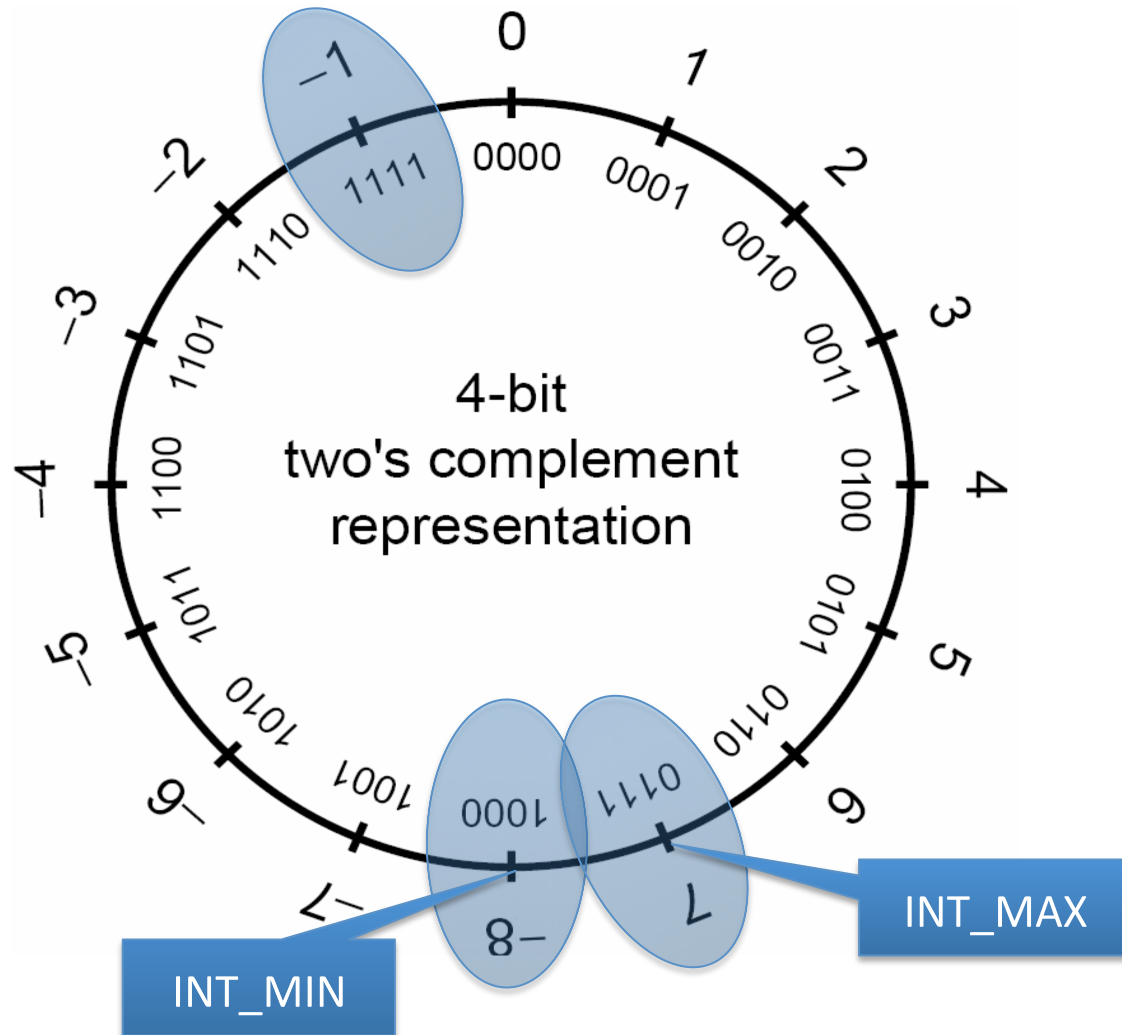
Integer Representation: Two's Complement

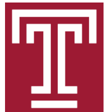
The two's complement form of a negative integer is created by adding one to the one's complement representation.

```
0 0 1 0   1 0 0 1           0 0 1 0   1 0 0 1
↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓           ↓ ↓ ↓ ↓   ↓ ↓ ↓ ↓
1 1 0 1   0 1 1 0 + 1 =     1 1 0 1   0 1 1 1
```



Signed Integer Representation





Unsigned Integer Representation

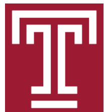
UINT_MAX

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

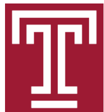
4-bit

two's complement
representation



Re-cap about Conversion for “a = b”

- A conversion to a type that can represent the value being converted is always well-defined
  - the value simply stays unchanged
  - E.g., signed -> larger signed; unsigned -> larger unsigned; unsigned short -> int
- A conversion to an unsigned type is always well-defined
  - y = x % (dst_type_max +1)
- A conversion to a signed type that cannot represent the value being converted results is implementation dependent
  - But in many systems: truncate the higher-order bits and re-interpret the remaining



Integer Conversion Rules for "a op b"

0. Integer Promotion always occur for char and short
1. If both operands have the same type and sign, no conversion is needed
2. If both operands have the same sign, the operand with the type of lesser rank is converted to the type of the operand with greater rank.
3. Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
4. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
5. Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

unsigned a; signed b;

Rank(a) >= Rank (b), apply rule 3

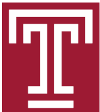
Rank(a) < Rank(b) and b's type has more bits, apply rule 4

Rank(a) < Rank(b) and b's type does not have more bits, apply rule 5

Integer Conversion Examples

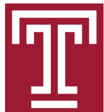
```
#include <iostream>

signed int s1 = -4;
unsigned int u1 = 2;

signed long int s2 = -4;
unsigned int u2 = 2;

signed long long int s3 = -4;
unsigned long int u3 = 2;

int main()
{
    std::cout << (s1 + u1) << "\n"; // 4294967294
    std::cout << (s2 + u2) << "\n"; // -2
    std::cout << (s3 + u3) << "\n"; // 18446744073709551614
}
```

- 64-bit system: int 32 bits, long 64, long long 64
- The three outputs apply the rules (3), (4), (5) respectively



Vulnerability due to Integer Overflows

- Even an operation as simple as a + b and a*b may lead to integer overflows, and may be exploitable
- E.g., malloc(element_size * count);

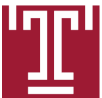


Vulnerability due to Integer Overflows

```
1.  bool func(char *name, long cbBuf) {
2.    unsigned short bufSize = cbBuf;
3.    char *buf = (char *)malloc(bufSize);
4.    if (buf) {
5.      memcpy(buf, name, cbBuf);
6.      return true;
7.    }
8.    return false;
9.  }
```

`cbBuf` is used to initialize **`bufSize`**, which is used to allocate memory for **`buf`**

`cbBuf` is declared as a **`long`** and used as the size in the **`memcpy()`** operation



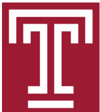
Mitigation

- Checking

```
unsigned int a, unsigned int b;
if(a <= UINT_MAX - b)
     malloc(a + b);

if(a <= UINT_MAX / b)
     malloc(a * b);
```

- Avoiding

```
int a, b; …; int m = (a + b) / 2;
// assume both a and b are positive
m = a + (b - a) / 2;
```

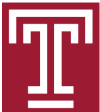


Mitigation

- Use GCC's built-in overflow detection functions
  - __builtin_[us](operation)(l?l?)_overflow

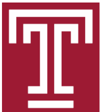
```
unsigned long a, b, c;
// mul means multiplication; l means long
if (__builtin_umull_overflow(a, b, &c)) {
   // returned non-zero: there has been an overflow
}
else
{
   // return zero: there hasn't been an overflow
}
```

- GCC 5+ and Clang 3.8+ additionally offer generic builtins that work without specifying the type of the values: **__builtin_add_overflow, __builtin_sub_overflow and __builtin_mul_overflow**



Outline

- Integer-related Attacks
- Format String Attacks

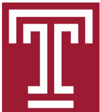


Vulnerable code example

- Is the following code vulnerable? Why?

```
void foo(char* user_provided) {
        printf(user_provided);
}
```

- Yes. E.g., user_provided can be “%08x %08x %08x %08x %08x”; this will leak information
- Why?



The format string can contain two types of data: printable characters and **format directives**, such as %d, %x, %u

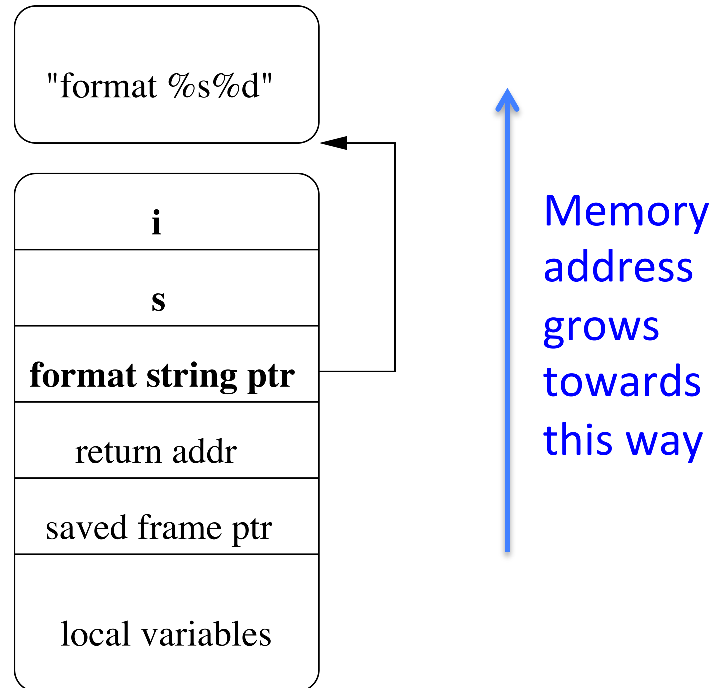
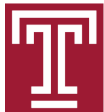


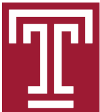
Figure 28. Activation record of *printf("format %s%d", s, i)* that has three parameters: the format string pointer, a string pointer, and an integer. A format function uses an internal stack pointer to access the parameters in the stack as it encounters the directives in the format string.

Buffer Overflow and Format String Overflow Vulnerabilities. Lhee and Chapin, 2002.



Width Specifier (specifies the minimum width)

- printf("%*d", 5, 10) will result in "   10" being printed, with a total width of 5 characters.
- printf("%*d", -5, 10) will result in "10   " being printed, with a total width of left-aligned 5 characters.
- Similarly,
- printf("%5d", 10) and printf("%-5d", 5, 10) can achieve the same effect
- printf("%05d", 10) prints "00010"
- printf("%#05x\n", 12); prints "0x00c" (# for 0x and the width of 0x00c is 10)

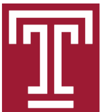


Attack one: viewing the memory

- Now you should be able to understand why the following code can leak information:

```
void foo(char* user_provided) {
        printf(user_provided);
}
```

- user_provided can be, e.g., “%08x %08x %08x %08x %08x” to leak information
- A special form of “buffer overflow”



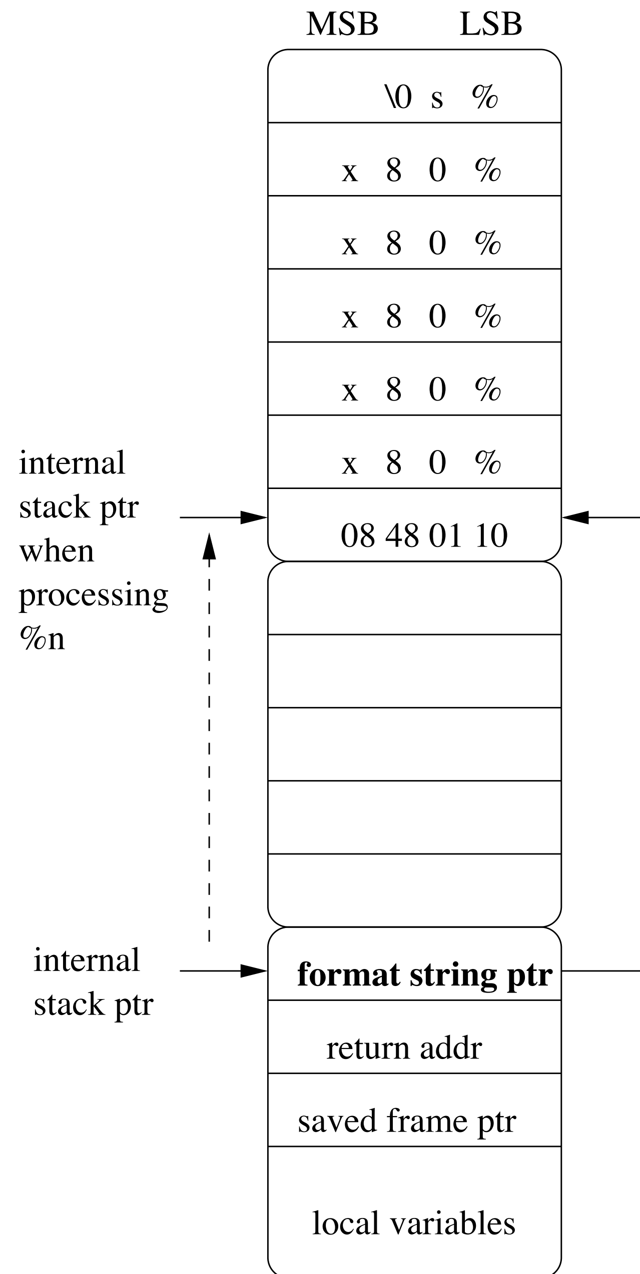
Attack two: Overwriting memory

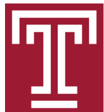
The %n directive writes an integer value, the number of characters that is written by the format function so far, at the location pointed by the the corresponding parameter. E.g.,

```
printf("\x10\x01\x48\x08%08x%08x%08x%08x%08x%n");
```

Writes 44 at 0x8480110 (four characters for "\x10\x01\x48\x08" and eight characters for each of the five "%08x")

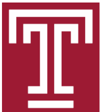
You can make use of the Width Specifier, e.g., %10000d, to increase the value you want to write





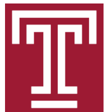
Attack three: Overflow a buffer

- char buf[100];
- // user-provided is “%104d\x80\x48\x55\66”
- sprintf(buf, user-provided)



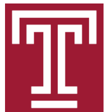
Suggestions

- Avoid using user-provided format string; try your best to use fixed predefined format string
- GCC can detect the mismatch (in terms of type and parameter number) between the directives and parameters when you use “-Wall or –Wformat”



Writing Assignments

- Does “a – b” concern you? How? What should you do?
- unsigned int x = -1; // What is x’s value? Why?
- short x = -1; int y = 0; // x > y? Why?
- short x = -1; unsigned int y = 0; // x > y? why?



References

- Secure Coding in C and C++. 2nd edition Seacord. 2013.
  - http://ptgmedia.pearsoncmg.com/images/0321335724/samplechapter/seacord_ch05.pdf
- C++ implicit Conversion
  - http://stackoverflow.com/a/19274544/577165
  - http://stackoverflow.com/a/17833338
- How to detect integer overflow in C/C++
  - http://stackoverflow.com/a/1514309
  - http://stackoverflow.com/a/20956705
- Printf format string
  - https://en.wikipedia.org/wiki/Printf_format_string
- Buffer Overflow and Format String Overflow Vulnerabilities. Lhee and Chapin, 2002.