COSC 458-647
Application Software Security

Lecture 10

Integer Overflow

Example: Simple string concatenate

```
int concatString( char *buf1, char *buf2,
                  size_t len1, size_t len2 ) {
   char buf[256]; // Fixed size buffer! Watch out
   if(len1 + len2 > 256) // Nice bound checking. Or not?
      return -1;
   memcpy(buf, buf1, len1);
   memcpy(buf + len1, buf2, len2);
   return 0;
}
```

Is this function safe?

Example: Simple string concatenate

- The previous function is not safe, because integer types have a finite precision.

```
(size_t)0xFFFFFFFF + (size_t)0x2 = 0x1
```

or, euqivalently,

```
(size_t)4294967295 + (size_t)2 = 1
```

- So the previous function could have an *integer overflow*, which in turn (in this example) can lead to a buffer overflow

Integer overflow

- Integer overflows occur when a larger integer is needed to accurately represent the results of an arithmetic operation.
- Let's look at a simple example using an unsigned char (8-bit) integer type:

```
unsigned char a = 255;
unsigned char b = 2;
unsigned char result = a + b;
```

- When compiled and run, we get the following binary representation:

```
     11111111      (a)
+    00000010      (b)
     ----------------------
     100000001     (result)
```

Integer Overflow

- Integer overflow: an arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space.
- Example:

```
Test 1:
short x = 30000;
short y = 30000;
printf("%d\n", x+y);
```

```
Test 2:
short x = 30000;
short y = 30000;
short z = x + y;
printf("%d\n", z);
```

Will two programs output the same? What will they output?

C Data Types

• **short int**	16 bits	[-32,768; 32,767]
• **unsigned short int**	16 bits	[0; 65,535]
• **unsigned int**	**32** bits	[0; 4,294,967,295]
• **int**	32 bits	[-2,147,483,648; 2,147,483,647]
• **long int**	32 bits	[-2,147,483,648; 2,147,483,647]
• **signed char**	8 bits	[-128; 127]
• **unsigned char**	8 bits	[0; 255]

Integer Overflow

- Ashcraft & Engler [IEEE S&P 2002]: “Many programmers appear to view integers as having arbitrary precision, rather than being fixed-sized quantities operated on with modulo arithmetic.”

```
bool DoSomething(const char* server)
{
   unsigned char namelen = strlen(server) + 2;
   if (namelen < 255) // Is this checking safe?
   {
      char* UncName = malloc(namelen);
      if(UncName != 0)
         sprintf(UncName, "\\\\%s", server);
      //do more things here
   }
}
```

The solution

- Solving integer overflow problems is not easy: they are very subtle

```
int ConcatString(char *buf1,char *buf2,size_t len1,size_t len2)
{
    char buf[256];
    if !((len1 < 256 - len2) && (len2 < 256 - len1))
      return -1;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
    return 0;
}
```

The solution

- For multiplication, in mathematical terms, the general problem is:

  A * B > MAX_INT → Error!

- In order to keep your test from depending on an overflow, we can rewrite it as follows:

  A > MAX_INT/B → Error!

- For addition, the unsigned case is relatively easy:

  A + B > MAX_INT → Error!

- In order to keep your test from depending on an overflow, we can rewrite it as follows:

  A > MAX_INT – B → Error!

The solution

- Testing a signed addition operation is considerably more difficult. We actually have four cases—both numbers are positive, both numbers are negative, and two mixed-sign cases:

```
• if ( !((rhs ^ lhs) < 0) ) //test for +/- combo
  {  //either two negatives, or 2 positives
     if (rhs < 0) {  //two negatives
        if (lhs < MinInt() - rhs) //remember rhs < 0
           throw ERROR_ARITHMETIC_OVERFLOW;
     } else {  //two positives
        if (MaxInt() - lhs < rhs)
           throw ERROR_ARITHMETIC_OVERFLOW;
     }
  }
  return lhs + rhs; //else overflow not possible
```

When casting occurs in C?

- When assigning to a different data types.

- For binary operators +, -, *, /, %, &, |, ^.
  - if either operand is an unsigned long, both are cast to an unsigned long.
  - in all other cases where both operands are 32-bits or less, the arguments are both upcast to int, and the result is an int.

- For unary operators
  - ~ changes type, e.g., ~((unsigned short)0) is int.
  - ++ and -- does not change type.

Where Does Integer Overflow Matter?

- Allocating spaces using calculation.
- Calculating indexes into arrays
- Checking whether an overflow could occur

- Direct causes:
  - Truncation; Integer casting

Integer Overflow Vulnerabilities Example

```
int main(int argc, char *argv[]) {
  unsigned short s;
  int i;
  char buf[80];

  if (argc < 3) {
      return -1;
  }

  i = atoi(argv[1]);

  s = i;

  if(s >= 80) {
      printf("No you don't!\n");
      return -1;
  }
  printf("s = %d\n", s);
  memcpy(buf, argv[2], i);
  buf[i] = '\0';
  printf("%s\n", buf);

  return 0;
}
```

C Data Types

• short int	16 bits	[-32,768; 32,767]
• unsigned short int	16 bits	[0; 65,535]
• unsigned int	16 bits	[0; 4,294,967,295]
• int	32 bits	[-2,147,483,648; 2,147,483,647]
• long int	32 bits	[-2,147,483,648; 2,147,483,647]
• signed char	8 bits	[-128; 127]
• unsigned char	8 bits	[0; 255]

Integer Overflow Vulnerabilities Example

- **Extra credit quiz**

```
const   long MAX_LEN = 20K;
char    buf[MAX_LEN];
short   len = strlen(input);

if (len < MAX_LEN) {
  strcpy(buf, input);
}
```

Can a buffer overflow attack occur? If so, how long does input needs to be?

Another Example

The function is supposed to return false when (x + y) overflows unsigned short. Does the function do it correctly?

```
bool  isValidAddition(unsigned short x,
      unsigned short y) {
      if (x + y < x)
            return false;
      return true;
}
```

Integer overflow in new[]

- This operator performs an implicit multiplication that is unchecked:

```
int* allocate_integers(int howmany)
{
    return new int[howmany];
}
```

- If you study the code generation for this, it comes out:

```
mov  eax, [esp+4] ; eax = howmany
shl  eax, 2       ; eax = howmany * sizeof(int)
push eax
call operator new ; allocate that many bytes
pop  ecx
retd 4
```

- The multiplication by sizeof(int) is not checked for overflow!!

Integer overflow in new[]

```
class MyClass {
public:
  MyClass(); // constructor
  int stuff[256];
};
```

```
MyClass* allocate_myclass(int howmany) {
  return new MyClass[howmany];
}
```

- This class also contains a constructor, so allocating an array of them involves two steps:
  1. Allocate the memory,
  2. Then construct each object.

Integer overflow in new[]

```
mov  eax, [esp+4] ; howmany
shl  eax, 10      ; howmany * sizeof(MyClass)
push esi
push eax
call operator new ; allocate that many bytes

...
push OFFSET MyClass::MyClass
push [esp+12]     ; howmany
push 1024         ; sizeof(MyClass)
push esi          ; memory block

call `vector constructor iterator`
mov  eax, esi
jmp  loop
fail:
  xor  eax, eax
done:
  pop  esi
  retd 4
```

- Unchecked multiplication
- Tries to allocate that many bytes
- Tells the vector constructor iterator to call the ctor (MyClass::MyClass) that many times.

If somebody calls allocate_myclass(0x200001), the multiplication overflows and only 1024 bytes are allocated.

This allocation succeeds, and then the vector ctor tries to initialize 0x200001 items

=> You walk off the end of the memory block and start a memory overflow

Integer overflow in new[]

Solution

```
template<typename T>
T* NewArray(size_t n)
{
  if (n <= (size_t)-1 / sizeof(T))
    return new T[n];

  // n is too large
  return NULL;
}
```

You can now use this template to allocate arrays in an overflow-safe manner.

Integer overflow in new[]

- But how could you get tricked into an overflow situation?

- The most common way of doing this is by reading the value out of a file or some other storage location. For example, if your code is parsing a file that has a section whose format is "length followed by data“

- See GIF, JPG, BMP vulnerabilities

Security Audit of Mozilla's .bmp image parsing

- In August 2004 by Gael Delalleau
- There are multiple integer overflows in the code used to parse and display .bmp images. Some of them leads to exploitable security bugs.
- These integer overflow can be found:
  - In the .bmp decoder of the libpr0n module
  - In the OS-specific code used to display images. For instance,
    - the following file is used on Windows systems:
    - mozilla/gfx/src/windows/nsImageWin.cpp

Denial Of Service while parsing a malformed .bmp image

- in `nsBMPDecoder.cpp` line 287:

```
mRow = new PRUint8[(mBIH.width*mBIH.bpp)/8 + 4];
```

- `mBIH.width*mBIH.bpp` can wrap around and thus the allocated buffer can be made very small.

- On Linux (Mozilla 1.7.2 developpement version) a negative value in the new[] operator will crash the client (uncatched exception?)

Denial Of Service while parsing a malformed .bmp image

In nsImageWin.cpp line 160:

```
  mRowBytes = CalcBytesSpan(mBHead->biWidth);
```

with :

```
1498  nsImageWin::CalcBytesSpan(PRUint32  aWidth)
1499  {
1500        PRInt32 spanBytes;
1503        spanBytes = (aWidth * mBHead->biBitCount) >> 5;
                  (...)
1510        spanBytes <<= 2;
1513        return(spanBytes);
1514  }
```

Denial Of Service while parsing a malformed .bmp image

- **`aWidth*mBHead->biBitCount`** is actually similar to **`mBIH.width*mBIH.bpp`**. It can wrap around, so `mRowBytes` can be very small.

- `mRowBytes` finds its way in `nsBMPDecoder.cpp` in the `mBpr` variable, which is used in line 374 to allocate a buffer (which can be very small due to the integer wrap around we just explained):

```
mDecoded = (PRUint8*)malloc(mBpr);
```

Denial Of Service while parsing a malformed .bmp image

- Thus at lines 420 to 428 we can end in a big loop (`lpos`==width iterations) which writes bytes to memory pointed by `p`, which is actually our small `mDecoded` buffer:

```
421  while (lpos > 0) {
422    SetPixel(d, p[2], p[1], p[0]);
423    p += 2;
424    --lpos;
425    if (mBIH.bpp == 32)
426      p++; // Padding byte
427      ++p;
428    }
```

- The `SetPixel` function writes the 3 bytes to d and increments d by 3.
- Thus, memory is being copied from the `p` buffer allocated in the heap, to the `mDecoded` buffer allocated later on the same heap. Both buffers can be made very small, so the loop will soon copy a part of the heap to another part of the heap.

Consequences

- The client will crash when this big loop reaches the end of the allocated memory for the heap (write attempt to an unmapped memory area).

- It might be possible to trigger arbitrary code execution if another thread uses the corrupted heap before the crash happens, or if we can allocate enough memory on the same heap to make it big enough to avoid the crash.

Integer overflow

- Integer overflows are becoming a new security attack vector.
- A solution for C++ is using templates: see “Integer Handling with the C++ SafeInt Class” - David LeBlanc
- The article discusses some of the ways you can protect yourself against integer overflow attacks.