Integer overflow in the new[] operator

devblogs.microsoft.com/oldnewthing/20040129-00

January 29, 2004

Raymond Chen

Integer overflows are becoming a new security attack vector. Mike Howard's article discusses some of the ways you can protect yourself against integer overflow attacks.

One attack vector he neglects to mention is integer overflow in the new[] operator. 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 to

```
  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
```

Notice that the multiplication by sizeof(int) is not checked for overflow. Somebody can trick you into under-allocating memory by passing a value like howmany = 0x40000001. For larger structures, multiplication overflow happens sooner.

Let's look at a slightly longer example:

```
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: allocate the memory, then construct each object. The allocate_myclass function compiles to this:

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

This function does an unchecked multiplication of the size, then tries to allocate that many bytes, then tells the vector constructor iterator to call the constructor (MyClass::MyClass) that many times.

If somebody tricks you into calling allocate_myclass(0x200001), the multiplication overflows and only 1024 bytes are allocated. This allocation succeeds, and then the vector constructor tries to initialize 0x200001 of those items, even though in reality only one of them got allocated. So you walk off the end of the memory block and start corrupting memory.

That's a bad thing.

To protect against this, you can wrap an integer overflow check around the array allocation.

```
template<typename T>
T* NewArray(size_t n)
{
  if (n <= (size_t)-1 / sizeof(T))
    return new T[n];
  // n is too large - act as if we
  // ran out of memory
  return NULL;
}
```

Note: If you use a throwing “new”, then replace the “return NULL” with an appropriate throw.

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

```
MyClass *allocate_myclass(int howmany)
{
  return NewArray<MyClass>(howmany);
}
```

This generates the following code:

```
  push edi
  mov  edi, [esp+8] ; howmany
  cmp  edi, 4194303 ; overflow?
  ja   overflow
  mov  eax, edi
  shl  eax, 10
  push esi
  push eax
  call operator new
  mov  esi, eax
  test esi, esi
  pop  ecx
  je   failed
  push OFFSET MyClass::MyClass
  push edi
  push 1024
  push esi
  call
  call `vector constructor iterator`
  mov  eax, esi
  jmp  done
failed:
  xor  eax, eax
done:
  pop  esi
  jmp  exit
overflow:
  xor     eax, eax
exit:
  pop  edi
  retd 4
```

Notice the new code that checks for a possible integer multiplication overflow.

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”, somebody could intentionally put an overflow-inducing value into the

“length” field, then get somebody else to try to load the file.

This is particularly dangerous if the filetype is something that is generally considered “not dangerous”, like a JPG.

Raymond Chen

Follow