Reviewing Code for Integer Manipulation Vulnerabilities

Michael Howard
Secure Windows Initiative

April 28, 2003

Summary: Michael Howard dishes on integer manipulation vulnerabilities and outlines a security plan that you can use to safeguard your own applications. (8 printed pages)

A couple of years ago, very few people had heard of integer overflow attacks, and now it seems there's a new one every few days. Below is a short list of some integer overflow security bugs found in the last few months:

- Sun RPC xdr_array
- OpenSSH authentication
- Apache Chunked Encoding
- Microsoft JScript
- FreeBSD socket and system calls
- Snort TCP Packet Reassembly

In this month's column, I'll outline how these bugs occur, how you can hunt for them in your code, and how you can fix them.

Oh, before I get into the core of this article, I'm happy to announce that Writing Secure Code received the RSA Conference Award for Industry Innovation at the RSA Security Conference held in San Francisco, April 2003. You can read more about it at http://www.rsaconference.net/rsa2003/pages/press_body_awards1.html.

Okay, back to integer attacks.

I'm not going to explain what an integer is, I assume you know what they are, and you know that there are basically two flavors—signed and unsigned—where signed integers have the high-bit set to 1 when the value is negative, this is a consequence of 2's complement arithmetic. And you also know there are different size integers, the most common being 64-bit, 32-bit, 16-bit, and 8-bit in length. That's all I'm going to say about integers, and for the purposes of this paper, that's all you need know.

There are three main integer manipulations that can lead to security vulnerabilities:

- Overflow and underflow
- Signed versus unsigned errors
- Truncation

On their own, these issues may not cause security errors. However, if your code exhibits one or more of these issues and manipulates memory, the potential for a buffer overrun error or application failure increases. Let's look at each in detail.

Overflow and Underflow

Quick, what's wrong with this code?
bool func(size_t cbSize) {
    if (cbSize < 1024) {
        // we never deal with a string trailing null
        char *buf = new char[cbSize-1];
        memset(buf,0,cbSize-1);
        // do stuff
        delete [] buf;
        return true;
    } else {
        return false;
    }
}

The code is correct, right? It validates that cbSize is no larger than 1 KB, and new or malloc should always allocate 1 KB correctly, right? Let's ignore the fact that the return value of new or malloc should be checked for the moment. Also, cbSize cannot be a negative number, because it's a size_t. But what if cbSize is zero? Look at the code that allocates the buffer—it subtracts one from the buffer size request. Subtracting one from zero causes a size_t variable, which is an unsigned integer, to wrap under to 0xFFFFFFFF (assuming a 32-bit value), or 4 GB. Your application just died—or worse!

Here's a similar issue:

bool func(char *s1, size_t len1, char *s2, size_t len2) {
    if (1 + len1 + len2 > 64)
        return false;
    // accommodate for the trailing null in the addition
    char *buf = (char*)malloc(len1+len2+1);
    if (buf) {
        StringCchCopy(buf,len1+len2,s1);
        StringCchCat(buf,len1+len2,s2);
    }
    // do other stuff with buf
    if (buf) free(buf);
    return true;
}

Once again, the code looks well written; it checks the data sizes, it verifies that malloc succeeded, and it uses the safe string handling functions, StringCchCopy and StringCchCat (you can read more about these string handling functions at http://msdn.microsoft.com/library/en-us/dnsecure/html/strsafe.asp). However, this code suffers from an integer overflow. What if len1 is 64, and len2 is 0xFFFFFFFF? The code that determines the buffer size is legal adds 1, 64 and 0xFFFFFFFF together, which yields 64, because the addition operation wraps around. Next, the code allocates only 64 bytes, and the code then builds a new string of 64 bytes in length, and then concatenates 0xFFFFFFFF bytes to the string. Once again, the application dies, and in some cases, some code when attacked with carefully crafted sizes can lead to exploitable buffer overrun attacks.

There's another lesson here — safe string handling functions are not safe if you get the buffer size calculation incorrect.

The JScript Overflow Attack

The same kind of overflow bug can occur when multiplying, which is what happened in the Microsoft JScript bug. The bug only manifests itself when using the JScript sparse array support:
var arr = new Array();
arr[1] = 1;
arr[2] = 2;
arr[0x40000001] = 3;

In this example, the array has three elements and a length of 0x40000001 (1073741825 decimal). But since this example uses sparse arrays, it only consumes the memory of a three-element array.

The C++ code that implements the JScript customized sort routine allocates a temporary buffer on the heap, copies the three elements into the temporary buffer, sorts the temporary buffer using the custom function, and then moves the contents of the temporary buffer back into the array. Here's the code that allocates the temporary buffer:

```
TemporaryBuffer = (Element *)malloc(ElementCount * sizeof(Element));
```

An `Element` is a 20-byte data structure used to hold an array entry. It looks like the program will attempt to allocate around 20 GB for the temporary buffer. You might think that since most people don't have 20 GB of memory in their machines the allocation attempt will fail. Then the JScript regular out-of-memory handling routines will handle the problem. Unfortunately, that is not what happens.

When using 32-bit integer arithmetic, we get an integer overflow attack because the result (0x0000000500000014) is too large to hold in a 32-bit value:

```
0x40000001  *  0x00000014  =   0x0000000500000014
```

C++ throws away all the bits that don't fit, so we get 0x00000014. This is why the allocation does not fail—the allocation does not attempt to allocate twenty billion bytes, but rather it attempts to allocate only twenty bytes. The sort routine then assumes that the buffer is large enough to hold the three elements in the sparse array, so it copies the 60 bytes that make up the three elements into a 20 byte buffer, overrunning the buffer by 40 bytes. Oops!

**Signed vs. Unsigned Errors**

Take a quick look at the following code. It's similar to the first example. See if you can spot the error, and if you do, try to determine what the result is.

```
bool func(char *s1, int len1,
          char *s2, int len2) {
    char buf[128];
    if (1 + len1 + len2 > 128)
        return false;
    if (buf) {
        strncpy(buf,s1,len1);
        strncat(buf,s2,len2);
    }
    return true;
}
```

The problem here is the string sizes are stored as signed integers, so `len1` can be larger than 128 as long as `len2` is negative, hence the sum is less than 128 bytes. However, a call to `strncpy` will overflow the `buf` buffer.

**Truncation Errors**

Now let's look at the last attack type, by way of, you guessed it, a code example.

```
bool func(byte *name, DWORD cbBuf) {
```
unsigned short cbCalculatedBufSize = cbBuf;
byte *buf = (byte*)malloc(cbCalculatedBufSize);
if (buf) {
    memcpy(buf, name, cbBuf);
    // do stuff with buf
    if (buf) free(buf);
    return true;
}
return false;

The attack, or at least the result, is a little like the JScript bug outlined earlier. What if cbBuf is 0x00010020? cbCalculatedBufSize is only 0x20 because only the lower 16-bits from 0x00010020 are copied. Hence only 0x20 bytes are allocated, and 0x00010020 bytes are copied into the newly allocated target buffer. Note that compiling this code with Microsoft Visual C++® /W4 option yields:

warning C4244: 'initializing' : conversion from 'DWORD' to 'unsigned short', possible loss of data

Be aware that operations like the following do not flag a warning:

int len = 16;
memcpy(buf, szData, len);

The last argument to memcpy is a size_t, yet the argument, len, is signed. The warning is not issued because memcpy always assumes the third argument is unsigned and putting a cast to unsigned would not change the function outcome.

Note you will get a warning if you attempt to assign a DWORD with a size_t, not because there is a potential loss of data on 32-bit platforms, but because there will be data loss on 64-bit platforms:

warning C4267: '=' : conversion from 'size_t' to 'DWORD', possible loss of data

You get this warning because all default C++ projects compile with the –Wp64 option, which tells the compiler to watch for 64-bit portability issues.

Integer Manipulation Issues in Managed Code

Integer manipulation errors can occur in managed languages, such as C# and Visual Basic® .NET, but the potential for damage is greatly reduced as your code does not have direct access to memory. However, calling into native code (assuming your code is granted the permission to call unmanaged code.) can still cause security issues like those noted above. Integers in the Common Language Specification (CLS) are signed, and one mistake is to validate signed integer arguments in managed code, when the variables are treated as unsigned integers in the unmanaged code.

This is a specific case of the more general advice of always check what you're passing to unmanaged code. Many integer manipulation bugs in managed code can cause reliability errors in Visual Basic .NET, because all such operations will raise the System.OverflowException if an overflow or underflow occurs.

By default, C# does not throw these exceptions. Use the checked keyword if you wish to check for these issues:

UInt32 i = 0xFFFFFFFF0;
UInt32 j = 0x00000100;
UInt32 k;
checked {k = i + j;}

Remedies
Who would have thought that simply manipulating integers could lead to security problems? A simple remedy to vulnerable code like this:

```c
if (A + B > MAX) return -1;
```

Is to use this code with unsigned integers:

```c
if (A + B >= A && A + B < MAX) {
    // cool!
}
```

The first operation `A + B >= A`, checks for the wrap-around, and the second makes sure the sum is less than the target size.

As for the multiplication issue in JScript, you can check that the number of elements is no larger than a predetermined value that is smaller than the largest amount of memory you are willing to allocate. For example, the following code could potentially allocate up to 64 MB:

```c
const size_t MAX = 1024 * 1024 * 64;
const size_t ELEM_SIZE = sizeof(ELEMENT);
const size_t MAX_ELEMS = MAX / ELEM_SIZE;
if (cElems >= MAX_ELEMS)
    return false;
```

Finally, use unsigned integers (such as DWORD and size_t) for array indexes, buffer sizes, and the like.

### Key Code Reviewing Points

Keep the following in mind when compiling or reviewing code for integer-related issues:

- Compile C and C++ code with the highest possible warning level, `/W4`.
- Use `size_t` or DWORD for buffer sizes and element counts. There is no reason to use a signed value for such constructs.
- Keep in mind that a `size_t` is a different type depending on the platform you use. A `size_t` is the size of a memory address, so it is a 32-bit value on a 32-bit platform, but a 64-bit value on a 64-bit platform.
- If your code performs any kind of integer manipulation (addition, multiplication, and so on) where the result is used to index into an array or calculate a buffer size, make sure the operands fall into a small and well-understood range.
- Be wary of signed arguments to memory allocation functions (`new`, `malloc`, `GlobalAlloc`, and so on) because they are treated as unsigned integers.
- Watch out for operations that yield C4018, C4389, and C4244 warnings.
- Watch out for casts that cast away the C4018, C4389, and C4244 warnings.
- Investigate all use of `#pragma warning(disable, Cnnnn)` that disable the C4018, C4389, and C4244 warnings. In fact, comment them out, re-compile, and check all new integer-related warnings.
- Code migrated from other platforms or compilers may make different assumptions about data sizes. Watch out!
- If calling unmanaged code from managed code, make sure you confirm that it is signed correctly. Many arguments to Win32 APIs are unsigned ints or DWORDs, and many managed code variables are signed.
- Finally, if you are using managed code, make sure you catch `OverflowExceptions`, if appropriate.

### Spot the Security Flaw

Many people worked out my last flaw. It was an integer overflow attack. So what's wrong with this C#
string Status = "No";
string sqlstring = "";
try {
    SqlConnection sql = new SqlConnection(
        "data source=localhost;" + 
        "user id=sa;password=password;");
    sql.Open();
    sqlstring="SELECT HasShipped" + 
        " FROM detail WHERE ID='" + Id + "'";
    SqlCommand cmd = new SqlCommand(sqlstring,sql);
    if ((int)cmd.ExecuteScalar() != 0)
        Status = "Yes";
    catch (SqlException se) {
        Status = sqlstring + " failed\n\r"
        foreach (SqlError e in se.Errors) {
            Status += e.Message + "\n\r"
        }
    } catch (Exception e) {
        Status = e.ToString();
    }

Michael Howard is a Senior Security Program Manager in the Secure Windows Initiative group at Microsoft and is the coauthor of Writing Secure Code, now in its second edition, and the main author of Designing Secure Web-based Applications for Windows 2000. His main focus in life is making sure people design, build, test, and document nothing short of a secure system. His favorite line is "One person's feature is another's exploit."