15
16
17
18
19
20
21
22
23

# 24 DEADLY SINS OF SOFTWARE SECURITY

## Programming Flaws and How to Fix Them

Covers

Windows <

UNIX <

Linux <

Mac OS X <

C <

C++ <

C# <

Java <

PHP <

Perl <

Python <

Ruby <

Visual Basic <

Web <

Smart-Client < Applications

Mobile < Applications

Michael Howard David LeBlanc John Viega

Foreword by Dan Kaminsky, Director of Penetration Testing, IOActive

# 24 DEADLY SINS OF SOFTWARE SECURITY

## Programming Flaws and How to Fix Them

Michael Howard, David LeBlanc, and John Viega

Mc Graw Hill

New York Chicago San Francisco Lisbon
London Madrid Mexico City Milan New Delhi
San Juan Seoul Singapore Sydney Toronto

Copyright © 2010 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-162676-7

MHID: 0-07-162676-X

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-162675-0, MHID: 0-07-162675-1

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please e-mail us at bulksales@mcgraw-hill.com.

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. ("McGraw-Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

# SIN 5

# BUFFER OVERRUNS

## OVERVIEW OF THE SIN

Buffer overruns have long been recognized as a problem in low-level languages. The core problem is that user data and program flow control information are intermingled for the sake of performance, and low-level languages allow direct access to application memory. C and C++ are the two most popular languages afflicted with buffer overruns.

Strictly speaking, a buffer overrun occurs when a program allows input to write beyond the end of the allocated buffer, but there are several associated problems that often have the same effect. One of the most interesting is format string bugs, which we cover in Sin 6. Another incarnation of the problem occurs when an attacker is allowed to write at an arbitrary memory location outside of an array in the application, and while, strictly speaking, this isn't a classic buffer overrun, we'll cover that here too.

A somewhat newer approach to gaining control of an application is by controlling pointers to C++ objects. Figuring out how to use mistakes in C++ programs to create exploits is considerably harder than just overrunning a stack or heap buffer—we'll cover that topic in Sin 8, "C++ Catastrophes."

The effect of a buffer overrun is anything from a crash to the attacker gaining complete control of the application, and if the application is running as a high-level user (root, administrator, or local system), then control of the entire operating system and any other users who are currently logged on, or will log on, is in the hands of the attacker. If the application in question is a network service, the result of the flaw could be a worm. The first well-known Internet worm exploited a buffer overrun in the finger server, and was known as the Robert T. Morris (or just Morris) finger worm. Although it would seem as if we'd have learned how to avoid buffer overruns since one nearly brought down the Internet in 1988, we continue to see frequent reports of buffer overruns in many types of software.

Now that we've gotten reasonably good at avoiding the classic errors that lead to a stack overrun of a fixed-size buffer, people have turned to exploiting heap overruns and the math involved in calculating allocation sizes—integer overflows are covered in Sin 7. The lengths that people go to in order to create exploits is sometimes amazing. In "Heap Feng Shui in JavaScript," Alexander Sotirov explains how a program's allocations can be manipulated in order to get something interesting next to a heap buffer that can be overrun.

Although one might think that only sloppy, careless programmers fall prey to buffer overruns, the problem is complex, many of the solutions are not simple, and anyone who has written enough C/C++ code has almost certainly made this mistake. The author of this chapter, who teaches other developers how to write more secure code, has shipped an off-by-one overflow to customers. Even very good, very careful programmers make mistakes, and the very best programmers, knowing how easy it is to slip up, put solid testing practices in place to catch errors.

## CWE REFERENCES

This sin is large enough to deserve an entire category:

CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer

There are a number of child entries that express many of the variants covered in this chapter:

- CWE-121: Stack-based Buffer Overflow
- CWE-122: Heap-based Buffer Overflow
- CWE-123: Write-what-where Condition
- CWE-124: Boundary Beginning Violation ('Buffer Underwrite')
- CWE-125: Out-of-bounds Read
- CWE-128: Wrap-around Error
- CWE-129: Unchecked Array Indexing
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-193: Off-by-one Error
- CWE-466: Return of Pointer Value Outside of Expected Range
- CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")

## AFFECTED LANGUAGES

C is the most common language used to create buffer overruns, closely followed by C++. It's easy to create buffer overruns when writing in assembler, given it has no safeguards at all. Although C++ is inherently as dangerous as C, because it is a superset of C, using the Standard Template Library (STL) with care can greatly reduce the potential to mishandle strings, and using vectors instead of static arrays can greatly reduce errors, and many of the errors end up as nonexploitable crashes. The increased strictness of the C++ compiler will help a programmer avoid some mistakes. Our advice is that even if you are writing pure C code, using the C++ compiler will result in cleaner code.

More recently invented higher-level languages abstract direct memory access away from the programmer, generally at a substantial performance cost. Languages such as Java, C#, and Visual Basic have native string types, provide bounds-checked arrays, and generally prohibit direct memory access. Although some would say that this makes buffer overruns impossible, it's more accurate to say that buffer overruns are much less likely.

In reality, most of these languages are implemented in C/C++, or pass user-supplied data directly into libraries written in C/C++, and implementation flaws can result in buffer overruns. Another potential source of buffer overruns in higher-level code exists because the code must ultimately interface with an operating system, and that operating system is almost certainly written in C/C++.

C# enables you to perform without a net by declaring unsafe sections; however, while it provides easier interoperability with the underlying operating system and libraries written in C/C++, you can make the same mistakes you can in C/C++. If you primarily program in higher-level languages, the main action item for you is to continue to validate data passed to external libraries, or you may act as the conduit to their flaws.

Although we're not going to provide an exhaustive list of affected languages, most older languages are vulnerable to buffer overruns.

## THE SIN EXPLAINED

The classic incarnation of a buffer overrun is known as "smashing the stack." In a compiled program, the stack is used to hold control information, such as arguments, where the application needs to return to once it is done with the function and because of the small number of registers available on x86 processors, quite often registers get stored temporarily on the stack. Unfortunately, variables that are locally allocated are also stored on the stack. These stack variables are sometimes inaccurately referred to as statically allocated, as opposed to being dynamically allocated heap memory. If you hear someone talking about a *static* buffer overrun, what they really mean is a *stack* buffer overrun. The root of the problem is that if the application writes beyond the bounds of an array allocated on the stack, the attacker gets to specify control information. And this is critical to success; the attacker wants to modify control data to values of his bidding.

One might ask why we continue to use such an obviously dangerous system. We had an opportunity to escape the problem, at least in part, with a migration to Intel's 64-bit Itanium chip, where return addresses are stored in a register. The problem is that we'd have to tolerate a significant backward compatibility loss, and the x64 chip has ended up the more popular chip.

You may also be asking why we just don't all migrate to code that performs strict array checking and disallows direct memory access. The problem is that for many types of applications, the performance characteristics of higher-level languages are not adequate. One middle ground is to use higher-level languages for the top-level interfaces that interact with dangerous things (like users!), and lower-level languages for the core code. Another solution is to fully use the capabilities of C++, and use string libraries and collection classes.

For example, the Internet Information Server (IIS) 6.0 web server switched entirely to a C++ string class for handling input, and one brave developer claimed he'd amputate his little finger if any buffer overruns were found in his code. As of this writing, the developer still has his finger, no security bulletins were issued against the web server in two years after its release, and it now has one of the best security records of any major web server. Modern compilers deal well with templatized classes, and it is possible to write very high-performance C++ code.

Enough theory—let's consider an example:

```
#include <stdio.h>
void DontDoThis(char* input)
{
      char buf[16];
      strcpy(buf, input);
      printf("%s\n", buf);
}
int main(int argc, char* argv[])
{
      // So we're not checking arguments
      // What do you expect from an app that uses strcpy?
      DontDoThis(argv[1]);
      return 0;
}
```

Now let's compile the application and take a look at what happens. For this demonstration, the author used a release build with debugging symbols enabled and stack checking disabled. A good compiler will also want to inline a function as small as DontDoThis, especially if it is only called once, so he also disabled optimizations. Here's what the stack looks like on his system immediately prior to calling strcpy:

```
0x0012FEC0  c8 fe 12 00  Èþ.. <- address of the buf argument
0x0012FEC4  c4 18 32 00  Ä.2. <- address of the input argument
0x0012FEC8  d0 fe 12 00  Ðþ.. <- start of buf
0x0012FECC  04 80 40 00  .□ @.
0x0012FED0  e7 02 3f 4f  ç.?O
0x0012FED4  66 00 00 00  f... <- end of buf
0x0012FED8  e4 fe 12 00  äþ.. <- contents of EBP register
0x0012FEDC  3f 10 40 00  ?.@. <- return address
0x0012FEE0  c4 18 32 00  Ä.2. <- address of argument to DontDoThis
0x0012FEE4  c0 ff 12 00  Àÿ..
0x0012FEE8  10 13 40 00  ..@. <- address main() will return to
```

Remember that all of the values on the stack are backward. This example is from a 32-bit Intel system, which is "little-endian." This means the least significant byte of a value comes first, so if you see a return address in memory as "3f104000," it's really address 0x0040103f.

Now let's look at what happens when buf is overwritten. The first control information on the stack is the contents of the Extended Base Pointer (EBP) register. EBP contains the frame pointer, and if an off-by-one overflow happens, EBP will be truncated. If the attacker can control the memory at 0x0012fe00 (the off-by-one zeros out the last byte), the program jumps to that location and executes attacker-supplied code.

If the overrun isn't constrained to one byte, the next item to go is the return address. If the attacker can control this value and is able to place enough assembly into a buffer that he knows the location of, you're looking at a classic exploitable buffer overrun. Note that the assembly code (often known as *shell code* because the most common exploit is to invoke a command shell) doesn't have to be placed into the buffer that's being overwritten. It's the classic case, but in general, the arbitrary code that the attacker has placed into your program could be located elsewhere. Don't take any comfort from thinking that the overrun is confined to a small area.

Once the return address has been overwritten, the attacker gets to play with the arguments of the exploitable function. If the program writes to any of these arguments before returning, it represents an opportunity for additional mayhem. This point becomes important when considering the effectiveness of stack tampering countermeasures such as Crispin Cowan's Stackguard, IBM's ProPolice, and Microsoft's /GS compiler flag.

As you can see, we've just given the attacker at least three ways to take control of our application, and this is only in a very simple function. If a C++ class with virtual functions is declared on the stack, then the virtual function pointer table will be available, and this can easily lead to exploits. If one of the arguments to the function happens to be a function pointer, which is quite common in any windowing system (for example, the X Window System or Microsoft Windows), then overwriting the function pointer prior to use is an obvious way to divert control of the application.

Many, many more clever ways to seize control of an application exist than our feeble brains can think of. There is an imbalance between our abilities as developers and the abilities and resources of the attacker. You're not allowed an infinite amount of time to write your application, but attackers may not have anything else to do with their copious spare time than figure out how to make your code do what they want. Your code may protect an asset that's valuable enough to justify months of effort to subvert your application. Attackers spend a great deal of time learning about the latest developments in causing mayhem, and they have resources like www.metasploit.com, where they can point and click their way to shell code that does nearly anything they want while operating within a constrained character set.

If you try to determine whether something is exploitable, it is highly likely that you will get it wrong. In most cases, it is only possible to prove that something is either exploitable or that you are not smart enough (or possibly have not spent enough time) to determine how to write an exploit. It is extremely rare to be able to prove with any confidence at all that an overrun is not exploitable. In fact, the guidance at Microsoft is that all writes to any address other than null (or null, plus a small, fixed increment) are must-fix issues, and most access violations on reading bad memory locations are also must-fix issues. See http://msdn.microsoft.com/en-us/magazine/cc163311.aspx by Damien Hasse for more details.

The point of this diatribe is that the smart thing to do is to just fix the bugs! There have been multiple times that "code quality improvements" have turned out to be security fixes in retrospect. This author just spent more than three hours arguing with a development team about whether they ought to fix a bug. The e-mail thread had a total of eight people on it, and we easily spent 20 hours (half a person-week) debating whether to fix the problem or not because the development team wanted proof that the code was exploitable. Once the security experts proved the bug was really a problem, the fix was estimated at one hour of developer time and a few hours of test time. That's an incredible waste of time.

The one time when you want to be analytical is immediately prior to shipping an application. If an application is in the final stages, you'd like to be able to make a good guess whether the problem is exploitable to justify the risk of regressions and destabilizing the product.

It's a common misconception that overruns in heap buffers are less exploitable than stack overruns, but this turns out not to be the case. Most heap implementations suffer from the same basic flaw as the stack—the user data and the control data are intermingled. Depending on the implementation of the memory allocator, it is often possible to get the heap manager to place four bytes of the attacker's choice into the location specified by the attacker.

The details of how to attack a heap are somewhat arcane. A recent and clearly written presentation on the topic, "Reliable Windows Heap Exploits," by Matthew "shok" Conover & Oded Horovitz, can be found at http://cansecwest.com/csw04/csw04-Oded+Connover.ppt. Even if the heap manager cannot be subverted to do an attacker's bidding, the data in the adjoining allocations may contain function pointers, or pointers that will be used to write information. At one time, exploiting heap overflows was considered exotic and hard, but heap overflows are now some of the more frequent types of exploited errors. Many of the more recent heap implementations now make many of the attacks against the heap infrastructure anywhere from extremely difficult to impractical due to improved checking and encoding of the allocation headers, but overwriting adjoining data will always be an issue, except with heaps specialized to trade off efficiency for reliability.

## 64-bit Implications

With the advent of commonly available x64 systems, you might be asking whether an x64 system might be more resilient against attacks than an x86 (32-bit) system. In some respects, it will be. There are two key differences that concern exploiting buffer overruns. The first is that whereas the x86 processor is limited to 8 general-purpose registers (eax, ebx, ecx, edx, ebp, esp, esi, edi), the x64 processor has 16 general-purpose registers.

Where this fact comes into play is that the standard calling convention for an x64 application is the fastcall calling convention—on x86, this means that the first argument to a function is put into a register instead of being pushed onto the stack. On x64, using fastcall means putting the first four arguments into registers. Having a lot more registers (though still far less than RISC chips, which typically have 32–64 registers, or ia64, which has 128) not only means that the code will run a lot faster in many cases, but that many values that were previously placed somewhere on the stack are now in registers where they're much more difficult to attack—if the contents of the register just never get written to the stack, which is now much more common, it can't be attacked at all with an arbitrary write to memory.

The second way that x64 is more difficult to attack is that the no-execute (NX) bit is always available, and most 64-bit operating systems enable this by default. This means that the attacker is limited to being able to launch return-into-libC attacks, or exploiting any pages marked write-execute present in the application. While having the NX bit always available is better than having it off, it can be subverted in some other interesting ways, depending on what the application is doing. This is actually a case where the higher-level languages make matters worse—if you can write the byte code, it isn't seen as executable at the C/C++ level, but it is certainly executable when processed by a higher-level language, such as C#, Java, or many others.

The bottom line is that the attackers will have to work a little harder to exploit x64 code, but it is by no means a panacea, and you still have to write solid code.

## Sinful C/C++

There are many, many ways to overrun a buffer in C/C++. Here's what caused the Morris finger worm:

```
char buf[20];
gets(buf);
```

There is absolutely no way to use gets to read input from stdin without risking an overflow of the buffer—use fgets instead. More recent worms have used slightly more subtle problems—the blaster worm was caused by code that was essentially strcpy, but using a string terminator other than null:

```
while (*pwszTemp != L'\\')
     *pwszServerName++ = *pwszTemp++;
```

Perhaps the second most popular way to overflow buffers is to use strcpy (see the previous example). This is another way to cause problems:

```
char buf[20];
char prefix[] = "http://";
```

```
strcpy(buf, prefix);
strncat(buf, path, sizeof(buf));
```

What went wrong? The problem here is that strncat has a poorly designed interface. The function wants the number of characters of available buffer, or space left, not the total size of the destination buffer. Here's another favorite way to cause overflows:

```
char buf[MAX_PATH];
sprintf(buf, "%s - %d\n", path, errno);
```

It's nearly impossible, except for in a few corner cases, to use sprintf safely. A critical security bulletin for Microsoft Windows was released because sprintf was used in a debug logging function. Refer to bulletin MS04-011 for more information (see the link in the section "Other Resources" in this chapter).

Here's another favorite:

```
char buf[32];
strncpy(buf, data, strlen(data));
```

So what's wrong with this? The last argument is the length of the incoming buffer, not the size of the destination buffer!

Another way to cause problems is by mistaking character count for byte count. If you're dealing with ASCII characters, the counts are the same, but if you're dealing with Unicode, there are two bytes to one character (assuming the Basic Multilingual Plane, which roughly maps to most of the modern scripts), and the worst case is multibyte characters, where there's not a good way to know the final byte count without converting first. Here's an example:

```
_snwprintf(wbuf, sizeof(wbuf), "%s\n", input);
```

The following overrun is a little more interesting:

```
bool CopyStructs(InputFile* pInFile, unsigned long count)
{
      unsigned long i;

      m_pStructs = new Structs[count];

      for(i = 0; i < count; i++)
      {
            if(!ReadFromFile(pInFile, &(m_pStructs[i])))
                  break;
      }
}
```

How can this fail? Consider that when you call the C++ new[] operator, it is similar to the following code:

```
ptr = malloc(sizeof(type) * count);
```

If the user supplies the count, it isn't hard to specify a value that overflows the multiplication operation internally. You'll then allocate a buffer much smaller than you need, and the attacker is able to write over your buffer. The C++ compiler in Microsoft Visual Studio 2005 and later contains an internal check to detect the integer overflow. The same problem can happen internally in many implementations of calloc, which performs the same operation. This is the crux of many integer overflow bugs: It's not the integer overflow that causes the security problem; it's the buffer overrun that follows swiftly that causes the headaches. But more about this in Sin 7.

Here's another way a buffer overrun can get created:

```
#define MAX_BUF 256
void BadCode(char* input)
{
      short len;
      char buf[MAX_BUF];

      len = strlen(input);

      //of course we can use strcpy safely
      if(len < MAX_BUF)
            strcpy(buf, input);
}
```

This looks as if it ought to work, right? The code is actually riddled with problems. We'll get into this in more detail when we discuss integer overflows in Sin 7, but first consider that literals are always of type signed int. The strlen function returns a size_t, which is an unsigned value that's either 32- or 64-bit, and truncation of a size_t to a short with an input longer than 32K will flip len to a negative number; it will get upcast to an int and maintain sign; and now it is always smaller than MAX_BUF, causing an overflow.

A second way you'll encounter problems is if the string is larger than 64K. Now you have a truncation error: len will be a small positive number. The main fix is to remember that size_t is defined in the language as the correct type to use for variables that represent sizes by the language specification. Another problem that's lurking is that input may not be null-terminated. Here's what better code looks like:

```
const size_t MAX_BUF = 256;
void LessBadCode(char* input)
{
        size_t len;
        char buf[MAX_BUF];

        len = strnlen(input, MAX_BUF);

        //of course we can use strcpy safely
        if(len < MAX_BUF)
              strcpy(buf, input);
}
```

## Related Sins

One closely related sin is integer overflows. If you do choose to mitigate buffer overruns by using counted string handling calls, or you are trying to determine how much room to allocate on the heap, the arithmetic becomes critical to the safety of the application. Integer overflows are covered in Sin 7.

Format string bugs can be used to accomplish the same effect as a buffer overrun, but they aren't truly overruns. A format string bug is normally accomplished without overrunning any buffers at all.

A variant on a buffer overrun is an unbounded write to an array. If the attacker can supply the index of your array, and you don't correctly validate whether it's within the correct bounds of the array, a targeted write to a memory location of the attacker's choosing will be performed. Not only can all of the same diversion of program flow happen, but also the attacker may not have to disrupt adjacent memory, which hampers any countermeasures you might have in place against buffer overruns.

## SPOTTING THE SIN PATTERN

Here are the components to look for:

- Input, whether read from the network, a file, or the command line
- Transfer of data from said input to internal structures
- Use of unsafe string handling calls
- Use of arithmetic to calculate an allocation size or remaining buffer size

## SPOTTING THE SIN DURING CODE REVIEW

Spotting this sin during code review ranges from being very easy to extremely difficult. The easy things to look for are usage of unsafe string handling functions. One issue to be aware of is that you can find many instances of safe usage, but it's been our experience that there are problems hiding among the correct calls. Converting code to use only safe calls has a very low regression rate (anywhere from 1/10th to 1/100th of the normal bug-fix regression rate), and it will remove exploits from your code.

One good way to do this is to let the compiler find dangerous function calls for you. If you undefined strcpy, strcat, sprintf, and similar functions, the compiler will find all of them for you. A problem to be aware of is that some apps have re-implemented all or a portion of the C run-time library internally, or perhaps they wanted a strcpy with some other terminator than null.

A more difficult task is looking for heap overruns. In order to do this well, you need to be aware of integer overflows, which we cover in Sin 3. Basically, you want to first look for allocations, and then examine the arithmetic used to calculate the buffer size.

The overall best approach is to trace user input from the entry points of your application through all the function calls. Being aware of what the attacker controls makes a big difference.

## TESTING TECHNIQUES TO FIND THE SIN

*Fuzz testing,* which subjects your application to semi-random inputs, is one of the better testing techniques to use. Try increasing the length of input strings while observing the behavior of the app. Something to look out for is that sometimes mismatches between input checking will result in relatively small windows of vulnerable code. For example, someone might put a check in one place that the input must be less than 260 characters, and then allocate a 256-byte buffer. If you test a very long input, it will simply be rejected, but if you hit the overflow exactly, you may find an exploit. Lengths that are multiples of two and multiples of two plus or minus one will often find problems.

Other tricks to try are looking for any place in the input where the length of something is user specified. Change the length so that it does not match the length of the string, and especially look for integer overflow possibilities—conditions where length + 1 = 0 are often dangerous.

Something that you should do when fuzz testing is to create a specialized test build. Debug builds often have asserts that change program flow and will keep you from hitting exploitable conditions. On the other hand, debug builds on modern compilers typically contain more advanced stack corruption detection. Depending on your heap and operating system, you can also enable more stringent heap corruption checking.

One change you may want to make in your code is that if an assert is checking user input, change the following from

```
assert(len < MAX_PATH);
```

to

```
if(len >= MAX_PATH)
{
      assert(false);
      return false;
}
```

You should always test your code under some form of memory error detection tool, such as AppVerifier on Windows (see link in the section "Other Resources") to catch small or subtle buffer overruns early.

Fuzz testing does not have to be fancy or complicated—see Michael Howard's SDL blog post "Improve Security with 'A Layer of Hurt'" at http://blogs.msdn.com/sdl/archive/2008/07/31/improve-security-with-a-layer-of-hurt.aspx. An interesting real-world story about how simple fuzzing can be comes from the testing that went into Office 2007. We'd been using some fairly sophisticated tools and were hitting the limits of what the tools could find. The author was speaking with a friend who had found some very interesting bugs, and inquired as to how he was doing it. The approach used was very simple: take the input and replace one byte at a time with every possible value of that byte. This approach obviously only works well for very small inputs, but if you reduce the number of values you try to a smaller number, it works quite well for even large files. We found quite a few bugs using this very simple approach.

## EXAMPLE SINS

The following entries, which come directly from the Common Vulnerabilities and Exposures list, or CVE (http://cve.mitre.org), are examples of buffer overruns. An interesting bit of trivia is that as of the first edition (February 2005), 1,734 CVE entries that match "buffer overrun" exist. We're not going to update the count, as it will be out of date by the time this book gets into your hands—let's just say that there are many thousands of these. A search of CERT advisories, which document only the more widespread and serious vulnerabilities, yields 107 hits on "buffer overrun."

### CVE-1999-0042

Buffer overflow in University of Washington's implementation of IMAP and POP servers.

#### Commentary

This CVE entry is thoroughly documented in CERT advisory CA-1997-09; it involved a buffer overrun in the authentication sequence of the University of Washington's Post Office Protocol (POP) and Internet Message Access Protocol (IMAP) servers. A related vulnerability was that the e-mail server failed to implement least privilege, and the exploit granted root access to attackers. The overflow led to widespread exploitation of vulnerable systems.

Network vulnerability checks designed to find vulnerable versions of this server found similar flaws in Seattle Labs SLMail 2.5 as reported at www.winnetmag.com/Article/ArticleID/9223/9223.html.

### CVE-2000-0389–CVE-2000-0392

Buffer overflow in krb_rd_req function in Kerberos 4 and 5 allows remote attackers to gain root privileges.

Buffer overflow in krb425_conv_principal function in Kerberos 5 allows remote attackers to gain root privileges.

Buffer overflow in krshd in Kerberos 5 allows remote attackers to gain root privileges.

Buffer overflow in ksu in Kerberos 5 allows local users to gain root privileges.

### Commentary

This series of problems in the MIT implementation of Kerberos is documented as CERT advisory CA-2000-06, found at www.cert.org/advisories/CA-2000-06.html. Although the source code had been available to the public for several years, and the problem stemmed from the use of dangerous string handling functions (strcat), it was only reported in 2000.

## CVE-2002-0842, CVE-2003-0095, CAN-2003-0096

Format string vulnerability in certain third-party modifications to mod_dav for logging bad gateway messages (e.g., Oracle9*i* Application Server 9.0.2) allows remote attackers to execute arbitrary code via a destination URI that forces a "502 Bad Gateway" response, which causes the format string specifiers to be returned from dav_lookup_uri() in mod_dav.c, which is then used in a call to ap_log_rerror().

Buffer overflow in ORACLE.EXE for Oracle Database Server 9*i*, 8*i*, 8.1.7, and 8.0.6 allows remote attackers to execute arbitrary code via a long username that is provided during login as exploitable through client applications that perform their own authentication, as demonstrated using LOADPSP.

Multiple buffer overflows in Oracle 9*i* Database Release 2, Release 1, 8*i*, 8.1.7, and 8.0.6 allow remote attackers to execute arbitrary code via (1) a long conversion string argument to the TO_TIMESTAMP_TZ function, (2) a long time zone argument to the TZ_OFFSET function, or (3) a long DIRECTORY parameter to the BFILENAME function.

### Commentary

These vulnerabilities are documented in CERT advisory CA-2003-05, located at www.cert.org/advisories/CA-2003-05.html. The problems are one set of several found by David Litchfield and his team at Next Generation Security Software Ltd. As an aside, this demonstrates that advertising one's application as "unbreakable" may not be the best thing to do whilst Mr. Litchfield is investigating your applications.

## CAN-2003-0352

Buffer overflow in a certain DCOM interface for RPC in Microsoft Windows NT 4.0, 2000, XP, and Server 2003 allows remote attackers to execute arbitrary code via a malformed message, as exploited by the Blaster/MSblast/LovSAN and Nachi/Welchia worms.

### Commentary

This overflow is interesting because it led to widespread exploitation by two very destructive worms that both caused significant disruption on the Internet. The overflow was in the heap and was evidenced by the fact that it was possible to build a worm that was very stable. A contributing factor was a failure of principle of least privilege: the interface should not have been available to anonymous users. Another interesting note is that overflow countermeasures in Windows 2003 degraded the attack from escalation of privilege to denial of service.

More information on this problem can be found at www.cert.org/advisories/CA-2003-23.html, and www.microsoft.com/technet/security/bulletin/MS03-039.asp.

## REDEMPTION STEPS

The road to buffer overrun redemption is long and filled with potholes. We discuss a wide variety of techniques that help you avoid buffer overruns, and a number of other techniques that reduce the damage buffer overruns can cause. Let's look at how you can improve your code.

### Replace Dangerous String Handling Functions

You should, at minimum, replace unsafe functions like strcpy, strcat, and sprintf with the counted versions of each of these functions. You have a number of choices of what to replace them with. Keep in mind that older counted functions have interface problems and ask you to do arithmetic in many cases to determine parameters.

As you'll see in Sin 7, computers aren't as good at math as you might hope. Newer libraries include strsafe, the Safe CRT (C run-time library) that shipped in Microsoft Visual Studio 2005 (and is on a fast track to become part of the ANSI C/C++ standard), and strlcat/strlcpy for *nix. You also need to take care with how each of these functions handles termination and truncation of strings. Some functions guarantee null termination, but most of the older counted functions do not. The Microsoft Office group's experience with replacing unsafe string handling functions for the Office 2003 release was that the regression rate (new bugs caused per fix) was extremely low, so don't let fear of regressions stop you.

### Audit Allocations

Another source of buffer overruns comes from arithmetic errors. Learn about integer overflows in Sin 7, and audit all your code where allocation sizes are calculated.

### Check Loops and Array Accesses

A third way that buffer overruns are caused is not properly checking termination in loops, and not properly checking array bounds prior to write access. This is one of the most difficult areas, and you will find that, in some cases, the problem and the earth-shattering kaboom are in completely different modules.

## Replace C String Buffers with C++ Strings

This is more effective than just replacing the usual C calls but can cause tremendous amounts of change in existing code, particularly if the code isn't already compiled as C++. You should also be aware of and understand the performance characteristics of the STL container classes. It is very possible to write high-performance STL code, but as in many other aspects of programming, a failure to Read The Fine Manual (RTFM) will often result in less than optimal results. The most common replacement is to use the STL std::string or std::wstring template classes.

## Replace Static Arrays with STL Containers

All of the problems already noted apply to STL containers like vector, but an additional problem is that not all implementations of the vector::iterator construct check for out-of-bounds access. This measure may help, and the author finds that using the STL makes it possible for him to write correct code more quickly, but be aware that this isn't a silver bullet.

## Use Analysis Tools

There are some good tools on the market that analyze C/C++ code for security defects; examples include Coverity, Fortify, PREfast, and Klocwork. As in many aspects of the security business, which tool is best can vary quite rapidly—research what is out there by the time you read this. There is a link to a list in the section "Other Resources" in this chapter. Visual Studio 2005 (and later) includes PREfast (used as /analyze) and another tool called Source Code Annotation Language (SAL) to help track down security defects such as buffer overruns. The best way to describe SAL is by way of code.

In the (silly) example that follows, you know the relationship between the data and count arguments: data is count bytes long. But the compiler doesn't know; it just sees a char * and a size_t.

```
void *DoStuff(char *data, size_t count) {
    static char buf[32];
    return memcpy(buf, data, count);
}
```

This code looks okay (ignoring the fact we loath returning static buffers, but humor us). However, if count is larger than 32, then you have a buffer overrun. A SAL-annotated version of this would catch the bug:

```
void *DoStuff(_In_bytecount_ (count) char *data, size_t count) {
    static char buf[32];
    return memcpy(buf, data, count);
}
```

This annotation, _In_bytecount_(N), means that *data is an "In" buffer that is only read from, and its byte count is the "count" parameter. This is because the analysis tool knows how the data and count are related.

The best source of information about SAL is the sal.h header file included with Visual C++.

## EXTRA DEFENSIVE MEASURES

Consider additional defensive measures the same way you think of seat belts or airbags in your car. Seat belts will often reduce the severity of a crash, but you still do not want to get into an accident. I can't think of anyone who believes that they've had a good day when they've needed their airbags! It's important to note that for every major class of buffer overrun mitigation, previously exploitable conditions that are no longer exploitable at all exist; and for any given mitigation technique, a sufficiently complex attack can overcome the technique completely. Let's look at a few of them.

### Stack Protection

Stack protection was pioneered by Crispin Cowan in his Stackguard product and was independently implemented by Microsoft as the /GS compiler switch. At its most basic, stack protection places a value known as a canary on the stack between the local variables and the return address. Newer implementations may also reorder variables for increased effectiveness. The advantage of this approach is that it is cheap, has minimal performance overhead, and has the additional benefit of making debugging stack corruption bugs easier. Another example is ProPolice, a Gnu Compiler Collection (GCC) extension created by IBM.

In Visual C++ 2008 and later, /GS is enabled by default from the command line and the IDE.

Any product currently in development should utilize stack protection.

You should be aware that stack protection can be overcome by a variety of techniques. If a virtual function pointer table is overwritten and the function is called prior to return from the function—virtual destructors are good candidates—then the exploit will occur before stack protection can come into play. That is why other defenses are so important, and we'll cover some of those right now.

### Nonexecutable Stack and Heap

This countermeasure offers considerable protection against an attacker, but it can have a significant application compatibility impact. Some applications legitimately compile and execute code on the fly, such as many applications written in Java and C#. It's also important to note that if the attacker can cause your application to fall prey to a return-into-libC attack, where a legitimate function call is made to accomplish nefarious ends, then the execute protection on the memory page may be removed.

Unfortunately, while most of the hardware currently available is able to support this option, support varies with CPU type, operating system, and operating system version as well. As a result, you cannot count on this protection being present in the field, but you must test with it enabled to ensure that your application is compatible with a nonexecutable stack and heap, by running your application on hardware that supports hardware protection, and with the target operating system set to use the protection. For example, if you are targeting Windows, then make sure you run all your tests on a Windows Vista or later computer using a modern processor. On Windows, this technology is called Data Execution Prevention (DEP); it is also known as No eXecute (NX.)

Windows Server 2003 SP1 also supports this capability. PaX for Linux and OpenBSD also support nonexecutable memory.

## OTHER RESOURCES

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, "Public Enemy #1: Buffer Overruns"
- "Heap Feng Shui in JavaScript" by Alexander Sotirov: http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html
- "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows Server 2003" by David Litchfield: www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
- "Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP" by David Litchfield: www.ngssoftware.com/papers/non-stack-bo-windows.pdf
- "Blind Exploitation of Stack Overflow Vulnerabilities" by Peter Winter-Smith: www.ngssoftware.com/papers/NISR.BlindExploitation.pdf
- "Creating Arbitrary Shellcode In Unicode Expanded Strings: The 'Venetian' Exploit" by Chris Anley: www.ngssoftware.com/papers/unicodebo.pdf
- "Smashing the Stack for Fun and Profit" by Aleph1 (Elias Levy): www.insecure.org/stf/smashstack.txt
- "The Tao of Windows Buffer Overflow" by Dildog: www.cultdeadcow.com/cDc_files/cDc-351/
- Microsoft Security Bulletin MS04-011/Security Update for Microsoft Windows (835732): www.microsoft.com/technet/security/Bulletin/MS04-011.mspx
- Microsoft Application Compatibility Analyzer: www.microsoft.com/windows/appcompatibility/analyzer.mspx
- Using the Strsafe.h Functions: http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp

- More Secure Buffer Function Calls: AUTOMATICALLY!: http://blogs.msdn.com/michael_howard/archive/2005/2/3.aspx
- Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries: http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx
- "strlcpy and strlcat—Consistent, Safe, String Copy and Concatenation" by Todd C. Miller and Theo de Raadt: www.usenix.org/events/usenix99/millert.html
- GCC extension for protecting applications from stack-smashing attacks: www.trl.ibm.com/projects/security/ssp/
- PaX: http://pax.grsecurity.net/
- OpenBSD Security: www.openbsd.org/security.html
- Static Source Code Analysis Tools for C: http://spinroot.com/static/

## SUMMARY

- **Do** carefully check your buffer accesses by using safe string and buffer handling functions.
- **Do** understand the implications of any custom buffer-copying code you have written.
- **Do** use compiler-based defenses such as /GS and ProPolice.
- **Do** use operating system–level buffer overrun defenses such as DEP and PaX.
- **Do** use address randomization where possible such as ASLR in Windows (/dynamicbase).
- **Do** understand what data the attacker controls, and manage that data safely in your code.
- **Do not** think that compiler and OS defenses are sufficient—they are not; they are simply extra defenses.
- **Do not** create new code that uses unsafe functions.
- **Consider** updating your C/C++ compiler, since the compiler authors add more defenses to the generated code.
- **Consider** removing unsafe functions from old code over time.
- **Consider** using C++ string and container classes rather than low-level C string functions.

# SIN 7
# INTEGER OVERFLOWS

## OVERVIEW OF THE SIN

Integer overflows, underflows, and arithmetic overflows of all types, especially floating point errors, have been a problem since the beginning of computer programming. Integer overflows have been a subject of security research once the easy stack-smashing attacks were largely replaced by heap exploits. While integer overflows have been involved in exploits for quite some time, in the last several years, they're frequently the root cause of many reported issues.

The core of the problem is that for nearly every binary format in which we can choose to represent numbers, there are operations where the result isn't what you'd get with pencil and paper. There are exceptions—some languages implement variable-size integer types, but these are not common and do come with some overhead.

Other languages, such as Ada, implement a range-checked integer type, and if these types are consistently used, they reduce the chances of problems. Here's an example:

```
type Age is new Integer range 0..200;
```

The nuances of the problem vary from one language to another. C and C++ have true integer types; and modern incarnations of Visual Basic pack all the numbers into a floating point type known as a "Variant," so you can declare an int, divide 5 by 4, and expect to get 1. Instead, you get 1.25. Perl displays its own distinctive behavior; C# makes the problem worse by generally insisting on signed integers, but then turns around and makes it better by creating a "checked" keyword (more on this in the section "Sinful C#"). Java is even less helpful because of its insistence on signed integers and lack of template support—it would be possible, but hard, to make classes to contain each of the int types that implemented checking.

## CWE REFERENCES

The following CWE references discuss this topic. CWE-682 is the parent entry for this class of error.

- CWE-682: Incorrect Calculation
- CWE-190: Integer Overflow or Wraparound
- CWE-191: Integer Underflow (Wrap or Wraparound)
- CWE-192: Integer Coercion Error

## AFFECTED LANGUAGES

All common languages are affected, but the effects differ, depending on how the language handles integers internally. C and C++ are arguably the most dangerous and are likely to turn an integer overflow into a buffer overrun and arbitrary code execution; but all languages are prone to denial of service and logic errors.

## THE SIN EXPLAINED

The effects of integer errors range from crashes and logic errors to escalation of privilege and execution of arbitrary code. A current incarnation of the attack hinges on causing an application to make errors determining allocation sizes; the attacker is then able to exploit a heap overflow. The error can be anything from an underallocation to allocating zero bytes. If you typically develop in a language other than C/C++, you may think you're immune to integer overflows, but this would be a mistake. Logic errors related to the truncation of integers resulted in a bug several years ago in Network File System (NFS) where any user can access files as root. Problems with integers have also caused problems as serious as catastrophic failures in spacecraft.

## Sinful C and C++

Even if you're not a C or C++ programmer, it's worthwhile to look at the dirty tricks that C/C++ can play on you. Being a relatively low-level language, C sacrifices safety for execution speed and has the full range of integer tricks up its sleeve. Because of the low-level capabilities of C/C++, integer problems that show up when using these languages illustrate the issues that the processor encounters.

Most other languages won't be able to do all of the same things to your application, and some, like C#, can do unsafe things if you tell them to. If you understand what C/C++ can do with integers, you'll have a better shot at knowing when you're about to do something wrong, or even why that Visual Basic .NET application keeps throwing those pesky exceptions. Even if you only program in a high-level language, you'll eventually need to make system calls, or access external objects written in C or C++. The errors you made in your code can show up as overflows in the code you call.

### Casting Operations

There are a few programming patterns and issues that most frequently lead to integer overflows. One of the first is a lack of awareness of casting order and implicit casts from operators. For example, consider this code snippet:

```
const long MAX_LEN = 0x7fff;

short len = strlen(input);

if(len < MAX_LEN)
      //do something
```

Aside from truncation errors, what's the order of the cast that happens when len and MAX_LEN are compared? The language standard states that you have to promote to like types before a comparison can occur; so what you're really doing is upcasting len from a signed 16-bit integer to a signed 32-bit integer. This is a straightforward cast because both types are signed. In order to maintain the value of the number, the type value is sign-extended until it is the same size as the larger type. In this case, you might have this as a result:

```
len = 0x0100;
(long)len = 0x00000100;
```

or

```
len = 0xffff;
(long)len = 0xffffffff;
```

As a result, if the attacker can cause the value of len to exceed 32K, len becomes negative, because once it's upcast to a 32-bit long it's still negative, and your sanity check to see if len is larger than MAX_LEN sends you down the wrong code path.

Understanding how integers get converted is half of the solution. In the following cases, we'll use the word "int" to mean integer type, not the 32-bit signed integer you might commonly think of. Here are the conversion rules for C and C++:

**Signed int to Larger signed int** The smaller value is sign-extended; for example, (char)0x7f cast to an int becomes 0x0000007f, but (char)0x80 becomes 0xffffff80.

**Signed int to Same-Size unsigned int** The bit pattern is preserved, though the value will change if the input is negative. So (char)0xff (–1) remains 0xff when cast to an unsigned char, but –1 clearly has a different meaning than 255. Casts between signed and unsigned integers are always danger signs to watch out for.

**Signed int to Larger unsigned int** This combines the two behaviors: The value is first sign-extended to a larger signed integer and then cast to preserve the bit pattern. This means that positive numbers behave as you'd expect, but negative numbers might yield unexpected results. For example, (char)–1 (0xff) becomes 4,294,967,295 (0xffffffff) when cast to an unsigned long.

**Unsigned int to Larger unsigned int** This is the best case: the new number is zero-extended, which is generally what you expect. Thus (unsigned char)0xff becomes 0x000000ff when cast to an unsigned long.

**Unsigned int to Same-Size signed int** As with the cast from signed to unsigned, the bit pattern is preserved, and the meaning of the value may change, depending on whether the uppermost (sign) bit is a 1 or 0.

**Unsigned int to Larger signed int** This behaves very much the same as casting from an unsigned int to a larger unsigned int. The value first zero-extends to an unsigned int the same size as the larger value and then is cast to the signed type. The value of the number is maintained and won't usually cause programmer astonishment.

The last phrase is a reference to *The Tao of Programming,* which asserts that user astonishment is always a bad thing. Programmer astonishment is perhaps worse.

**Downcast** Assuming that any of the upper bits are set in the original number, you now have a truncation, which can result in general mayhem. Unsigned values can become negative or data loss can occur. Unless you're working with bitmasks, always check for truncation.

## Operator Conversions

Most programmers aren't aware that just invoking an operator changes the type of the result. Usually, the change will have little effect on the end result, but the corner cases may surprise you. Here's some C++ code that explores the problem:

```
template <typename T>
void WhatIsIt(T value)
{
      if((T)-1 < 0)
            printf("Signed");
      else
            printf("Unsigned");

      printf(" - %d bits\n", sizeof(T)*8);
}
```

For simplicity, we'll leave out the case of mixed floating point and integer operations. Here are the rules:

- If either operand is an unsigned long, both are upcast to an unsigned long. Academically, longs and ints are two different types, but on a modern compiler, they're both 32 or 64-bit values; for brevity, we'll treat them as equivalent.
- 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.
- If one of the operands is 64-bit, the other operand is also upcast to 64-bit, with an unsigned 64-bit value being the upper bound.

Most of the time, this results in the right thing happening, and implicit operator casting can actually avoid some integer overflows. There are some unexpected consequences, however. The first is that on systems where 64-bit integers are a valid type, you might expect that because an unsigned short and a signed short get upcast to an int, and the correctness of the result is preserved because of the operator cast (at least unless you downcast the result back to 16 bits), an unsigned int and a signed int might get cast up to a 64-bit int (_int64). If you think it works that way, you're unfortunately wrong—at least until the C/C++ standard gets changed to treat 64-bit integers consistently.

The second unexpected consequence is that the behavior also varies depending on the operator. The arithmetic operators (+, -, *, /, and %) all obey the preceding rules as you'd expect. What you may not expect is that the binary operators (&, |, ^) also obey the same rules; so, (unsigned short) | (unsigned short) yields an int! The Boolean operators (&&, ||, and !) obey the preceding rules in C programs but return the native type bool in C++.

To further add to your confusion, some of the unary operators tamper with the type, but others do not. The one's complement (~) operator changes the type of the result (the same way the other binary operators behave); so ~((unsigned short)0) yields an int, but the pre- and postfix increment and decrement operators (++, – –) do not change the type. An even more unexpected operator cast comes from the unary – (negation) operator. This will cast values smaller than 32-bit to an int and, when applied to a 32- or 64-bit unsigned int, will result in the same bitwise operation, but the result is still unsigned—the result would be very unlikely to make any sense.

As an illustration, a senior developer with many years of experience proposed using the following code to check whether two unsigned 16-bit numbers would overflow when added together:

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

      return true;
}
```

It looks like it ought to work. If you add two positive numbers together and the result is smaller than either of the inputs, you certainly have a malfunction. The exact same code does work if the numbers are unsigned longs. Unfortunately for our senior developer, it will never work, because the compiler will optimize out the entire function to true!

Recalling the preceding behavior, what's the type of unsigned short + unsigned short? It's an int. No matter what we put into two unsigned shorts, the result can never overflow an int, and the addition is always valid. Next, you need to compare an int with an unsigned short. The value x is then cast to an int, which is never larger than x + y. To correct the code, all you need to do is cast the result back to an unsigned short, like so:

```
if((unsigned short)(x + y) < x)
```

The same code was shown to a blackhat who specializes in finding integer overflows, and he missed the problem as well, so the experienced developer has plenty of company! Something to remember here is that if a very experienced programmer can make this type of mistake, the rest of us are on very thin ice!

## Arithmetic Operations

Be sure to understand the implications of casts and operator casts when thinking about whether a line of code is correct—an overflow condition could depend on implicit casts. In general, you have four major cases to consider: unsigned and signed operations involving the same types, and mixed-type operations that could also be mixed sign. The simplest of all is unsigned operations of the same type; signed operations have more complexity, and when you're dealing with mixed types, you have to consider casting behavior. We'll cover example defects and remedies for each type of operation in later sections.

**Addition and Subtraction** The obvious problem with these two operators is wrapping around the top and bottom of the size you declared. For example, if you're dealing with unsigned 8-bit integers, 255 + 1 = 0. Or 2 – 3 = 255. In the signed 8-bit case, 127 + 1 = –128. A less obvious problem happens when you use signed numbers to represent sizes. Now someone feeds you a size of –20, you add that to 50, come up with 30, allocate 30 bytes, and then proceed to copy 50 bytes into the buffer. You're now hacked. Something to remember, especially when dealing with languages where integer overflows are anywhere from difficult to impossible, is that subtracting from a positive and getting less than you started with is a valid operation; it won't throw an overflow exception, but you may not have the program flow you expect. Unless you've previously range-checked your inputs and are certain that the operation won't overflow, be sure to validate every operation.

**Multiplication, Division, and Modulus** Unsigned multiplication is fairly straightforward: any operation where a * b > MAX_INT results in an incorrect answer. A correct but less efficient way to check the operation is to convert your test to b > MAX_INT/a. A more efficient way to check the operation is to store the result in the next larger integer where available, and then see if there was an overflow. For small integers, the compiler will do that for you. Remember that short * short yields an int. Signed multiplication requires one extra check to see if the answer wrapped in the negative range.

You may be wondering how division, other than dividing by zero, can be a problem. Consider a signed 8-bit integer: MIN_INT = –128. Now divide that by –1. That's the same thing as writing –(–128). The negation operator can be rewritten as ~x + 1. The one's complement of –128 (0x80) is 127, or 0x7f. Now add 1, and you get 0x80! So you see that minus negative 128 is still minus 128! The same is true of any minimum signed integer divided by –1. If you're not convinced that unsigned numbers are easier to validate yet, we hope this convinces you.

The modulus (remainder) operator returns the remainder of a division operation; thus, the answer can never have a larger magnitude than the numerator. You may be wondering how this can overflow. It can't actually overflow, but it can return an incorrect answer, and this is due to casting behavior. Consider an unsigned 32-bit integer that is equal to MAX_INT, or 0xffffffff, and a signed 8-bit integer that has a value of –1. So –1 mod 4,294,967,295 ought to yield 1, right? Not so fast. The compiler wants to operate on like numbers, so the –1 has to be cast to an unsigned int. Recall from earlier how that happens. First you sign-extend until you get to 32 bits, so you'll convert 0xff to 0xffffffff. It then converts (int)(0xfffffff) to (unsigned int)(0xffffffff). You see that the remainder of –1 divided by 4 billion is zero, or at least according to our computer! The same problem will occur any time you're dealing with unsigned 32- or 64-bit integers mixed with negative signed integers, and it applies to division as well—1/4,294,967,295 is really 1, which is annoying when you've expected to get zero. An additional problem with modulus is that the sign of the return can be implementation-dependent.

### Comparison Operations

Surely something as basic as equality ought to work, or one would hope. Unfortunately, if you're dealing with mixed signed and unsigned integers, there's no such guarantee—at least if the signed value isn't a larger type than the unsigned value. The same problem we outlined with division and modulus will cause problems.

Another way that comparison operations will get you is when you check for a maximum size using a signed value: your attacker finds some way to cause the value to be negative, and that's always less than the upper limit you expected. Either use unsigned numbers, which is what we recommend, or be prepared to make two checks: first that the number is greater than or equal to zero, and second that it is smaller than your limit.

### Binary Operations

Binary operations, like binary AND, OR, and XOR (exclusive or), ought to work, but again, sign extension will mix things up. Let's look at an example:

```
int flags = 0x7f;
char LowByte = 0x80;

if((char)flags ^ LowByte == 0xff)
    return ItWorked;
```

You might think that the result of this operation ought to be 0xff, which is what you're checking for, but then the pesky compiler gets ambitious and casts both values to an int. Recall from our operator conversions that even binary operations convert to int when given smaller values—so flags gets extended to 0x0000007f, which is just fine, but LowByte gets extended to 0xffffff80, and our result is really 0xfffffffff, which isn't equal to 0x000000ff!

### 64-bit Portability Issues

There are four standard integer types that can change size, depending on whether the build is 32- or 64-bit. Each of the following are guaranteed to always have the same size as a pointer – sizeof(x) == sizeof(void*):

- size_t
- ptrdiff_t
- uint_ptr
- int_ptr

The size_t type is unsigned, and ptrdiff_t is signed. Both of these are interesting because size_t is used to return (you guessed it) size values for the C run-time library (CRT), and ptrdiff_t is the type that results when you take the difference of two pointers. If you are using any of these four types, or are doing pointer math, these are something to pay attention to.

The first typical problem is that you might see something along the lines of

```
int cch = strlen(str);
```

Sometimes you see this because the code is old—strlen did return an int several years back. In other cases, you see this construct because the developer isn't thinking far enough ahead. On a 32-bit system, the justification is that you simply can't allocate more than 2GB on most operating systems, and certainly not in one chunk. If you're never going to port the code to a 64-bit system, that might be okay. On a 64-bit system, it might be completely possible to allocate 2GB at once—as of this writing (September 2008), systems that can handle up to 16GB are common, and an exploit has already been seen that works with inputs larger than 2GB on 64-bit BSD systems (see http://www.securityfocus.com/bid/13536/info).

The second problem is a lot more subtle. Consider this code (where x is a reasonably small number):

```
unsigned long increment = x;
if( pEnd - pCurrent < increment )
  pCurrent += increment;
else
  throw;
```

If there is an error, and pEnd – pCurrent becomes negative, do we throw, or do we increment the pointer? Don't feel bad if you miss this—a lot of very good devs have given the wrong answer. A useful technique for dealing with this type of problem is to put the casts in place for the types involved, like so:

```
if( (ptrdiff_t)(pEnd – pCurrent) < (unsigned long)increment )
```

On a 32-bit system, a ptrdiff_t would be a signed 32-bit int. When comparing a signed 32-bit int to an unsigned long, the signed value is cast to unsigned. If the signed value is negative, then when it is cast to unsigned, the value will be very large, and the resulting code will behave as if you'd written this:

```
if( pEnd - pCurrent < increment && pEnd – pCurrent >= 0)
```

The code will also throw a signed-unsigned comparison warning—be careful with these!

On a 64-bit system, a ptrdiff_t is a signed 64-bit integer, and the casts look like this:

```
if( (__int64)(pEnd – pCurrent) < (unsigned long)increment )
```

Now the comparison upcasts increment to __int64, no signed-unsigned warning will be thrown because the cast from unsigned 32-bit to signed 64-bit preserves value, and you'll now take the code path that increments the pointer! The answer is that either code path could be taken, depending on the build environment.

Take care when dealing with types that change size, and especially take care with pointer math, which emits a type that changes size. You also need to be aware of compiler differences—the size of the native types are not guaranteed—the only thing you can be sure of is that a char consumes a byte.

### Sinful Compiler Optimizations

The C/C++ standard decrees that the result of pointer arithmetic that causes an integer overflow is *undefined.* This means that anything can happen—a compiler could choose to throw exceptions—really anything. Technically, any operation beyond the current buffer (plus 1) is undefined. We can check for an unsigned addition overflow by performing the addition and checking to see if the result is smaller, and some programmers do the same with pointers, like so:

```
if( p + increment < p ) throw;
```

Because this operation is undefined, it is completely within the bounds of the standard for the compiler to assume that this statement must always be false and optimize it away. If you want to ensure that this check works the way you would like, write it like this:

```
if( (size_t)p + increment < (size_t)p ) throw;
```

The results of an unsigned integer wraparound are defined, and the compiler can't just throw away the results.

## Sinful C#

C# is very much like C++, which makes it a nice language if you already understand C/C++, but in this case, C# has most of the same problems C++ has. One interesting aspect of C# is that it enforces type safety much more stringently than C/C++ does. For example, the following code throws an error:

```
byte a, b;
a = 255;
b = 1;

byte c = (b + a);

error CS0029: Cannot implicitly convert type 'int' to 'byte'
```

If you understand what this error is really telling you, you'll think about the possible consequences when you get rid of the error by writing:

```
byte c = (byte)(b + a);
```

A safer way to get rid of the warning is to invoke the Convert class:

```
byte d = Convert.ToByte(a + b);
```

If you understand what the compiler is trying to tell you with all these warnings, you can at least think about whether there's really a problem. However, there are limits to what it can help with. In the preceding example, if you got rid of the warning by making a, b, and c signed ints, then overflows are possible, and you'd get no warning.

Another nice feature of C# is that it uses 64-bit integers when it needs to. For example, the following code returns an incorrect result when compiled in C, but works properly on C#:

```
int i = -1;
uint j = 0xffffffff; //largest positive 32-bit int

if(i == j)
    Console.WriteLine("Doh!");
```

The reason for this is that C# will upcast both numbers to a long (64-bit signed int), which can accurately hold both numbers. If you press the issue and try the same thing with a long and a ulong (which are both 64-bit in C#), you get a compiler warning that you need to convert one of them explicitly to the same type as the other. It's the author's opinion that the C/C++ standard should be updated so that if a compiler supports 64-bit operations, it should behave as C# does in this respect.

## Checked and Unchecked

C# also supports the checked and unchecked keywords. You can declare a block of code as checked as this example shows:

```
byte a = 1;
byte b = 255;

checked
{
    byte c = (byte)(a + b);
    byte d = Convert.ToByte(a + b);

    Console.Write("{0} {1}\n", b+1, c);
}
```

In this example, the cast of a + b from int to byte throws an exception. The next line, which calls Convert.ToByte(), would have thrown an exception even without the checked keyword, and the addition within the arguments to Console.Write() throws an exception because of the checked keyword. Because there are times where integer overflows are intentional, the unchecked keyword can be used to declare blocks of code where integer overflow checking is disabled.

You can also use both checked and unchecked to test individual expressions as follows:

```
checked(c = (byte)(b + a));
```

A third way to enable checked behavior is through a compiler option—passing in /checked to the compiler on the command line. If the checked compiler option is enabled, you'll need to explicitly declare unchecked sections or statements where integer overflows are actually intended.

## Sinful Visual Basic and Visual Basic .NET

Visual Basic seems to undergo periodic language revisions, and the transition from Visual Basic 6.0 to Visual Basic .NET is the most significant revision since the shift to object-oriented code in Visual Basic 3.0. One of the more fundamental changes is in the integer types as shown in Table 7-1.

In general, both Visual Basic 6.0 and Visual Basic .NET are immune to execution of arbitrary code through integer overflows. Visual Basic 6.0 throws run-time exceptions when overflows happen in either an operator or one of the conversion functions—for example, CInt(). Visual Basic .NET throws an exception of type System.OverflowException. As detailed in Table 7-1, Visual Basic .NET also has access to the full range of integer types defined in the .NET Framework.

Although operations within Visual Basic itself may not be vulnerable to integer overflows, one area that can cause problems is that the core Win32 API calls all typically take unsigned 32-bit integers (DWORD) as parameters. If your code passes signed 32-bit integers into system calls, it's possible for negative numbers to come back out. Likewise, it may be completely legal to do an operation like 2 – 8046 with signed numbers, but with an unsigned number, that represents an overflow. If you get into a situation where

| Integer Type | Visual Basic 6.0 | Visual Basic .NET |
|---|---|---|
| Signed 8-bit | Not supported | System.SByte |
| Unsigned 8-bit | Byte | Byte |
| Signed 16-bit | Integer | Short |
| Unsigned 16-bit | Not supported | System.UInt16 |
| Signed 32-bit | Long | Integer |
| Unsigned 32-bit | Not supported | System.UInt32 |
| Signed 64-bit | Not supported | Long |
| Unsigned 64-bit | Not supported | System.UInt64 |

**Table 7-1.** Integer Types Supported by Visual Basic 6.0 and Visual Basic .NET

you're obtaining numbers from a Win32 API call, manipulating those numbers with values obtained from or derived from user input, and then making more Win32 calls, you could find yourself in an exploitable situation. Switching back and forth between signed and unsigned numbers is perilous. Even if an integer overflow doesn't result in arbitrary code execution, unhandled exceptions do cause denial of service. An application that isn't running isn't making any money for your customer.

## Sinful Java

Unlike Visual Basic or C#, Java has no defense against integer overflows. As documented in the *Java Language Specification,* found at http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html#9151:

> The built-in integer operators do not indicate overflow or underflow in any way. The only numeric operators that can throw an exception (§11) are the integer divide operator / (§15.17.2) and the integer remainder operator % (§15.17.3), which throw an `ArithmeticException` if the right-hand operand is zero.

Like Visual Basic, Java also only supports a subset of the full range of integer types. Although 64-bit integers are supported, the only unsigned type is a char, which is a 16-bit unsigned value.

Because Java only supports signed types, most of the overflow checks become tricky; and the only area where you don't run into the same problems as C/C++ is when mixing signed and unsigned numbers would lead to unexpected results.

## Sinful Perl

Although at least two of the authors of this book are enthusiastic supporters of Perl, Perl's integer handling is best described as peculiar. The underlying type is a double-precision floating point number, but testing reveals some interesting oddities. Consider the following code:

```
$h = 4294967295;
$i = 0xffffffff;
$k = 0x80000000;

print "$h = 4294967295 - $h + 1 = ".($h + 1)."\n";
print "$i = 0xffffffff - $i + 1 = ".($i + 1)."\n";

printf("\nUsing printf and %%d specifier\n");
printf("\$i = %d, \$i + 1 = %d\n\n", $i, $i + 1);

printf("Testing division corner case\n");
printf("0x80000000/-1 = %d\n", $k/-1);
print "0x80000000/-1 = ".($k/-1)."\n";
```

The test code yields the following results:

```
[e:\projects\19_sins]perl foo.pl
4294967295 = 4294967295 - 4294967295 + 1 = 4294967296
4294967295 = 0xffffffff - 4294967295 + 1 = 4294967296

Using printf and %d specifier
$i = -1, $i + 1 = -1

Testing division corner case
0x80000000/-1 = -2147483648
0x80000000/-1 = -2147483648
```

At first, the results look peculiar, especially when using printf with format strings, as opposed to a regular print statement. The first thing to notice is that you're able to set a variable to the maximum value for an unsigned integer, but adding 1 to it either increments it by 1 or, if you look at it with %d, does nothing. The issue here is that you're really dealing with floating point numbers, and the %d specifier causes Perl to cast the number from double to int. There's not really an internal overflow, but it does appear that way if you try to print the results.

Due to Perl's interesting numeric type handling, we recommend being very careful with any Perl applications where significant math operations are involved. Unless you have prior experience with floating point issues, you could be in for some interesting learning experiences. Other higher-level languages, such as Visual Basic, will also sometimes internally convert upward to floating point as well. The following code and result shows you exactly what's going on:

```
print (5/4)."\n";
1.25
```

For most normal applications, Perl will just do the right thing, which it is exceedingly good at. However, don't be fooled into thinking that you're dealing with integers—you're dealing with floating point numbers, which are another can of worms entirely.

## SPOTTING THE SIN PATTERN

Any application performing arithmetic can exhibit this sin, especially when one or more of the inputs are provided by the user, and not thoroughly checked for validity. Focus especially on C/C++ array index calculations and buffer size allocations.

## SPOTTING THE SIN DURING CODE REVIEW

C/C++ developers need to pay the most attention to integer overflows. Now that many developers are better about checking sizes when directly manipulating memory, the next line of attack is on the math you use to check what you're doing. C# and Java are next. You may not have the issue of direct memory manipulation, but the language lets you make nearly as many mistakes as C/C++ allows.

One comment that applies to all languages is to check input before you manipulate it! A very serious problem in Microsoft's IIS 4.0 and 5.0 web server happened because the programmer added 1 and then checked for an overly large size afterward—with the types he was using, 64K – 1 + 1 equals zero! There is a link to the bulletin in the section "Other Resources" in this chapter.

### C/C++

The first step is to find memory allocations. The most dangerous of these are where you're allocating an amount you calculated. The first step is to ensure that you have no potential integer overflows in your function. Next, go look at the functions you called to determine your inputs. The author of this chapter has seen code that looked about like this:

```
THING* AllocThings(int a, int b, int c, int d)
{
      int bufsize;
      THING* ptr;

      bufsize = IntegerOverflowsRUs(a, b, c, d);

      ptr = (THING*)malloc(bufsize);
      return ptr;
}
```

The problem is masked inside the function used to calculate the buffer size, and made worse by cryptic, nondescriptive variable names (and signed integers). If you have time to be thorough, investigate your called functions until you get to low-level run-time or system calls. Finally, go investigate where the data came from: How do you know the function arguments haven't been tampered with? Are the arguments under your control, or the control of a potential attacker?

According to the creators of the Perl language, the first great virtue of a programmer is laziness! Let's do things the easy way—all these integers are hard enough—the compiler can help us. Turn up the warning level to /W4 (Visual C++) or –Wall or

–Wsign-compare (gcc), and you'll find potential integer problems popping up all over the place. Pay close attention to integer-related warnings, especially signed-unsigned mismatches and truncation issues.

In Visual C++, the most important warnings to watch for are C4018, C4389, C4242, C4302 and C4244.

In gcc, watch for "warning: comparison between signed and unsigned integer expressions" warnings.

Be wary of using #pragma to ignore warnings; alarm bells should go off if you see something like this in your code:

```
#pragma warning(disable : 4244)
```

The next thing to look for are places where you've tried to ensure writes into buffers (stack and heap buffers) are safe by bounding to the destination buffer size; here, you must make sure the math is correct. Here's an example of the math going wrong:

```
int ConcatBuffers(char *buf1, char *buf2,
                size_t len1, size_t len2){
    char buf[0xFF];
    if((len1 + len2) > 0xFF) return -1;
    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);
    // do stuff with buf
    return 0;
}
```

In this code, the two incoming buffer sizes are checked to make sure they are not bigger than the size of the destination buffer. The problem is if len1 is 0x103, and len2 is 0xfffffffc, and you add them together, they wrap around on a 32-bit CPU to 255 (0xff), so the data squeaks by the sanity check. Then the calls to mempcy attempt to copy about 4GB of junk to a 255-byte buffer!

Someone may have been trying to make those pesky warnings go away by casting one type to another. As you now know, these casts are perilous and ought to be carefully checked. Look at every cast, and make sure it's safe. See the earlier section "Casting Operations" on C/C++ casting and conversion.

Here's another example to watch for:

```
int read(char*buf, size_t count) {
    // Do something with memory
}

    ...
    while (true) {
        BYTE buf[1024];
        int skip = count - cbBytesRead;
        if (skip > sizeof(buf))
            skip = sizeof(buf);

        if (read(buf, skip))
            cbBytesRead += skip;
        else
            break;
    ...
```

This code compares the value of skip with 1024 and, if it's less, copies skip bytes to buf. The problem is if skip calculates out to a negative number (say, –2), that number is always smaller than 1024 and so the read() function copies –2 bytes, which, when expressed as an unsigned integer (size_t), is almost 4GB. So read() copies 4GB into a 1K buffer. Oops!

Another overlooked example is calling the C++ new operator. There is an implicit multiply:

```
Foo *p = new Foo(N);
```

If N is controlled by the bad guys, they could overflow operator new, because N * sizeof(Foo) might overflow. Some compilers do currently check for integer overflows when doing the math and will fail the allocation.

## C#

Although C# doesn't typically involve direct memory access, it can sometimes call into system APIs by declaring an unsafe section and compiling with the /unsafe flag. Any calculations used when calling into system APIs need to be checked. Speaking of checked, it is a great keyword or better yet compiler switch to use. Turn it on, and pay close attention when you end up in the exception handler. Conversely, use the unchecked keyword sparingly, and only after giving the problem some thought.

Pay close attention to any code that catches integer exceptions—if it's done improperly, just swallowing an exception may lead to exploitable conditions.

In short, any C# code compiled with /unsafe should have all integer arithmetic reviewed (see the preceding section, "C/C++," for ideas) to make sure it's safe.

## Java

Java also doesn't allow direct memory access, and it isn't quite as dangerous as C/C++. But you should still be wary: like C/C++, the language itself has no defense against integer overflows, and you can easily make logic errors. See the section "Redemption Steps" later in the chapter for programmatic solutions.

### Visual Basic and Visual Basic .NET

Visual Basic has managed to turn integer overflows into a denial of service problem—much the same situation as using the checked keyword in C#. A key indication of problems shows up when the programmer is using the error handling mechanism to ignore errors due to mishandling integers. Ensure the error handling is correct. The following in Visual Basic (not Visual Basic .NET) is a warning that the developer is lazy and does not want to handle any exception raised by the program at run time. Not good.

```
On Error Continue
```

### Perl

Perl is cool, but floating point math is a little strange. Most of the time, it will do the right thing, but Perl is different in many ways, so be careful. This is especially true when calling into modules that may be thin wrappers over system calls.

## TESTING TECHNIQUES TO FIND THE SIN

If the input is character strings, try feeding the application sizes that tend to cause errors. For example, strings that are 64K or 64K – 1 bytes long can often cause problems. Other common problem lengths are 127, 128, and 255, as well as just on either side of 32K. Any time that adding one to a number results in either changing sign or flipping back to zero, you have a good test case.

In the cases where you're allowed to feed the programmer numbers directly—one example would be a structured document—try making the numbers arbitrarily large, and especially hit the corner cases.

## EXAMPLE SINS

A search on "integer overflow" in the Common Vulnerabilities and Exposures (CVE) database yields 445 entries as of this writing (September 2008—a rate of about 100 per year since we wrote the first edition). Here are a few.

### Multiple Integer Overflows in the SearchKit API in Apple Mac OS X

From the CVE (CVE-2008-3616) description:

> Multiple integer overflows in the SearchKit API in Apple Mac OS X 10.4.11 and 10.5 through 10.5.4 allow context-dependent attackers to cause a denial of service (application crash) or execute arbitrary code via vectors associated with "passing untrusted input" to unspecified API functions.

## Integer Overflow in Google Android SDK

From the CVE (CVE-2008-0986) description:

> Integer overflow in the BMP::readFromStream method in the libsgl.so library in Google Android SDK m3-rc37a and earlier, and m5-rc14, allows remote attackers to execute arbitrary code via a crafted BMP file with a header containing a negative offset field.

Here is an interesting note from the Core Security Technologies advisory on this issue (www.coresecurity.com/corelabs):

> Several vulnerabilities have been found in Android's core libraries for processing graphic content in some of the most used image formats (PNG, GIF, and BMP). While some of these vulnerabilities stem from the use of outdated and vulnerable open source image processing libraries, others were introduced by native Android code that uses them or that implements new functionality.
>
> Exploitation of these vulnerabilities to yield complete control of a phone running the Android platform has been proved possible using the emulator included in the SDK, which emulates phones running the Android platform on an ARM microprocessor.

## Flaw in Windows Script Engine Could Allow Code Execution

From the CVE (CAN-2003-0010) description:

> Integer overflow in JsArrayFunctionHeapSort function used by Windows Script Engine for JScript (JScript.dll) on various Windows operating systems allows remote attackers to execute arbitrary code via a malicious web page or HTML e-mail that uses a large array index value that enables a heap-based buffer overflow attack.

The interesting thing about this overflow is that it allows for arbitrary code execution by a scripting language that doesn't allow for direct memory access. The Microsoft bulletin can be found at www.microsoft.com/technet/security/bulletin/MS03-008.mspx.

## Heap Overrun in HTR Chunked Encoding Could Enable Web Server Compromise

Shortly after this problem was announced in June 2002, widespread attacks were seen against affected IIS servers. More details can be found at www.microsoft.com/technet/security/Bulletin/MS02-028.mspx, but the root cause was because the HTR handler accepted a length of 64K –1 from the user, added 1—after all, we needed room for the null terminator—and then asked the memory allocator for zero bytes. It's not known whether

Bill Gates really said 64K ought to be enough for anybody or if that's an Internet legend, but 64K worth of shell code ought to be enough for any hacker to cause mayhem!

## REDEMPTION STEPS

Redemption from integer overflows can only truly be had by carefully studying and understanding the problem. That said, there are some steps you can take to make the problem easier to avoid. The first is to use unsigned numbers where possible. The C/C++ standard provides the size_t type for (you guessed it) sizes, and a smart programmer will use it. Unsigned integers are much, much easier to verify than signed integers. It makes no sense to use a signed integer to allocate memory!

### Do the Math

Algebra usually isn't any fun, but it is useful. A good way to prevent integer overflows is to just work out the math involved as you would back in Algebra I. Let's consider a couple of typical allocation calculations:

$$Size = (elements * sizeof\,(element)) + sizeof\,(header)$$

If *Size* is greater than MAX_INT, there's a problem. You can then rewrite this as:

$$MaxInt \leq (elements * sizeof\,(element)) + sizeof\,(header)$$

Which leads to:

$$MaxInt - sizeof\,(header) \leq elements * sizeof\,(element)$$

And finally:

$$\frac{MaxInt - sizeof\,(header)}{sizeof\,(element)} \leq elements$$

A nice aspect of this check is that it will work out to a compile-time constant. Working out the math on a scratch pad or whiteboard can help you to write some very efficient checks for whether a calculation is valid.

### Don't Use Tricks

Avoid "clever" code—make your checks for integer problems straightforward and easy to understand. Here's an example of a check for addition overflows that was too smart by half:

```
int a, b, c;

c = a + b;

if(a ^ b ^ c < 0)
  return BAD_INPUT;
```

This test suffers from a lot of problems. Many of us need a few minutes to figure out just what it is trying to do, and then it also has a problem with false positives and false negatives—it only works some of the time. Another example of a check that only works some of the time follows:

```
int a, b, c;

c = a * b;

if(c < 0)
  return BAD_INPUT;
```

Even allowing for positive inputs to start with, the code only checks for some overflows—consider (2^30 + 1) * 8; that's 2^33 + 8—and once truncated back to 32-bit, it yields 8, which is both incorrect and not negative. A safer way to do the same thing is to store a 32-bit multiplication in a 64-bit number, and then check to see if the high-order bits are set, indicating an overflow.

For code like this:

```
unsigned a,b;
...
if (a * b < MAX) {
    ...
}
```

you could simply bind the a and b variables to a value you know is less than MAX. For example:

```
#include "limits.h"

#define MAX_A 10000
#define MAX_B 250

assert(UINT_MAX / MAX_A >= MAX_B); // check that MAX_A and MAX_B are small enough
if (a < MAX_A && b < MAX_B) {
    ...
}
```

## Write Out Casts

A very good defense is to annotate the code with the exact casts that happen, depending on the various operations in play. As a somewhat contrived example, consider this:

```
unsigned int x;
short a, b;

// more code ensues

if( a + b < x) DoSomething();
```

The operator cast resulting from the addition results in an int—so you effectively have this:

```
if( (int)(a + b) < x ) DoSomething();
```

You now have a signed-unsigned comparison, and the int has to be cast to unsigned before the comparison can take place, which results in this:

```
if( (unsigned int)(int)(a + b) < x ) DoSomething();
```

You can now analyze the problem completely—the results of the addition aren't a problem, because anything you can fit into two shorts can be added and fit into an int. The cast to unsigned may be a problem if the intermediate result is negative.

Here's another example discovered during the development of SafeInt: a compile-time constant was needed to find the minimum and maximum signed integers based on a template argument. Here's the initial code:

```
template <typename T>
T SignedIntMax()
{
  return ~( 1 << sizeof(T)*8 - 1);
}
```

For most integer types, this worked just fine, but there was a problem with 64-bit ints. This was missed in code review by some excellent developers and one of my coauthors. Let's take a closer look using a casting analysis. For a 64-bit integer, we have

```
return ~( (int)1 << 63 );
```

At this point, the problem ought to be clear: a literal is an int, unless it is something too large to fit in an int (for example, 0x8000000 is an unsigned int). What happens when we left-shift a 32-bit number by 63 bits? According to the C/C++ standard, this is undefined. The Microsoft implementation will just shift by the modulus of the shift argument and the number of bits available (and not warn you). The correct code is

```
return ~( (T)1 << sizeof(T)*8 - 1 );
```

We don't recommend leaving the casts in place in the actual code. If someone changes a type, this could introduce problems. Just annotate it temporarily, or in a comment until you've figured out what will happen.

## Use SafeInt

If you'd like to thoroughly armor your code against integer overflows, you can try using the SafeInt class, written by David LeBlanc (details are in the section "Other Resources" in this chapter). Be warned that unless you catch the exceptions thrown by the class, you've exchanged potential arbitrary code execution for a denial of service. Here's an example of how you can use SafeInt:

```
size_t CalcAllocSize(int HowMany, int Size, int HeaderLen)
{
     try{
       SafeInt<size_t> tmp(HowMany);
       return tmp * Size + SafeInt<size_t>(HeaderLen);
     }
      catch(SafeIntException)
      {
           return (size_t)~0;
      }
}
```

Signed integers are used as an input for illustration—this function should be written exclusively with the size_t type. Let's take a look at what happens under the covers. The first is that the value of HowMany is checked to see if it is negative. Trying to assign a negative value to an unsigned SafeInt throws an exception. Next, operator precedence causes you to multiply a SafeInt by Size, which is an int and will be checked both for overflow and valid range. The result of SafeInt * int is another SafeInt, so you now perform a checked addition. Note that you need to change the incoming int to a SafeInt, because a negative header length would be valid math but doesn't make sense—sizes are best represented as unsigned numbers. Finally, in the return, the SafeInt<size_t> is cast back to a size_t, which is a no-op. There's a lot of complex checking going on, but your code is simple and easy to read.

If you're programming with C#, compile with /checked, and use unchecked statements to exempt individual lines from checking.

## EXTRA DEFENSIVE MEASURES

If you use gcc, you can compile with the –ftrapv option. This catches signed integer overflows by calling into various run-time functions, but it works *only* for signed integers. The other bit of bad news is these functions call abort() on overflow.

Microsoft Visual C++ 2005 and later automatically catches calls to operator new that overflow. Note, your code must catch the ensuing std::bad_alloc exception, or your application will crash —which is generally preferable to running attacker-supplied shell code!

Some static analysis tools are starting to find integer problems. In the five years that we've been working with integer overflows, we've gone from believing that it wasn't possible to find integer overflows with static analysis to having a limited ability to find issues. If you have tools available that do find problems, then by all means you should use them. A problem that we've seen with everything we've reviewed to date is that a manual analysis will find a lot more than the tools will. Run the tools, then go review your allocations yourself—integer overflows are really tricky, so look at the tools as a helper, not a complete solution.

## OTHER RESOURCES

- SafeInt—available at www.codeplex.com/SafeInt. SafeInt is supported on both Visual Studio and gcc.
- "Reviewing Code for Integer Manipulation Vulnerabilities" by Michael Howard: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp
- "Expert Tips for Finding Security Defects in Your Code" by Michael Howard: http://msdn.microsoft.com/msdnmag/issues/03/11/SecurityCodeReview/default.aspx
- "Integer Overflows —The Next Big Threat" by Ravind Ramesh: http://star-techcentral.com/tech/story.asp?file=/2004/10/26/itfeature/9170256&sec=itfeature
- DOS against Java JNDI/DNS: http://archives.neohapsis.com/archives/bugtraq/2004-11/0092.html

## SUMMARY

- **Do** check all calculations used to determine memory allocations to check the arithmetic cannot overflow.
- **Do** check all calculations used to determine array indexes to check the arithmetic cannot overflow.
- **Do** use unsigned integers for array offsets and memory allocation sizes.
- **Do** check for truncation and sign issues when taking differences of pointers, and working with size_t.
- **Do not** think languages other than C/C++ are immune to integer overflows.

# SIN 9

# CATCHING EXCEPTIONS

## OVERVIEW OF THE SIN

Exception handling is an often misused feature of programming languages and operating systems. Basically, if something's gone wrong, and you don't know exactly how to correct it, then the only safe thing you can do is to exit the application. Trying to do anything else may lead to an unstable application, and an unstable application is typically some amount of work away from being an exploitable application.

Three related sins are Sin 11, "Failure to Handle Errors"; Sin 13, "Race Conditions"; and Sin 12, "Information Leakage."

## CWE REFERENCES

CWE also recognizes catching broad exceptions as an issue.

- CWE-396: Declaration of Catch for Generic Exception

## AFFECTED LANGUAGES

As is often the case, C and C++ allow you to get into the most trouble. However, as we'll explain in a moment, there are variants specific to different operating systems, and some languages that you may not normally think of as being low level may allow you to access the operating system APIs—Perl is an example. While higher-level languages such as C# and Java do use exception handling semantics, the automatic garbage collection features tend to make exception handling errors less likely to be exploitable.

## THE SIN EXPLAINED

Exception handling comes in several flavors. Broadly, we have try-catch blocks implemented in several languages; Windows operating systems have structured exception handling (as does Objective C++), which includes three types of blocks, try, except, and finally; and UNIX-based operating systems (including Linux and Mac OS) all can utilize signal handling. Windows also implements a very small subset of signal handling, but because the supported signals are so incomplete, it is very rare to see programs that use signals on Windows—there are also many other ways to get the same thing accomplished.

### Sinful C++ Exceptions

The basic concept behind C++ exceptions is fairly simple. You put code that might do something wrong into a try block, and then you can handle errors in a catch block. Here's an example of how it is used (and abused):

```
void Sample(size_t count)
{
    try
    {
        char* pSz = new char[count];
    }
    catch(...)
    {
        cout << "Out of memory\n";
    }
}
```

You'd first do something that could fail inside a try block, and then you catch exceptions in a catch block. If you want to create an exception, you do this with the `throw` keyword. Try-catch blocks can be nested, so if an exception isn't caught by the first catch block in scope, it can be caught by the next catch block.

The preceding sample is also an example of how to cause problems. When catch(…) is used, this is a special construct that tells the compiler to handle all C++ exceptions inside this catch block. You typically won't handle operating system exceptions or signals like access violations (also called segmentation faults) in C++ catch blocks, and we'll cover the issues with that shortly. In the trivial example just shown, the only thing that can possibly go wrong is for the allocation to fail. However, in the real world, you'd be doing a lot of operations, and more could go wrong than just an allocation—you'd treat them all the same! Let's take a look at something a little more complicated and see how this works.

```
void Sample(const char* szIn, size_t count)
{
   try
   {
      char* pSz = new char[count];
      size_t cchIn = strnlen( szIn, count );
      // Now check for potential overflow
      if( cchIn == count )
         throw FatalError(5);
      // Or put the string in the buffer
   }
   catch( ... )
   {
      cout << "Out of memory\n";
   }
}
```

If you've just done this, you've treated bad inputs the same as being out of memory. The right way to do this is with the following catch statement:

```
catch( std::bad_alloc& err )
```

This catch statement will only catch the exceptions new throws—std::bad_alloc. You'd then have another try-catch at a higher level that would catch FatalError exceptions, or possibly log them, and then throw the exception again to let the app go ahead and exit.

Of course, it's not always this simple because in some cases operator::new might throw a different exception. For example, in the Microsoft Foundation Classes, a failed new operator can throw a CMemoryException, and in many modern C++ compilers (Microsoft Visual C++ and gcc, for example), you can use std::nothrow to prevent the new operator from raising an exception. The following two examples will both fail to catch the correct exception because the first examples won't throw an exception, and the second does not throw a std::bad_alloc exception, it throws CMemoryExceptions.

```
try
{
    struct BigThing { double _d[16999];};
    BigThing *p = new (std::nothrow) BigThing[14999];
    // Use p
}
catch(std::bad_alloc& err)
{
    // handle error
}
And
try
{
    CString str = new CString(szSomeReallyLongString);
    // use str
}
catch(std::bad_alloc& err)
{
    // handle error
}
```

If you're paying attention, you're probably thinking, "But they caused a memory leak! Don't these authors ever write real code?" If so, very good—and there's a reason for making the example do this. Proper code that deals with exceptions has to be exception safe. The right thing to have done would be to have used a pointer holder of some type that will release the memory once you exit the try block—even if the exit mechanism is a thrown exception. If you've decided to use exceptions in your application, good for you—you've just potentially made your code much more efficient; if done properly, it will utilize predictive pipelining in the processor better, and error handling can be done a lot more cleanly. You've also signed yourself up to write exception-safe code. If you're really paying attention, you might have noticed that we caught the exception by reference, not by value, and not by a pointer to an exception—there are good reasons for this, and if you'd like it explained, Scott does an excellent job of it in "Effective C++."

We've actually packed several lessons into this example. Another common error would be to write code like this:

```
catch(...)
{
   delete[] pSz;
}
```

This presumes that pSz has been correctly initialized and declared outside of the try block. If pSz has not been initialized, you're looking at an exploit. An even worse disaster that Richard van Eeden found during a code review looked like this:

```
catch(...)
{
   // Geez pSz is out of scope - add it here
   char* pSz;
   delete pSz;
}
```

The programmer had declared pSz inside of the try block, noticed that it didn't compile, and just created a new, uninitialized variable with the same name—this is very easily an exploit. Again, the right thing to do is to contain the pointer with a pointer holder that will properly initialize itself to null, release the memory when it goes out of scope, and set itself back to null on the way out.

## Sinful Structured Exception Handling (SEH)

Current Microsoft Windows operating systems support a feature called structured exception handling (SEH). SEH includes the keywords `__try`, `__except`, and `__finally`. Each keyword precedes a block of code, and effectively allows C programs to mimic what C++ accomplishes with `try`, `catch`, and destructors. The SEH analogue to the C++ keyword `throw` is the RaiseException API call. Here's an example:

```
int Filter( DWORD dwExceptionCode )
{
   if( dwExceptionCode == EXCEPTION_INTEGER_OVERFLOW )
      return EXCEPTION_EXECUTE_HANDLER;
   else
```

```
      return EXCEPTION_CONTINUE_SEARCH;
}
void Foo()
{
    __try
   {
      DoSomethingScary();
   }
   __except( Filter( GetExceptionCode() ) )
   {
      printf("Integer overflow!\n");
      return E_FAIL;
   }
   __finally
   {
      // Clean up from __try block
   }
}
```

Here's the way it works: any exception raised within the __try block will invoke the exception handler in the first __except block found. If you don't create one yourself, the operating system will create one for you, which is how you get a pop-up informing you the application has suffered an inconvenience. When the __except block is entered, the filter expression is called. Filter expressions typically just make decisions based on the exception code, but they can get much more detailed information about the exception if the filter expression also has an argument that passes in the result of calling GetExceptionInformation. If you're interested in the details of how SEH works, consult MSDN.

The __except block will be entered if the filter expression returns EXCEPTION_EXECUTE_HANDLER, where it can then take appropriate action. If the filter returns EXCEPTION_CONTINUE_SEARCH, the next exception handler up the chain is called, and if the filter returns EXCEPTION_CONTINUE_EXECUTION, then the instruction that originally caused the exception is attempted again.

If the __try block is exited either normally or through an exception, then the __finally block is executed. Be warned that all of this comes with fairly significant overhead and run-time cost, and it isn't advisable to substitute __finally blocks for "goto CleanUp" statements—goto can be criticized for being used to create overly complex code, but using it to ensure proper cleanup and enforcing one exit point for a function is a very valid use.

Like a catch(…) block, __except(EXCEPTION_EXECUTE_HANDLER) just handles all exceptions and is never advisable. Out of all sinful SEH, this is perhaps the most common flaw. A poorly written filter expression would be the next most common flaw, and an example of this will be shown later in this chapter. A less common, but very exploitable problem is to perform cleanup in a __finally block when memory has been corrupted. If you write a __finally block, make sure you call the AbnormalTermination macro within the block to know whether you got there by way of an unexpected exit or not—a gotcha is that return and goto both count as an abnormal termination—you need to Read The Fine Manual if you're going to use these.

An exception filter that returns EXCEPTION_CONTINUE_EXECUTION should be used only rarely, and only if you know exactly what you're doing. The most common scenario is if you're using memory-mapped files, and you get an exception indicating that a needed page has not been mapped. You can map the page into memory and try again. Assuming user-mode programming, this is the only situation we're aware of in which continuing execution is warranted, though one could create a contrived example where handling a divide by zero can be dealt with by patching up a variable and continuing.

As an aside, mixing C++ exceptions and SEH exceptions can be tricky. Giving thorough instructions on how to use these in the same application is beyond the scope of this book, but if you find yourself having to do this, isolate the code with __try-__except blocks from the C++ code by refactoring into different functions.

The next example is just about as sinful as you can get on Windows:

```
char *ReallySafeStrCopy(char *dst, const char *src) {
      __try {
            return strcpy(dst,src);
      }
          except(EXCEPTION_EXECUTE_HANDLER)
          // mask the error
      }
      return dst;
}
```

If strcpy fails because src is larger than dst, or src is NULL, you have no idea what state the application is in. Is dst valid? And depending on where dst resides in memory, what state is the heap or stack? You have no clue—and yet the application might keep running for a few hours until it explodes. Because the failure happens so much later than the incident that caused the error, this situation is impossible to debug. Don't do this.

## Sinful Signal Handling

On operating systems derived from UNIX, signal handlers are used to process the various signals that can be passed into a process, errors that happen internal to a process, or user-defined signals that might be used instead of multithreaded programming. Signal handling can also lead to race conditions when a process is interrupted by a signal during a critical series of calls—we'll cover this much more thoroughly in Sin 13, "Race Conditions."

Most of the problems you can encounter are similar to those we've already covered; if you have an application in an unstable state and attempt to either recover or perform cleanup tasks, you're at risk of causing additional problems.

An issue which is often seen when programming for the various operating systems derived from UNIX—BSD, System V, and Linux, to name the three most common—is that some system calls behave very differently across the different branches. Signal handling system calls are noteworthy in this respect due to some fairly drastic differences, and your code should be careful to not make assumptions about the behavior of the signal() system call.

Some signals in particular lead to disaster—signal handlers will resume at the instruction that raised the signal just like a SEH handler that returns EXCEPTION_CONTINUE_EXECUTION. This means that if you wrote a signal handler for a numeric error, such as divide by zero, your application can easily get into an infinite loop.

Trying to handle a memory fault (SIG_SEGV or segmentation fault) by doing anything more than logging the error is playing into the hands of the attackers. The documentation on signal function points out that handling a SIG_SEGV signal may result in undefined behavior, such as uploading sensitive files somewhere, installing a rootkit, and broadcasting your system address on IRC.

## Sinful C#, VB.NET, and Java

The code example that follows shows how not to catch exceptions. The code is catching every conceivable exception and, like the Windows SEH example, could be masking errors.

```
try
{
    // (1) Load an XML file from disc
    // (2) Use some data in the XML to get a URI
    // (3) Open the client certificate store to get a
    //     client X.509 certificate and private key
    // (4) Make an authenticated request to the server described in (2)
    //     using the cert/key from (3)
}
catch (Exception e)
{
    // Handle any possible error
    // Including all the ones I know nothing about
}
```

All the functionality in the preceding code includes a dizzying array of possible exceptions. For .NET code, this includes SecurityException, XmlException, IOException, ArgumentException, ObjectDisposedException, NotSupportedException, FileNotFoundException, and SocketException. Does your code really know how to handle all these exceptions correctly?

Don't get me wrong—there are situations when you should catch all exceptions, such as at the boundary of a COM interface. If you do this, ensure that you have translated the error into something that the caller will understand as a failure; returning an HRESULT of E_UNEXPECTED might be one way to handle it, logging the error and terminating the application could be another valid approach.

### Sinful Ruby

This code example is somewhat similar to the preceding C# code in that it handles all possible exceptions.

```
begin
    # something dodgy
rescue Exception => e
    # handle the exception, Exception is the parent class
end
```

## SPOTTING THE SIN PATTERN

Any application that has the following pattern is at risk of this sin:

- Using catch(…).
- Using catch(Exception).
- Using __except(EXCEPTION_EXECUTE_HANDLER), and even if EXCEPTION_EXECUTE_HANDLER is not hard-coded, filter expressions need to be reviewed.
- Using signal, or sigaction, though whether this represents a security problem depends on the signal being handled, and perhaps more important, how it is handled.

## SPOTTING THE SIN DURING CODE REVIEW

If the code is C++, look for catch blocks that catch all exceptions, and also examine whether exception-safe techniques are in use. If not, be especially wary of uninitialized variables. If the code is C++ and compiled with the Microsoft compiler, verify that the /EHa compiler option has not been set; this flag causes structured exceptions to land in catch blocks. The current consensus is that creating this option wasn't a good idea, and using it is worse.

If the code uses __try, review the __except blocks and the filter expressions. If __finally blocks are present, review these for proper use of the AbnormalTermination macro. Just as with catch blocks, be wary of uninitialized variables. An example of an improperly written exception handler is shown here:

```
int BadFilter( DWORD dwExceptionCode )
{
   switch( dwExceptionCode )
   {
   case EXCEPTION_ACCESS_VIOLATION:
      // At least we won't ignore these
      return EXCEPTION_CONTINUE_SEARCH;
   case EXCEPTION_MY_EXCEPTION:
      // User-defined - figure out what's going on,
      // and do something appropriate.
      return HandleMyException();
   default:
      // DO NOT DO THIS!!!
      return EXCEPTION_EXECUTE_HANDLER;
}
```

If the code uses signal or sigaction, review the signals that are handled for issues, ensure that only functions known to be safe within signal handlers are used—see the Async-signal-safe functions topic within the man page for signal(7). While you're at it, review for race conditions, which are detailed in Sin 13.

Some static analysis tools can find error-related issues. For example, Microsoft VC++ /analyze will find code that catches all exceptions:

```
void ADodgeyFunction() {
      __try {
      }
      __except( 1 ) {
      }
}
```

This yields the following warning:

```
warning C6320: Exception-filter expression is the constant
EXCEPTION_EXECUTE_HANDLER. This might mask exceptions that were not
intended to be handled.
```

For .NET code, FxCop will yield DoNotCatchGeneralExceptionTypes warnings for code that catches all exceptions.

Fortify's static analysis tools also flag overly zealous exception handing code in .NET code and Java, referring to them as "Overly-Broad Catch Block" errors.

## TESTING TECHNIQUES TO FIND THE SIN

Most of these sins are better found through code review than testing. A skillful tester can sometimes find erroneous SEH exception handling by attaching a debugger and causing it to break on all first-chance exceptions, and then seeing whether these get propagated or end up bubbling up through the application. Be warned that you'll hit a lot of these calls that are internal to the operating system.

## EXAMPLE SINS

The following entry on the Common Vulnerabilities and Exposures (CVE) web site (http://cve.mitre.org/) is an example of improper exception handling.

### CVE-2007-0038

This is addressed in Microsoft security bulleting MS07-017, "Windows Animated Cursor Remote Code Execution Vulnerability." Technically, this vulnerability is due to a buffer overrun, but it was made worse by an __except block that just invoked the handler on all exceptions. Handing all the exceptions allowed the attackers to completely overcome address space layout randomization (ASLR), which caused the severity of the issue to be critical on Windows Vista.

## REDEMPTION STEPS

Examine your code for code that catches exceptions or handles signals. Ensure that any cleanup is happening on properly initialized objects.

### C++ Redemption

The first step to redemption is unfortunately somewhat like the automobile repair manual that states "First, remove the engine." Correctly structuring exception handling in a C++ application is as much art as science, it can be complex, and we could all debate for quite some time the nuances of how to do it. Your first task if your application uses exception handling is to ensure that all objects that acquire resources are exception safe and properly clean up on destruction. Consistently documenting which methods do and do not throw exceptions can be very helpful.

The second task, which is equally daunting, is to ensure that you're catching the exceptions you need to catch, and in the right places. Be careful to catch all C++ exceptions prior to exiting a callback function—the operating system or library call that is calling your code may not handle exceptions properly, or worse yet, it may swallow them. Ideally, all of your catch(…) blocks will be considered to be a bug if any exceptions reach them and indicate that you need to handle those exceptions explicitly elsewhere.

If you've done the first step well, you should not have any problems with this, but it is worth mentioning: carefully audit any cleanup steps performed in a catch block (regardless of scope), and make sure that nothing operates on uninitialized or partially initialized variables or objects. One word of warning about older MFC (Microsoft Foundation Classes) code: constructs exist to catch C++ exceptions with a set of macros that include CATCH_ALL. It won't quite catch all the exceptions; it is actually looking for a pointer to a CException class, but catching exceptions by pointer is in general a bad plan—what if the exception goes out of scope, or if it is on the heap, what if you can't allocate a new one?

### SEH Redemption

If your code uses structured exceptions, find your try-except blocks, and ensure that you do not have __except blocks that handle exceptions other than those that you know exactly how to handle. If you find code that handles access violations, and people start asking what regressions we might cause by removing the handler, be sure to remind them that not only is this code already broken now, but you're not getting any crash data to tell you where the problem might be. Also look out for macros that implement exception handlers.

### Signal Handler Redemption

Carefully audit signal handlers to ensure that only safe functions are called from within signal handlers—Read The Fine man pages for your operating system for a list of these. Do not ever attempt to handle segmentation faults. Additionally, read Sin 13 for more information on race conditions and signal handlers.

## OTHER RESOURCES

- *Programming with Exceptions in C++* by Kyle Loudon (O'Reilly, 2003)
- "Structured Exception Handling Basics" by Vadim Kokielov: http://www.gamedev.net/reference/articles/article1272.asp
- "Exception handling," Wikipedia: http://en.wikipedia.org/wiki/Exception_handling
- "Structured Exception Handling," Microsoft Corporation: http://msdn.microsoft.com/en-us/library/ms680657.aspx
- Lessons learned from the Animated Cursor Security Bug: http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx
- "Exception Handling in Java and C#" by Howard Gilbert: http://pclt.cis.yale.edu/pclt/exceptions.htm

## SUMMARY

- **Do** catch only specific exceptions.
- **Do** handle only structured exceptions that your code can handle.
- **Do** handle signals with safe functions.
- **Do not** catch(…).
- **Do not** catch (Exception).
- **Do not** __except(EXCEPTION_EXECUTE_HANDLER).
- **Do not** handle SIG_SEGV signals, except to log.

# SIN 11
# FAILURE TO HANDLE ERRORS CORRECTLY

## OVERVIEW OF THE SIN

Many security risks are possible when programmers fail to handle an error condition correctly. Sometimes a program can end up in an insecure state, but more often the result is a denial of service issue, as the application simply dies. This problem is significant in even modern languages, such as C#, Ruby, Python, and Java, where the failure to handle an exception usually results in program termination by the run-time environment or operating system.

The unfortunate reality is that any reliability problem in a program that leads to the program crashing, aborting, or restarting is a denial of service issue and therefore can be a security problem, especially for server code.

A common source of errors is sample code that has been copied and pasted. Often sample code leaves out error return checking to make the code more readable.

## CWE REFERENCES

The Common Weakness Enumeration project includes the following entries relating to the error-handling issues explained in this chapter.

- CWE-81: Failure to Sanitize Directives in an Error Message Web Page
- CWE-388: Error Handling
- CWE-209: Error Message Information Leak
- CWE-390: Detection of Error Condition Without Action
- CWE-252: Unchecked Return Value

## AFFECTED LANGUAGES

Any language that uses function error return values, such as ASP, PHP, C, and C++; and any language that relies on exceptions, such as C#, Ruby, Python, VB.NET, and Java.

## THE SIN EXPLAINED

There are five variants of this sin:

- Yielding too much information
- Ignoring errors
- Misinterpreting errors

- Using useless return values
- Using non-error return values

Let's look at each in detail.

## Yielding Too Much Information

We talk about this issue in numerous places in the book, most notably in Sin 12. It's a very common issue: an error occurs and, in the interest of "usability," you tell the user exactly what failed, why, and, in some cases, how to fix the issue. The problem is you just told the bad guy a bunch of really juicy information, too—data he can use to help him compromise the system.

## Ignoring Errors

Error return values are there for a very good reason: to indicate a potential failure condition so that your code can react accordingly. Admittedly, some errors are not serious errors; they are informational and often optional. For example, the return value of printf is very rarely checked; if the value is positive, then the return indicates the number of characters printed. If it's –1, then an error occurred. Frankly, for most code, it's not a big issue, though if you've redirected stdout to a device of some sort, failure to check this exact error can result in a serious bug, which happened to hit a team one of the authors previously worked with.

For some code, the return value really does matter. For example Windows includes many impersonation functions, such as ImpersonateSelf(), ImpersonateLogonUser(), and SetThreadToken(). If these fail for any reason, then the impersonation failed and the token still has the identity associated with the process token. This could potentially lead to a privilege elevation bug if the process is running as an elevated identity, such as Local System.

Then there's file I/O. If you call a function like fopen(), and it fails (access denied, file locked, or no file), and you don't handle the error, subsequent calls to fwrite() or fread() fail too. And if you read some data and deference the data, the application will probably crash.

Languages like Java try to force the programmer to deal with errors by checking to ensure they catch exceptions at compile time (or, at least, delegate responsibility for catching the exception to the caller). There are some exceptions, however, that can be thrown from so many parts of the program that Java doesn't require they be caught, particularly the NullPointerException. This is a pretty unfortunate issue, since the exception getting thrown is usually indicative of a logic error; meaning that, if the exception does get thrown, it is really difficult to recover properly, even if you are catching it.

Even for the errors Java does force the programmer to catch, the language doesn't force them to be handled in a reasonable manner. A common technique for circumventing the compiler is to abort the program without trying to recover, which is still a denial of service problem. Even worse, but sadly much more common, is to add an empty exception handler, thus propagating the error.

## Misinterpreting Errors

Some functions are just weird, take recv(), which can return three values. Upon successful completion, recv() returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recv() returns 0. Otherwise, –1 is returned and errno is set to indicate the error. The C realloc() function is similar; it does not behave the same way as malloc() or calloc(), viz:

For malloc(), if the size argument is 0, malloc() allocates a zero-length item returns a valid pointer to that item. If the size argument is >0 and there isn't enough memory available, malloc() returns NULL.

For realloc(), if the size argument is zero, then the block pointed to by the memblock argument is freed and the return value is NULL. If the size argument is >0 and there isn't enough memory available, realloc() returns NULL.

So realloc() can return NULL in two distinct cases.

One final example is fgets(), which returns NULL if there's an error or if the code is at the end of the file. You have to use feof()/ferror() to tell the difference.

It is this kind of inconsistency that makes it easy to misinterpret errors and leads to bugs, sometimes bugs that are hard to spot. Attackers often take advantage of small coding errors like this.

## Using Useless Return Values

Some of the C standard run-time functions are simply dangerous—for example, strncpy(), which returns no useful value, just a pointer to the destination buffer, regardless of the state of the destination buffer. If the call leads to a buffer overrun, the return value points to the start of the overflowed buffer! If you ever needed more ammunition against using these dreaded C run-time functions, this is it!

## Using Non-Error Return Values

An example of this is the MulDiv() function found on Windows operating systems. The function has been around a long time; it was meant to allow a programmer to do a little bit of 64-bit math before there was support for 64-bit integers. The function is equivalent to writing

```
int result = ((long long)x * (long long)y)/z;
```

This allows the multiplication to overflow harmlessly if the divisor brings the result back into the range supported by a 32-bit signed integer. The problem is that the function returns –1 on error, which is a perfectly acceptable result for many inputs.

## Sinful C/C++

In the code sample that follows, the developer is checking the return from a function that yields a completely useless value—the return from strncpy() is a pointer to the start of the destination buffer. It's of little use, but it allows chaining of function calls—at least that was the original intent in C. Assuming, of course, there is no buffer overrun along the way!

```
char dest[19];
char *p = strncpy(dest, szSomeLongDataFromAHax0r,19);
if (p) {
    // everything worked fine, party on dest or p
}
```

The variable p points to the start of dest, regardless of the outcome of strncpy(), which, by the way will not terminate the string if the source data is equal to, or longer than, dest. Looking at this code, it looks like the developer doesn't understand the return value from strncpy; she's expecting a NULL on error. Oops!

The following example is common also. Sure, the code checks for the return value from a function, but only in an assert, which goes away once you no longer use the debug option. There is no validity checking for the incoming function arguments, but that's another issue altogether.

```
DWORD OpenFileContents(char *szFilename) {
    assert(szFilename != NULL);
    assert(strlen(szFilename) > 3);
    FILE *f = fopen(szFilename,"r");
    assert(f);

    // Do work on the file

    return 1;
}
```

## Sinful C/C++ on Windows

As we mentioned earlier, Windows includes impersonation functions that may fail. In fact, since the release of Windows Server 2003 in 2003, a new privilege was added to the OS to make impersonation a privilege granted only to specific accounts, such as service accounts (local system, local service, and network service) and administrators. That simply means your code could fail when calling an impersonation function, as shown:

```
ImpersonateNamedPipeClient(hPipe);
DeleteFile(szFileName);
RevertToSelf();
```

The problem here is if the process is running as Local System, and the user calling this code is simply a low-privileged user, the call to DeleteFile() may fail because the user does not have access to the file, which is what you would probably expect. However, if the impersonation function fails, the thread is still executing in the context of the process, Local System, which probably can delete the file! Oh no, a low-privileged user just deleted the file!

### Related Sins

There is a class of sins somewhat related to error handling, and those are exception handling sins; most notably catching all exceptions and catching the incorrect exceptions.

## SPOTTING THE SIN PATTERN

There is really no way to define the sin pattern easily. A code review is by far the most efficient way to spot these.

## SPOTTING THE SIN DURING CODE REVIEW

As this is such a broad bug type, you should verify the correctness of all functions that do not check the return value from functions with a non-void return type. In the case of Windows, this is especially true for all impersonation functions, including RevertToSelf() and SetThreadToken().

## TESTING TECHNIQUES TO FIND THE SIN

As noted earlier, the best way to find the sin is through code review. Testing is pretty difficult, because it assumes you can drive functions to fail systematically. From a cost effectiveness and human effort perspective, code review is the cheapest and most effective remedy.

## EXAMPLE SIN

The following entries in Common Vulnerabilities and Exposures (CVE) at http://cve.mitre.org/ are examples of this sin.

### CVE-2007-3798 tcpdump print-bgp.c Buffer Overflow Vulnerability

This buffer overrun bug was caused by incorrectly calculating the buffer size from calls to snprintf() because the function can return –1 when the buffer overflows when using some older version of the function, such as the implementation in glibc 2.0.

### CVE-2004-0077 Linux Kernel do_mremap

This is one of the most famous "forgot to check the return value" bug in recent history because many Internet-connected Linux machines were compromised through this bug. There's a great write-up by the finders, and sample exploit code at http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt.

**NOTE** There were a cluster of Linux Kernel memory manager security bugs in late 2003 and early 2004, including two bugs in this area, so do not confuse this bug with the other remap bug: CVE-2003-0985.

## REDEMPTION STEPS

The only real redemption step is to make sure you check return values when appropriate.

### C/C++ Redemption

In the code that follows, rather than check just a bunch of asserts, we're going to check all arguments coming into the code, and then handle the return from fopen() appropriately.

The guideline for using asserts is they should only check for conditions that should never happen.

```
DWORD OpenFileContents(char *szFilename) {
    if (szFilename == NULL || strlen(szFile) <= 3)
        return ERROR_BAD_ARGUMENTS;
    FILE *f = fopen(szFilename,"r");
    if (f == NULL)
        return ERROR_FILE_NOT_FOUND;

    // Do work on the file

    return 1;
```

#### C/C++ When Using Microsoft Visual C++

Microsoft also added a code annotation that helps enforce return checking on various functions such as impersonation functions. For example, the Foo() function in the following code must always have its return checked.

```
_Check_return_  bool Foo() {
    // do work
}
```

If the return from Foo() is not checked anywhere in the code, the following warning is issued:

```
warning C6031: Return value ignored: 'Foo'
```

## OTHER RESOURCES

- *Code Complete, Second Edition* by Steve McConnell (Microsoft Press, 2004), Chapter 8, "Defensive Programming"
- Linux Kernel mremap() Missing Return Value Checking Privilege Escalation: www.osvdb.org/displayvuln.php?osvdb_id=3986

## SUMMARY

- **Do** check the return value of every security-related function.
- **Do** check the return value of every function that changes a user setting or a machine-wide setting.
- **Do** make every attempt to recover from error conditions gracefully, to help avoid denial of service problems.
- **Consider** using code annotations if they are available, for example in Microsoft Visual C++.
- **Do not** rely on error checking solely using assert().
- **Do not** leak error information to untrusted users.