replayed by the attacker. For example, the following C# code hashes a username and password and uses the result as a key in a HTTP field to identify the user:

```
SHA256Managed s = new SHA256Managed();
byte [] h = s.ComputeHash(UTF8Encoding.UTF8.GetBytes(uid + ":" + pwd));
h = s.ComputeHash(h);
string b64 = Convert.ToBase64String(h); // base64 result
```

Or, similar code in JavaScript (from HTML or ASP) calls CAPICOM on Windows:

```
// Hex hash result
var oHash = new ActiveXObject("CAPICOM.HashedData");
oHash.Algorithm = 0;
oHash.Hash(uid + ":" + pwd);
oHash.Hash(oHash.Value);
var b64 = oHash.Value; // Hex result
```

Or, similar code in Perl also hashes the user's name and password:

```
use Digest::SHA1 qw(sha1 sha1_base64);
my $s = $uid . ":" . $pwd;
my $b64 = sha1_base64(sha1($s)); # base64 result
```

Note that all these examples hash the hash of the concatenated string to mitigate a vulnerability called *length extension attacks.* An explanation of the vulnerability is outside the scope of this book, but for all practical uses, don't just hash the concatenated data, do one of the following:

```
Result = H(data1, H(data2))
```

or

```
Result = H(H(data1 CONCAT data2))
```

This issue is covered in a little more detail in Sin 21, "Using the Wrong Cryptography."

But even code that uses sound cryptographic defenses could be vulnerable to attack! Imagine a username and password hashes down to "xE/f1/XKonG+/XFyq+Pg4FXjo7g=" and you tack that onto the URL as a "verifier" once the username and password have been verified. All an attacker need do is view the hash and replay it. The attacker doesn't need to view the password! All that fancy-schmancy crypto bought you nothing! You can fix this with channel encryption technology like SSL, TLS, or IPSec.

Attacker Predicts the Data

In this scenario, a user connects with a username and password, possibly over SSL/TLS, and then your server code verifies the account information and generates an auto-incrementing value to represent that user. Every interaction by that user uses the value to identify them without requiring the server to go through the authentication steps. This can be attacked easily over SSL/TLS. Here's how: A valid but malicious user connects to the server and provides his valid credentials. He gets an identifier value, 7625, back from the server. This value might be in the form of a URL or a cookie. He then closes the browser and tries again with the same valid username and password. This time he gets the value 7627 back. It looks like this is an incrementing value, and someone else possibly logged on between the first user's two logons. Now all the attacker need do to hijack the other user's session is connect (over SSL/TLS!), setting the connection identifier to 7626. Encryption technologies don't help protect against predictability like this. You could set the connection identifier using cryptographically random numbers, using code like this JavaScript and CAPICOM:

```
var oRNG = new ActiveXObject("CAPICOM.Utilities");
var rng = oRNG.GetRandom(32,0);
```

NOTE CAPICOM calls into the CryptGenRandom function on Windows.

Or PHP on Linux or UNIX (assuming the operating system supports /dev/random or /dev/urandom):

```
// @ before to prevent fopen from dumping too much info to the user
$hrng = @fopen("/dev/urandom","r");
if ($hrng) {
    $rng = base64_encode(fread($hrng,32));
    fclose($hrng);
}
```

Or in Java:

```
try {
    SecureRandom rng = SecureRandom.getInstance("SHA1PRNG");
    byte b[] = new byte[32];
    rng.nextBytes(b);
} catch(NoSuchAlgorithmException e) {
    // Handle exception
}
```

Or in VB.Net:

```
Dim rng As New RNGCryptoServiceProvider()
Dim b(32) As Byte
rng.GetBytes(b)
```

NOTE The default implementation of Java's SecureRandom has a very small entropy pool. It may be fine to use for session management and identity in a web application, but it is probably not good enough for long-lived keys.

All this being said, there is still one potential problem with using unpredictable random numbers: if the attacker can view the data, the attacker can simply view the random value and then replay it! At this point, you may want to consider using channel encryption, such as SSL/TLS. Again, it depends on the threats that concern you.

Attacker Changes the Data

Finally, let's assume you're not really worried about an attacker viewing the data, but you are worried about an attacker changing valid data. This is the "hidden form field with the price embedded" problem. If you need to support this scenario, you can place a message authentication code (MAC) as a form field entry; and if the MAC returned from the browser fails to match the MAC you sent, or the MAC is missing, then you know the data has been changed. Think of a MAC as a hash that includes a secret key as well as data you would normally hash. The most commonly used MAC is the keyed-hash message authentication code (HMAC), so from now on, we'll just use the term HMAC. So for a form, you would concatenate all the hidden text in the form (or any fields you want to protect), and hash this data with a key held at the server. In C#, the code could look like this:

```
HMACSHA256 hmac = new HMACSHA256(key);
byte[] data = UTF8Encoding.UTF8.GetBytes(formdata);
string result = Convert.ToBase64String(hmac.ComputeHash(data));
```

Or in Perl:

```
use strict;
use Digest::HMAC_SHA1;

my $hmac = Digest::HMAC_SHA1->new($key);
$hmac->add($formdata);
my $result = $hmac->b64digest;
```

PHP does not have an HMAC function, but the PHP Extension and Application Repository (PEAR) does. (See the section "Other Resources" for a link to the code.)

The result of the HMAC could then be added to the hidden form, viz:

```
<INPUT TYPE = HIDDEN NAME = "HMAC" VALUE = "X8lbKBNG9cVVeF9+9rtB7ewRMbs">
```

When your code receives the hidden HMAC form field, the server code can verify the form entries have not been tampered with by the repeating the concatenation and hash steps.

Don't use a hash for this work. Use an HMAC because a hash can be recomputed by the attacker; an HMAC cannot unless the attacker has the secret key stored at the server.

EXTRA DEFENSIVE MEASURES

There are no extra defensive measures to take.

OTHER RESOURCES

- Common Weakness Enumeration: http://cwe.mitre.org/
- W3C HTML Hidden Field specification: www.w3.org/TR/REC-html32#fields
- *Practical Cryptography* by Niels Ferguson and Bruce Schneier (Wiley, 1995), §6.3 "Weaknesses of Hash Functions"
- PEAR HMAC: http://pear.php.net/package/Crypt_HMAC
- "Hold Your Sessions: An Attack on Java Session-Id Generation" by Zvi Gutterman and Dahlia Malkhi: http://research.microsoft.com/~dalia/pubs/GM05.pdf

SUMMARY

- **Do** test all web input, including forms and cookies with malicious input.
- **Do** understand the strengths and weaknesses of your designs if you are not using cryptographic primitives to redeem yourself.
- **Do not** embed confidential data in any HTTP or HTML construct, such as the URL, cookie, or form, if the channel is not secured using an encryption technology such as SSL, TLS, or IPSec, or it uses application-level cryptographic defenses.
- **Do not** trust any data, confidential or not, in a web form, because malicious users can easily change the data to any value they like, regardless of SSL use or not.

- **Do not** use HTTP referer [sic] headers as an authentication method.
- **Do not** use predictable data for authentication tokens.
- **Do not** think the application is safe just because you plan to use cryptography; attackers will attack the system in other ways. For example, attackers won't attempt to guess cryptographically random numbers; they'll try to view them.

PART II

IMPLEMENTATION SINS

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.