Abstract:

Buffer overflow exploits, or “stack smashings”, are among the most common attacks used against computer systems today. They are easy to implement and allow malicious code to execute with administrator privileges on the target system. These combined factors make buffer overflow attacks a very real concern for computer systems today. This paper will discuss the details of what a stack-based buffer overflow is and how the overflow is "exploited" in the form of a malicious attack against a computer system. It will then take a look at what the overflow looks like using code samples and a view of the program stack. This paper will also provide a brief discussion of how these attacks may be prevented and a look at some of the notable buffer overflow exploits to date, including the Microsoft - AOL Instant Messaging "War".

Outline

1 Introduction to Buffer Overflows

1.1 What is a buffer overflow?

"Buffer overflows constitute the largest single threat to enterprises today" (McAfee). This is due in part to the fact that they are extremely common and extremely easy to exploit. What is a buffer overflow? Consider the scenario where a program is using an array of characters as a local variable. This array is stored on the program stack and is allocated some finite space. An overflow occurs when data greater than the size allocated is placed into the array. In C, this is a common problem because there is no automatic bounds-checking on arrays. Moreover, several of the library functions provided in the standard C library are unsafe and allow buffer overflow conditions to persist (Baratloo). These functions include but are not limited to strcat(), strcpy(), gets(), sprintf(), and vsprintf().

1.2 What is the program stack and how is it used to exploit a buffer?

When the buffer overflows, the data has to go somewhere. Where does it go? The program that is currently executing is using the current frame on the program stack. The stack stores data like local variables, the base pointer of the current frame, and the calling program's return address. If the overflow of a string buffer is large enough, the overflowing data will overwrite the base pointer of the current frame and continue overwriting the return address on the caller frame. If this situation occurs randomly, then the result will be a segmentation fault (on Unix) and the program will simply terminate (Wikipedia). However, this overflow also provides the opportunity for a hacker to intentionally overwrite the original return address of a program with a call to a malicious piece of code.

How does the malicious code get to a place where it can be called by the "new" return address? A common way introduce the malicious code into the target environment is to simply pass that code along into the buffer that is overflowing. The return address would then simply point back into the buffer to call the malicious code (Adelph1). The trick to finding the address of the buffer in the stack lies in the fact that for every program, the stack will start at the same address. The hacker can print the value of the stack pointer to find its location and try to guess from there where the buffer is located (Adelph1). This is definitely not efficient. For the malicious code to execute, the return call has to point to it exactly. A way to increase the chances of overwriting the return address with the desired procedure call is to fill the first part of the buffer with "nop" instructions, then include the shellcode, then end the data string with many entries of the call to the address of the malicious code in the buffer (Adelph1).

Clearly, a buffer overflow attack can neatly achieve two goals at the same time: it can introduce attack code into a currently running process and it can force the currently running process to actually execute that attack code (Baratloo). The malicious code, itself, is typically some form of shellcode and may run with administrator privileges and cause great harm to the system and its resources (MacAfee).

2 Examples

2.1 What a buffer overflow looks like

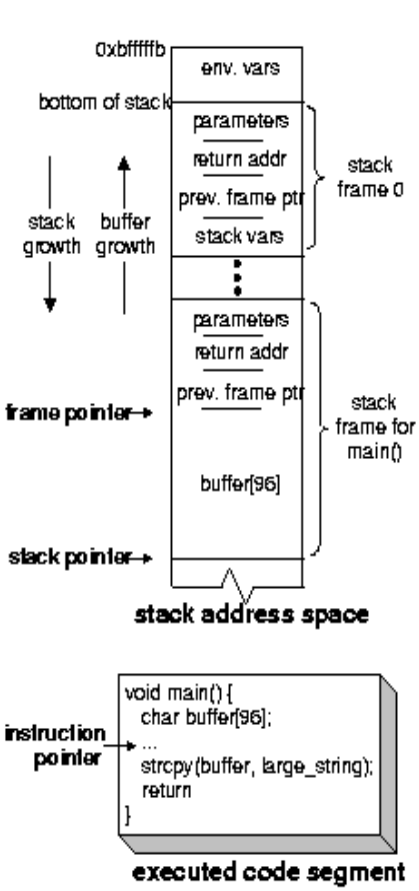
Adelph1 provides the following example of a program which has a buffer overflow error:

```
void function (char *str) {
  char buffer[16];
  strcpy(buffer, str);
}

void main() {
  char large_string[256];
  int i;

  for ( i=0; i<255; i++)
        large_string[i] = 'A';

  function(large_string);
}
```

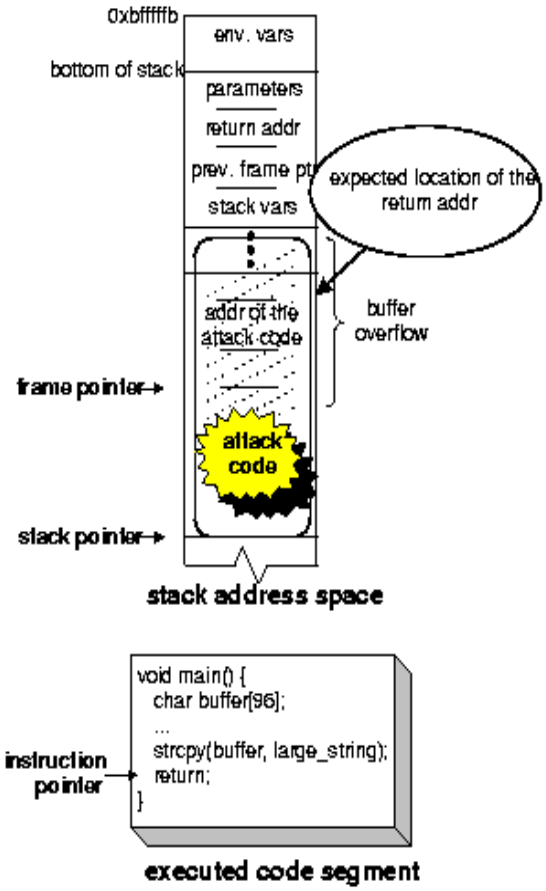
After filling large_string with 255 'A's, the entire contents of large_string is copied using strcpy() (one of the "unsafe" functions which has no inherent bounds checking) into buffer[]. Unfortunately, buffer is only 16 characters long and so clearly not big enough to hold all of the items in large_string. Thus, a buffer overflow occurs and there is a segmentation fault. In effect, since buffer[] can hold 16 characters, the remaining 250 characters overflow on the stack and overwrite the next 250 bytes (including the return address) with a series of 'A's, or 0x41.

2.2 How a program stack is manipulated during a buffer overflow

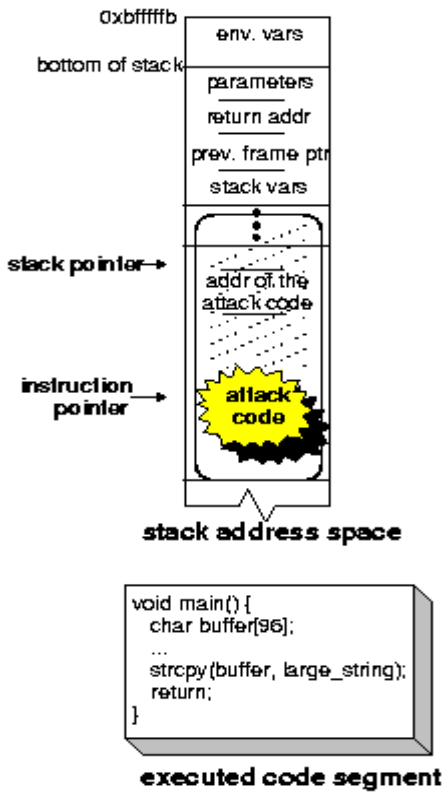
"A Process Undergoing a Stack Smashing Attack" (Baratloo)



(a) Before the attack



(b) After attack code is injected



(c) Executing the attack code

In step (a) we see the current frame, which has allocated a buffer to hold 96 characters. In addition, we can see that the calling frame is also on the stack with its return address.
In step (b) the actual smashing of the stack occurs. In this example, large_string actually contains 128 characters. Large_string contains the malicious code and when it is copied using the unsafe strcpy() function, we see that the end of the buffer is reached and the data overflows into the rest of the stack to overwrite the end of the current frame and overwrite the return address of the calling program.
Lastly, when the program returns in step (c), the attack code is called and executed.

3 Prevention

3.1 How to prevent buffer overflow exploits

3.1.1 As a programmer

Obviously, the choice of programming language will dictate some of a system's vulnerability to buffer overflow attacks. Languages such as Java and Pascal include bounds checking automatically, whereas C and C++ do not. If the use of C is mandated, or already in place, then the programmer can choose to manually ensure that the bounds checking is done

before operations with arrays and strings are called. In addition, the programmer can avoid references to “unsafe” functions in the C standard library, and use others, like strncpy(), instead.

3.1.2 Tools

3.1.2.1 Unix/Linux Based

One way to protect from buffer overflow attacks is to “randomize the address space” (Wikipedia) so that the memory is not writable and not executable. In other words, a non-executable stack will thwart an attempt made to execute shellcode that has been placed on the stack. PaX and exec-shield are both patches for the Linux Kernel which provide these capabilities. Note that these patches will not prevent the overflow from occurring, but may prevent the subsequent execution of malicious code.

Sun’s Sparc and the newer 64-bit Intel processors flag some areas of memory with a special “NX bit” to prevent execution. Intel calls their solution “XD” for “eXecute Disabled” (Wikipedia).

3.1.2.2 Windows Based

Microsoft and other 3rd party providers have created “Executable Space Protection” software aimed at preventing the execution of malicious code (Wikipedia). Specifically, Microsoft has introduced DEP (Data Execution Prevention) which protects against the SEH exploit, i.e. where a buffer is overflowed to access an exception handler’s address space. DEP is built on top of Intel/AMD’s extension of PAE (Page Addressing Extensions), which used the special NX bit to mark non-executable regions of data (Wikipedia).

Microsoft has an additional mechanism for Windows Server 2003 which places a “canary” on the stack which can be checked and verified to see if a change was made to the stack.

3.1.2.3 Other

Intrusion Detection Software can be configured to block anything which contains a large number of NOP instructions, although this is generally unreliable (Wikipedia). Specific to stack smashing are products which check explicitly to see whether or not the stack has been altered. StackGuard and ProPolice are two examples and are extensions to gcc (Wikipedia).

Janus is a run-time environment which creates a restricted sand-box for each application based on the access it determines that the application needs (Baratloo). Unfortunately, this product does not work well with applications that genuinely need administrative-like privileges. McAfee Entercept also provides security against buffer overflow exploits.

Because the buffer overflow scenario is so common and easily exploited, the best hope is to stay current with software patches which block the exploits as they are discovered

(Mixter). No software protection will be perfect, since they, too, are by definition 'software' and therefore also prone to vulnerability and attack (Mixter)!

4 Notable Exploits

4.1 A look at some of the notable buffer overflow exploits to date

4.1.1 Timeline of exploits

The Morris Worm in 1988 was perhaps the earliest buffer overflow exploit to attract attention. This famous internet worm exploited a buffer overflow in the FINGER command which allowed it to execute code on the host system (Bryant 209). Once the worm started running it would replicate itself and basically "take over". After this outbreak, buffer overflow exploits were somewhat ignored until 1995 when Thomas Lopatic published his findings regarding buffer overflow exploits on the Bugtraq security mailing list (Wikipedia). A year later Elias Levy (aka Adelph One) published his cook-book like approach to creating and exploiting buffer overflows in his article, "Smashing the Stack for Fun and Profit" (Wikipedia). More recently in 2003, Microsoft SQL Server 2000 was attacked with the SQLSlammer worm which also exploited a buffer overflow.

4.1.2 The AOL/Microsoft Instant Messaging War

One of the most famous instances of the buffer overflow exploit was during the battle between Microsoft and AOL. According to Bryant, Microsoft introduced their Instant Messaging system in 1999 which allowed it's users to communicate, or "chat", with AOL instant messager users (210). Shortly thereafter, the Microsoft users were denied chatting ability with AOL users. AOL was essentially exploiting their own code by viewing select memory locations and returning them to the server. The AOL client actually sends back too much information to overflow its own buffer (Nelson). If the client failed to do this or failed to contain the correct information, the service was denied because the client was not an AOL customer (Bryant 210). If Microsoft were to uncover the appropriate addresses in memory to use, then AOL would simply use different address locations.

This scenario was uncovered by an employee of Microsoft who sent an email under the guise of a "Phil Bucking" to a security expert, Richard Smith (Bryant 210). The controversy arose from the fact that while AOL had the right to offer exclusive service to their customers, their method of enforcing the customers-only policy was suspect. It basically opened the door to attack code that could have been placed on user systems. Ultimately, AOL never admitted to using any sort of exploitation (Bryant 211).

Works Cited

Adelph One. “Smashing the Stack for Fun and Profit”. Phrack. 15 Nov. 2005. http://www.phrack.org/phrack/49/P49-14.

Baratloo, Arash, Navjot Singh, and Timothy Tsai. “Transparent Run-Time Defense Against Stack Smashing Attacks”. 16 Nov. 2005. http://www.research.avayalabs.com/project/libsafe/doc/usenix00/paper.html#/sec:introduction

Bryant, Randall, and David R. O’Hallaron. Computer Systems: A Programmer’s Perspective. Upper Saddle River: Pearson Education, Inc., 2003.

“Buffer overflow”. Wikipedia. 16 Nov. 2005. <http://en.wikipedia.org/wiki/Buffer_overflows>.

“Buffer Overflow Exploits: The Why and How”. McAfee.com 15 Nov. 2005. http://www.mcafee.com.

Mixter. “Writing buffer overflow exploits – a tutorial for beginners”. 15 Nov. 2005. <http://mixter.void.ru/exploit.html>.

Nelson, Matthew. “AOL’s AIM gets bugged”. InfoWorld 20 Aug. 1999. CNN.com. 15 Nov. 2005. <http://www.cnn.com/TECH/computing/9908/20/aolbug.idg/>.

“Stack Smashing”. 15 Nov. 2005. <http://homepages.wmich.edu/~m1tahir/stacksmaching.html>.