avoiding buffer overflows and related problems

by Steven Alexander

Steven programs in C/C++, assembly languages and UniBASIC. He has experience with UNIX and Windows security, firewalls, and IDSes.

alexander.s@mccd.edu

Attackers use buffer overflows and format string vulnerabilities to manipulate software both to gain access to and to raise privilege on computer systems. This paper details the means by which these vulnerabilities can be prevented in C programs. This introduction to current exploitation techniques will motivate and explicate why precautions are necessary.

Buffer overflows are nothing new. One of the means by which the Morris worm spread was a buffer overflow in fingerd. The technique didn't become popular, however, until the release of two papers [3, 4] that detailed discovery and exploitation of these vulnerabilities. A number of defenses are covered in [2].

Over time, different techniques and tools for preventing the exploitation of these vulnerabilities have been proposed and, time and time again, defeated. The problem lies in the fact that C allows low-level control with very little abstraction from the machine. Proposed solutions such as StackGuard, StackShield [11], and PaX [25] are not fully able to prevent exploitation, since their protection mechanisms are applied to poorly written code after its creation. These programs do complicate the job of the attacker and are a useful stopgap for preventing the exploitation of the occasional bug left by a security-conscious programmer, but software written without security in mind will continue to be victimized.

The goal of this paper is to introduce the reader to the concepts behind various buffer overflow techniques and the techniques required to prevent them. Format strings are also discussed, because they use similar methods for exploiting software. Some examples are given using assembly language for 32-bit Intel processors. Most readers should be able to follow along without previous assembly language experience.

This paper examines exploits from the perspective of a UNIX-based operating system; Windows exploitation is covered in [23] and [24]. Readers used to programming in C on either platform should have no trouble with the discussion.

Buffer overflows are not the only security problems that exist in software. The interested reader should also study [1] for an overview of other security considerations. Subsequent sections of this paper describe the concepts behind buffer overflow and format string attacks. The material on exploitation is simplified to introduce the reader to problems of which she should be aware without requiring her to acquire an expert knowledge of the techniques. Serious readers will want to digest the papers listed in the references. They can be read in roughly the order they are listed.

Exploiting a Buffer Overflow

If you're already familiar with the concepts behind a buffer overflow exploit, you can skip this section. The concepts in the section are more fully described in [3] and [4].

Suppose you have a program that looks like this:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
      char buf[256];
      if (argc < 2) {
```

```
        printf("Oops.\n");
        return -1;
    }
    strcpy(buf, argv[1]);
    return 0;
}
```

Example 1. Simple exploitable program

This program is trivially vulnerable to a buffer overflow. The strcpy function performs no bounds checking on buf and will blindly copy argv[1] until the program crashes or strcpy encounters a null character '\0'.

Before entering main, the operating system exec call pushes the return instruction pointer onto the stack. Upon entering main, the frame pointer is pushed onto the stack. Then the stack pointer is copied over the frame pointer to mark the local stack frame. Finally, the stack pointer is decremented to make room for local variables, growing "downward." The function prologue for main typically looks like this in assembly language:

```
pushl %ebp          ; save frame pointer
movl %esp,%ebp      ; create new frame
subl %esp, $0x100   ; make room for local vars
```

Example 2. Typical function invocation prologue

The stack should now look like Figure 1.

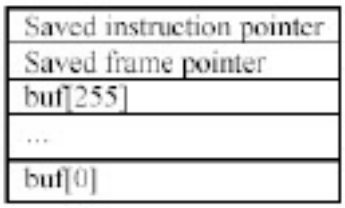

Figure 1

The function epilogue (executed as the function returns) consists of popping the saved frame pointer from the stack and executing a return instruction. Intel machines use the ret instruction to tell the processor to take the next value from the stack and move it into the program counter. Program execution then resumes at whatever address that value contains.

An attacker can carefully craft input to cause this program to execute a command shell (or anything else on the system). Here is one method to do this:

First, an attacker writes a code snippet to execute a command shell (this is called "shellcode"). This is normally done by compiling something similar to the following:

```
#include <stdio.h>

void main() { system ("/bin/sh"); }
```

Example 3. Short program to run a shell

The exec functions are also commonly used. This code is compiled to assembly language for easy modification. It is then changed to reduce code size, to remove null bytes, and to ensure that the string "/bin/sh" can be stored in a location that the attacker has permission to write to. Luckily for the attacker, it is easy to find already written shellcode on the Internet for most operating systems. Shellcode can be a lot more complicated than the previous example if the vulnerable program has enough room to store it. When attacking network software, shellcode is used that can bind "/bin/sh" (or cmd.exe for Windows) to a network port. After modification, the shellcode is assembled into machine-executable instructions. It is beyond the scope of this paper to go further into the details of writing shellcode.

The next step is to find the address of buf[0] (see Example 1). Sometimes, this is found using a debugger like gdb. Other times, it can be approximated, as is detailed in Aleph One's paper [4].

Some of the most dangerous problems occur when pointers of any type can be overwritten.

Finally, argv[1] (which is strcpy'd into buf) is filled with the following:

Figure 2

The shellcode is written into buf[0]. The rest of the buffer is filled with null instructions (NOPs). On Intel's x86 processor line, a NOP is encoded as 0x90, which is the same as xchg eax, eax. Swapping the eax register with itself has, of course, no effect. The next four bytes overwrite the saved frame pointer. The last four bytes overwrite the saved instruction pointer with the address of buf[0], which contains shellcode. When the function returns, this code will be executed. When the address is not known exactly, it can be guessed by prepending the NOPs to the shellcode and attempting to point to any location within the series of NOPs. The base stack address for the system should be known, so with a 256-byte buffer, the process can be repeated using 200-byte increments from the base stack address downward. Shellcode tends to be shorter than 50 bytes.

Advanced Buffer Overflow Techniques

Many advanced buffer overflow techniques were developed to defeat protection mechanisms such as StackGuard, StackShield, and PaX. Others exploit particular situations such as a one-byte overflow. As the referenced papers show, even subtle mistakes can be exploited. The reader interested in learning how everything really works should make an attempt to read through all of the references.

Heap Overflows

Not every vulnerable program is as straightforward as the one presented in Example 1. Sometimes, the memory being written to is on the heap or in the bss segment rather than the stack. In this case, the saved instruction pointer can't be written over (not in a straightforward manner, anyway), but other important data may be vulnerable. Conover details some of the possibilities in [5]. Some of the most dangerous problems occur when pointers of any type can be overwritten. Overwriting function pointers to point at illicit code will cause that code to be executed the next time the function is called. Overwriting other pointers can sometimes cause saved instruction pointers, function pointers, or important structures like _atexit or .dtors to be overwritten by a subsequent instruction.

One of the more interesting heap overflow techniques involves overwriting the boundary tags on areas of memory used by malloc in order to cause the unlink or frontlink macros to overwrite a function pointer or a saved instruction pointer. These techniques are detailed in [16] and [17].

Techniques have also been developed to exploit C++ code [19]. All of the usual C techniques still apply. However, C++ implements virtual function pointers in order to provide classes, and these can, in some cases, be overwritten and cause other code to be executed instead.

Defeating Protection Mechanisms

StackGuard and StackShield are two tools that complicate exploitation by protecting return addresses. StackGuard works by placing a "canary" value between the saved frame and instruction pointers. If the canary is overwritten, the program will quit rather than resume execution at the saved address. StackShield works by saving the

return address in a secure location rather than the stack. These techniques were bypassed in [11]. StackGuard was patched before the publication of [11] to protect the saved address more strongly. StackGuard had previously used either a random canary value (assigned to each function at run time) or a null canary which contained string-terminating characters such as '\0' and '\n'. The new technique, proposed by Aaron Grier, saves the result of XORing the return address with the assigned random canary value. During the function epilogue, the saved value is XORed with the assigned value and compared to the return address. If the return address is not what is expected, the program exits. This can be circumvented by overwriting function pointers or entries from .dtors [20], _atexit [21], PLT, and GOT [14].

PaX is a set of kernel patches that also alleviate program exploitation. One of its features is to make stack and heap memory non-executable. A non-executable stack patch for Linux was first released by Solar Designer but was later circumvented by Solar Designer [9] and Rafal Wojtczuk [8] using a technique called return-into-libc that is used when an attacker cannot provide his own code to be executed (as is the case with a non-executable stack). Instead, the attacker finds the address of a call to system and arranges to have the string "/bin/sh" on the stack. An improvement of this attack that uses mmap and strcpy to set up its own executable area of memory was used against PaX [10].

PaX also uses a feature called Address Space Layout Randomization [25]. With ASLR, entropy is introduced into stack and library function addresses. Reference [12] shows that programs can be exploited with PaX ASLR running. ASLR can in some circumstances be brute-forced; this will generate a lot of noise, as was intended [25]. However, log files can be trimmed once root access is gained.

Subtle Mistakes

Even a one-byte overflow can be enough to exploit a program. Klog [7] shows how writing one byte past the end of a buffer can be used to overwrite the least significant byte of the saved frame pointer. In some conditions, this can be used to cause the calling function to retrieve its saved instruction pointer from the wrong location.

For example, say that function1 calls function2. The least significant byte of the saved frame pointer is overwritten in function2 and the function returns. In function1, the saved frame pointer is copied to the stack pointer. If this happens near the return of function1 (so the program doesn't crash), the instruction pointer will be retrieved from a location lower down on the stack than it should be. In some situations it is possible to force the program to retrieve its return address from user-supplied input stored on the stack. An attacker simply needs to provide an address containing shellcode she would like executed.

A program can also be exploited simply by filling a buffer without a terminating null character. Take the following snippet, for example:

```
#include<stdio.h>
void main(){
      char buf[256];
      char tmp[64];
      strncpy(tmp, argv[1], 64);
      strncpy(buf, argv[2], 256);
          . . .
```

Even a one-byte overflow can be enough to exploit a program.

Finding all of the flaws in old software can be a real headache.

Example 4. Copying supplied arguments to logically joined variables
Twitch's paper [6] describes attacks in which the input to tmp (argv[1]) is exactly the size of the buffer. In C, strings are terminated with a null character '\0'. strcpy and strncpy both terminate strings with a null character, as should be expected. The primary difference is that strncpy uses an extra argument, the maximum number of characters to copy. However, strncpy does not null-terminate a string unless the string's length is less than the provided maximum number of characters . If argv[1] in the above example is 64 characters or longer, tmp will not be null-terminated. As such, any future references to tmp that are not bounded will read past the end of tmp and into buf. This is a common mistake because programmers assume that the string is safe after using strncpy the first time.

Twitch demonstrated methods to exploit a program in which the string saved "lower" on the stack (in this case tmp) was later copied using an unbounded copy. In this situation, the contents of the string above it (in this case buf) can be used to copy over other data, even a saved instruction pointer.

Format String Vulnerabilities

Format string exploits are deadly but easy to prevent. Consider the following program:

```
#include <stdio.h>
void main() {
      char buf[512];
      char tmp[512];
      read(0, buf, 512);
      sprintf(tmp, buf);
}
```

Example 5. sprintf without a format string
The last line of code should read sprintf(tmp, "%s", buf);. Unfortunately, the format specifier was omitted. As a result, an attacker can provide his own format specifiers in their input. An exploit is possible because of the %n specifier.

The %n specifier saves the number of bytes written so far to the memory address pointed to by the corresponding argument. Programs are exploited in this manner by writing to the saved instruction pointer (or another function pointer, _atexit, etc.). To write a 32-bit address, one or two bytes are written at a time. An attacker could for instance write to &function_pointer, &function_pointer+1, &function_pointer+2, and &function_pointer+3.

The attack isn't very complicated but takes awhile to explain. Since the aim of this paper is awareness and avoidance, see [13] for an introduction and [14] and [15] for more advanced techniques. Because of their ability to write over arbitrary locations in memory, format strings are one of the most flexible exploitation techniques. Fortunately, they are also not very common.

Writing Secure Code

The need for rigorous bounds checking should be clear by now. Fortunately, this isn't difficult when writing new software. Finding all of the flaws in old software can be a real headache, however.

Note that even an astute programmer will never write perfect code with regard to any useful metric. However, by using the following guidelines, it will be difficult to find exploitable boundary conditions in your programs. Readers should refer to [1] after

they are finished with this paper. The examples and usage below should be compared with the documentation for your system.

Integer Overflows

Care should be taken when converting from signed to unsigned integers [22]. Exploitable conditions sometimes occur because type conversion leads to integers being interpreted differently than intended. As an example, note that the signed 32-bit integer -1 equates to 0xFFFFFFFF, the maximum possible value that can be stored in an unsigned integer. When integers are converted from signed to unsigned, or vice versa, they should be checked to make sure they are still within an acceptable range of values.

The gets() Function

gets() is perhaps the most insecure function a programmer can use. It takes only one argument, a buffer pointer that is never verified for integrity. fgets should be used instead:

```
char *fgets(char *str, int size, FILE *stream);
```

fgets will read at most size-1 characters from stream. The input characters are written to str and are null-terminated. Below is a simple example of proper use:

```
#include <stdio.h>
int main() {
      char buf[256];
      fgets(buf, sizeof(buf), stdin);
      printf("%s\n", buf);
      return(0);
}
```

Example 6. Proper way to input character strings

strcpy()

strcpy has no bounds checking and should be replaced with strncpy:

```
char *strncpy(char *dst, const char *src, size_t len);
```

strncpy will only null-terminate a string if it is less than len characters in length. The following snippet is a proper use of strncpy:

```
strncpy(buf, buf_with_user_input, sizeof(buf) -1);
buf[sizeof(target) - 1] = '\0';
```

Example 7. Proper use of strncpy

strcat()

strcat also has no bounds checking; use strncat instead. Using strncat is trickier than other functions, though, because it doesn't write to the beginning of a buffer. It has this template:

```
char *strncat(char *s, const char* append, size_t count);
```

strncat appends the null-terminated string append to the null-terminated string s. It appends at most count non-null characters, then adds a terminating '\0'. The following example is a proper usage of strncat:

```
strncat(buf, "something else to say", sizeof(buf) -
   strlen(buf) - 1);
```

REFERENCES

[1] Matt Bishop, "How to Write a Setuid Program," *;login:* 12:1 (January-February 1987), pp. 5–11. *http://nob.cs.ucdavis.edu/~bishop/papers/Pdf/1987-sproglogin.pdf*

[2] Crispin Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" (2000). *http://www.immunix.org/StackGuard/discex00.pdf*

[3] Mudge, "How to Write Buffer Overflows" (October 1995). *http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html*

[4] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine* 49 (November 1996). *http://www.phrack.org/phrack/49/P49-14*

[5] Matt Conover, "w00w00 on Heap Overflows" (January 1999). *http://www.w00w00.org/files/articles/heaptut.txt*

[6] twitch, "Taking Advantage of Non-Terminated Adjacent Memory Spaces," *Phrack Magazine* 56 (May 2000). *http://www.phrack.org/phrack/56/p56-0x0e*

[7] klog, "The Frame Pointer Overwrite," *Phrack Magazine* 55 (September 1999). *http://www.phrack.org/phrack/55/P55-08*

[8] Rafal Wojtczuk, "Defeating Solar Designer's Non-Executable Stack Patch" (February 1998). *http://www.securityfocus.com/archive/1/8470*

[9] Solar Designer, "Getting Around Non-Executable Stack (and Fix)." *http://www.securityfocus.com/archive/1/7480*

[10] Nergal, "The Advanced Return-Into-Libc Exploits: PaX Case Study," *Phrack Magazine* 58 (December 2001). *http://www.phrack.org/phrack/58/p58-0x04*

[11] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack Magazine* 56 (May 2000). *http://www.phrack.org/phrack/56/p56-0x05*

[12] Anonymous, "Bypassing PaX ASLR Protection," *Phrack Magazine* 59 (July 2002). *http://www.phrack.org/phrack/59/p59-0x09*

[13] Pascal Bouchareine, "Format String Vulnerability" (July 2000). *http://www.hert.org/papers/format.html*

[14] scut, Team Teso, "Exploiting Format String Vulnerabilities" (September 2001). *http://www.team-teso.net/articles/formatstring/*

[16] Michel Kaempf, "Vudo Malloc Tricks," *Phrack Magazine* 57 (August 2001). *http://www.phrack.org/phrack/57/p57-0x0b*

[17] anonymous, "Once upon a free()," Phrack Magazine 57 (August 2001). *http://www.phrack.org/phrack/57/p57-0x0c*[16]

Example 8. Proper use of strncat

The OpenBSD project introduced two new string functions, strlcpy and strlcat, both of which require the size of the buffer to be passed to them rather than the maximum number of characters to write [18]. This eases the programmer's job. Remember, it only takes one byte to make a program exploitable. If the -1 had been forgotten in Example 8, it would be a potentially exploitable program.

One of the other advantages of the OpenBSD strlcpy function is speed. Unfortunately, strncpy zero-fills the end of a string rather than adding just a single null character. This can degrade performance when the strings being copied are significantly smaller than the buffer they are copied into (as is often the case).

sprintf()

sprintf has no bounds checking; snprintf should be used instead:

```
int snprintf(char *str, size_t size, const char *format, ...);
```

snprintf writes at most size-1 characters to str and appends a terminating '\0'. Any additional characters are discarded. snprintf can be used as follows:

```
snprintf(buf, sizeof(buf), "%s", other_buffer);
```

Example 9. Proper use of snprintf

memcpy()

A few exploits have occurred in the wild because memcpy was used improperly to copy strings. The number of bytes copied should be, at most, the size of the smaller buffer minus one. The string should be manually null-terminated. The prototype for memcpy is as follows:

```
void *memcpy(void *dst, void *src, size_t, len);
```

Proper usage for a string buffer would be:

```
maxlen = (sizeof(buf1) < sizeof(buf2) )  ? sizeof(buf1)
                                         : sizeof(buf2);
memcpy(buf1, buf2, maxlen -1);
buf1[sizeof(buf1)-1] = '\0';
```

Example 10. Proper use of memcpy

The scanf() family

The scanf family of functions has the following prototypes:

```
int scanf(const *char format, ...);
int fscanf(FILE *stream, const *char format, ...);
int sscanf(const char *str, const *char format, ...);
```

The format specifiers used with this set of functions should limit the size of the input, as in the following example:

```
char buf[64];
fscanf(stdin, "%63s", buf);
```

Example 11. Limiting input string sizes in scanf

read()

The read system call has the following prototype:

```
ssize_t read(int d, void *buf, size_t nbytes);
```

read is meant for inputting raw data; it does not null-terminate its destination buffer. If you use read to get string data, remember to null-terminate the buffer manually. The

better option for user input is usually fgets. The following example uses read correctly (for inputting string data):

```
read(0, buf, sizeof(buf)-1);
buf[sizeof(buf)-1] = '\0';
```

Example 12. Proper use of read when inputting string

Pointers

Unfortunately, it is all too common to see pointers misused:

```
void some_function(char *string) {
    char buf[256];
    int i;
    for(i=0;i<=256;i++) {
        buf[i]=string[i];
    }
}
```

Example 13. Filling a local buffer

This example will copy up to 257 bytes from string into buf and can overwrite the saved frame pointer. This is exactly the problem that klog describes in [7]. The code should read:

```
void some_function(char *string) {
    char buf[256];
    int i;
    for(i=0;i<255;i++) {
        buf[i]=string[i];
    }
    buf[255] = '\0';
}
```

Example 14. Better code for filling a buffer

Notice that <=256 was changed to <255, which copies two fewer bytes. The two-byte difference prevents overwriting the frame pointer and allows room to null-terminate the string.

Conclusion

A program can be exploited with as little as a single byte buffer overflow, a missing null, or a missing format string. Code should be carefully written and rechecked from time to time. Nobody writes perfect code, but well-written code is much harder to exploit.

Programmers should always check the bounds of the input to their programs. Strings should always be null-terminated. Format strings should always be provided.

Using PaX or StackGuard is recommended if it is available for your system. They can't fix bad code, but they will make an attacker's job more difficult.

[18] Todd C. Miller and Theo de Raadt, "strlcpy and strlcat: Consistent, Safe String Copy and Concatenation." *http://www.openbsd.org/papers/strlcpy-paper.ps*

[19] rix, "Smashing C++ VPTRS," *Phrack Magazine* 56 (May 2000). *http://www.phrack.org/phrack/56/p56-0x08*

[20] Juan Bello Rivas, "Overwriting the .dtors Section" (March 2001). *http://www.synnergy.net/papers/dtors.txt*

[21] Pascal Bouchareine, "__atexit in Memory Bugs: Specific Proof of Concept with Statically Linked Binaries and Heap Overflows." *http://community.core-sdi.com/~juliano/heap_atexit.txt*

[22] blexim, "Basic Integer Overflows," *Phrack Magazine* 60 (December 2002). *http://www.phrack.org/phrack/60/p60-0x0a.txt*

[23] dark spyrit, "Win32 Buffer Overflows," *Phrack Magazine* 55 (September 1999). *http://www.phrack.org/phrack/55/P55-15*

[24] David Litchfield, "Windows 2000 Format String Vulnerabilities" (2001). *http://community.corest.com/~juliano/win32format.doc*

[25] PaX: *http://pax.grsecurity.net/*