Securely Programming in C

Sayed Ahmed

Securely Programming in C
Sayed Jamil Ahmed
September 2002

Preface

Three men are in a car; a programmer, an engineer and a designer. As the car is about to go down a hill the brakes fail on the car and it crashes at the bottom. The men walk out of the car and discuss what happened. The designer says 'let's get the drawings of the car and analyze the design and find out the flaws in the design'. The engineer says no, no let's take a look at the wreckage and analyze each and every component to see what failed'. The programmer says 'I think we should push the car to the top of the hill and see if it happens again' •

Programmers have generally borne the brunt of criticism over security vulnerabilities. I believe that security related bugs in software are more down to education than any thing else. Security is simply not addressed at an academic level (at least wasn't when I was at University) and is seldom addressed in Commerce (according to my experience). Education is key in the quest to make software more secure.

Introduction

If one subscribes to the any one of the security advisories mailing lists such as CERT or Security focus you will see new security alerts being issued on almost a daily basis. The vast majority of these alerts are software related caused by bad coding practices or by programming errors. What is worse is that the situation doesn't seem to be getting any better. More and more security flaws are being found resulting in more and more compromised systems. The question is why? Is it laziness on the part of the programmers or a lack of education? Having been a (C) programmer for 10 years working for major international banks and other organizations and not heard about buffer overflows or even heard any of my colleagues talk about it until I entered the security field I believe it's a lack of education. It seems as though there is a divide between security practitioners who know about security issues and non security practitioners who know very little.

This paper will discuss what I feel are the main issues in secure programming in the C programming language in a UNIX environment (Buffer Overflows, Format Strings and Race Conditions), topics such as overflows are relevant in Windows too. The issues will be described so that the reader can understand the nature of the vulnerability. The mechanisms of the exploit will also be described where they have not been sufficiently covered in other papers. After discussing these vulnerabilities secure programming tips and automated tools are described.

Buffer Overflows

A buffer overflow occurs when data is written into buffer which is smaller than the data e.g. if you try and write a 10 character string into an array of 5 characters. Obviously 10 characters don't fit into an array of 5, so what happens is that 10 characters are written into the array, the first five occupy the space allocated for the array and the remaining 5 overflow the array into the adjacent memory. The contents of the adjacent memory and the amount of overflow are of significance depending on the type of overflow described below. The table below shows a simplified picture of a programs process space

Segment	**Usage**
Stack	**local variables, function arguments**
Heap	**dynamic variables**
Data/BSS	**global, static, initialised and uninitialised**

The target for overflows is usually a function or instruction pointer. Manipulating one of these would allow an attacker to run his own code (exploit). Other specific variables can also be targeted such as variables for file names or access privileges. This requires specific knowledge of what a program does and which variables are used. Variables that can be targeted depend on their location (Stack / Heap etc) as these are allocated in different ways. The Stack grows downwards from a high address to lower address, so only variables at higher address can be targeted. The Heap grows in the opposite direction so only variables at lower addresses can be targeted. The ability to run exploit code is preferred.

Exploit code is usually a set of instructions to open a command shell. This code can be stored in the buffer to be overflowed (if it is big enough), in an environment variable or can even be passed as a command line argument. The position of this code can be important. Command line arguments, environment variables, local variables and function arguments are stored on the stack. If the stack is non executable then the exploit code placed on the stack will not run. Patches are available to make these segments non executable but these are seldom applied. The privilege level of the shell depends on the program that has been exploited; if it was owned by root or was setuid root then the attacker has root privileges in the shell.

Stack Overflows

(Stack overflows have been covered extensively in many papers such as [1] & [3] so only a brief description is included here)

Since all stack variables are stored in contiguous memory locations any overflow would result in overwriting other stack variables that are above (higher memory address) than the overflowed buffer. If an attacker can figure out the position of variables on the stack he could manipulate specific variables as long as they were above (higher address) the buffer to be overflowed.

Beyond program variables are the registers. These are usually the target for attackers specifically the EIP (Extended Instruction Pointer). This pointer points to the next instruction to execute. An attacker will place his own code (exploit) at a (roughly) known address in memory and attempt to overflow the buffer with a specially crafted string to reach the EIP. The address of the exploit code is positioned in this crafted string such that it will exactly overwrite the EIP. When the program looks for the next instruction to execute it is diverted to the exploit code.

Heap Overflows

Dynamically allocated memory is stored in an area called the heap. This area is managed using tree structures and double linked lists to keep track of free and used memory. Arbitrary overflows of these areas therefore damages these structures usually resulting in a program crash. However if one has knowledge of the implementation of dynamic memory allocation and the data structures used it is possible to write an exploit to run code of an attackers choice. There are a number of possible ways to exploit dynamic memory management the example below describes how to exploit the unlinking of a *chunk* of memory in Doug Leas *malloc* library (used by glibc) to run exploit code [4].

Memory Chunks are arranged in a double linked list as shown in figure 1 below. *Boundary Tags* precede each Chunk of memory which contains the information for linked list management (sizes, forward and backward pointers). For efficiency reasons when an allocated chunk of memory is to be free'ed the next chunk is checked to see if it is unallocated, if it is then the freeing process merges the two chunks into one and adjusts the pointers accordingly to maintain the double linked list i.e. if Chunk 2 is unused when Chunk 1 is free'ed, then Chunk 1 and 2 are merged and Chunk 1 is linked to Chunk 3 and Chunk 3 linked back to Chunk 1. Chunk 2 is said to be *unlinked*. To check if a Chunk is in use or not the malloc libraries check the next Chunks SZ field. The two low order bits are used to store this information, if they are not set then the previous Chunk is free.

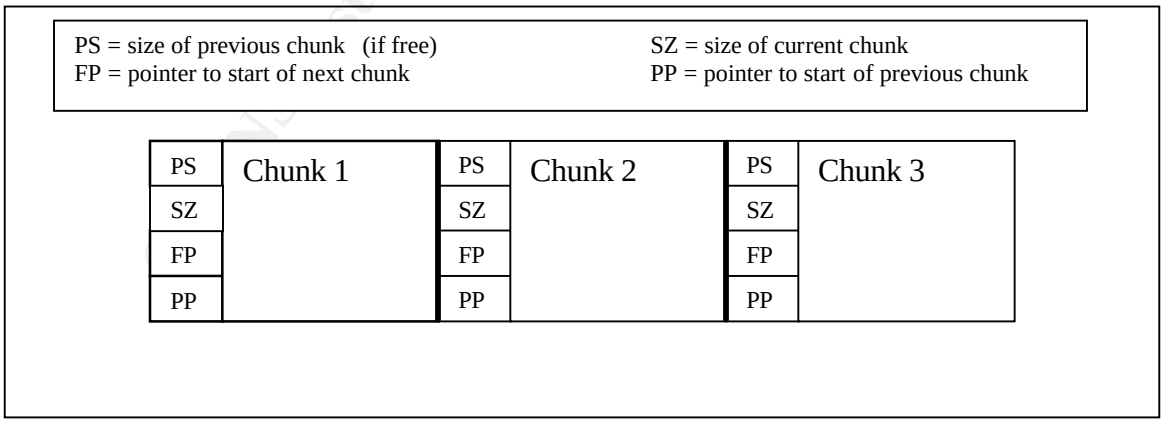

Figure 1: Double linked list of memory Chunks

For example if the buffer we can overflow is allocated in Chunk 1 then we have ensure that Chunk 2 is free so that it can be unlinked and merged with Chunk 1. To ensure that

this is the case we have to fool the unlinking process into believing that Chunk 2 is free. This information is stored in Chunk 3s SZ field. Rather than following the forward pointer on Chunk 2 the libraries use the SZ field on Chunk 2 to calculate an offset to Chunk 3. The trick is to use our overflow string (from Chunk 1) to overwrite the boundary tags of Chunk 2 to place a negative offset (-4) in the SZ field, thus fooling the libraries into thinking that Chunk 3 starts 4 bytes before Chunk 2. When the SZ field of Chunk 3 is checked for the availability of Chunk 2 it actually reads the PS field of Chunk 2. Our overflow string has to ensure that the 2 low order bytes of this field are zero so that unlinking can take place. To actually unlink Chunk 2, FP of Chunk1 and PP of Chunk 3 must be made to point to each other.
This is done using the FP and PP pointers in Chunk 2 only. The following two copies occur

1. Copy PP of Chunk2 to FP + 12 (this changes Chunk3 PP to point to Chunk 1)
2. Copy FP of Chunk2 to PP + 8 (this changes Chunk 1 FP to point to Chunk 3)

Therefore FP and PP of Chunk 2 are used to calculate the address to change, if these are manipulated (by the original overflow from Chunk 1) such that FP is the address of a function pointer and PP is the address of some exploit code then an attacker can run arbitrary exploit code. This would mean that the two copies above now do the following

1. Copies the address of the exploit code to the address of a function pointer + 12 bytes.
2. Copies the address of the function pointer to the address of the exploit code + 8 bytes.

Since the address of the function pointer is adjusted by twelve bytes the actual address stored in FP must be twelve bytes short to compensate. Also the second copy would write an address at the start of the exploit code which would render it useless. The exploit code must be written such that it jumps over this address and into the real code.

So what about the actual values (addresses) for the function pointer and the exploit code? An attacker has many choices for both, exploit code can be placed in an environment variable, command line argument or in a program buffer if there is space the function pointer can also be taken from the global offset table which contains pointers to the standard library functions such as *free()*. If the address of *free()* is used then once the overflow occurs any subsequent call to *free()* would run the exploit code not the function *free()*.

The overflow string must be constructed as follows

- Overflow to end of Chunk 1 (to reach the boundary tags of Chunk 2)
- zero low order bytes (PS of Chunk 2 - impersonating SZ of Chunk 3)
- -4 (SZ of Chunk 2 - spoofs start address of Chunk 3)
- address of free() – 12 bytes (FP of Chunk 2 - address of function pointer)
- address of exploit code (PP of Chunk 2 - address of exploit code)

Data/BSS Overflows

As with stack overflows program variables can be manipulated since one variable would overflow into another. However the Data/BSS segments grow in the opposite direction to the stack so target variables would have be below (lower memory address) that the buffer to be overflowed. Target variables could be of any type such as integers or strings or even function pointers, in which case an attacker would be able to run arbitrary code as with stack and heap overflows.

Format Strings

Format strings define the format and types of program variables that are substituted into an input or output string. These are used by many functions such as printf(), sprintf() and scanf(). Consider the following statement

```
printf("%s %d\n", buf_ptr, i) ;
```

The format string is highlighted above, it is a sequence of formatting instructions and conversion characters. Using this string the *printf()* function knows what to print and how to print it. The character *%* in the string denotes a substitution of a program variable the character following the *%* is its type (aka. conversion character), *s* for string and *d* for integer. The *\n* denotes a new line. Conversion characters exist for all conversion types. Any character that is not a conversion or a format character is simply copied to the output. A seldom used conversion character is *%n*. This does not substitute an argument into the string but writes the number of bytes output so far in the string to the corresponding argument in the argument list. If we add *%n* to the above statement and assign values to its arguments we can see the following result

```
n=0 ;
printf("%s %d %n\n", buf_ptr, i, &n ) ;
```

if buf_ptr = "Hello" and i = 5, then the output would be

```
Hello 5 8
```

The value of *n* is now 8, since that is the number of characters that were output by that time.

It's this property that can be exploited. Its use in programs is quite rare, so why is such a concern ?. Its exploitation occurs when functions such as *printf()* that require a format string are coded without one. Instances where trusting programmers just want to print out a string supplied by a user. Consider the following statement which is common in many logging functions.

```
printf(buf_ptr) ;
```

This is legal C, is it a *printf()* without a format string ?, no, buf_ptr is the format string. The programmer expects it to contain only printable characters which will be copied to the output not conversion characters, and certainly not a *%n*. Conversion characters require arguments for substitution, but here there are no arguments. At compile time the contents of *buf_ptr* is not known so an error cannot be generated. At run time if *buf_ptr* contains conversion characters then *printf()* looks for the arguments on the program stack. The *%n* requires an address argument, somewhere to write the number of characters printed, this also will be taken from the stack. Since the user (attacker) provided format string is also placed on the stack the attacker has control through this string a portion of the stack. By adding *%n* characters with format width specifiers into the string an attacker can write any value to any location in memory. Exactly how this is done is the subject of a paper in its self, fortunately many have been written [5][6]. Needless to say this allows an attacker to run arbitrary code. As with buffer overflows attackers can place shell code in known memory locations and divert the normal course of a programs execution to it.

Race Conditions

Race conditions occur when two or more processes access a shared resource in an order which was not expected by the program. Unordered access to resources is common in multitasking environments. If a program requires that a resource be exclusively accessed or accessed in a certain order by different processes then it should take steps to ensure that ordering. Two kinds of race conditions will be discussed here the first describes race conditions in signal handlers and the second in file handling.

Signal Handlers

A signal is a notification to a process that an event has occurred, signals can be sent by one process to another process or sent by the kernel to a process. Example Signals are

SIGBUS	: Bus error
SIGSEGV	: Segmentation Violation
SIGWINCH	: Window size change
SIGUSR1	: User defined signal

When a signal is sent to a process its normal control flow is interrupted and a particular function known as a signal handler is called (if it is defined). Signal handler code that includes function calls that are not reentrant safe are vulnerable to exploitation. Consider the following signal handler (described in [7]) which is shared by two signals SIGHUP and SIGTERM. (this handler is similar to that which sendmail used up to version 8.11.3)

```
/* global variables */

void *global1, *global2;
char *msg;

/* signal handler used by SIGHUP and SIGTERM */

void sig_handler(int dummy) {
     syslog(LOG_NOTICE,"%s\n",msg);
     free(global2);
     free(global1);
     sleep(10); exit(0);
}
```

The *syslog()* function (glibc) is not reentrant safe. It dynamically allocates two buffers (using the two global pointers) and copies the message (*msg*) into the buffers before writing to the syslog. Following the syslog the global pointers are de-allocated in the signal handler. If two signals are sent to this program closely following each other a race condition occurs which could lead to arbitrary code execution. The first signal (SIGHUP) would enter the handler, call the syslog() function and allocate memory to the two global pointers. After exiting the syslog function the global pointers are free'ed. During the free if a second signal is received the handler code is reentered. Memory that has been free'ed is then reused. Carefully crafting the buffer contents of the syslogs would allow the dynamic memory structures to become corrupted as in the unlink example above leading arbitrary code execution.

File Handling

Race Conditions in file handling revolve around *Time of check Time of use* (TOCTOU) issues. Some C library functions can be subverted into accessing files other than those that were intended. This is best explained with an example.

The Solaris *ps* utility (list currently running processes) would as part of its processing open a known file in the */tmp* directory and change its ownership to *root*. This could be exploited in the following way by an attacker

- First slow down the system
- Run *ps*
- Delete the tmp file created
- Create a new file with the same name with SUID bit set
- Edit this file to start a shell (root)

(This problem was reported in 1995 and has now been resolved) To increase the chances of successfully exploiting this condition a script / program can be used to carry out the above and placed in a loop to repeatedly try the exploit. TOCTOU issues are heavily dependant on timing often brute force is required to exploit. This example created a new

file in place of the real file. Symbolic links can also be used in File system attacks. These kinds of attacks rely on knowing the name of the temporary and access rights that the file has. All files whether temporary or not should have the minimum level of privileges i.e. don't grant write access (or even read access) to users or groups that don't require it. C library functions exist to create temporary files with unique names such as *tmpnam()*, unique filenames can help thwart attacks as the attacker will not know the name of the file that will be created. However a determined attacker who analyses the filenames created may be able to predict a filename and thus carry out an attack [2].

Secure Programming Tips

Avoiding Overflows

1. Use copy functions that copy a maximum number of bytes rather than ones that rely on NULL terminated strings (e.g. use strncpy instead of strcpy).
2. Check the length of input from the user to make sure it's not longer than expected.
3. Make the stack/heap non executable, If an attackers places malicious code somewhere on the stack/heap in the hope of exploiting an overflow to execute it then it will not run even if he successfully overflows a buffer

   Stack protection
   - For Linux systems the Openwall project supplies a patch [8]
   - In Solaris a configuration parameter suffices [9]
   - Stack guarding (see below)

   Heap protection
   - Patch provided by the PaX Team [10]

   (Attackers can still place code elsewhere so these are not complete solutions)

Format Strings

4. Always provide a format string argument.

Signal Handlers

5. Keep signal handlers simple and use only reentrant safe functions [7]

File Handling

6. Always use full path names to files, as modification of the UNIX PATH variable can change the order of the directories a file is searched for.
7. Use file handling functions that identify files using file descriptors instead of file names (e.g. *fchown()* instead of *chown()*).

8. Avoid writing temporary files or configuration files to world writeable directories such as */tmp*.
9. Minimise the access rights on files and directories to the absolute minimum required for correct functionality.
10. Set the sticky bit on */tmp*, this will prevent users other than the owner or root from removing or linking to any files in the directory.
11. Use unique filenames.

General

12. Almost all security related problems stem from user input. It is essential that user input is validated such that

    - It only contains valid characters e.g. a telephone number is sequence of numbers only, if any other characters appear in the users input then the input should be rejected.
    - Does not exceed the maximum length of any buffer it is written to.

13. Always check the return codes of library functions for failure.

Automated tools

New programs can be designed with security in mind but there exist billions of lines of code in systems all around the world. These need to be reviewed to fix vulnerabilities. A number of tools exist to help with task.

Code Scanners

Various groups have attempted to write automated tools to scan code for vulnerabilities (*its4*[11], *rats*[12]). These can be useful in identifying potential problems especially if you have very large programs. Their databases cover a wide range of vulnerabilities including the ones discussed in this paper. They cover a number of languages such as C, C++, perl and python. These tools perform static checking of code; that is they do not check for vulnerabilities at runtime. However they use a simplistic method of analyzing source code and are easy to trip up, as the following example shows

```
char x[5];
strncpy(x, argv[1], 11);
```

As one can see the variable x is of size 5 characters but the copy tries to copy 11, an obvious overflow which isn't picked up by *its4* (v1.1.1) or *rats* (v1.5)

Stack guarding

Two methods can be used to protect the stack. One is to insert a known *canary* value on the stack and check to see if it has been overflowed and modified [13] and the other is to protect the return addresses that are the targets of overflows [14]. Both tools compile code into your program to implement their protection schemes. Determined attackers can however bypass both these protection schemes.

Conclusion

Education is key in the quest to make software more secure. At university I was taught, no, told to write structured programs to always check user input and to do bounds checking but never the most important reason why. Programmers need to be educated not only on good programming practices but also what the consequences of bad programming are. It's not just that their program could crash or fail to do what it's supposed to do, their program could result in a major security risk for their company.

Software needs to be properly tested, particularly if it crashes as this is the first sign that something like an overflow has occurred. Close attention should be paid to validating user input as this is a source of an attack whether it's from user input at the keyboard, from a file, socket, pipe etc.

Tools are available to help programmers to find and deal with security holes but these are not complete solutions. Compilers that insert protection code are simply papering over the cracks. Ingenious attackers have found ways around them, there is always somewhere to place exploit code. The best advice I can give is to learn about security and apply the knowledge. The tips included above are not a complete list but will make programs more secure.

References

[1] Smashing the stack for fun and profit, Aleph One, Phrack 49 http://www.phrack.com/show.php?p=49
[2] Secure Programming for Linux and Unix HOWTO http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html
[3] Inside the Buffer Overflow Attack: Mechanism, Method, & Prevention, Mark E. Donaldson April 3, 2002, http://rr.sans.org/code/inside_buffer.php
[4] vudoo malloc tricks / Once upon a free http://www.phrack.org/show.php?p=57
[5] Exploiting Format String Vulnerabilities http://www.team-teso.net/articles/formatstring/
[6] Article by Christophe Blaess at http://community.core-sdi.com/~juliano/ follow user supplied format string link
[7] Signals for fun and profit by Michal Zalewski http://razor.bindview.com/publish/papers/signals.html
[8] Openwall Project (see kernel patches) http://www.openwall.com/linux/
[9] Solaris : disable executable stack http://ist.uwaterloo.ca/security/howto/1999-06-22.html
[10] PaX http://pageexec.virtualave.net/pax-linux-2.4.19.patch
[11] its4 http://www.cigital.com/its4/
[12] rats http://www.securesoftware.com/rats.php
[13] StackGuard: http://immunix.org
[14] Stack shield http://www.angelfire.com/sk/stackshield/