

Toward a Secure Programming Paradigm

Benjamin Collins
CPSC 481 – Fall 2002

November 1, 2002

Contents

1	Introduction	2
2	Common Security Issues	2
2.1	The buffer overflow	2
2.2	Race conditions	3
2.3	Numeric overflow	4
3	Techniques	5
3.1	Avoiding numeric and buffer overflows	5
3.2	Avoiding race conditions	6
4	Application to Curriculum	6
A	Examples	8
A.1	Buffer Overflow	8
A.2	Race Condition	9
B	Improved examples	11
B.1	Buffer overflow	11
B.2	Race condition	12

1 Introduction

Engineering secure software has been at best a challenge for the software industry. Many companies that sell software as its primary product do not give the necessary attention or time to security in their products, and some hardly give it a passing thought. Software engineers graduating from Texas A&M University ought to bring to their employers a skill that is becoming highly valued, and will soon be an essential part of a software engineer's skillset: writing secure code. It is difficult to be sure that code cannot be abused or exploited to gain unauthorized access to resources — it is even more difficult if the individuals writing the code do not have any training or experience in securing code. What follows is an attempt to explain simple concepts that have been used to exploit software time and time again over the last few decades and give methods for avoiding the pitfalls that occur as a result of neglecting them.

2 Common Security Issues

There could be any number of things that make a particular piece of code insecure. There are, however, a small number of mistakes that engineers make that are frequently exploited by malicious auditors in order to gain access to resources. I will explain some of these often-used exploits in this section. These brief explanations are certainly not comprehensive in any way, even of the most common exploitable conditions. I am neglecting many others, including exploitation involving format strings, environment variables, network protocols, and many others.

2.1 The buffer overflow

This is a class of problems unto itself. Buffer overflows have been a problem as long as software engineering has been a profession. The basic idea of a buffer overflow is to find data of a fixed size that is being stored on the stack. If precautions are not taken, the data, usually in the form of an

array of some kind, can be over-written in a way that causes the return address for the previous context in the process being run can be changed to an arbitrary value. Once this is accomplished, an attacker can run arbitrary code with the privileges of the exploited process. Often, the “arbitrary” code is a set of instructions to run a shell or to change permissions on a file. Sometimes this can be achieved by overflowing a buffer by only one byte. klog describes in detail how the code listed in appendix A.1 can be exploited in this way [7]. As you can see, in `func()`, if the string pointed to by `sm` is 257 bytes long, the character array `buffer` will be overflowed. The stack frame for `func()` is also shown in appendix A.1 [7]. As you can see, if you write one byte past `buffer`, you can modify the last byte of the saved frame pointer which will be loaded as the current frame pointer when `func` returns to `main()`. The ability to change only one byte of the saved frame pointer will not allow a completely arbitrary modification of the flow of execution, but if there is a user-controlled buffer in a nearby region of memory, an exploit can be found. In the article, klog writes code for executing a shell into `buffer` and then modifies the last byte of the frame pointer so that when `func()` returns, the flow of execution will enter the shell code.

2.2 Race conditions

Race conditions can occur when a program tries to access a resource that it expects to be there but is not available or has changed in some way. This situation typically occurs with temporary files. Many programs create files in a fixed folder, usually `/tmp`. If another process or a user with sufficient privileges moves, deletes, or modifies the file, a race condition can occur. For example, it is common for a program to use the system calls `access()` or `stat()` [5] to verify certain properties of a file such as ownership or permissions. Once the file has been checked, the programs then usually use system calls like `open()`, `chown()`, `chgrp()`, `chmod()`, and `unlink()` [5] to perform various operations on the file. However, there could be a significant amount of time lapse between the system calls. The file in question could be altered after being checked and before being operated on, thus the program might unwittingly make a change in the system that would allow unauthorized access to system resources.

An example of a program that is vulnerable to a race condition attack is

listed in appendix A.2 [6]. Notice that all the checks on the condition of the file are complete on line 29; the file is opened on line 31. This leaves an opportunity for an attacker to change the properties of the file. Assume that the program, called `ex_01`, is being run with root privileges, and that the attacker wants to write the following line into `/etc/shadow`: `root::1:99999:::..`. The attacker should create a file and run the program as follows:

```
$ touch tmp_file
$ ./ex_01 tmp_file \"root::1:9999:::..\" &
```

During the time after `ex_01` checks the conditions of `tmp_file` and before it actually opens the file for writing, the attacker could delete the file and create a symbolic link to `/etc/shadow` using the name `tmp_file`.

```
$ rm -f tmp_file
$ ln -s /etc/shadow ./tmp_file
```

Then, when the program opens `tmp_file` for writing, it will actually open `/etc/shadow` and the attacker will have access passwordless root account.

2.3 Numeric overflow

A numeric overflow is similar to a buffer overflow in that security issues result when a malicious user violates data boundaries in a process. However, it is different than the common buffer overflow in that it exploits the definition of data rather than the structure of the data. Consider the following example from [5]. Unix user IDs (UIDs) are represented by 16-bit integer values on some systems. NFS uses a 32-bit integer to represent a UID and does not allow a UID of 0 (`root`) to be used; a UID of 0 by default is mapped to the value 65534 (-2 because of the overflow condition). If an attacker gave the NFS server a request using a UID of 2^{17} , the server will check to see if the UID is 0 — and its not. The server will then give the UID to the UNIX kernel for access control since all filesystem requests must pass through the kernel, even for NFS. The kernel only deals with 16-bit UIDs and will discard the high order bits for our UID of 2^{17} , leaving the UID as 0 — `root`.

3 Techniques

There are some general practices that can be applied to a wide range of situations that might occur in a software engineering environment. Many of them are very simple and would not require a greater amount of work on the part of the engineer, only awareness of common security issues. Some of those issues are more complex and require a significant work-around by the engineer.

3.1 Avoiding numeric and buffer overflows

In the example discussed in section 2.1, the “fix” for the single-byte overflow is to change the for loop so that the index is always less than 256, not less than or equal to 256. The corrected code is given in B.1. In more general terms, there are a number of ways to correct errors that might lead to buffer overflows. Among them are input validation. Input into programs or functions should always be validated, but particularly when the input is user input or otherwise untrusted [5].

One way to validate untrusted input is to check the length of the input. Buffer overflows are frequently the result of untrusted input being too large for some buffer used in that part of the code; this is true both for buffers stored on the stack and on the heap — although there are a different set of circumstances when dealing with the heap and they are beyond the scope of this paper. There is no set method for checking the length of input, although common practices are to truncate the input to the desired length or indicate an error has occurred, either through return values or throwing exceptions. Often, functions that do not enforce length restrictions are a target of buffer overflow attacks; these include `strcpy()`, `gets()`, `strcat()`, `sprintf()`, and others. Simply put, these functions copy one string to a fixed string buffer without regard for the length of the input string. In order to prevent these functions from being abused to the benefit of some malicious attacker, special care must be taken by an engineer. Either the engineer should check the length of input to these functions independently or use more secure versions of the functions. For the previously shown functions, these are `strncpy()`, `fgets()`, `strncat()`, and `snprintf()` (although this is not standard on all systems). All of these improved replacements take a parameter dealing with length: number of

characters to copy, number of characters to write, etc.

If the input being dealt with is not in the form of a string of some kind, it is often either numbers or network infrastructure data [5]. In the case that the input is a number, checks should include information like magnitude and sign. For network data, information from relevant RFCs should be used for validations.

3.2 Avoiding race conditions

Permission-checking cannot be separated from access.

— *Steve Bellovin* [4]

This is a very important general principle when dealing with race conditions. This approach can be implemented by using `stat()` and `lstat()` [3] to check the status of a file, including permissions, ownership, and whether a file is a symbolic link or not. When using these two functions, information about the file in question is stored in a struct. Later, when the program is ready to open the file, `fstat()` should be used to check the status of the opened file with the information in the previous structs. If there are discrepancies, then either abort the operation or deal with the problem in some other appropriate way. If the information matches, then go on with the operation on the assumption that the opened file is valid and correct. There are other ways to avoid the race condition. One conceptually simple method is to avoid using fixed common directories [3]. Instead of writing a temporary file to `/tmp`, perhaps it would be more secure to create a directory in `/root` and write the file there [6, 3]. Another method that could help is to temporarily drop the level of privileges while accessing files that could potentially be altered. This does not solve the problem altogether, but it would help to avoid a compromise of the `root` account. The code listed in A.2 can be fixed by checking the status of the file from a file descriptor instead of a pathname. The corrected code is listed in B.2.

4 Application to Curriculum

In conclusion, I suggest some ways that the Department of Computer Science at Texas A&M University could integrate some of these ideas and

principles into the existing curriculum. Many of the least time-expensive methods could be integrated into lower-level programming courses such as CPSC 111 and CPSC 206. Many students learn the fundamentals of programming in these courses. Would it not be beneficial for them to learn correct practices and increase their awareness of security issues? Inevitably, new security issues will come up in the future. Software engineers will be far more prepared to come up with correct ways to address those issues if they have been taught to think about security. Also, a course that focuses solely on secure software engineering could be offered as an anchor for the integration of these concepts into the curriculum. A senior level course — maybe given a course number like 470 — would be most appropriate. After having the **Computer Architecture**, **Operating Systems**, and **Networks and Distributed Computing** courses, upperclassmen would be prepared to do more advanced study about how to audit code for potential security problems as well as find new and creative solutions to those problems. Of course, such a class would have to be offered as a technical elective, but students who would choose this course would help to solidify their ability to engineer secure software and they would have a competitive advantage over most other computer science graduates who go on to be software engineers.

A Examples

A.1 Buffer Overflow

```
#include <stdio.h>

func(char *sm)
{
    char buffer[256];
    int i;
    for(i=0;i<=256;i++)
        buffer[i]=sm[i];
}

main(int argc, char *argv[])
{
    if (argc < 2) {
        printf('missing args\n');
        exit(-1);
    }

    func(argv[1]);
    return 0;
}
```

The stack frame for func():

```
saved_eip
saved_ebp
char buffer[255]
char buffer[254]
...
char buffer[000]
int i
```

A.2 Race Condition

```
1  /* ex_01.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7
8  int
9  main (int argc, char * argv [])
10 {
11     struct stat st;
12     FILE * fp;
13
14     if (argc != 3) {
15         fprintf (stderr, "usage : %s file message\n", argv [0]);
16         exit(EXIT_FAILURE);
17     }
18     if (stat (argv [1], & st) < 0) {
19         fprintf (stderr, "can't find %s\n", argv [1]);
20         exit(EXIT_FAILURE);
21     }
22     if (st . st_uid != getuid ()) {
23         fprintf (stderr, "not the owner of %s \n", argv [1]);
24         exit(EXIT_FAILURE);
25     }
26     if (! S_ISREG (st . st_mode)) {
27         fprintf (stderr, "%s is not a normal file\n", argv[1]);
28         exit(EXIT_FAILURE);
29     }
30
31     if ((fp = fopen (argv [1], "w")) == NULL) {
32         fprintf (stderr, "Can't open\n");
33         exit(EXIT_FAILURE);
34     }
35     fprintf (fp, "%s\n", argv [2]);
36     fclose (fp);
```

```
37     fprintf (stderr, 'Write Ok\n');
38     exit(EXIT_SUCCESS);
39 }
```

B Improved examples

B.1 Buffer overflow

```
#include <stdio.h>

func(char *sm)
{
    char buffer[256];
    int i;
    for(i=0;i<256;i++) /*this is the changed line of code*/
        buffer[i]=sm[i];
}

main(int argc, char *argv[])
{
    if (argc < 2) {
        printf('missing args\n');
        exit(-1);
    }

    func(argv[1]);
    return 0;
}
```

From [6].

B.2 Race condition

```
1  /* ex_02.c */
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/stat.h>
7  #include <sys/types.h>
8
9  int
10 main (int argc, char * argv [])
11 {
12     struct stat st;
13     int fd;
14     FILE * fp;
15
16     if (argc != 3) {
17         fprintf (stderr, "usage : %s file message\n", argv [0]);
18         exit(EXIT_FAILURE);
19     }
20     if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
21         fprintf (stderr, "Can't open %s\n", argv [1]);
22         exit(EXIT_FAILURE);
23     }
24     fstat (fd, & st);
25     if (st . st_uid != getuid ()) {
26         fprintf (stderr, "%s not owner !\n", argv [1]);
27         exit(EXIT_FAILURE);
28     }
29     if (! S_ISREG (st . st_mode)) {
30         fprintf (stderr, "%s not a normal file\n", argv[1]);
31         exit(EXIT_FAILURE);
32     }
33     if ((fp = fdopen (fd, "w")) == NULL) {
34         fprintf (stderr, "Can't open\n");
35         exit(EXIT_FAILURE);
36     }
```

```
37     fprintf (fp, "%s", argv [2]);
38     fclose (fp);
39     fprintf (stderr, "Write Ok\n");
40     exit(EXIT_SUCCESS);
41 }
```

References

- [1] Peteanu, Razvan. *Best Practices for Secure Development* v4.03, Oct 2001
Available online from http://member.rogers.com/razvan.peteanu/best_prac_for_sec_dev4.pdf
- [2] Rönning, Juha. *Software Considered Harmful: Why Software is Insecure* Oulu University Secure Programming Group, Presentation, 2002.
- [3] Karas, Benjamin. *Writing Privileged Programs* Crimelabs Security Group, 2000.
Available online from <http://www.crimelabs.net/docs/sec-programming.ps>
- [4] Bellovin, Steve. *Shifting the Odds: Writing (More) Secure Software*, AT&T Research, Murray Hill, 1996.
- [5] Bishop, Matt. *How Attackers Break Programs, and How To Write Programs More Securely* SANS, Baltimore, 2001.
Available online from <http://http://nob.cs.ucdavis.edu/~bishop/secprog/sans2001.pdf>
- [6] Raynal, Frédéric; Blaess, Christophe; and Grenier, Christophe. *Secure Programming* Articles #[182,183,190,191,198,203], LinuxFocus 2001.
Available online from <http://www.linuxfocus.org>
- [7] klog. “The Frame Pointer Overwrite”, *Phrack Magazine* vol. 9, issue 55, article 8. 9/9/99.
Available online from <http://www.phrack.org>