Writing Privileged Programs

Benjamin Karas

June 12, 2000

Contents

1 Abstract

Writing privileged programs able to resist a concerted attacker has always been a very difficult process. The difficulty lies in the complex interactions between various system calls and the program's environment. Vulnerabilities can result from operational faults: configuration and policy errors; coding faults: faults of commission, omission, or programming logic; or environmental faults [Asl95]. However, some vulnerabilities are common enough to warrant a general avoidance scheme.

2 Organization

This paper outlines a number of common vulnerabilities that affect a wide range of applications. The last section gives a few general methods for reducing the number of bugs in a program.

3 Common vulnerabilities

These sections show various methods that have been used to exploit programs. Some are theoretical, but most have been used in the real world to gain privileged access to data, resources, or computers. For each, I present a description of the vulnerability, the origin, how to exploit it, and various fixes. Often there are problems with the fixes themselves, which I will try to note in each section.

3.1 Common buffer overflow

The common buffer overflow is a fancy name for writing data beyond the end of an array. These most commonly occur when processing user-input string data. Often, a program will call a function that does check to see if it is writing past the end of an array. By overwriting other areas of memory, an attacker can often execute arbitrary code, such as to open a shell or to change permissions on something.

Origin

Many standard library functions do not properly limit the amount of data they write. These include:

Unchecked	**Checked**
`strcpy(3)`	`strncpy(3)`, `strlcpy(3)`
`strcat(3)`	`strncat(3)`, `strlcat(3)`
`sprintf(3)`, `vsprintf(3)`	`snprintf(3)`, `vsnprintf(3)`
`gets(3)`	`fgets(3)`

According to David Wheeler, the following functions may also allow buffer overflows: `realpath(3)`, `getopt(3)`, `getpass(3)`, `strecpy(3)`, `strecat(3)`, `strtrns(3)` [Whe00].

Exploit

Exploiting these types of overflows is not usually difficult. If the buffer is located on the stack, the attacker is usually able to overwrite the return address for a function. When the function attempts to return, it will execute the attackers code. Methods of doing this are fairly well know, especially after the publication of "Smashing the Stack for Fun and Profit" by Aleph One [One96].

Fixes

The best solution to this vulnerability is to rewrite the code such that it checks the length of buffers. For most insecure functions, there is a more secure function that allows you to specify the length of the destination buffer. Unfortunately, there are remaining issues with many of these functions.

`strncpy(3)` This function will not NIL terminate a string if it runs out of space. Be sure to either manually add a `'\0'`, or reserve an extra byte of memory for the NIL termination. See the next section concerning vulnerabilities involving `strncpy(3)`.

`strncat(3)` Be careful to pass the amount of *remaining* space in the destination instead of the total length of the destination buffer. It will append a NIL to the destination string, so be sure to leave room for this byte.

`strlcpy(3)`, `strlcat(3)` These wonderful functions were created for use in OpenBSD [MdR]. Unfortunately, this means the library containing them is not always installed. You must pass the full length of the destination to the function, which will NIL terminate the string for you.

There are quite a few other methods to add resistance to these types of overflows. You can link your program against a number of libraries, such as libsafe [BTS99] or StackGuard [CPMea]. Solar designer has released a patch to the Linux kernel that makes the program's stack non-executable [Des]. Unfortunately, it is possible to work around these restrictions, so do not depend solely on them. On the other hand, the extra layers of security are nice.

3.2 One-byte overflows

The method described by Aleph One [One96] requires the ability to overflow the stack by multiple bytes. Unfortunately, as discussed by klog [klo99], it is possible to execute arbitrary code even if you overflow a buffer by a single byte. This amazing feat works if the buffer is located first in the frame. In this case, you are able to set the least significant byte of the stored frame pointer. This lets you set the frame arbitrarily, which can lead to an exploit.

Origin

The reason this type of exploit is mentioned specifically is because of the ease with which one can introduce an off-by-one error in a program. klog successfully wrote an exploit for the following program:

phrack5508.c

```
/* From Phrack-55-08, klog */
#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]);
}
```

phrack5508.c

As you can see, the mistake is the `<=` sign in the for loop. This should have been a plain `<` sign. This shows just how easy off-by-one errors are.

Exploit

The exploit involved changing the saved frame pointer so that it would point to somewhere in a user controlled buffer. When the main program tries to return, it restores the instruction pointer from the user's buffer and starts executing the user's code. [klo99]

Fixes

Stackguard and libsafe cannot protect against these types of vulnerabilities. Stackguard protects the return instruction pointer [CPMea] and libsafe only replaces certain system calls [BTS99]. Solar Designer's patch should protect against some of these attacks, but it still does not prevent a byte of memory from being overwritten. The best method, therefore, is avoidance.

These errors are more difficult to spot. The best way to avoid them is to look at boundary conditions. What happens when `i` is `0`? What happens when it is `255` or `256`? One convention I use is to always use the total length in the for loop:

This:

```
for (i=0; i<256; i++)
```

instead of this:

```
for (i=0; i<=255; i++)
```

The biggest rule is to be consistent. The advantage to the above rule is that you immediately see the size of the buffer.

3.3 Protected installations

One misconception is that Stack Guard, libsafe, or Stack Shield will completely protect a system from harm. Unfortunately, a shorter buffer overflow will still overwrite variables while not triggering alarms. Attacks can be made against stacks that are non-executable and protected by any of the above libraries.

Origin

This type of overflow is very similar to the last one. The attacker overwrites a pointer, which then allows the attacker to modify data elsewhere. Unlike previous overflows, this class of vulnerability requires the following three conditions:

1. There needs to be a pointer physically located on the stack after the buffer.
2. The attacker needs to be able to overwrite this pointer.
3. The program must use the new pointer in a `strcpy(3)`-like function.

For instance, the following program, as presented by Bulba and Kil3r, is vulnerable to attack [BK00]:

phrack5605.c

```c
/* From Phrack 56-0x05, Bulba and Kil3r */
int f (char ** argv)
{
        int pipa;       // useless variable
        char *p;
        char a[30];

        p=a;

        strcpy(p,argv[1]);        // <== vulnerable strcpy()
        strncpy(p,argv[2],16);
        printf("After second strcpy ;)\n");
}

main (int argc, char ** argv) {
        f(argv);
        execl("back_to_vul","",0);  //<-- The exec that fails
        printf("End of program\n");
}
```

phrack5605.c

Exploit

The exploit works by overwriting the pointer and then using it to write data to an arbitrary location in memory. The authors suggest a number of places to overwrite, including the return address, the list of functions registered with `atexit(3)` or `on_exit(3)`, or the Global Offset Table [BK00].

This is the generic attack route. It still leaves open the possibility of overwriting critical program data, such as whether a person has permission to perform a certain operation.

Fixes

Obviously none of the quick fix libraries or kernel patches will help us here. The best thing to do is avoid buffer overflows in general. The ability to overwrite any portion of data is very damaging to the robustness of a program's security.

3.4 `snprintf(3)`

Thusfar we have learned that there are no quick fixes to all our problems. The best this to do is to migrate to safer function calls. Unfortunately, our problems do not end here as the standard function calls are not robust.

The problem is that a number of function calls do not NIL terminate strings for us. For instance, the following code will result in a buffer overflow [Twi00]:

phrack560e.c

```
/* From Phrack 56-0x0e, twitch */
int main(int argc, char **argv)
{
  char buf1[1024];
  char buf2[256];

  strncpy(buf, argv[1], 1024);
  strncpy(buf2, argv[2], 256);

  ...

  if(somecondition)
         print_error(buf2);

}

void print_error(char *p)
{
  char mybuf[263];

  sprintf(mybuf, "error: %s", p);
}
```

phrack560e.c

Origin

The problem lies in forgetting to NIL terminate the string after the `snprintf(3)` call, which will not do this for you. Other function calls that do not NIL terminate strings for you are: `fread(3)`, `read(2)`, `readv(2)`, `pread(2)`, `memcpy(3)`, `memccpy(3)`, `memmove(3)`, `bcopy(3)`, `gethostname(2)`, and `strncat(3)`.

Exploit

The exploit is the same as any other buffer overflow. Using the two buffers together, the attacker is able to overwrite data somewhere, possibly executing arbitrary code.

Fixes

One method is to manually insert a NIL character after each call to this function. A second method is to allocate an extra byte for the string and to always keep a NIL character there.

For example, you might recode the above code as follows:

phrack560e.fixed.c

```
/* Modified code from Phrack 56-0x0e, twitch */

#define BIGSIZE    1024
#define SMALLSIZE   256

int main(int argc, char **argv)
{
  /* Allocate space for one extra byte */
  char buf1[BIGSIZE + 1];
  char buf2[SMALLSIZE + 1];

  /* Place NIL terminator at end of each string */
  buf1[BIGSIZE] = '\0';
  buf2[SMALLSIZE]  = '\0';

  strncpy(buf, argv[1], BIGSIZE);
  strncpy(buf2, argv[2], SMALLSIZE);

  ...

  if(somecondition)
         print_error(buf2);

}

void print_error(char *p)
{
  char mybuf[263];

  sprintf(mybuf, "error: %s", p);
}
```

phrack560e.fixed.c

The big thing here is to use a consistent convention when coding. You might want to specify your convention in a coding style document distributed with your code.

3.5 `snprintf(buffer, user_string)` and %

There are a whole class of problems involving using a user-supplied string as a format string to any of the `printf` family of functions. If the user supplied string contains any `'%'` characters, an attacker can do two things:

1. The attacker can telescope the string, possibly cutting off information at the end of the string. This can be disastrous if, say, the hostname of a logging function was cut off.

2. The attacker can, using `'%n'` write a value to a piece of memory. Combine this with a bug in `vsnprintf(3)`, and you can write an arbitrary value to a location in memory [Pri00].

Origin

This is a prime example of not validating a user's input. The programmer, in this case, trusted that the user would use safe characters in the user supplied string, thus making it safe to use it as a format string. This is bad in two ways. First, all user input should be checked to make sure it is acceptable. Second, it is probably bad practice to not provide an explicit format string.

Exploit

Exploiting this kind of bug is much more of an intellectual challenge than most overflow exploits. However, prizm was able to exploit Qpopper 2.53 successfully, so you really should watch out for this kind of thing. Other programmers have been caught not providing an explicit (`"%s"`) format string, too.

Another problem is that it allows an attacker to telescope the string by using `"%p"`. First, this can push data off the end of the string, which might make logging hard. Second, if the programmer for some reason uses an unchecked `sprintf(3)` afterward, a regular buffer overflow may result, as in the following code fragment:

`rabbit1.c`

```c
/* rabbit example code #1, 6-8-2000 */
void _getmessage_int(char *dest, char *src)
{
        snprintf(dest, 20, "%s", src);
        printf("src=%s\ndest=%s\n\n",src,dest);
}

int getmessage(char *dest, int len, char *src)
{
        char C[101];
        /* _getmessage_int fills first 20 of C */
        if (len >= 20)
                _getmessage_int(C, src);
        else
                snprintf(C, len, "Error: too small\n");
```

```

        /* return length by doing snprintf */
        snprintf(dest, len, C);
        printf("C=%s\ndest=%s\n\n",C,dest);
}

int main(int argc, char *argv[])
{
        char A[101], B[21];

        getmessage(A, 101, argv[1]);

        /* we know that _getmessage_int fills only 20
         * bytes, so we can use sprintf  (*WRONG*)   */
        sprintf(B, A);
        printf("A:%s\nB:%s\n",A,B);
}
```

rabbit1.c

The problem lies in the interaction between the various calls to `snprintf(3)` and `sprintf(3)`. The code is certainly contrived, but if a complex program is coded sloppily, then things like this just might occur. The following output shows the program being buffer overflowed, though not exploited:

rabbit1.c.run

```
$ ./rabbit1 "rabbit is a hacker; blah blah blah blah"
src=rabbit is a hacker; blah blah blah blah
dest=rabbit is a hacker;

C=rabbit is a hacker;
dest=rabbit is a hacker;

A:rabbit is a hacker;
B:rabbit is a hacker;


$ ./rabbit1 "%p%p%prabbit is a hacker; blah blah blah blah"
src=%p%p%prabbit is a hacker; blah blah blah blah
dest=%p%p%prabbit is a h

C=%p%p%prabbit is a h
dest=0x702570250x617270250x74696262rabbit is a h

A:696262rabbit is a h
B:0x702570250x617270250x74696262rabbit is a h
```

rabbit1.c.run

As you can see, the extra "`%p`"s in the input cause a buffer overflow. Had the final `printf` call been checked (`snprintf(3)`), none of the message would have survived being pushed off the end of the string.

Fixes

The easiest fix is to always validate all inputs, and to always provide a format string to `printf` functions. Doing these two things will safeguard your code from these types of attacks.

3.6 Tempfile race conditions

There is a whole class of problems involving file system race conditions. The most common of these is the tempfile race condition. In this problem, a program attempts to create a new file, but mistakenly follows a symbolic link that was setup by an attacker. This can trick a program into opening an arbitrary file on the system. There are other variations on this theme, each basically resulting in a privileged program opening a file other than the one it expected.

Origin

The problems lies in the interaction of the `stat(2)` family of functions, the `open(2)` family of functions, and various other functions that access files. There is considerable debate as to whether the library implementation is at fault, or if programmers are being lazy. Unfortunately, there is even more debate as to how to fix this problem.

The problem is further exacerbated in the inability for the `O_EXCL` flag to work across NFS mounted volumes, and the fact that a SUID program usually does not check file permissions when opening files. More discussion is given in the **notes** section below.

Exploit

The exploit is fairly trivial. The attacker writes a program that repeatedly creates a symlink in `/tmp` and deletes while at the same time starting and restarting the vulnerable program. The attacker is trying to time events like so:

1. The attacker starts the vulnerable program.
2. The program chooses a filename to create, and checks that it does not exist in `/tmp`. If the file exists, it makes sure it is not a symlink.
3. The attacker guesses the filename the vulnerable program will use.
4. The attacker creates a symlink or FIFO to this filename.
5. The program opens the filename. If it is a symlink, it opens some arbitrary file, like `/etc/passwd` or /code/etc/nologin. If the file is a FIFO, the attacker can read and write arbitrary data to the program.

Where the exploit goes from here usually depends on the goals of the attacker and the specifics of the vulnerable program. Ideas include accessing arbitrary files, changing permissions on files, passing invalid data through the pipe to the program, etc.

Fixes

This vulnerability is difficult to fix. This is due, in part, to the multitude of opinions on the correct fix:

1. Do not use a common directory. This only works if the file can be private. If more than one process need to access the file, finding a suitable spot without administrator intervention can be difficult.

2. Create a subdirectory owned by the privileged user in `/tmp`. Change directories there, and compare inodes and permissions to make sure you got to the right spot. Since no one else can create files here, you can safely create a file and use it.

3. When opening a file for reading, you should be able to use lstat to make sure it is the correct file, and that it is not a symlink.

   > I believe that the correct approach (in general) is to lstat() the file, check that it's suitable (e.g. not a symlink), open() it, fstat() the descriptor, and compare the device/inode pair with those from the lstat(), to check that you did actually open the file which you thought that you were opening.
   > [Cle98]

4. When creating an empty file, perform lots of checks after using `open(2)` and truncate the file [Sho00]. Other variations exist on this theme.

5. Drop privileges temporarily while accessing files. This prevents the attacker from opening some files, but the attacker may still be able to do naughty things.

As you can see, there is no consensus here. It would be nice if a well documented decision were to be presented by someone more knowledgeable than I. The other solution is to look at the source code of a program that performs an operation similar to your own and that is known to be secure. Another consideration is that many of the source code examples are long and complex, which also makes them prone to errors and bugs.

There are a few good, easy, things you can do to minimize this vulnerability. First, if you need a random name, use the `mkstemp(3)` function. It is more secure than the older `mktemp(3)` function. Second, check that the file you are opening is a regular file, or at least not a symlink. If it is, respond reasonably by making a log entry and gracefully exiting. Also, check inode numbers using `lstat(2)` and `fstat(2)` to make sure you opened the file you expected to open.

Notes

The issue with `O_EXCL` is that it is supposed to allow a process to create a new file only if none is currently there. The problem with it not working under NFS is that should your program try to open a file on an NFS volume, you are not guaranteed this. The oft-quoted manual page gives a longer explanation:

`O_EXCL` When used with `O_CREAT`, if the file already exists it is an error and the open will fail. `O_EXCL` is broken on NFS file systems, programs which rely on it for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same fs (e.g., incorporating hostname and pid), use link(2) to make a link to the lockfile and use stat(2) on the unique file to check if its link count has increased to 2. Do not use the return value of the link() call.

The OpenBSD `mkstemp(3)` man page gives further insight into the complexities of this problem [ope]:

> BUGS
>
> For mktemp() there is an obvious race between file name selection and file creation and deletion: the program is typically written to call tmpnam(3), tempnam(3), or mktemp(). Subsequently, the program calls open(2) or fopen(3) and erroneously opens a file (or symbolic link, fifo or other device) that the attacker has created in the expected file loca- tion. Hence mkstemp() is recommended, since it atomically creates the file. An attacker can guess the filenames produced by mktemp(). Whenev- er it is possible, mkstemp() or mkdtemp() should be used instead.
>
> For this reason, ld(8) will output a warning message whenever it links code that uses the mktemp().
>
> The mkdtemp() and mkstemps() functions are nonstandard and should not be used if portability is required.
>
> EXAMPLES
>
> Quite often a programmer will want to replace a use of mktemp() with mkstemp(), usually to avoid the problems described above. Doing this correctly requires a good understanding of the code in question.
>
> For instance, code of this form:

```
char sfn[15] = "";
FILE *sfp;

strcpy(sfn, "/tmp/ed.XXXXXX");
if (mktemp(sfn) == NULL || (sfp = fopen(sfn, "w+")) == NULL) {
        fprintf(stderr, "%s: %s\n", sfn, strerror(errno));
        return (NULL);
}
return (sfp);
```

> should be rewritten like this:

```
char sfn[15] = "";
FILE *sfp;
int fd = -1;

strcpy(sfn, "/tmp/ed.XXXXXX");
if ((fd = mkstemp(sfn)) == -1 ||
    (sfp = fdopen(fd, "w+")) == NULL) {
        if (fd != -1) {
                unlink(sfn);
                close(fd);
        }
        fprintf(stderr, "%s: %s\n", sfn, strerror(errno));
        return (NULL);
}
return (sfp);
```

Often one will find code which uses mktemp() very early on, perhaps to globally initialize the template nicely, but the code which calls open(2) or fopen(3) on that filename will occur much later. (In almost all cas- es, the use of fopen(3) will mean that the flags O_CREAT — O_EXCL are not given to open(2), and thus a symbolic link race becomes possible, hence making necessary the use of fdopen(3) as seen above). Furthermore, one must be careful about code which opens, closes, and then re-opens the file in question. Finally, one must ensure that upon error the temporary file is removed correctly.

There are also cases where modifying the code to use mktemp(), in concert with open(2) using the flags O_CREAT — O_EXCL, is better, as long as the code retries a new template if open(2) fails with an errno of EEXIST.

4 Methods

References

[Asl95] Taimur Aslam. A taxonomy of security faults in the unix operating system. Master's thesis, August 1995. http://www.cerias.purdue.edu/coast/coast-library.html.

[BK00] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56(05), May 01, 2000. http://phrack.infonexus.com/search.phtml?view&article=p55-5.

[BTS99] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks, December 25, 1999. http://www.bell-labs.com/org/11356/libsafe.html.

[Cle98] Glynn Clements. Re: Xinetd /tmp race? Posted to *bugtraq* mailing list, November 12, 1998.
http://www.securityfocus.com/templates/archive.pike?list=1&date=1998-11-8&msg=13899.18573.790989.190546@cerise.sensei.co.uk.

[CPMea] Crispin Cowan, Calton Pu, Dave Maier, and et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. Proceeding of Usenix 1998.
http://immunix.org/stackguard.html.

[Des] Solar Designer. Linux kernel patch from the openwall project.
http://www.openwall.com/linux/README.

[klo99] klog. The frame pointer overwrite. 55(08), September 09, 1999.
http://phrack.infonexus.com/search.phtml?view&article=p55-8.

[MdR] Todd C. Miller and Theo de Raat. strlcpy and strlcat — consistent, safe, string copy and concatenation. Proceedings of Usenix 1999.
http://www.usenix.org/events/usenix99/millert.html.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49(14), November 08, 1996.
http://phrack.infonexus.com/search.phtml?view&article=p49-14.

[ope] `mkstemp(3)` openbsd man page. june 4, 1993.
http://www.openbsd.org/cgi-bin/man.cgi?query=mkstemp&sektion=3&apropos=0&manpath=OpenBSD+Current.

[Pri00] Prizm. Remote shell via qpopper2.53. buffer0verfl0w security, May 23, 2000.
http://www.digibel.org/ b0f/advisors/b0f5-Qpopper.txt.

[Sho00] Adam Shostack. Security code review guidelines, May 02, 2000.
http://www.homeport.org/ adam/review.html.

[Twi00] Twitch. Taking advantage of non-terminated adjacent memory spaces. 56(0x0e), May 1, 2000.
http://phrack.infonexus.com/search.phtml?view&article=p56-14.

[Ven] Vendicator. Stack shield: A 'stack smashing' technique protection tool for linux.
http://www.angelfire.com/sk/stackshield/.

[Whe00] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO, version 2.00.* April 21, 2000.
http://www.dwheeler.com/secure-programs/.