More About
- Bulletins
- Challenges
- Crypto FAQ
- CryptoBytes
- RSA Algorithm
- PKCS
- Advanced Encryption Standard
- Tech Notes
- Staff & Associates
- Standards

Countermeasures against Buffer Overflow Attacks

Niklas Frykholm (nfrykholm@rsasecurity.com)
30 November, 2000

1. Introduction

Buffer overflow problems are responsible for a large number of security vulnerabilities. Of the 44 CERT advisories published between 1997 and 1999, 24 were related to buffer overflow issues. [13]

Buffer overflow problems are caused by programming errors. In theory, if every programmer learned to write better, more defensive code, the problem would go away. In practice this is not likely to occur. Human beings are error-prone and experience shows that it is highly unrealistic to expect programmers to produce bug free code.

The purpose of this document is to examine to what extent automated tools can be used to reduce the risk of buffer overflow vulnerabilities. We look at the possible ways of dealing with buffer overflows, survey the existing tools and compare the tradeoffs they make between security and efficiency. We also discuss whether there are any good buffer overflow prevention techniques that have not yet been automated.

2. Buffer Overflow Attacks

To understand how we can defend against buffer overflow attacks, we must first understand how these attacks work.

The term *buffer* refers to an allocated chunk of memory, such as a pointer, array or string. In C and C++ there is no automatic bounds checking on buffers, which means that it is easy for the programmer to write past the end of the buffer. The code below shows an example:

```
void f() {
 int a[10];
 a[20] = 3;
}
```

In most cases, writing past the end of the buffer causes the program to crash with a segmentation fault error, but does not result in a security vulnerability. For a security vulnerability to occur, two conditions must be fulfilled:

1. The attacker must be able to control the data written into the buffer.
2. There must be security sensitive variables stored after the buffer in memory.

For example, consider the following example:

```
int main(int argc, char *argv[]) {
```

```
 char passwd_ok = 0;
 char passwd[8];
 strcpy(passwd, argv[1]);
 if (strcmp(passwd, "niklas")==0)
  passwd_ok = 1;

 if (passwd_ok) {
  ...
}
```

The layout in memory looks something like this:

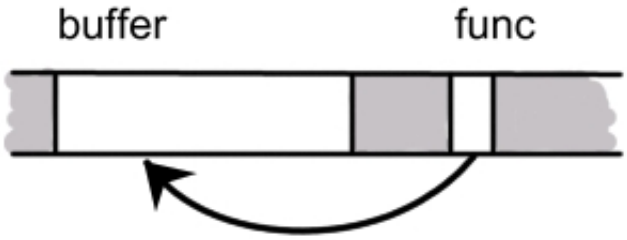
The `strcpy` function makes no check that `argv[1]` contains at most 8 characters, so an attacker that passes a longer string can overflow the `passwd` buffer. If the string is long enough, the `passwd_ok` flag will be overwritten and the password will be accepted by the program, no matter what it is.

Since `strcpy` does not take any parameter that specifies the size of the destination buffer it has for long been recognized as especially vulnerable to buffer overflow attacks. Other vulnerable functions of the same type are `gets`, `sprintf`, `scanf` and `strcat`.

Security flags, as the one above, are not the only sensitive variables. A more common, but less obvious vulnerability is caused by the use of function pointers. The code below shows an example:

```
int main(int argc, char *argv[]) {
 void (*func)() = f;
 char buffer[100];
 gets(buffer);
 func();
 ...
}
```

Since `gets` does not check the length of the buffer, an attacker can overflow it and overwrite `func`. Later, when `func` is called, the program will jump to a memory location determined by the attacker. The attacker can jump to a function in the program or one of its libraries, or he can be even more clever and jump into the buffer that he himself has supplied.



Since the attacker determines the content of this buffer, he can get the program to execute arbitrary code. (Well, not completely arbitrary. The code cannot contain any `\n` bytes since that would terminate the `gets` function.)

The good news for the attacker is that there is *always* a function pointer to overwrite, whether one has been declared or not. To understand why, we must look at what happens (on most computer platforms) when a C function is called.

When a function is called in C, the caller begins by pushing the function parameters to the stack. Thereafter, the caller pushes the address of its next instruction --- the address where execution should continue when the function returns --- to the stack and jumps to the function. The callee, in turn, makes room on the stack for its local variables. In most computer architectures the stack grows from high memory addresses to low, so the memory layout after the call will look something like this:

By overflowing a buffer in the local variables, the attacker can overwrite the return address. This means that when the function is done, it will not return to the caller, instead it will jump to an address determined by the attacker. Thus, the effect is similar to overwriting a function pointer as in the previous example.

It should be noted that the attacker does not only control the address that is jumped to, but also the entire content of the stack. Since the stack is where function parameters are stored, this means that the attacker can in fact call any function in the program or in the libraries used by it, and specify arbitrary parameters. (Again, not completely arbitrary. Many overflows are stopped by a zero byte. If that is the case, the only place where the attacker will be able to insert a zero byte is at the very end of the parameter list.) Thus, it is not necessary for the attacker to inject his own code into the buffer, since he can usually find an existing function with sufficiently devastating effects.

We will use the name **stack attack** to refer to attacks where the target is the return address on the stack. Attacks where other sensitive variables are overwritten will be referred to as **variable attacks**. (Sometimes attacks of this type are called **heap attacks** but that is a bit confusing, because the sensitive variables could be stored on the stack as well as on the heap.)

3. Countermeasures

As we have seen, a buffer overflow attack requires two things. First, a buffer overflow must occur in the program. Second, the attacker must be able to use the buffer overflow to overwrite a security sensitive piece of data (a security flag, function

pointer, return address, etc).

If we want to prevent buffer overflows completely we must stop one of these two things, i. e. either:

1. Prevent all buffer overflows *or*
2. Prevent all sensitive information from being overwritten

Both these solutions are costly in terms of efficiency and many programs therefore settle for a partial goal, such as:

- Prevent use of dangerous functions: `gets`, `strcpy`, etc.
- Prevent return addresses from being overwritten
- Prevent data supplied by the attacker from being executed (stops the attacker from jumping into his own buffer)

There are several possible levels where a defense mechanism can be inserted. At the **language level** we can make changes to the C language itself to reduce the risk of buffer overflows. At the **source code level** we can use static or dynamic source code analyzers to check our code for buffer overflow problems. At the **compiler level** we can change the compiler so that it does bounds checking or protects certain addresses from overwriting. At the **operating system level** we can change the rules for which memory pages that should be allowed to hold executable content.

In the following sections, we will look at each of these options in turn.

3.1 Language Tools

The simplest solution at the language level is to switch to a language that provides automatic bounds checking of buffers, such as Java, Perl or Python. However, in most projects this is not an option.

A better solution is to use a library module that implements "safe", bounds-checked buffers, such as the standard C++ **string** module or **libmib** [9]. There are two problems with this solution. First, it requires a complete rewrite of all the source code for the project. This alone makes the solution improper for anything but recently started projects. Second, most programs have to interface with prewritten library code. Since this code will use the ordinary "unsafe" buffers, the program will not be able to completely avoid them. Instead it will constantly have to convert between "safe" and "unsafe" buffers. Whenever a buffer is in the "unsafe" mode, buffer overflow problems can occur. There is also a risk that programmer's will forget to convert buffers back to the "safe" mode.

An alternative to making every buffer access safe is to target only those specific functions in C which are known to be dangerous, e. g. `strcpy`, `strcat`, `gets` and `sprintf`. This takes care of buffer overflows caused by these functions, but doesn't handle buffer overflows caused by other code, such as user written functions.

The simplest solution to securing the dangerous function is to disallow them. They all have "safe" counterparts that can be substituted, such as `strncpy` and `snprintf`, which in addition to the buffers also take a size parameter. (Of course, these functions are only "safe" if the size parameter is specified correctly.) Replacing the functions with safe calls is probably the best solution provided that the source code to the program is available. To make sure that the unsafe versions are not used by mistake, their prototypes can be removed from the header files.

The **libsafe** library from Bell Labs provides a way of securing calls to these functions, even if the source code is not available [1]. It does this by replacing the implementation of the dangerous functions in the shared libc library with safe versions.

libsafe makes use of the fact that stack frames are linked together by frame pointers (this is implementation dependent, but many C compilers on many platforms use this solution). When a buffer is passed as argument to one of the unsafe functions, libsafe follows the frame pointers to find the stack frame where the buffer was allocated (if it is not found, it is assumed that the buffer resides on the heap). It then checks the distance to the closest return address on the stack. When the function executes it makes sure that this address is not overwritten. If an attempt to overwrite this address is made, the program terminates with a vulnerability warning.

Finding the correct stack frame and protecting the return address requires some overhead which depends on how deep on the stack buffers are buried and how many calls to unsafe functions the program makes. Usually the overhead is quite small.

libsafe can only protect against stack attacks, not variable attacks, and only against vulnerabilities caused by one of the dangerous functions. Also, it is only really useful when we don't have access to the source code (otherwise, it is better to replace the unsafe calls). Thus, it is typically used by end users who want to add an extra measure of safety and not by programmers who want to make sure that their code is correct.

3.2 Source Code Tools

A source code tool analyzes the source code of a program and tries to determine whether it contains any dangerous constructions that could lead to buffer overflows.

We cannot expect a source code tool to be able to detect all possible instances of buffer overflow while at the same time yielding no false positives, since that is a task of the same difficulty as solving the halting problem. Constructing good source code tools will therefore always be a heuristic task.

The source code tools available today make only a limited, local analysis of the code and are therefore heavily restricted in the types of buffer overflow problems that they can detect.

its4 is a source code tool that checks for the use of dangerous function. It is a bit smarter than a plain `grep` search in that it can rule out some cases where the use of a dangerous function usually does not pose a problem. [3]

L0pht Heavy Industries have a web page advertising **slint**, a source code analyzer capable of detecting some buffer overflow problems [7]. However, it seems to be vaporware. L0pht does not give any information about what the program does or what classes of errors it is able to detect and people who have tried to order the product have been told that it is not ready yet [6].

An analyzer that could perform more extensive, cross-function analysis would be a valuable tool in a security audit. Constructing such a tool would however be a major undertaking. The tool would have to try to keep track of the size of all buffers and the possible ranges of variables to be able to detect when a buffer overflow might occur.

Dynamic analysis tools such as **purify** are an alternative to static analysis tools

[12]. A dynamic analysis tool analyzes the memory use of a program as it is run. Dynamic analysis tools can detect buffer overflow problems if they occur in a test run of a program. However, errors that occur in test runs can usually be detected even without an analysis tools, since they typically cause the program to crash. The main advantage of dynamic analyzers is that they allow for the error to be swiftly located once it has been detected.

3.3 Compiler Tools

A compiler tool changes the way a program is compiled, so that protection against buffer overflow is automatically compiled in with the program. No changes to the program's source code are thus necessary.

Buffer overflows can be prevented by adding bounds checking to all buffers. To do this, the compiler must add code for keeping track of the size of buffers and for checking that every buffer access falls within the allocated size. Herman ten Brugge has written a patch that adds bounds checking to the gcc compiler [2]. Another project for adding bounds checking to gcc is managed by Greg McGary [8].

The problem with adding bounds checking to every pointer is that it results in a great performance hit. Code size and execution time may grow by 200 % or more.

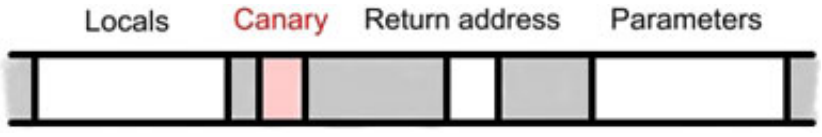
Instead of preventing all buffer overflows we might try to protect the return address from being overwritten. This does not protect against variable attacks, only against the more common stack attacks.

One possible way of doing this is to write the return address to a "safe" place (far from the local buffers) at the start of a function and then restore it just before the function returns. Since the function can call other functions which in turn need to store their return addresses we will need a stack to keep track of all the stored return addresses. In practice this solution thus means separating the stack used for return addresses from the stack used for local variables. This requires a lot of changes to the compiler. A program called **StackShield** implements this approach, but is still in beta [15].

Instead of moving the return address, we can move the buffers. For example, we could allocate all buffers in heap space instead of in stack space. The disadvantage of this solution is that the heap is substantially slower than the stack. I do not know of anyone who has implemented this solution, but the idea has circulated [11].

A slightly different approach is taken by the **StackGuard** program [4]. StackGuard does not prevent the return address from being overwritten, instead it tries to detect when it happens and take the appropriate action (terminating the program before any damage is done).

StackGuard accomplishes this in an ingenious way. Whenever a function is called, code is added for pushing a small value, called a "canary" value, to the stack. This value thus ends up between the local variables and the return address.



When the function exits it checks that the canary value has not been modified before returning. The idea is that a buffer overflow in one of the local variables cannot overwrite the return address without simultaneously destroying the integrity of the

canary value. It thus becomes possible to detect whether a buffer overflow has occurred before the function returns.

For this to work, the attacker must not be able to guess the canary value. If the attacker can correctly guess the canary value, he can overwrite the return address without being detected. StackGuard can set the canary value in one of two possible ways. A random value may be used, which is hard for the attacker to guess or the value 0 may be used, which is easy to guess but hard for the attacker to put into his buffer (since most sensitive buffer operations, such as string copying, are terminated by a zero value).

StackGuard can only protect against stack attacks, not against variable attacks, but there is ongoing work to add canary values to function pointers too, since they are also a vulnerable target. The added canary checks increase function call and return times with 40 - 80 %. If the program contains many small function calls and inlining is not used, the total overhead can be in this range. If the program does not contain as many function calls, the overhead will be smaller.

3.4 Operating System Tools

Some people have tried to solve the buffer overflow problem at the level of the operating system. The OS can impose some restrictions which make buffer overflow attacks harder.

Solar Designer has created a patch for Linux that makes the stack non-executable [5]. This means that the attacker can no longer inject his own code into the stack and run it. The patch also maps `libc` to the `0x00...` memory range, so that it is impossible to call `libc` functions without having a zero in the buffer. It does not, however, prevent the attacker from calling arbitrary functions in the program with arbitrary arguments. Usually, by making the right calls, the attacker can achieve the desired effect [17] [14], but the required effort will be greater than on a system where the patch is not used.

The disadvantage of having a non-executable stack is that some legitimate programs (though not very many) actually execute content on the stack. These programs will cease to work if the patch is applied.

4. Conclusions

No tool can completely solve the buffer overflow problem, but tools can increase the probability that a buffer overflow is detected and reduce the attacker's chance of successfully exploiting a vulnerability.

The Solar Designer patch makes attacks slightly harder for the attacker without reducing the program's performance. Its main drawback is that it is incompatible with a small number of programs. For a typical server machine that only runs a small fixed set of programs, this is not a big issue, because it is easy to check all programs for incompatibilities. Thus, I would recommend all Linux servers to run the Solar Designer patch. However, it is important not to be lulled in to a false sense of security, since its protection mechanisms can easily be bypassed.

For the same reason it makes sense to run libsafe on server machines. Performance before and after its installation should be measured and compared.

StackGuard is capable of stopping many types of buffer overflow attacks, but there is a performance penalty in using it. Depending on the type of application this may or may not be acceptable. There is also a psychological penalty involved. Releasing

a program under StackGuard is equivalent to admitting that there might be security related bugs in the application and this could have a negative effect on a company's image. This drawback is shared by all compiler tools, especially those with a non-negligible performance cost. In this respect, source code and operating system tools are superior.

There are at least three possible areas of future development for buffer overflow tools:

1. Many of the compiler patches and libraries only support Linux or other Unix derivatives and the gcc compiler. There is a need for developing similar tools with support for other compilers and operating systems.
2. There is a need for a powerful source analyzer to assist security auditors in locating potential buffer overflow problems. This is a major programming project.
3. The possibility of allocating all buffers on the heap should be investigated. In particular, it should be examined how this affects the performance of real-world programs.

References

1. Bell Labs. *libsafe*. http://www.bell-labs.com/org/11356/libsafe.html
2. Herman ten Brugge. *Bounds checking patch for gcc*. http://web.inter.nl.net/hcc/Haj.Ten.Brugge/
3. Cigital Security. *ITS4: Software Security Tool*. http://www.cigital.com/its4/
4. Crispin Cowan et al. *StackGuard*. http://www.cse.ogi.edu/DISC/projects/__immunix/StackGuard/usenixsc98_html/
5. Solar Designer. *Linux kernel patch from the Openwall project*. http://www.openwall.com/linux/
6. Marc Heuse. *Re: imapd4r1 v12.264 and security implications*. http://lists.suse.com/archives/suse-security/2000-Apr/0136.html
7. L0pht Heavy Industries. *SLINT - source code security analyzer*. http://www.l0pht.com/slint.html
8. Greg McGary. *Bounds checking projects*. http://gcc.gnu.org/projects/bp/main.html
9. Mib Software. *Libmib allocated string functions*. http://www.mibsoftware.com/libmib/astring/
10. Aleph One. *Smashing the stack for fun and profit*. http://www.linux.com/security/newsitem.phtml?sid=11&aid=5043
11. Aleph One. *BugTraq: Frequently asked questions*. http://www.nationwide.net/~aleph1/FAQ
12. Rational. *Rational Purify for Windows*. http://www.rational.com/products/purify_nt/index.jsp
13. Evan Thomas. *Attack Class: Buffer Overflows*. Hello World!. April, 1999. http://www.cosc.brocku.ca/~cspress/HelloWorld/1999/04-apr/attack_class.html
14. Linus Torvalds. *Re: [PATCH] [SECURITY] suid procs exec'd with bad 0,1,2 fds*. http://lwn.net/980806/a/linus-noexec.html
15. Vendicator. *StackShield*. http://www.angelfire.com/sk/stackshield/index.html
16. David A. Wheeler. *Secure programming for Linux and Unix howto*. http://www.linuxdoc.org/HOWTO/Secure-Programs-HOWTO/index.html
17. Rafal Wojtczuk. *Defeating Solar Deisgner non-executable stack patch*. http://www.securityfocus.com/archive/1/8470

United States: 1-877-RSA-4900 or 781 515 5000, Europe, Middle East, Africa: +44 (0)1344 781000,
Asia/Pacific: +65 733 5400, Japan: +81 3 5222 5200
Home | Contact Us | Search | Terms of Use and Privacy Statement