Buffer Overflow

From OWASP

This Page has been flagged for review. Please help OWASP and review this Page to FixME.
Comment: No real edits since 2009

*This is a **Vulnerability**. To view all vulnerabilities, please see the Vulnerability Category page.*

Last revision (mm/dd/yy): **06/29/2016**

Vulnerabilities Table of Contents

Related Security Activities

Description of Buffer Overflow

See the OWASP article on Buffer Overflow Attacks.

How to Avoid Buffer Overflow Vulnerabilities

See the OWASP Development Guide article on how to Avoid Buffer Overflow Vulnerabilities.

How to Review Code for Buffer Overflow Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for Buffer Overruns and Overflows Vulnerabilities.

How to Test for Buffer Overflow Vulnerabilities

See the OWASP Testing Guide article on how to Test for Buffer Overflow Vulnerabilities.

Overview

A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.

Description

Buffer overflow is probably the best known form of software security vulnerability. Most software developers know what a buffer overflow vulnerability is, but buffer overflow attacks against both legacy and newly-developed applications are still quite common. Part of the problem is due to the wide variety of ways buffer overflows can occur, and part is due to the error-prone techniques often used to prevent them.

Buffer overflows are not easy to discover and even when one is discovered, it is generally extremely difficult to exploit. Nevertheless, attackers have managed to identify buffer overflows in a staggering array of products and components.

In a classic buffer overflow exploit, the attacker sends data to a program, which it stores in an undersized stack buffer. The result is that information on the call stack is overwritten, including the function's return pointer. The data sets the value of the return pointer so that when the function returns, it transfers control to malicious code contained in the attacker's data.

Although this type of stack buffer overflow is still common on some platforms and in some development communities, there are a variety of other types of buffer overflow, including Heap buffer overflow and Off-by-one Error among others. Another very similar class of flaws is known as Format string attack. There are a number of excellent books that provide detailed information on how buffer overflow attacks work, including Building Secure Software [1], Writing Secure Code [2], and The Shellcoder's Handbook [3].

At the code level, buffer overflow vulnerabilities usually involve the violation of a programmer's assumptions. Many memory manipulation functions in C and C++ do not perform bounds checking and can easily overwrite the allocated bounds of the buffers they operate upon. Even bounded functions, such as strncpy(), can cause vulnerabilities when used incorrectly. The combination of memory manipulation and mistaken assumptions about the size or makeup of a piece of data is the root cause of most buffer overflows.

Buffer overflow vulnerabilities typically occur in code that:

- Relies on external data to control its behavior
- Depends upon properties of the data that are enforced outside of the immediate scope of the code
- Is so complex that a programmer cannot accurately predict its behavior

Buffer Overflow and Web Applications

Attackers use buffer overflows to corrupt the execution stack of a web application. By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code – effectively taking over the machine.

Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web applications use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks.

Buffer overflows can also be found in custom web application code, and may even be more likely given the lack of scrutiny that web applications typically go through. Buffer overflow flaws in custom web applications are less likely to be detected because there will normally be far fewer hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the flaw (other than to crash the application) is significantly reduced by the fact that the source code and detailed error messages for the application are normally not available to the hacker.

Consequences

- Category:Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe-string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or mis¬use of mitigating technologies.

Environments Affected

Almost all known web servers, application servers, and web application environments are susceptible to buffer overflows, the notable exception being environments written in interpreted languages like Java or Python, which are immune to these attacks (except for overflows in the Interpretor itself).

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed, depending on environment.

Required resources

Any

Severity

Very High

Likelihood of exploit

High to Very High

How to Determine If You Are Vulnerable

For server products and libraries, keep up with the latest bug reports for the products you are using. For custom application software, all code that accepts input from users via the HTTP request must be reviewed to ensure that it can properly handle arbitrarily large input.

How to Protect Yourself

Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products. Periodically scan your web site with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications. For your custom application code, you need to review all code that accepts input from users via the HTTP request and ensure that it provides appropriate size checking on all such inputs. This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

Risk Factors

TBD

Examples

Example 1.a

The following sample code demonstrates a simple buffer overflow that is often caused by the first scenario in which the code relies on external data to control its behavior. The code uses the gets() function to read an arbitrary amount of data into a stack buffer. Because there is no way to limit the amount of data read by this function, the safety of the code depends on the user to always enter fewer than BUFSIZE characters.

```
...
char buf[BUFSIZE];
gets(buf);
...
```

Example 1.b

This example shows how easy it is to mimic the unsafe behavior of the gets() function in C++ by using the >> operator to read input into a char[] string.

```
...
char buf[BUFSIZE];
cin >> (buf);
...
```

Example 2

The code in this example also relies on user input to control its behavior, but it adds a level of indirection with the use of the bounded memory copy function memcpy(). This function accepts a destination buffer, a source buffer, and the number of bytes to copy. The input buffer is filled by a bounded call to read(), but the user specifies the number of bytes that memcpy() copies.

```
...
char buf[64], in[MAX_SIZE];
printf("Enter buffer contents:\n");
read(0, in, MAX_SIZE-1);
printf("Bytes to copy:\n");
scanf("%d", &bytes);
memcpy(buf, in, bytes);
...
```

Note: This type of buffer overflow vulnerability (where a program reads data and then trusts a value from the data in subsequent memory operations on the remaining data) has turned up with some frequency in image, audio, and other file processing libraries.

Example 3

This is an example of the second scenario in which the code depends on properties of the data that are not verified locally. In this example a function named lccopy() takes a string as its argument and returns a heap-allocated copy of the string with all uppercase letters converted to lowercase. The function performs no bounds checking on its input because it expects str to always be smaller than BUFSIZE. If an attacker bypasses checks in the code that calls lccopy(), or if a change in that code makes the assumption about the size of str untrue, then lccopy() will overflow buf with the unbounded call to strcpy().

```
char *lccopy(const char *str) {
        char buf[BUFSIZE];
        char *p;

        strcpy(buf, str);
        for (p = buf; *p; p++) {
                 if (isupper(*p)) {
                        *p = tolower(*p);
                }
        }
        return strdup(buf);
}
```

Example 4

The following code demonstrates the third scenario in which the code is so complex its behavior cannot be easily predicted. This code is from the popular libPNG image decoder, which is used by a wide array of applications, including Mozilla and some versions of Internet Explorer.

The code appears to safely perform bounds checking because it checks the size of the variable length, which it later uses to control the amount of data copied by png_crc_read(). However, immediately before it tests length, the code performs a check on png_ptr->mode, and if this check fails a warning is issued and processing continues. Because length is tested in an else if

block, length would not be tested if the first check fails, and is used blindly in the call to png_crc_read(), potentially allowing a stack buffer overflow.

Although the code in this example is not the most complex we have seen, it demonstrates why complexity should be minimized in code that performs memory operations.

```
if (!(png_ptr->mode & PNG_HAVE_PLTE)) {
        /* Should be an error, but we can cope with it */
        png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette) {
        png_warning(png_ptr, "Incorrect tRNS chunk length");
        png_crc_finish(png_ptr, length);
        return;
}
...
png_crc_read(png_ptr, readbuf, (png_size_t)length);
```

Example 5

This example also demonstrates the third scenario in which the program's complexity exposes it to buffer overflows. In this case, the exposure is due to the ambiguous interface of one of the functions rather the structure of the code (as was the case in the previous example).

The getUserInfo() function takes a username specified as a multibyte string and a pointer to a structure for user information, and populates the structure with information about the user. Since Windows authentication uses Unicode for usernames, the username argument is first converted from a multibyte string to a Unicode string. This function then incorrectly passes the size of unicodeUser in bytes rather than characters. The call to MultiByteToWideChar() may therefore write up to (UNLEN+1)*sizeof(WCHAR) wide characters, or (UNLEN+1)*sizeof(WCHAR)*sizeof(WCHAR) bytes, to the unicodeUser array, which has only (UNLEN+1)*sizeof(WCHAR) bytes allocated. If the username string contains more than UNLEN characters, the call to MultiByteToWideChar() will overflow the buffer unicodeUser.

```
void getUserInfo(char *username, struct _USER_INFO_2 info){
        WCHAR unicodeUser[UNLEN+1];
        MultiByteToWideChar(CP_ACP, 0, username, -1,
                                                unicodeUser, sizeof(unicodeUser));
NetUserGetInfo(NULL, unicodeUser, 2, (LPBYTE *)&info);
}
```

Related Attacks

- Format string attack

Related Vulnerabilities

- Heap buffer overflow

Related Controls

- Category:Input Validation
- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.
- Pre-design through Build: Compiler-based canary mechanisms such as StackGuard, ProPolice and the Microsoft Visual Studio / GS flag. Unless this provides automatic bounds checking, it is not a complete solution.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Categories: FIXME/pls review | Vulnerability | FIXME