CHAPTER

18

Source Code Auditing: Finding Vulnerabilities in C-Based Languages

Auditing software with the source code is often the most effective way to discover new vulnerabilities. A large amount of widely deployed software is open source, and some commercial vendors have shared their operating system source code with the public. With some experience, it is possible to detect obvious flaws quickly and more subtle flaws with time. Although binary analysis is always a possibility, the availability of source code makes auditing much easier. This chapter covers auditing source code written in C-based languages for both simple and subtle vulnerabilities, and mainly focuses on detecting memory-corruption vulnerabilities.

Many people audit source code, and each has his or her own reasons for doing so. Some audit code as part of their jobs or as a hobby, whereas others simply want an idea of the security of the applications they run on their systems. There are undoubtedly people out there who audit source code to find ways to break into systems. Whatever the reason for auditing, source code review is arguably the best way to discover vulnerabilities in applications. If the source code is available, use it.

The argument about whether it's more difficult to find bugs or to exploit them has been thrown around a fair bit, and cases can be made for either side. Some vulnerabilities are extremely obvious to anyone reading the source but turn out to be nearly unexploitable in any practical situation. However, the opposite is more common, and in my experience, the bottleneck in vulnerability

research is most often the discovery of quality vulnerabilities and not their exploitation.

Some vulnerabilities are immediately recognizable and can quickly be spotted. Others are quite difficult to see, even when they are pointed out to you. Different software packages offer different difficulty levels and different challenges. There is undoubtedly a lot of badly written software out there, but at the same time, very secure open source software exists as well.

Successful auditing is based on recognition and understanding. Many vulnerabilities that have been discovered in different applications are quite similar, and if you can find or recognize a vulnerability in one application, there's a good chance that the same mistake was made somewhere else. Locating more subtle issues requires deeper understanding of the application, and in general has a scope much larger than any single function. An in-depth knowledge of the application being audited is very helpful.

Quite honestly, there are a lot more people doing source code auditing now than was the case several years ago, and as time goes on, the more obvious and easy-to-spot bugs will be found. Developers are becoming more aware of security issues and less likely to repeat mistakes. Simple mistakes that make it into release software are usually quickly spotted and pounced upon by vulnerability researchers. It would be easy to argue that vulnerability research will only become more difficult as time goes on; however, there is new code being authored on a continual basis, and new bug classes are unearthed occasionally. You can rely on the fact that security vulnerabilities remain in every major software application. It is simply a matter of finding them.

Tools

Source code auditing can be a painful task if you're armed only with a text editor and grep. Fortunately, some very useful tools are available that make source code auditing much easier. In general, these tools have been written to aid software development but work just as well for auditing. For small applications, it's not always necessary to use any specialized tools, but for larger applications that span multiple files and directories, these tools become very useful.

Cscope

Cscope is a source code browsing tool that is very useful for auditing large source code trees. It was originally developed at Bell Labs, and has been made publicly available under the BSD license by SCO. You can find it at `http://cscope.sourceforge.net/`.

Cscope can locate the definition of any symbol or all references to a symbol of a given name, among other things. It can also locate all calls to a given function or locate all functions called by a function. When run, Cscope generates a database of symbols and references, and can be used recursively. It will easily handle the source code for an entire operating system and can make searching for specific vulnerability types across a large code base much easier. It will work on virtually every Unix variant with curses support, and there are precompiled Windows binaries available for download. Cscope can be invaluable for auditing and is used by many security researchers on a regular basis.

Cscope support is built into many editors, including Vim and Emacs, and it can be invoked from within those editors.

Ctags

Ctags is useful specifically for locating any language tags (symbols) within a large code base. Ctags creates a tag file that contains location information for language tags in files scanned. Many editors support this tag file format, which can allow for easy browsing of source code from within your favorite editor. Tag files can be created for many languages, including, most importantly, C and C++. One of Ctags's useful features is its ability to immediately go to a tag highlighted by the cursor, and then to return to the previous location or to a location farther up the tag stack. This feature allows your source code browsing to approximate the flow of execution. Ctags can be downloaded from `http://ctags.sourceforge.net/`; in addition, many Linux distributions offer a precompiled package.

Editors

Which text editor you use when viewing source code can make a big difference in ease of auditing. Certain editors offer features that are more conducive to development and source code auditing and make better choices. Two of these editors—Vim, the enhanced version of vi, and Emacs—offer complementary features, in addition to many features that are specifically added to make writing and searching through large amounts of code easy. Many editors offer features such as bracket-matching, which allows you to locate the partner of any opening or closing bracket. This can be very useful when auditing code with many nested brackets in complex patterns.

Many people have strong opinions about text editors and use their preferred editor religiously. Although some editors are inherently better suited for the task than others, the most important thing when choosing an editor is to pick something you're familiar with and comfortable using.

Cbrowser

Many other tools offer similar functionality to Cscope and Ctags. Cbrowser, for example, offers a graphical front-end for Cscope and can be useful for people who audit in a GUI environment.

Automated Source Code Analysis Tools

There are several publicly available tools that attempt to perform static analysis of source code and automatically detect vulnerabilities. Most of these are useful as a starting point for a novice auditor, but none of them have progressed to the level of replacing a thorough audit by an experienced person. Many large software vendors use static analysis tools in-house to detect simple vulnerabilities before they make it into production code. However, the shortfalls of these tools are obvious. Nonetheless, they can be a useful place to get a quick start on a large and relatively un-audited source tree.

Splint is a static-analysis tool designed to detect security problems within C programs. With annotations added to programs, Splint has the ability to perform relatively strong security checking. The analysis engine has in the past been shown to detect security problems such as the BIND TSIG overflow automatically (albeit after they were already known). Although Splint has trouble dealing with large and complex source trees, it's still worth looking at. It is developed by the University of Virginia and you can find it at `www.splint.org/`.

CQual is an application that evaluates annotations that have been added to C source code. It extends the standard C type qualifiers with additional qualifiers such as *tainted*, and has logic to infer the type of variables whose qualifiers have not been explicitly defined. CQual can detect certain vulnerabilities such as format strings; however, it will not find some of the more advanced issues that can be discovered by manual analysis. CQual was written by Jeff Foster and can be downloaded from `http://www.cs.umd.edu/~jfoster/cqual/`.

Other tools, such as RATS offered by Secure Software, are available, but they were generally designed to locate simplistic vulnerabilities not commonly found in modern software. Some bug classes better lend themselves to detection via static analysis, and several other publicly available tools automatically detect potential format string vulnerabilities.

In general, the current set of static-analysis tools is lacking when it comes to detecting the relatively complicated vulnerabilities found in modern software. Though these may be good for a beginner, most serious auditors will go far beyond the subset of vulnerabilities for which these programs can check.

Methodology

Sometimes security researchers can get lucky when auditing an application without following any concrete plan. They might just happen to read the right piece of code at the right time and see something that has gone unnoticed before. If, however, your goal is to find a definite vulnerability in a particular application or to attempt to find all bugs in one application (as is the case in any professional source code audit), then a more well-defined methodology is needed. What that methodology turns out to be depends on your goals and the types of vulnerabilities you are looking for. Some possible ways to audit source code are outlined here.

Top-Down (Specific) Approach

In the *top-down* approach to source code auditing, an auditor looks for certain vulnerabilities without needing to gain a larger understanding of how the program functions. For example, an auditor might search an entire source tree for format-string vulnerabilities affecting the `syslog` function without reading the program line-by-line. This method can be quick, of course, because the auditor does not have to gain an in-depth understanding of the application, but there are drawbacks to auditing this way. Any vulnerabilities that require deep knowledge of the context of the program or that span more than one part of the code will probably be missed. Some vulnerabilities can be easily located simply by reading one line of code; those are the types of bugs that will be found by the top-down method. Anything requiring more comprehension will need to be located another way.

Bottom-Up Approach

In the *bottom-up* approach to source code auditing, an auditor attempts to gain a very deep understanding of the inner workings of an application by reading a large portion of the code. The obvious place to start the bottom-up approach is the `main` function, reading from there to gain an understanding of the program from its entry point to its exit point. Although this method is time consuming, you can gain a comprehensive understanding of the program, which allows you to more easily discover more subtle bugs.

Selective Approach

Both the previous approaches have problems that prevent them from being effective and timely methods for locating bugs. Using a combination of the

two, however, can be more successful. Most of the time, a significant percentage of any code base is dead code when it comes to looking for security vulnerabilities. For example, a buffer overflow in the code for parsing a root-owned configuration file for a Web server is not a real security issue. To save time and effort, it's more effective to focus auditing on sections of code most likely to contain security issues that would be exploitable in real-world situations.

In the *selective* approach to source code auditing, an auditor will locate code that can be reached with attacker-defined input and focus extensive auditing energy on that section of the code. It's useful, however, to have a deep understanding of what these critical portions of code do. If you don't know what the piece of code you're auditing does or where it fits into an application, you should take time to learn it first, so that you do not waste time on an unprofitable audit. There's nothing more frustrating than finding a great bug in dead code or in a place in which you can't control input.

In general, most successful auditors use the selective approach. The selective methodology for source code auditing is generally the most effective manual method of locating true vulnerabilities in an application.

Vulnerability Classes

A list of bug classes that are commonly or not so commonly found in applications is always valuable. Although our list is definitely not all-inclusive, it does attempt to list most of the bug classes found in applications today. Every few years, a new bug class is unearthed and a new slew of vulnerabilities are found almost immediately. The remaining vulnerabilities are found as time progresses; the key to finding them at all is recognition.

Generic Logic Errors

Though the *generic logic error* class of vulnerabilities is the most non-specific, it is often the root cause of many issues. You must understand an application reasonably well in order to find flaws in programming logic that can lead to security conditions. It is helpful to understand internal structures and classes specific to an application and to brainstorm ways in which they might be misused. For example, if an application uses a common buffer structure or string class, an in-depth understanding will allow you to look for places in the application where members of the structure or class are misused or used in a suspicious way. When auditing a reasonably secure and well-audited application, hunting for generic logic errors is often the next best course of action to take.

(Almost) Extinct Bug Classes

A few vulnerabilities that were commonly found in open source software five years ago have now been hunted to near-extinction. These types of vulnerabilities generally took the form of well-known unbounded memory copy functions such as `strcpy`, `sprintf`, and `strcat`. While these functions can be and still are used safely in many applications, it was historically common to see these functions misused, which could lead to buffer overflows. In general, however, these types of vulnerabilities are no longer found in modern open source software.

`Strcpy`, `sprintf`, `strcat`, `gets`, and many similar functions have no idea of the size of their destination buffers. Provided that the destination buffers have been properly allocated, or verification of the input size is done before any copying, it is possible to use most of these functions without danger. When adequate checking is not performed, however, these functions become a security risk. We now have a significant amount of awareness about the security issues associated with these functions. For example, the man pages for `sprintf` and `strcpy` mention the dangers of not doing bounds-checking before calling these functions.

An overflow in the University of Washington IMAP server, fixed in 1998, provides an example of this type of vulnerability. The vulnerability affected the `authenticate` command and was simply an unbounded string copy to a locate stack buffer.

```
char tmp[MAILTMPLEN];
AUTHENTICATOR *auth;
                              /* make upper case copy of mechanism name */
ucase (strcpy (tmp,mechanism));
```

The conversion of the input string to uppercase provided an interesting challenge for exploit developers at the time; however, it does not present any real obstacle today. The fix for the vulnerability was simply to check the size of the input string and reject anything too long.

```
                                /* cretins still haven't given up */
if (strlen (mechanism) >= MAILTMPLEN)
  syslog (LOG_ALERT|LOG_AUTH,"System break-in attempt, host=%.80s",
          tcp_clienthost ());
```

Format Strings

The *format string* class of vulnerabilities surfaced sometime in the year 2000 and has resulted in the discovery of significant vulnerabilities in the past few years. The format string class of bugs is based on the attacker being able to

control the format string passed to any of several functions that accept `printf`-style arguments (including `*printf`, `syslog`, and similar functions). If an attacker can control the format string, he or she can pass directives that will result in memory corruption and arbitrary code execution. The exploitation of these vulnerabilities has largely been based on the use of the previously obscure `%n` directive in which the number of bytes already printed is written to an integer pointer argument.

Format strings are very easy to find in an audit. Only a limited number of functions accept `printf`-style arguments; it is quite often enough to identify all calls to these functions and verify whether an attacker can control the format string. For example, the following vulnerable and non-vulnerable `syslog` calls look strikingly different.

Possibly Vulnerable

```
syslog(LOG_ERR,string);
```

Non-Vulnerable

```
syslog(LOG_ERR,"%s",string);
```

The "possibly vulnerable" example could be a security risk if `string` is controllable by an attacker. You must often trace back the flow of data by several functions to verify whether a format string vulnerability does indeed exist. Some applications implement their own custom implementations of `printf`-like functions, and auditing should not be limited to only a small set of functions. Auditing for format string bugs is quite cut and dried, and it is possible to determine automatically whether a bug exists.

The most common location in which to find format string bugs is logging code. It is quite common to see a constant format string passed to a logging function, only to have the input printed to a buffer and passed to `syslog` in a vulnerable fashion. The following hypothetical example illustrates a classic format string vulnerability within logging code:

```
void log_fn(const char *fmt,…) {

     va_list args;
     char log_buf[1024];

     va_start(args,fmt);

     vsnprintf(log_buf,sizeof(log_buf),fmt,args);

     va_end(args);

     syslog(LOG_NOTICE,log_buf);

}
```

Format string vulnerabilities were initially exposed in the wu-ftpd server and then subsequently found in many other applications. However, because these vulnerabilities are very easy to find in an audit, they have been nearly eliminated from most major open source software packages.

Generic Incorrect Bounds-Checking

In many cases, applications will make an attempt at bounds-checking; however, it is reasonably common that these attempts are done incorrectly. Incorrect bounds-checking can be differentiated from classes of vulnerabilities in which no bounds-checking is attempted, but in the end, the result is the same. Both of these types of errors can be attributed to logic errors when performing bounds-checking. Unless in-depth analysis of bounds-checking attempts is performed, these vulnerabilities might not be spotted. In other words, don't assume that a piece of code is not vulnerable simply because it makes some attempt at bounds-checking. Verify that these attempts have been done correctly before moving on.

The Snort RPC preprocessor bug found by ISS X-Force in early 2003 is a good example of incorrect bounds-checking. The following code is found within vulnerable versions of Snort:

```
while(index < end)
{
        /* get the fragment length (31 bits) and move the pointer to the
           start of the actual data */
        hdrptr = (int *) index;

        length = (int)(*hdrptr & 0x7FFFFFFF);

        if(length > size)
        {
            DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad
"
                    "length: %d\n", length);
            return;
        }
        else
        {
            total_len += length;
            index += 4;
            for (i=0; i < length; i++,rpc++,index++,hdrptr++)
                *rpc = *index;
        }
}
```

In the context of this application, `length` is the length of a single RPC fragment, and `size` is the size of the total data packet. The output buffer is the

same as the input buffer and is referenced in two different locations by the variables `rpc` and `index`. The code is attempting to reassemble RPC fragments by removing the headers from the data stream. Through each iteration of the loop, the position of `rpc` and `index` is incremented, and `total_len` represents the size of data written to the buffer. An attempt was made at bounds-checking; however, that check is flawed. The length of the current RPC fragment is checked against the total data size. However, the correct check would be to compare the total length of all RPC fragments, including the current one, against the size of the buffer. The check is insufficient, but still present, and if you make a cursory examination of the code, you might simply dismiss the check as being valid—this example highlights the need for verifying all bounds-checking in important areas.

Loop Constructs

Loops are a very common place to find buffer overflow vulnerabilities, possibly because their behavior is a little more complicated, from a programming perspective, than linear code. The more complex a loop, the more likely that a coding error will introduce a vulnerability. Many widely deployed and security-critical applications contain very intricate loops, some of which are insecure. Often, programs contain loops within loops, leading to a complex set of interactions that is prone to errors. Parsing loops or any loops that process user-defined input are a good place to start when auditing any application, and focusing on these areas can give good results with minimal effort.

A good example of a complex loop gone wrong is the vulnerability found in the `crackaddr` function within Sendmail, by Mark Dowd. The loop in this particular function is far too large to show here and is up there on the list of complex loops found in open source software. Due to its complexity and the number of variables manipulated within the loop, a buffer overflow condition occurs when several patterns of input data are processed by this loop. Although many checks had been put in place in Sendmail to prevent buffer overflows, there were still unexpected consequences to the code. Several third-party analyses of this vulnerability, including one by the Polish security researcher group, The Last Stage of Delirium, understated the exploitability of this bug simply because they missed one of the input patterns that led to buffer overflow.

Off-by-One Vulnerabilities

Off-by-one vulnerabilities, or off-by-a-few vulnerabilities, are common coding errors in which one or a very limited number of bytes are written outside the bounds of allocated memory. These vulnerabilities are often the result of incorrect

null-termination of strings and are frequently found within loops or introduced by common string functions. In many cases, these vulnerabilities are exploitable, and they have been found in some widely deployed applications in the past.

For example, the following code was found in Apache 2 prior to 2.0.46, and was patched somewhat silently:

```
            if (last_len + len > alloc_len) {
                char *fold_buf;
                alloc_len += alloc_len;
                if (last_len + len > alloc_len) {
                   alloc_len = last_len + len;
                }
                fold_buf = (char *)apr_palloc(r->pool, alloc_len);
                memcpy(fold_buf, last_field, last_len);
                last_field = fold_buf;
            }
           memcpy(last_field + last_len, field, len +1); /* +1 for nul */
```

In this code, which deals with MIME headers sent as part of a request to the Web server, if the first two `if` statements are true, the buffer allocated will be 1 byte too small. The final memcpy call will write a null byte out of bounds. Exploitability of this bug proved to be very difficult due to the custom heap implementation; however, it is a blatant case of an off-by-one dealing with null-termination.

Any loop that null-terminates a string at the end of it should be double-checked for an off-by-one-condition. The following code, found within the OpenBSD ftp daemon, illustrates the problem:

```
    char npath[MAXPATHLEN];
        int i;

        for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++)
{
                npath[i] = *name;
                if (*name == '"')
                        npath[++i] = '"';
        }
        npath[i] = '\0';
```

Although the code attempts to reserve space for a null byte, if the last character at the boundary of the output buffer is a quote, an off-by-one condition occurs.

Certain library functions introduce off-by-one conditions if used improperly. For example, the function `strncat` always null-terminates its output string and will write a null byte out of bounds if not properly called with a

third argument equal to the space remaining in the output buffer less one for a null byte.

The following example shows incorrect usage of `strncat`:

```
strcpy(buf,"Test:");
strncat(buf,input,sizeof(buf)-strlen(buf));
```

The safe usage would be:

```
strncat(buf,input,sizeof(buf)-strlen(buf)-1);
```

Non-Null Termination Issues

For strings to be handled securely, they must generally be properly null-terminated so that their boundaries can be easily determined. Strings not properly terminated may lead to an exploitable security issue later in program execution. For example, if a string is not properly terminated, adjacent memory may be considered to be part of the same string. This situation can have several effects, such as significantly increasing the length of the string or causing operations that modify the string to corrupt memory outside the bounds of the string buffer. Some library functions inherently introduce issues associated with null-termination and should be looked for when auditing source code. For example, if the function `strncpy` runs out of space in the destination buffer, it will not null-terminate the string it writes. The programmer must explicitly do null-termination, or risk a potential security vulnerability. The following code, for example, would not be safe:

```
char dest_buf[256];
char not_term_buf[256];

strncpy(not_term_buf,input,sizeof(non_term_buf));

strcpy(dest_buf,not_term_buf);
```

Because the first strncpy will not null-terminate `non_term_buf`, the second `strcpy` isn't safe even though both buffers are of the same size. The following line, inserted between the `strncpy` and `strcpy`, would make the code safe from buffer overflow:

```
not_term_buf[sizeof(not_term_buf) - 1] = 0;
```

Exploitability of these issues is somewhat limited by the state of adjacent buffers, but in many cases these bugs can lead to arbitrary code execution.

Skipping Null-Termination Issues

Some exploitable coding errors in applications are the result of being able to skip past the null-terminating byte in a string and continue processing into undefined memory regions. Once the null-terminating byte has been skipped, if any further processing results in a write operation, it may be possible to cause memory corruption that leads to arbitrary code execution. These vulnerabilities usually surface in string processing loops, especially where more than one character is processed at a time or where assumptions about string length are made. The following code example was present until recently in the `mod_rewrite` module in Apache:

```
        else if (is_absolute_uri(r->filename)) {
            /* it was finally rewritten to a remote URL */

            /* skip 'scheme:' */
            for (cp = r->filename; *cp != ':' && *cp != '\0'; cp++)
                ;
            /* skip '://' */
            cp += 3;
```

where `is_absolute_uri` does the following:

```
int i = strlen(uri);
if (   (i > 7 && strncasecmp(uri, "http://",   7) == 0)
    || (i > 8 && strncasecmp(uri, "https://",  8) == 0)
    || (i > 9 && strncasecmp(uri, "gopher://", 9) == 0)
    || (i > 6 && strncasecmp(uri, "ftp://",    6) == 0)
    || (i > 5 && strncasecmp(uri, "ldap:",     5) == 0)
    || (i > 5 && strncasecmp(uri, "news:",     5) == 0)
    || (i > 7 && strncasecmp(uri, "mailto:",   7) == 0) ) {
    return 1;
}
else {
    return 0;
}
```

The issue here is the line `c += 3;` in which the processing code is attempting to skip past a `://` in the URI. Notice, however, that within `is_absolute_uri`, not all of the URI schemes end in `://`. If a URI were requested that was simply `ldap:a`, the null-terminating byte would be skipped by the code. Going further into the processing of this URI, a null byte is written to it, making this vulnerability potentially exploitable. In this particular case, certain rewrite rules must be in place for this to work, but issues like this are still quite common throughout many open source code bases and should be considered when auditing.

Signed Comparison Vulnerabilities

Many coders attempt to perform length checks on user input, but this is often done incorrectly when signed-length specifiers are used. Many length specifiers such as `size_t` are unsigned and are not susceptible to the same issues as signed-length specifiers such as `off_t`. If two signed integers are compared, a length check may not take into account the possibility of an integer being less than zero, especially when compared to a constant value.

The standards for comparing integers of different types are not necessarily obvious from the behavior of compiled code, and we must thank a friend for pointing out the actual correct compiler behavior. According to the ISO C standard, if two integers of different types or sizes are compared, they are first converted to a signed `int` type and then compared. If any of the integers are larger in type than a signed `int` size, both are converted to the larger type and then compared. An unsigned type is larger than a signed type and will take precedence when an unsigned `int` and a signed `int` are compared. For example, the following comparison would be unsigned:

```
if((int)left < (unsigned int)right)
```

However, this comparison would be signed:

```
if((int)left < 256)
```

Certain operators, such as the `sizeof()` operator, are unsigned. The following comparison would be unsigned, even though the result of the `sizeof` operator is a constant value:

```
if((int) left < sizeof(buf))
```

The following comparison would, however, be signed, because both short integers are converted to a signed integer before being compared:

```
if((unsigned short)a < (short)b)
```

In most cases, especially where 32-bit integers are used, you must be able to directly specify an integer in order to bypass these size checks. For example, it won't be possible in a practical case to cause `strlen()` to return a value that can be cast to a negative integer, but if an integer is retrieved directly from a packet in some way, it will often be possible to make it negative.

A signed comparison vulnerability led to the Apache chunked-encoding vulnerability discovered in 2002 by Mark Litchfield of NGSSoftware. This piece of code was the culprit:

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;

len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

In this case, `bufsiz` is a signed integer specifying the amount of space left in the buffer, and `r->remaining` is a signed `off_t` type specifying the chunk size directly from the request. The variable `len_to_read` is meant to be the minimum of either `bufsiz` or `r->remaining`, but if the chunk size is a negative value it is possible to bypass this check. When the negative chunk size is passed to `ap_bread`, it is cast to a very large positive value, and an extremely large `memcpy` results. This bug was obviously and easily exploitable on Win32 via an SEH overwrite, and the Gobbles Security Group cleverly proved that this was exploitable on BSD due to a bug in their `memcpy` implementation.

These types of vulnerabilities are still present throughout software today and are worth auditing for any time you have signed integers used as length specifiers.

Integer-Related Vulnerabilities

Integer overflows seem to be a buzzword that security researchers use to describe a multitude of vulnerabilities, many of which aren't actually related to integer overflows. Integer overflows were first well defined in the speech "Professional Source Code Auditing," given at BlackHat USA 2002, although these overflows had been known and identified by security researchers for some time before that.

Integer overflows occur when an integer increases beyond its maximum value or decreases below its minimum value. The maximum or minimum value of an integer is defined by its type and size. A 16-bit signed integer has a maximum value of 32,767 (`0x7fff`) and a minimum value of –32,768 (`-0x8000`). A 32-bit unsigned integer has a maximum value of 4,294,967,295 (`0xffffffff`) and a minimum value of 0. If a 16-bit signed integer that has a value of 32,767 is incremented by one, its value becomes –32,768 as a result of an integer overflow.

Integer overflows are useful when you want to bypass size checks or cause buffers to be allocated at a size too small to contain the data copied into them. Integer overflows can generally be categorized into one of two categories: addition and subtraction overflows, or multiplication overflows.

Addition or subtraction overflows result when two values are added or subtracted and the operation causes the result to wrap over the maximum/minimum boundary for the integer type. For example, the following code would cause a potential integer overflow:

```
char *buf;
int allocation_size = attacker_defined_size + 16;

buf = malloc(allocation_size);
memcpy(buf,input,attacker_defined_size);
```

In this case, if `attacker_defined_size` has a value somewhere between –16 and –1, the addition will cause an integer overflow and the `malloc()` call will

allocate a buffer far too small to contain the data copied in the `memcpy()` call. Code like this is very common throughout open source applications. There are difficulties associated with exploiting these vulnerabilities, but nonetheless these bugs exist.

Subtraction overflows can be commonly found when a program expects user input to be of a minimum length. The following code would be susceptible to an integer overflow:

```
#define HEADER_SIZE 16

char data[1024],*dest;
int n;

n = read(sock,data,sizeof(data));
dest = malloc(n);
memcpy(dest,data+HEADER_SIZE,n - HEADER_SIZE);
```

In this example, an integer wrap occurs in the third argument to `memcpy` if the data read off the network is less than the minimum size expected (`HEADER_SIZE`).

Multiplication overflows occur when two values are multiplied by each other and the resulting value exceeds the maximum size for the integer type. This type of vulnerability was found in OpenSSH and in Sun's RPC library in 2002. The following code from OpenSSH (prior to 3.4) is a classic example of a multiplication overflow:

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

In this case, `nresp` is an integer directly out of an SSH packet. It is multiplied by the size of a character pointer, in this case 4, and that size is allocated as the destination buffer. If `nresp` contains a value greater than `0x3fffffff`, this multiplication will exceed the maximum value for an unsigned integer and overflow. It is possible to cause a very small memory allocation and copy a large number of character pointers into it. Interestingly enough, this particular vulnerability was exploitable on OpenBSD because of OpenBSD's more secure heap implementation, which doesn't store control structure in-line on the heap. For heap implementations with control structures in-line, wholesale corruption of any heap with pointers would lead to a crash on subsequent allocations such as that within `packet_get_string`.

Smaller-sized integers are more vulnerable to integer overflows; it is possible to cause an integer wrap for 16-bit integer types via common routines such as `strlen()`. This type of integer overflow was responsible for the overflow in `RtlDosPathNameToNtPathName_U` that led to the IIS WebDAV vulnerability described in Microsoft Security Bulletin MS03-007.

Integer-related vulnerabilities are very relevant and are still quite common. Although many programmers are aware of the dangers of string-related operations, they tend to be less aware of the dangers related to manipulating integers. Similar vulnerabilities will likely reappear in programs for many years to come.

Different-Sized Integer Conversions

Conversions between integers of different sizes can have interesting and unexpected results. These conversions can be dangerous if their consequences are not carefully considered, and if spotted within source code they should be examined closely. They can lead to truncation of values, cause the sign of an integer to change or cause values to be sign extended, and can sometimes lead to exploitable security conditions.

A conversion from a large integer type to a smaller integer type (32 to 16 bits, or 16 to 8 bits) can result in value truncation or sign switching. For example, if a signed 32-bit integer with a negative value of –65,535 were changed to a 16-bit integer, the resulting 16-bit integer would have a value of +1 due to the truncation of the highest 16 bits of the integer.

Conversions from smaller to larger integer types can result in sign extension, depending on the source and destination type. For example, converting a signed 16-bit integer with the value –1 to an unsigned 32-bit integer will result in a value of 4GB less one.

The chart in Table 18-1 may help you keep track of conversions of one integer size to another. It is accurate for recent GCC versions.

Table 18-1: Integer Conversion Table

SOURCE SIZE/ TYPE	SOURCE VALUE	DESTINATION SIZE/TYPE	DESTINATION VALUE
16-bit signed	–1 (0xffff)	32-bit unsigned	4294967295 (0xffffffff)
16-bit signed	–1 (0xffff)	32-bit signed	–1 (0xffffffff)
16-bit unsigned	65535 (0xffff)	32-bit unsigned	65535 (0xffff)
16-bit unsigned	65535 (0xffff)	32-bit signed	65535 (0xffff)
32-bit signed	–1 (0xffffffff)	16-bit unsigned	65535 (0xffff)
32-bit signed	–1 (0xffffffff)	16-bit signed	–1 (0xffff)

Table 18-1 *(continued)*

SOURCE SIZE/ TYPE	SOURCE VALUE	DESTINATION SIZE/TYPE	DESTINATION VALUE
32-bit unsigned	32768 (0x8000)	16-bit unsigned	32768 (0x8000)
32-bit unsigned	32768 (0x8000)	16-bit signed	–32768 (0x8000)
32-bit signed	–40960 (0xffff6000)	16-bit signed	24576 (0x6000)

Hopefully, this table helps clarify the inter-conversion of integers of different sizes. A recent vulnerability discovered in Sendmail within the `prescan` function is a good example of this type of vulnerability. A signed character (8 bits) was taken from an input buffer and converted to a signed 32-bit integer. This character was sign extended to a 32-bit value of –1, which also happens to be the definition for a special situation `NOCHAR`. This leads to a failure in the bounds-checking of that function and a remotely exploitable buffer overflow.

The inter-conversion of integers of different sizes is admittedly somewhat complicated and can be a source of errors if not well thought out in applications. Few real reasons exist for using different-sized integers in modern applications; if you spot them while auditing, it is worth the time to do an in-depth examination of their use.

Double Free Vulnerabilities

Although the mistake of freeing the same memory chunk twice may seem pretty benign, it can lead to memory corruption and arbitrary code execution. Certain heap implementations are immune or resistant to these types of flaws, and their exploitability is limited to certain platforms.

Most programmers do not make the mistake of freeing a local variable twice (although we have seen this). *Double free vulnerabilities* are found most commonly when heap buffers are stored in pointers with global scope. Many applications will, when a global pointer is freed, set the pointer to null afterward to prevent it being reused. If an application does not do something similar to this, it is a good idea to begin hunting for places in which a memory chunk can be freed twice. These types of vulnerabilities can also occur in C++ code when you are destroying an instance of a class from which some members have already been freed.

A recent vulnerability in zlib was discovered in which a global variable was freed twice when a certain error was triggered during uncompression. In addition, a recent vulnerability in the CVS server was also the result of a double free.

Out-of-Scope Memory Usage Vulnerabilities

Certain memory regions in an application have a scope and lifetime for which they are valid. Any use of these regions before they are valid or after they become invalid can be considered a security risk. A potential result is memory corruption, which can lead to arbitrary code execution.

Uninitialized Variable Usage

Though it is relatively uncommon to see the use of uninitialized variables, these vulnerabilities do surface once in a while and can lead to real exploitable conditions in applications. Static memory such as that in the `.data` or `.bss` section of an executable is initialized to null on program startup. You have no such guarantee for stack or heap variables, and they must be explicitly initialized before they are read from to ensure consistent program execution.

If a variable is uninitialized, its contents are, by definition, undefined. However, it is possible to predict exactly what data an uninitialized memory region will contain. For example, a local stack variable that it uninitialized will contain data from previous function calls. It may contain argument data, saved registers, or local variables from previous function calls, depending on its location on the stack. If an attacker is lucky enough to control the right portion of memory, they can often exploit these types of vulnerabilities.

Uninitialized variable vulnerabilities are rare because they can lead to immediate program crashes. They are most often found in code that is not commonly exercised, such as code blocks rarely triggered because of uncommon error conditions. Many compilers will attempt to detect the use of uninitialized variables. Microsoft Visual C++ has some logic for detecting this type of condition, and gcc makes a good attempt at locating these issues also, but neither does a perfect job; therefore, the onus is on the developer not to make these sorts of errors.

The following hypothetical example shows an overly simplified case of the use of an uninitialized variable:

```
int vuln_fn(char *data,int some_int) {
        char *test;

        if(data) {
                test = malloc(strlen(data) + 1);
                strcpy(test,data);
                some_function(test);
        }

        if(some_int < 0) {
                free(test);
                return -1;
        }
```

```
        free(test);
        return 0;
}
```

In this case, if the argument `data` is null, the pointer `test` is not initialized. That pointer would then be in an uninitialized state when it is freed later in the function. Note that neither gcc nor Visual C++ would warn the programmer of the error at compile time.

Although this type of vulnerability lends itself to automatic detection, uninitialized variable usage bugs are still found in applications today (for example, the bug discovered by Stefan Esser in PHP in 2002). Although uninitialized variable vulnerabilities are reasonably uncommon, they are also quite subtle and can go undetected for years.

Use After Free Vulnerabilities

Heap buffers are valid for a lifetime, from the time they are allocated to the time they are deallocated via `free` or a `realloc` of size zero. Any attempts to write to a heap buffer after it has been deallocated can lead to memory corruption and eventually arbitrary code execution.

Use after free vulnerabilities are most likely to occur when several pointers to a heap buffer are stored in different memory locations and one of them is freed, or where pointers to different offsets into a heap buffer are used and the original buffer is freed. This type of vulnerability can cause unexplained heap corruption and is usually rooted out in the development process. Use after free vulnerabilities that sneak into release versions of software are most likely in areas of code that are rarely exercised or that deal with uncommon error conditions. The Apache 2 `psprintf` vulnerability disclosed in May of 2003 was an example of a use after free vulnerability, in which the active memory node was accidentally freed and then subsequently handed out by Apache's `malloc`-like allocation routine.

Multithreaded Issues and Re-Entrant Safe Code

The majority of open source applications are not multithreaded; however, applications that are do not necessarily take the precautions to ensure that they are thread-safe. Any multithreaded code in which the same global variables are accessed by different threads without proper locking can lead to potential security issues. In general, these bugs are not discovered unless an application is put under heavy load, and they may go undetected or be dismissed as intermittent software bugs that are never verified.

As outlined by Michal Zalewski in *Problems with Msktemp()* (August, 2002), the delivery of signals on Unix can result in execution being halted while

global variables are in an unexpected state. If library functions that are not re-entrant safe are used in signal handlers, this can lead to memory corruption.

Although there are thread- and re-entrant safe versions of many functions, they are not always used in multithreaded or re-entrant prone code. Auditing for these vulnerabilities requires keeping the possibility of multiple threads in mind. It is very helpful to understand what underlying library functions are doing, because these can be the source of problems. If you keep these concepts in mind, thread-related issues will not be exceptionally difficult to locate.

Beyond Recognition: A Real Vulnerability versus a Bug

Many times a software bug can be identified without it being a real security vulnerability. Security researchers must understand the scope and impact of a vulnerability before taking further steps. Though it is often not possible to confirm a bug's full impact until it has been successfully exploited, much of the more tedious security work can be done via simple source code analysis.

It is useful to trace backwards from the point of vulnerability to determine whether the necessary requirements can be met to trigger the vulnerability. Ensure that the vulnerability is indeed in active code and that an attacker can control all necessary variables, and verify that no obvious checks are in place farther back in code flow that might prevent the bug from being triggered. You must often check configuration files distributed with software to determine whether optional features are commonly turned on or off. These simple checks can save much exploit development time and help you avoid the frustration of attempting to develop exploit code for a non-issue.

Conclusion

Vulnerability research can sometimes be a frustrating task, but at other times it is a lot of fun. As an auditor, you will be searching for something that may not actually exist; you must have great determination in order to find anything worthwhile. Luck can help, of course, but consistent vulnerability research usually means hours of painstaking auditing and documentation. Time has proven repeatedly that every major software package has exploitable security vulnerabilities. Enjoy auditing.