iDEFENSE
A VeriSign Company

Attacking the Code: Source Code Auditing

David D. Rude II, Security Intelligence Engineer
drude@idefense.com

Sept. 20, 2006
An iDefense Security Report
The VeriSign® iDefense® Intelligence Team

VeriSign®

iDEFENSE
A VeriSign Company

VeriSign

Created and distributed by the iDefense Intelligence Operations Team

INSIDE THIS REPORT

iDEFENSE - A VeriSign Company

21345 Ridgetop Circle, Sterling, VA 20166
Toll Free: 877.516.2974 Main: 703.390.1230 Fax: 703.390.9456
www.idefense.com | di@idefense.com

1 Introduction

In the history of computing, security researchers have used many different techniques to find vulnerabilities in software, but source code auditing remains one of the most reliable and popular. Source code auditing is a means by which a security researcher (or auditor) examines and analyzes program source code (which makes up the "backbone" of any software application) for potential vulnerabilities or flaws. Such audits are vital to the software development process, and allow software vendors to address issues that might put their customers at risk to hackers, malicious code and other computer-based threats. This report examines the theory and practice of source code auditing as a means of discovering potentially damaging software vulnerabilities.

A good source code auditor must be a source code expert, but even a deep knowledge of programming does not guarantee fast and easy answers. Every piece of code becomes a learning experience for the auditor; and each new code teaches the auditor something they did not know about the language. Ultimately, high proficiency in source code auditing requires that the auditor know more about the programming language than the author of the software.

A skilled source code auditor must also become familiar with the many different types of security vulnerabilities and what causes them. This reinforces pattern recognition skills, which are essential for even the most basic audits. With so many different types of vulnerabilities in software today, pattern recognition becomes an asset that an auditor cannot live without. Auditing also requires creative thinking and strong logic skills, all of which come with time and practice. But strong pattern recognition skills alone will not help even the best auditor find flaws in the most efficient manner, and many professional source code auditors rely upon methodologies that allow them to gain the most benefit for time spent.

2 Types of Vulnerabilities

There are many types of vulnerabilities. Each vulnerability occurs due to programmer error. Commonly patterns can be seen within code which is vulnerable. This is because vulnerable code commonly is caused by similar circumstances. Vulnerabilities are commonly caused by misconceptions or a lack of understanding when using library functions or functions written by other authors as well as any arithmetic which may cause unexpected behavior. These misconceptions can lead to miscalculations of buffer or integer capacity, flawed logic statements, and other misunderstandings.

2.1 Buffer Overflows

Buffer overflow vulnerabilities can be categorized into two groups, heap-based and stack-based. Heap-based overflows occur when dynamically allocated memory is overflowed by filling that memory area with too much data, usually due to some sort of miscalculation by the programmer. Stack-based overflows occur when a static-sized local buffer is overflowed by attempting to store more data within the buffer than its fixed size allows.

2.1.1 Stack-Based

The following code sample demonstrates a stack-based overflow in cyrus-imapd, a commonly used IMAP server:

```
static int popd_canon_user(sasl_conn_t *conn, void *context,
               const char *user, unsigned ulen,
               unsigned flags, const char *user_realm,
               char *out, unsigned out_max, unsigned *out_ulen)
{
    char userbuf[MAX_MAILBOX_NAME+1], *p;
    size_t n;
    int r;

    if (!ulen) ulen = strlen(user);

    if (config_getswitch(IMAPOPT_POPSUBFOLDERS)) {
     //if popsubfolders
     //configed
    /* make a working copy of the auth[z]id */
    memcpy(userbuf, user, ulen); //the user variable is unchecked
    userbuf[ulen] = '';
    user = userbuf;
      …
}
```

The first thing an experienced auditor will notice here is that userbuf is on the stack and is of a static size (MAX_MAILBOX_NAME+1). The ulen variable is coming from the string length of the user argument. The "if" statement here is important, as it is checking to see if a configuration setting (popsubfolders) is set. If the popsubfolders configuration option is not set, this vulnerable code is never touched. If it is set the buffer overflow vulnerability is triggered when a large username is supplied. Next, the memcpy function is called using ulen to decide the number of bytes to copy to userbuf. This is wrong because the length should be the size of userbuf not the length of the source buffer. As long as the user string is longer than the userbuf variable which is MAX_MAILBOX_NAME+1 an overflow occurs on the stack.

2.1.2 Heap-Based

The following code sample illustrates a heap-based buffer overflow from the MPlayer media player for Unix/Linux operating systems:

```
    ibmpdemux/http.c:http_build_request (line 178):
       if( http_hdr->uri==NULL ) http_set_uri( http_hdr, "/");
       else {
          uri = (char*)malloc(strlen(http_hdr->uri)*2);
          if( uri==NULL ) {
             mp_msg(MSGT_NETWORK,MSGL_ERR,"Memory allocation failed\n");
             return NULL;
         }
         url_escape_string( uri, http_hdr->uri );
```

A source code auditor examining this code would first need to understand its purpose. This code allocates space for a string and then tries to escape or alter certain characters within that string. Looking closely at the allocation, the auditor can see that the malloc call has multiplication within it. Why is it multiplying by two? Because the escape code is expected to return an escaped string, which will be double the size of the original. What makes this code vulnerable is that the author never considered URL-encoded strings. URL-encoded characters represent one character by using three characters. For example, "A" is represented by the symbol combination "%41" when encoded. Therefore, the problem with this code lies in that it does not allocate enough memory for the url_escape_string function. When url_escape_string writes the escaped string into the uri variable, it will exceed the allocated space, causing a buffer overflow.

Buffer overflows have many subclasses. There are off-by-one buffer overflows, adjacent memory overflows, miss matched size calculations and many more. Off-by-one overflows occur when a buffer is overflowed by a single byte of data. Normally, this occurs during the process of "null terminating" a string or specifying the wrong size to a libc function. Adjacent memory overflows are very similar in that they rely upon null termination bugs to concatenate, or combine the two strings into one. Adjacent memory overflows get their name because two buffers must be next to each other in memory for this vulnerability to occur. When the null termination does not occur properly, the memory is read as continuing into the adjacent memory when strlen() functions, and others that rely upon the null termination, occur.

2.1.3 Off-By-One

The following code excerpt comes from thttpd, a fairly popular Web server on Unix/Linux operating systems. This code depicts an off-by-one vulnerability when indexing memory:

```
static int
b64_decode( const char* str, unsigned char* space, int size )
    {
    const char* cp;
    int space_idx, phase;
    int d, prev_d = 0;
    unsigned char c;

    space_idx = 0;
    phase = 0;
    for ( cp = str; *cp != '\0'; ++cp )
        {
        d = b64_decode_table[(int) *cp];
        if ( d != -1 )
            {
            switch ( phase )
                {
                case 0:
                ++phase;
                break;
                case 1:
                c = ( ( prev_d << 2 ) | ( ( d & 0x30 ) >> 4 ) );
                if ( space_idx < size )
                    space[space_idx++] = c;
                ++phase;
                break;
                case 2:
                c = ( (( prev_d & 0xf )<< 4 ) | ( ( d & 0x3c ) >> 2 ));
                if ( space_idx < size )
                    space[space_idx++] = c;
                ++phase;
                break;
                case 3:
                c = ( ( ( prev_d & 0x03 ) << 6 ) | d );
                if ( space_idx < size )
                    space[space_idx++] = c;
                phase = 0;
                break;
                }
            prev_d = d;
            }
        }
    return space_idx;
    }
```

This code looks confusing at first, which makes spotting the off-by-one overflow vulnerability difficult. This code is looking within a table to see if the characters can be decoded. If they can, then the code enters the switch-case statement. The "if" statements within the switch-case compares the space_idx variable to the size argument. If the space_idx variable is less than the size variable, then space_idx is increased in value by one (space_idx++ is the same as space_idx+1).

Consider the following scenario: space_idx is 255 and size is 256. Here, space_idx is less than size and will then be incremented to 256 when any of the 'space[space_idx++] = c;' statements are executed. This is where the off-by-one occurs. Arrays are indexed starting at zero. Any array declared like this (i.e., "char array[256];") will have the last valid index at 255 because indexing begins at zero. Thus sending this function the total size of the buffer where the data is being stored leads to an off-by-one buffer overflow.

2.1.4 Adjacent memory overflow

The next example is from syslogd, a commonly used logging daemon which captures and logs events, depicts an adjacent memory overflow:

```
/*
 * Validate that the remote peer has permission to log to us.
 */
int
validate(sin, hname)
struct sockaddr_in *sin;
   const char *hname;
{
   int i;
   size_t l1, l2;
   char *cp, name[MAXHOSTNAMELEN];
   struct allowedpeer *ap;

   if (NumAllowed == 0)
      /* traditional behaviour, allow everything */
      return 1;
```

```
                strncpy(name, hname, sizeof name);
                if (strchr(name, '.') == NULL) {
                   strncat(name, ".", sizeof name - strlen(name) - 1);
                   strncat(name, LocalDomain, sizeof name -strlen(name)-1);
                }

                ...
             }
```

In this example, suppose that hname is at least MAXHOSTNAMELEN bytes long and does not contain a period. This means that strncat gets the total size of name subtracted by the number of bytes stored in name subtracted by one ('sizeof(name) – strlen(name) – 1'), which is a negative value if the number of bytes in name are equal to the size of name. However, the size parameter is a signed integer, which means it will be willing to copy about 4 GB of data which is the maximum value an unsigned integer can represent. The 4 GB value is determined because when a signed value such as -1 is converted to an unsigned value -1 is represented as 4 GB. Thus, all of LocalDomain will be appended to name (which is already full), and a buffer overflow will occur.

It is important to understand what causes buffer overflow vulnerabilities. They are among the most common vulnerability class reported publicly. It is also important to understand how buffer overflows can be exploited because many of the other classes of vulnerabilities lead to buffer overflow conditions. There have been many papers written and published on the internet which should be referenced when attempting to learn more about buffer overflows. One such paper is called Smashing The Stack for Fun and Profit by Aleph1.

2.2 Integer Overflows

Integer overflows occur when a program makes a calculation and the results cannot be properly stored within an integer variable. Integer overflows can also occur if user-supplied integer values are explicitly trusted. User supplied values which are trusted and never tested for validity and then later on used to calculate size values or even offsets to data could cause integer overflows to occur. These types of vulnerabilities usually lead to buffer overflows due to the integer value commonly being used as an index or size value.

The following integer overflow example comes from the XDR RPC library which allows applications to communication over the RPC protocol:

```
bool_t xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
       XDR *xdrs;
       caddr_t *addrp; /* array pointer */
       u_int *sizep;          /* number of elements */
       u_int maxsize;         /* max numberof elements */
       u_int elsize;        /* size in bytes of each element */
        xdrproc_t elproc;    /* xdr routine to handle each element*/
  {
    u_int i;
    caddr_t target = *addrp;
    u_int c;           /* the actual element count */
    bool_t stat = TRUE;
    u_int nodesize;

    c = *sizep;
    if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
      {
        return FALSE;
      }
    nodesize = c * elsize;    /* [1] */

    ...

    *addrp = target = mem_alloc (nodesize);   /* [2] */

    ...

    for (i = 0; (i < c) && stat; i++)
      {
        stat = (*elproc) (xdrs, target, LASTUNSIGNED);/* [3] */
        target += elsize;
      }
```

In this example, nodesize is computed using the result of multiplying c and elsize. The c variable, which is assigned the value of sizep, and the elsize variable are both attacker-controlled values. An attacker specifying large values for both can create an integer overflow. When this happens, the mem_alloc function will incorrectly allocate a smaller value than intended. Then, during the "for" loop, elsize is used by itself as the size of each element which causes the buffer overflow. In the above example, elsize will be larger than nodesize, which is overflowed. Most integer overflows are similar to this example. The important thing to remember when analyzing integer overflows is the difference between signed and unsigned integers. An integer will always wrap around, or overflow, to the smallest possible value. In the case of a signed integer, the smallest value is -2,147,483,647 (on 32-bit computers). In unsigned integers, the smallest value is zero. Another important point to observe is that integer overflows can cause buffer overflows and logic bugs.

2.3 *Format Strings*

Format strings are commonly used when a function receives a dynamic amount of arguments. Format strings such as "%s", "%d", "%x", allow these functions to determine the number of arguments to use. Functions such as printf, fprintf, and scanf use format strings. There are two causes of format string vulnerabilities. The first and most widely known type is when too many arguments and not enough formats are specified by the programmer. The second is when the programmer uses formats that can be larger than the destination, leading to a buffer overflow.

The following format string vulnerability is in Xine, a popular media player for Unix/Linux operating systems:

```
static void print_formatted(char *title, const char *const *plugins) {
    const char  *plugin;
    char         buffer[81];
    int          len;
    char        *blanks = "     ";

    printf(title); //missing format

    sprintf(buffer, "%s", blanks);
    plugin = *plugins++;

    while(plugin) {

      len = strlen(buffer);

      if((len + (strlen(plugin) + 3)) < 80) {
      sprintf(buffer, "%s%s%s", buffer, (strlen(buffer) ==
      strlen(blanks)) ? "" : ", ", plugin);
      }
      else {
        printf(buffer); //missing format
        printf(",\n");
        snprintf(buffer, sizeof(buffer), "%s%s", blanks, plugin);
      }
```

In this example, the printf function is not given any format strings but it is given arguments. Attackers are able to specify format strings, and read and write to the stack, if the argument is controlled by an attacker (as is the case in the above example); such a circumstance could be disastrous for the vulnerable application. This type of situation could allow an attacker to write arbitrary data to arbitrary locations within the memory of the program. If this was to happen the program would no longer be able to trust its own memory.

The other type of format string issue occurs when a programmer either does not specify a bounds checking format specifier or specifies one which is not of the proper size. Another issue similar to this is when a programmer attempts to store formats which are completely user controlled into a buffer which cannot handle all of the data if the formats contained the most data they could represent.

Format string vulnerabilities are easy to spot and easy to prevent. Anytime a function contains an argument which is variable (in C this is done using '…' as an argument to a function) the function should contain format specifiers which match the number of arguments about to be passed. Checking the maximum size of the format specifiers and adding them together and then compare that value with the size of the buffer which is being used to store the output should be done each time formats specifiers are seen.

Format string vulnerabilities are especially dangerous as they commonly create a write-anything anywhere condition which means an attacker can write arbitrary data to arbitrary locations in memory. This easily gives an attacker enough leverage to change the flow of execution within an application.

2.4 *Information Leaks*

Information leaks are commonly caused by using uninitialized data or by a programmer improperly setting a pointer value. This commonly leads to the ability to read memory that normally would not be accessible.

A memory leak vulnerability was recently discovered in MySQL a common SQL database server. The following code sample shows an information leak flaw:

```
sql/sql_parse.cc: line: ~1589

case COM_TABLE_DUMP:
{
char *db, *tbl_name;
uint db_len= *(uchar*) packet;
uint tbl_len= *(uchar*) (packet + db_len + 1);

statistic_increment(thd->status_var.com_other, &LOCK_status);
thd->enable_slow_log= opt_log_slow_admin_statements;
db= thd->alloc(db_len + tbl_len + 2);
if (!db)
{
my_message(ER_OUT_OF_RESOURCES, ER(ER_OUT_OF_RESOURCES), MYF(0));
break;
}
tbl_name= strmake(db, packet + 1, db_len)+1;
strmake(tbl_name, packet + db_len + 2, tbl_len);

mysql_table_dump(thd, db, tbl_name, -1);
break;

}
```

This code does not check the packet length value; the length can be declared arbitrarily by an attacker. This leads to the attacker having the ability to read memory from any offset of the packet's memory location. Flaws like this could lead to more efficient exploits or sensitive information such as user names and passwords being disclosed in clear text.

Information leaks are useful when dealing with other vulnerabilities which are not easily exploited. This is because the information leaked can sometimes give an attacker better knowledge of where certain things are in memory. They can sometimes be useful alone depending on the information which is being leaked.

2.5 Logic Flaws

Logic bugs are usually very difficult for an auditor to discover. Finding this type of flaw requires extensive knowledge about the inner workings of the program and how all of its components and code fit together. This vulnerability commonly occurs due to improper or neglected bounds checking on return values from certain functions.

The following example is from the recent RealVNC issue, in which a logic bug leads to remote compromise without authentication:

```
// Inform the server of our decision
     if (secType != secTypeInvalid) {
       os->writeU8(secType);
       os->flush();
       vlog.debug("Choosing security type %s(%d)",
secTypeName(secType),secType);     }
```

The code here is trying to determine the security type setting that it should use for a remote connection. However, the code never checks to see if secType is "NULL," which is a legitimate value in this case. A secType of NULL means that no authentication is needed. This allows remote users to connect to the server without authentication.

2.6 Neglecting Return Values

Return values are commonly used to determine if a function call has failed. At times, programmers become accustomed to seeing certain functions succeed every time. While a function may succeed the majority of the time, an attacker may be able to influence the function call in a way that will force it to fail. This can have adverse affects on the program and in some cases can lead to an attacker gaining enough control to elevate privileges or do damage to data.

The security community has recently discovered several vulnerabilities regarding the checking of the setuid() function's return value on the Linux platform. The setuid() function is in charge of dropping privileges in privileged applications. This function rarely fails, and its success is often taken for granted. However, an attacker can cause it to fail by reaching process limits. When setuid() changes the user id to that of the user running the application and drops the suid privileges, it then increments the process count for that user. This means that if a user creates enough processes to reach the process limit set in the Linux kernel, the increment of the process count will bring that user to the process limit and setuid will fail, returning a value of -1.

The following example comes from the shadow package, which provides the passwd utility.

```
if (argc > 1 && argv[1][0] == '-' && strchr ("gfs", argv[1][1])) {
            char buf[200];

      setuid(getuid ());
```

The code is checking for command line arguments 'g,' 'f' or 's,' and then trying to drop privileges to that of the user running the application. However, the setuid() return value is never checked to see if it succeeded properly.

Neglecting to check return values can easily cause unexpected application behavior if that variable is capable of impacting application functionality. Any function which returns a value that is not void should be checked, even if that value is not expected to be able to effect the application.

2.7 Race Conditions

Race conditions occur within non-atomic operations. This means that any operation with a time delay between the next operations could contain a race condition vulnerability. Race conditions are commonly seen in two forms within programs: gathering information about files from the file system before opening them and signal handlers. Also, any kind of state machine that does not properly check the state prior to any operation may contain these types of flaws.

A race condition vulnerability within wu-ftpd, a commonly used FTP server, was discovered which allowed remote attackers to cause exploitable conditions. This issue specifically lies within code which handles out of bounds signals which are sent remotely.

```
static VOIDRET myoob FUNCTION((input), int input)
{
  ...
  if (getline(cp, 7, stdin) == NULL) {
    reply(221, "You could at least say goodbye.");
    dologout(0);
  }
  ...
}


void dologout(int status)
{
    ...
    if (logged_in) {
        delay_signaling();
        (void) seteuid((uid_t) 0);
        wu_logwtmp(ttyline, "", "");
    }
    if (logging)
        syslog(LOG_INFO, "FTP session closed");
    ...
}
```

The myoob function is the function which is called when an out of bounds signal is sent to the FTP server. The dologout function contains a call to syslog. The issue here is that syslog is constructed of non-atomic functions. This means that at any time a signal can occur and interrupt a potentially vital operation within the code. If an attacker can create a condition which causes memory to be in an inconsistent state, issues similar to the above example may become exploitable.

3 Auditor Preparation

As time goes by, an auditor will eventually be able to recognize patterns in the different types of vulnerabilities and spot them quickly within other, more benign code. As an inexperienced auditor trying to learn to audit code, it is often best to look at vulnerabilities that other auditors have found. Exploitation is a good way to gain an understanding of the way memory works, and is also helpful when looking at any future code. Studying patches on a regular basis is another good way to learn about the root causes of certain vulnerabilities. By looking at patches, an auditor might also be able to find flaws caused by the changes in functionality created by that patch. At times, software maintainers do not always understand the flaw and will patch the vulnerability incorrectly, making it more difficult to exploit, but allowing the vulnerability to remain in the code. Such an understanding is necessary before beginning any audit.

To prepare for an audit, the auditor needs to be critical of every minute detail. Reviewing the C Specification document for the code, looking specifically for the words "undefined behavior," is usually a good idea. This will give the auditor any information that he or she might not have. Figuring out how the target software is commonly used, and the features of the software, such as networking protocols is good preparatory information for any audit. This allows the auditor to collect some information about how the software works without looking at the code. It also gives the auditor an idea of which parts of the code might need the most scrutiny. The auditor should also research any information about the software that they do not fully understand. Encoding schemes and compression algorithms (if the software uses them) can yield useful information to any auditor. This research ensures that the auditor fully understands the concepts of any features the software implements, and gives the auditor a better understanding of any code they might audit.

4 Methodologies

There are three commonly used methodologies for auditing source code for vulnerabilities, each with its pros and cons. The most appropriate method for conducting a source code audit depends entirely upon some common factors including, the number of lines of source code, the functionality of the application and the amount of time being devoted to auditing the particular code, all these factors are essential when trying to determining the proper methodology to use when auditing.

It is important to remember that even small flaws can add up to large-impact issues. When conducting a source code audit, if the auditor comes across a potential security vulnerability, that auditor should make a note of it and continue with the audit.

4.1 Top Down

The top-down methodology provides great coverage and understanding of the source code being audited. This method consists of starting at the entry point in the program (the main function) and following all the possible code branches. Each branch should be followed and audited for possible security flaws and other potential bugs.

The con to this methodology is that it is extremely time consuming to properly follow every branch of large source trees. While the coverage is near perfect, it is often a waste of time to follow every branch of code if the user has no influence or input in the majority of branches.

4.2 Bottom Up

The bottom-up approach is the reverse of the top-down methodology. This method consists of finding places within the code that the auditor knows are influenced by the user. The auditor also looks for common insecure functions in the bottom-up approach. The auditor then looks for flaws within the current function and any functions which call the current function. The benefit of this method is that it saves time by only focusing on functions that are directly influenced by user input in any way.

This methodology, while a time saver, does often prevent the auditor from gaining a deep understanding of the application. Another draw back is the potential to miss subtle flaws which may have to do with logic or trust relationships due to a lack of understanding of the code. It is also difficult at times to determine if users can directly or indirectly influence anything within the functions.

4.3 Hybrid

The hybrid methodology attempts to utilize the pros of each of the first methods while limiting the cons of each. This can best be described as finding functions that take input, such as network input, and tracing forward through the code to find vulnerabilities in the way that input is handled.

This methodology saves time by skipping over parts of code that are inaccessible to users, and allows the auditor to gain a deeper understanding of the application while limiting the audit to parts of code which are only specifically influenced by its users.

Each methodology has its place and time to be used. It is up to the auditor to choose which one best fits the project that is about to be audited. The methodologies vary mainly in the amount of time which should be spent on auditing the code.

5 Taking Notes

The most important part of any source code audit is keeping detailed notes during the audit. This allows an auditor to maintain detailed records of each function's purpose and what is done within that function to arguments and local variables. Taking notes is certainly a key aspect in finding the more subtle bugs such as logic flaws or flaws within trust relationships. This forces the auditor to understand what the function is doing. Without an understanding of the code an auditor cannot properly write a function audit log.

Source code commenting is a method of taking notes within the code itself. This leaves a record for future auditors and serves as a reference point for future audits. Tags within the comments make them easier to find for reference. The best idea would be to create a note containing the auditor's initials and comments.

The following pieces of information should always appear in an auditor's notes for a function:

- Function name
- File location of the function and line number
- Arguments passed to the function
- Assignments or arithmetic performed on any of the arguments
- A summary of the function's purpose.

All these pieces of information are important to an auditor because they allow future auditors to look back and figure out what a specific called function is doing at any given time. Without these specific notes, future auditors would have to go back and look at the code multiple times to determine the same information.

Once an audit log is created the documents can be used in the future when looking at other versions of the same software. Even though the version may have changed, the documentation in the audit logs remains relevant and fairly accurate. An auditor should re-examine the audit logs during each new version of the software. By updating the logs, an auditor may be able to spot flaws in the changes made to a given function.

When looking at a function while taking notes, a good auditor will look quickly over the notes, to try and get a basic feel for the programmer's intention for the code. Auditors should write down any initial ideas about what the function is trying to achieve, and then take a deeper look. Auditors should start by looking at the types of variables the function is taking in, and then follow each variable to get an idea of what each is used for. Usually, the name of the variable will tell the auditor the programmer's intent; however, sometimes variable names can be misleading. Take note of any kind of variable arithmetic that takes place along with any kind of string manipulation. Good auditors will pay special attention to things such as null termination, pointer arithmetic and integer arithmetic. Once the auditor has an overview of the function, he can examine the branches of code leading to the function currently being reviewed, along with any branches of code that split from the current code, to see what type of impact the branches have on the starting branch.

6 Verifying Flaws

Verifying discovered flaws is often what separates a good source code auditor from a great one. At times vulnerabilities are reported which are not really vulnerabilities at all. This is because the code that would trigger the vulnerability is never reached or sanity checks are done prior to reaching the function in the code which prevents the vulnerability.

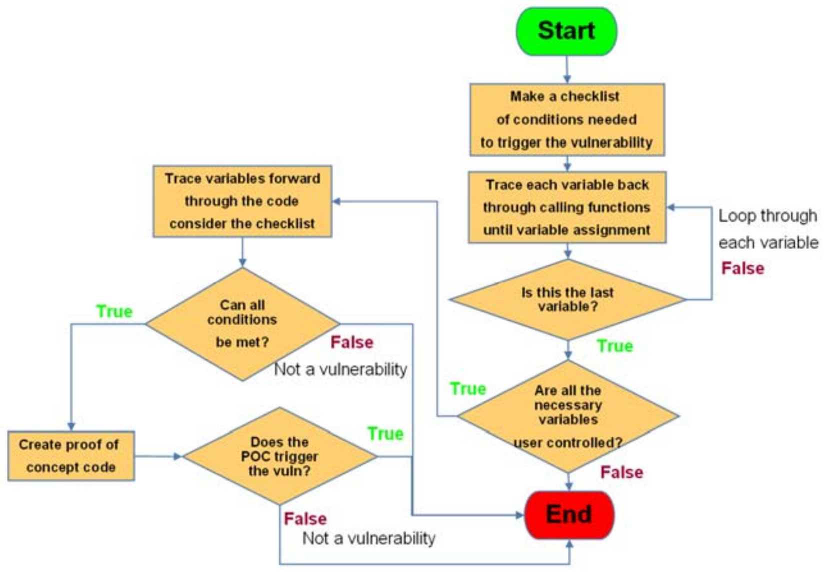
The first step in verifying code as vulnerable is to identify locations of the vulnerable code, or functions calling the code, within the larger code base. The auditor should closely examine the code of the functions calling the vulnerable code, look for any kind of limitations mitigating the vulnerability and make note of these limitations. This is necessary because there might be another, more subtle, bug that bypasses these limitations. If there are no limitations, the auditor should look for functions that call the current function. Auditors continue this recursive search for limitations until reaching the source of the input. The idea here is to walk back through the call stack. Another thing to take note of is often times a function is called in different locations. This means that the recursive limitation searching needs to be done for each location where the vulnerable function is called.

When looking at functions calling the vulnerable code it is often a good idea to create a check list of all the things which need to happen in order for the vulnerable code to be reached. This way checks can easily be done to see if the vulnerable code is reachable and reachable in a manner which will cause the vulnerability to trigger.

Once the recursive limitation checks have been completed and the check list has been created it is now time to follow the call stack forwards using the check list to make sure that each item on the list is done in order to reach the vulnerable code. Once the list is completely checked off the vulnerability has been statically verified. Lastly, a runtime check is performed to trigger the vulnerable code.

A runtime check is simply a test which triggers the vulnerable code. It is similar to creating an exploit in that the vulnerability is accessed, though actual exploitation is not performed. The runtime test simulates a real-world exploitation, and if successful verifies that the code in question is vulnerable to exploitation.

The following is a flow chart depicting the above stated procedure for verifying vulnerabilities:


Start
Make a checklist of conditions needed to trigger the vulnerability
Trace each variable back through calling functions until variable assignment
Loop through each variable
False
Is this the last variable?
True
Are all the necessary variables user controlled?
True
False
End
Trace variables forward through the code consider the checklist
Can all conditions be met?
True
False
Not a vulnerability
Create proof of concept code
Does the POC trigger the vuln?
True
False
Not a vulnerability

7 Auditing Tools

Source code auditors have many tools available to them, each with a different set of useful features to make auditing source code easier in the long term. The most important tool in an auditor's arsenal is a text editor for making changes to lines of code. When choosing a text editor, significant consideration should be given to the features of that program. Some of the most useful features include syntax highlighting, regular expression-searching capabilities, line numbering and the ability to leave multiple files open.

Jed and VI/VIM are two editors that have these capabilities and are commonly used by auditors. Jed is based on EMACS and contains useful functionality such as window splitting (allowing an auditor to view multiple files at once), a tag engine (which will search for programming language symbols within multiple files and many more. VI/VIM is a powerful editor with much of the same functionality as Jed.

Cscope and Ctags are other tools used by source code auditors, which differ primarily in interface design and Cscope's ability to perform advanced searches. Cscope's advanced search features allow the user to search by symbol definition, function usage or functions called from other functions. Cscope also allows auditors to use their preferred text editor application when viewing the source code. Ctags is similar to Cscope in that it creates a symbol file allowing for cross referencing and searching, though Ctags relies upon the text editor to provide that functionality. Using Ctags, an auditor can quickly jump between a function or symbol and the portion of the code where it is defined. Ctags and Cscope, if configured properly and combined with an auditor's preferred editor are extremely powerful tools in source code auditing.

The future may bring other tools to greatly reduce the amount of time spent performing a source code audit. Tools such as debuggers, which can filter sections of code based on user interaction, would be useful because the majority of existing methodologies rely upon starting where input is coming in and following the code paths from there.

8 Conclusion

Over time, source code auditing has remained the single most popular method of finding vulnerabilities in software code. The process of finding these flaws, while not a simple task by any means, is useful to not only the auditor but to the entire computing community. Researchers who find bugs and report them to the proper vendors ensure that future releases of that code will have fewer potential vulnerabilities. Such research and disclosure also gives software developers a chance to learn from previous mistakes and try to avoid similar issues in the future.

The importance of source code auditing cannot be denied; one needs only look at the daily bug reports present on the major security mailing lists. Every day, all over the world, private and corporate researchers use the methodologies discussed in this report to find and investigate potential vulnerabilities in source code. Until software developers have the time and resources to adequately audit their own source code, the security community and users at large will have to count on external auditors "attacking the code."