Practical Code Auditing

by
Lurene A. Grenier
lurene@daemonkitty.net

December 13, 2002

Abstract

A quick scan provides multiple papers detailing the exploitation of various software vulnerabilities, although there are very few comprehensive writings which cover the detection and repair of security related software flaws. This paper will attempt to cover the common categories of software vulnerability, and how to recognize them in either source code, or the actions of a "black box" binary. The focus will be on vulnerabilities which result in the execution of arbitrary code for the most part, touching on Denial of Service attacks only as they relate to the inproper handling of input data.

1 Acknowledgements

Much of the work and insight of this paper, as well as the format and presentation have benefitted from the discussion with and suggestions of several friends. I'd like to thank Dean McNamee for his understanding and explainations of memory chunks and free, Joakim Berg for his uncanny ability to dig up 0day and tutorials, Jose Nazario, Niels Provos, Todd Fries, and Sam Smith, and Julie Mallet for their eyes and suggestions.

2 Introduction

The first and most important aspect of code auting deals with understanding the forms of known vulnerabilities, and (obviously) being able to recognize them "in the wild". This paper will cover buffer overflow, heap overflow,integer overflow, format string vulnerabilities, bad free calls, and race conditions. As you can see simply from the names of these categories, the majority of vulnerabilities rely on the writing of memory past that which has been allocated for the purpose. Specifically, buffer overflows write to the stack, heap overflows write to malloc()'d memory (the heap), and integer overflows are a subset which result

in either buffer or heap overflows. Format string vulnerabilities of course work on a similiar theory, but get the result in a far different way.

3 Preliminary Source Code Auditing and Buffer Overflows

Our plan of attack will be to start with a wide net, then narrow our passes as we continue, beginning with the most commonly made errors and the most commonly exploited type of hole. This means we will sweep first for the non-error checking string functions which can result in an easily exploitable buffer overflow. This can be done with a tool such as grep, or perl, but we'll still need to go over our results. The functions we are most interested in are those such as strcpy, strcat, sprintf, or gets. We should also be concearned with functions such as strncat being used to copy a null byte one place past the end of the array, or strncpy'd strings being recopied as if they had a '\0' at the end. Other common buffer overflow points are home made "safe" string functions (ie. my_strncpy). Once we've grepped for these, we should output them to a file and immediately group them into areas of likelyhood. Strcat/ strcpy usage should be checked first, to be sure that if they are used at all, they are used with static strings that have had space allocated specifically for them. One should also double check the math to be sure the space is allocated correctly to allow for '\0's and be sure that they are there. Next, we will want to go over all instances of strncat and strncpy, making sure '\0's have been cat'd to the end, and that these '\0's are contained inside the allocated memory as opposed to one past, ie

buf[sizeof(buf)-1] = '\0'

as opposed to

buf[sizeof(buf)] = '\0'

Any usage of sprintf should be replaced with snprintf or similiar and the size limit should be checked. Any usage of gets should be an immediate clue that our program is vulnerable, not only at that point, but probably many others. It should be quickly replaced with a sane buffered and checked input loop, or exploited, depending on your purpose.

4 Heap Overflows and Free Bugs

While these same precations should be taken to in to acount to avoid heap overflows, there are also other interesting ways heap overflows can be accomplished, one such way being bad calls to free(). This is a very odd bug and bares a bit of explaination. Memory in C is looked at by free and malloc as something known as a chunk. Freed chunks are held in a doubly linked list and several things are

done with this list in order to minimize memory fragmentaion. Basically, when you are returned a pointer to malloc, the previous 4 bytes are used for chunk information, and after being free'd the 4 bytes directly after the memory area your pointer pointed to hold links to the previous and next chunks of free'd memory. In this header is a short containing a 1 or a 0 which tells free and malloc if the previous chunk is in use. None of this is terribly important unless we try to free a chunk of previous memory that has already been freed. In this case, free will check thenext chunk's prev in use variable, see it is not in use, and try to put this pointer into it's list of free'd memory to be reused. In the process of doing this, it will assume the first 4 bytes of your data are valid previous and next (relative) pointers. This can also be accomplished if an attacker can manage to overwrite the next block's prev in use variable. Essentually this allows the attacker to overwrite 4 bytes anywhere in memory, possibly a stack pointer. This is really simply a variation of heap overflow attacks. A prime real world example of this is the traceroute heap overflow.

Basically, a special string copying function mallocs a large chunk of memory, then breaks it up into several pieces of memory, handing out pointers to places inside the malloc'd memory. While this cuts down on the mallocing over-head, it becomes a problem when the rest of the code doesn't take this into consideration. Later in the code, those pointers are treated each as seperately malloced chunks of memory and each is free'd. While this cannot be spotted in any one section of code, it is certainly a reminder of our advice concearning specialized string manipulation functions, and a good example of a free induced heap overflow.

5 Format String Vulnerabilities

Format string vulnerabilities are the next thing to check for, being the easiest to spot, but also the easiest to write portable (and devistating, at least for ELF systems) exploits for. This is due to the GOT (global offset table) portion of ELF binaries. What this essentially means, is that any format string exploits will not only be able to bypass stack return address protections, but it will also not require the standard bufer overflow song and dance to get offsets, as the GOT always has the same address. (Please see the TESO paper for an exhaustive study of this). Thankfully, these are easy to find. All we need to do is grep for all printf functions, including those such as err, verr, warn, syslog, setproctitle, etc. Then we take that list and check for two things. The first and most important thing to check for is that our functions use format strings at all. For example, a vulnerable printf call might look like this

```
void foo(bar)
  char *bar;
{
  printf(bar);
}
```

While a clean printf will look thusly

```
void foo(bar)
  char *bar;
{
  printf("%s", bar);
}
```

The next check we need to make is for the now depricated use of size limits within the formatstring. All checks should be done outside of the print statement. Examples of this might take the form of formating such as %44s, which is supposed to limit the length of the printed string to 44 characters. This should no longer ever be used unless it's to align data, and you are assured that the string is smaller than the alignment size.

A pretty good example of a format string vulnerability as they are generally found in the wild can be found in the popular POP3 / IMAP daemon, perdition. (found by GOBBLES security) As usual the issue was with a syslog call rather than the familiar *printf calls. The line
syslog(priority, vl-¿buffer);

occurs in the function __vanessa_logger_log() in vanessa_logger.c. Obviously the only change needed to fix the issue is the addition of a "%s" argument in the middle of our call, as with the above example. However some developers seem to think that error and syslog calls are somehow a different beast than other format string calls, and you will almost certainly find more issues with these calls than with others.

6 Integer Overflows

Our next checks, and our most difficult, are now necessary mostly due to the recent OpenSSH vulnerability. Nothing can create more of a headache than hunting for integer overflows, primarily because you simply can't grep for them. You must read every line which deals with the copying of memory which is dynamically sized. For example, lets assume that we get from the network a number of structs we are to recieve. so
u_int num_of_structs = grab_from_input();

And so we naturally allocate memory next to hold this array of structs, like so

```
mystruct *mem = malloc(num_of_structs * sizeof(mystruct));
```

While both num_of_structs, and sizeof(mystruct) may be less than UINT_MAX, (and we can even check both to be sure they are individually under UINT_MAX), multiplying them together may result in a

small positive int. We have then created a buffer overflow when we try to copy into mem, even if we use safe copy functions. Pointer arithmetic with similarly flawed numbers can also cause an issue, forcing us to copy to memory either behind or far ahead of our allocated memory.

We must also be aware of issues of sign'dness. Note that many functions will take size_t which is generally cast to an unsigned integer. Mixing up signed and unsigned is the classic way to incur an integer overflow related buffer or heap issue. Make sure that function's which return an int to be used as size_t are checked for sanity. For example lets assume the following:
void make_buf(size_t);
int input();

void *buf = make_buf(input());

If input() errors and returns -1 for example, make_buf will interpret this as simply a very large number. (Specifically 2^bits where bits is the bit-width of the type.)

6.1 OpenSSH bug

As an example, of what a real world integer overflow might look like, let us examine the OpenSSH 3.3 patch, as this shows both the issue and the fix. The patch makes only one change, and this change is largely a check of protocol sanity (patch presented in unified diff format):

```
--- auth2-pam.c 22 Jan 2002 12:43:13 -0000 1.12
+++ auth2-pam.c 26 Jun 2002 10:12:31 -0000
@@ -140,6 +140,15 @@
nresp = packet_get_int(); /* Number of responses. */
debug("got %d responses", nresp);

+
+ if (nresp != context_pam2.num_expected)
+ fatal("%s: Received incorrect number of responses "
+     "(received %u, expected %u)", __func__, nresp,
+     context_pam2.num_expected);
+
+ if (nresp > 100)
+ fatal("%s: too many replies", __func__);
+
  for (i = 0; i < nresp; i++) {
  int j = context_pam2.prompts[i];
```

While it is hardly likely that nresp can or would overflow, one should note that nresp is used several times without checks, such as in the line:

response = xmalloc(nresp * sizeof(char*));

As long as we take heed of what is a sensable number of responses for the protocol, the problem is simply avoided. The issue however, is that finding or even deciding where these overflows can occur is quite difficult, and untilvery recently was never considered an issue.

One last quick and easy check should be done for integer overflow issues. A simple grep for atol and atoi should be done. While these functions are sometimes used safely, it is a bad practice to leave them in code you'd like to be sure is secure. This is because both atoi and atol have no boundry checking, and handle incorrect characters quite poorly. (for example "47x") All instances of these functions should be replaced by strtol.

7 Race Conditions

If our piece of software happens to be threaded, we should now search for any possible race conditions. This is another point, such as integer overflows that can lead to buffer and heap overflow vulnerabilities. This is another difficult category which must be done by hand, but can still be done systematically. First, find all variables that will be handled, or can be handled by multiple threads. Then, it's useful, but not necessary to flow out the activities and relativing timing of your various threads, or simply types of threads. Now, you can match these volatile variables to all possible places they will be changed in your flow chart. Once we know where everything can be accessed, we can match these points with source code, and make sure that we've placed the proper mutexes around these variables at the proper times. If you're particularly interested in being sure there are no vulnerabilities, you should also go through the lines and logic around the mutexes to be sure that all sensitive variables and changes are made within the lock, and nothing is omitted.

7.1 Binding Flaws

It can also lead to a more interesting class of bug, known as binding flaws. A good example of this invloves two simple calls. Access() is a call that returns an error if the uid cannot access a certain file in a certain way. The other function is open().

```
if (access(tmpfile, W_OK) == 0) {
if ((fd = open(tmpfile, O_WRONLY)) == -1)
/* complain and die */
/* write to file */
}
```

Lets assume the timeframe goes like this. tmpfile points at /tmp/foo, the access call goes just fine since our user can write to /tmp/foo. Right after access returns, but just before the open() call, we delete /tmp/foo and link /tmp/foo -¿ /bin/sh, and write our own malicious binary over the old /bin/sh.

To prevent this sort of mischief we need to make sure we have locked the file before the access call, and unlock it only after we're done writing and the file's been closed. To exploit it, we need perfect timing.

7.2 Signal Handlers

We should also be on the look out for race conditions resulting from unsafe signal handling. We will want to check for calls such as syslog(), or any functions which use malloc() inside a signal handler. The reason for this is that re-entering into malloc() can lead to a corruption of the heap. If the signal handler has no re-entry protection, it is likely that it is vulnerable to attack, (although the timing on this is very difficult, if doable at all.) One of the most glaring things to hunt for is SIGURG handled in remote daemons. If the signal handler happens to use an unsafe function, it's likely that it can be exploited remotely. A full list of functions which are safe to be called from signal handling functions can be found in the OpenBSD sigaction man page. (http://www.openbsd.org/cgi-bin/man.cgi?query=sigaction) In general, signal handlers should be combed through by hand to check for any non-atomic code. Ideally, we should find code which does nothing more than set a flag.

8 Binary Auditing

Generally, binary auditing is is not often attempted by developers, as they have the source code avaiable to them, and source code auditing is a much more complete method. Binary auditing is most useful in a few key situations though. These situations involve security sensitive pieces of software which have core dumped, even after a code audit, or pieces of software which are closed source in part or in full. It can also be useful in detecting hard to find integer overflows in network software. For the basic security auditing we're engaging in, we'll need a few simple tools, but not nearly as many as we'd use in forensic testing for the binary's functionality. Our purposes will require a debugger which can load core files (We'll use gdb), a system call tracing tool (We'll use ktrace on OpenBSD current, but strace and ptrace are also acceptable), and a packet injection tool (We'll use nemesis, because I've done some work on it, but I'm also partial to using the MIT pdos lab's modular click router for this - nemesis is a faster and easier tool, but click is more complete). Binary analysis occurs generally in two distinct steps. Firstly, we try to force the program in question to fail, where by fail I mean crash in some fashion (optimally leaving us a core file to work with) or even just act in some way it was not designed to or give us garbage output. The second stage involves examining the mode of failure, or the evidence our program leaves behind (usually in the form of a core file, or error log). From this secondary step we can determine if the problems we unearthed in the first step are indeed security flaws.

8.1 Stress Testing

The first and easiest approach to reaching that state of failure is simple stress testing. The auditor would create a program to generate random strings, and send them to the program's input stream, in an effort to reach a state conducive to step two. More than likely, however (we hope), the original author of the

program has done this already and has found and fixed most simple input problems. This is usually true in the case of closed source software of even dubious security, but stress testing is always a useful place to start. Often rather than fixing holes, authors will simply hide the problems with exception handling (in the case of languages which support this such as C++) and we will not see errors we have brought about. This also will not catch more complex issues, or even malloc overwrites, so obviously stress testing is only our first step. There are several utilities which can automate this, and shell scripts can be written in a matter of minutes as well, so we wont go into depth on exactly what to do, but as always, the goal here is to get it to coredump, and to keep in mind what we did to get it to do so.

8.2 Fault Injection

In this basic vein of auditing, we move on to a more sophisticated type of stress testing known as fault injection. This is the mose prominant and most successful method of binary auditing. While no binary auditing technique can be exhaustive as there is no finite set for input for programs of any complexity, fault injection allows us to work with inputs that are likely to cause undesireable results at critical locations, giving us a good idea of what is secure and what is not. Fault injection throws out the old garbage in, garbage out idea, and says instead garbage in, useful inofrmation about the flow of data within a program out. Our basic technique will take place in a series of 3 steps.

Our first step will be to determine likely points of failure in the piece of software. This can be accomplished in any of several ways, including call graphs, syscall monitors such as {p,k,s}trace, or, if you're so inclined, simply looking at the assembly.You should keep in mind, while hunting, the various software vulnerabilities we covered above.

Next, we'll craft a series of inputs for each point likely to produce evidence of a vulnerability. It is important to note that we are not trying to exploit the program now as this might be less than conducive to noticing faults if we fail. Our goal is to make the program output as much inofrmation as possible about what's going on at each point we test. An input should be created and tested for each possible vulnerability at each point, tailered to the method of failure for each vulnerability type. Quite often, we have missed possible points of failure and it is always beneficial to be exhaustive, as we may stimulate vulnerabilities we had not previously anticipated.

Finally we test our inputs at each point and carefully evaluate our output, keeping in mind which inputs are sane and which are not. Sometimes, a program which simply runs as normal when recieving an input which is not sane is a good indication that the programmer did not anticipate such an input. At all times it is useful to take what you know of the structure of the program (learned from traces, graphs, and assembly) and use your own programming knowledge to anticipate how to catch bad input. It can be very clear at times if this crucial step has been neglected.

Any output which is not desirable from the program, as well as any interperate non-sane input as sane should be flagged and tested further. For each tested point, we now have two sets of information, and we can therfore craft input with a bit more intellegence than we had previously at our disposal. The next set of inputs should be again, simply information gathering. If it is possible to output the stack, do so. If you can get some idea of the limits imposed on strings, that is also useful. More refined strings now allow us to decide not only what kind of vulnerability exists, but what it's limits are, and give us a good idea of how they can be exploited.

9 Static Analysis and Future Work

One of the most promising new techniques pioneered by David Wagner, and covered in is doctoral thesis, is static analysis. Building on two interesting assumptions, David and his team have been able to develope a piece of software that has detected buffer overrun vulnerabilities in security sensitive programs that were previously audited by hand several times. By treating strings as an abstract data type, ignoring primitive pointer operations, and modeling buffers as pairs of integer ranges (the allocated size, and the size currently in use), they have been able to model their auditing as an integer range analysis problem. While still in the beginning stages, the technique has show a good deal of promise, if limited to a very small, yet prevalent form of vulnerability. While the work also represents a significant leap in integer range analysis as well, the main issues we are interested in center around the sheer amount of vulnerabilities that occure not with complicated pointer arithmetic, but in simple standard C string manipulation functions.

10 Summary

Note that we are not deciding if it can be exploited, only how it can be exploited. It is the expressed oppinion of this author that there is honestly no such thing as an unexploitable vulnerability. Further, it is simply foolish for whitehats to assume that any vulnerability is safe to be left unpatched in the wild.

11 End Note

Several people, during the writing of this paper asked me if I was going to cover the use of automated code auditing software. I, at one point, gave ITS4, a commonly used software auditing tool, a try. It core dumped. Looking through the code I found several sections which had been commented with a special flag so that if anyone happened to audit ITS4 with ITS4 it'd come out clean. We had the idea that one could do rather humorous things with large company's auditing systems that accept code in email and run it through ITS4, then email back the output, but that is fodder for another paper.