FOCUS

Surviving Network Attacks

Check for updates

BEN LAURIE, A. L. DIGITAL

Network Forensics

The dictionary defines forensics as "the use of science and technology to investigate and establish facts in criminal or civil courts of law." I am more interested, however, in the usage common in the computer world: using evidence remaining after an attack on a computer to determine how the attack was carried out and what the attacker did.

The standard approach to forensics is to see what can be retrieved after an attack has been made, but this leaves a lot to be desired. The first and most obvious problem is that successful attackers often go to great lengths to ensure that they cover their trails. The second is that unsuccessful attacks often go unnoticed, and even when they are noticed, little information is available to assist with diagnosis.

OBFUSCATION AFTER A SUCCESSFUL ATTACK

Once an attack has succeeded, the attacker generally has complete access to the attacked system. The wily hacker can then remove evidence of the attack by modifying logs, deleting core dumps, and so forth. Indeed, many attacks install new software and modify the system so that the new software can't be seen using standard utilities such as ps. Once this has taken place, it can be very

Good detective work means paying attention before, during, and after the attack.

Network Forensics

difficult to clean the system, let alone determine what has happened.

UNSUCCESSFUL ATTACKS

Far more attacks fail than succeed. The most popular form of attack, despite years of warnings, is still the buffer overflow. In this attack, typically, some code is deposited on the stack and the return address of a function call is corrupted to cause the planted code to be called. The difficulty, from the attacker's point of view, is that the exact address the code will end up at is heavily dependent on the exact build of the target system (e.g., operating system, compiler version, exact compilation flags used, version of target code, options used when building, libraries linked in). Because of this, the attacker often has to try many slightly different versions of the attack until one works. An example that I'm personally familiar with is the Slapper worm, as I found and fixed the vulnerabilities that it exploited in OpenSSL.[1] The code contained an amazingly long table, part of which is provided in figure 1.

This table is used to determine addresses for the attack to use, but as can be seen, even when the worm is sure of the platform and Apache version (which may not always be the case), it sometimes still needs to try multiple addresses. If it uses the wrong addresses, then the usual result is that the target segfaults (segmentation faults), rather than succumbing to the attack.

In the good old days, this would have resulted in a core dump, which could potentially be a useful diagnostic tool (though bear in mind that since the stack is corrupt, even a core dump can be of limited use). Unfortunately, security concerns with core dumps, which can contain passwords, private keys, and so forth, have meant that most systems don't produce core dumps by default from the kinds of software that are likely to be targeted (e.g., sendmail, Apache, IMAP servers). Of course, this can generally be fixed, but you have to remember to do that *before* you get attacked.

Even given a core dump, determining how the attack worked can be difficult, since the direct evidence has already gone, and since many successful attacks corrupt important data structures, such as the stack or heap. Also, it is an unfortunate trend that many widely used packages disable stack frames as a performance enhancement, making core dumps very difficult to interpret. For example, in GCC (Gnu Compiler Collection) this is the `-fomit-frame-pointer` flag. If frame pointers are omitted, debuggers cannot display stack backtraces.

Despite these problems, it is certainly worth ensuring that services are run such that core dumps can be produced and that they are compiled so the core dumps are useful: in general, enabling debugging information and not disabling stack frames is the optimal approach.

CLASSES OF ATTACK

Attention is, unsurprisingly, focused on attacks that use the Internet as a transport medium—but there is another class of attack of concern to some: local attacks. These are worth discussing even in relation to Internet-based attacks. Broadly speaking, Internet-based attacks can be divided into two types: protocol attacks and malware. Of course, these problems are common to all networks; there's really nothing special about the Internet except its ubiquity.

PROTOCOL ATTACKS

Protocol attacks rely on flaws in protocols or implementations of protocols to cause undesired behavior in the software implementing the protocol. The best-known class of protocol attack is the widely discussed buffer overflow, but there are many others: less well-known examples are version-downgrade and related security-reducing attacks, or attacks that rely on unexpected input to cause incorrect behavior (for example, using ".." in paths to access directories outside the published tree, or SQL injection to cause arbitrary SQL to be executed by the server).

Others, such as cross-site scripting or cookie hijacking, are arguably not strictly protocol attacks, but I will classify them as such for two reasons:

1. They are based entirely on I/O to a program.
2. They do not use executable code as their primary means of exploitation—*primary* because buffer overflows usually do carry executable code as their payload, whereas malware uses locally executable code to gain control in the first place.

MALWARE

Malware is a term used to describe the range of software that ultimately relies on being executed in order to perform its "evil" (for want of a better word) mission. It covers viruses, Trojans, and worms, though the boundaries between these are beginning to become blurred. In general, these attacks rely on user error or ignorance to get executed, a well-known recent example being the MyDoom virus, which led recipients to believe they were being resent an e-mail that couldn't be delivered the first time around, thus enticing them to double-click the attachment, which then did Bad Things.

AFTER-THE-FACT FORENSICS

Usually, a new attack, be it malware or a protocol attack, is discovered after the attack has succeeded, because the behavior of the victim machine is altered. For example, in an article in process with Richard Clayton of the University of Cambridge Computer Laboratory, we are looking at the use of infected machines for the sending of spam by (among other things) analyzing the mail logs of a large Internet service provider. Interestingly, Clayton's software, designed to spot machines sending spam, rapidly picked up the spread of MyDoom, because it radically altered the e-mail habits of infected machines. More

Excerpt of Slapper Worm Table

```
struct archs {
char* desc;
int func_addr; /* objdump -R /usr/sbin/apache | grep free */
} architectures[] = {
{ "Caldera OpenLinux (apache-1.3.26)", 0x080920e0 },
{ "Cobalt Sun 6.0 (apache-1.3.12)", 0x8120f0c },
{ "Cobalt Sun 6.0 (apache-1.3.20)", 0x811dcb8 },
{ "Cobalt Sun x (apache-1.3.26)", 0x8123ac3 },
{ "Cobalt Sun x Fixed2 (apache-1.3.26)", 0x81233c3 },
{ "Conectiva 4 (apache-1.3.6)", 0x08075398 },
{ "Conectiva 4.1 (apache-1.3.9)", 0x0808f2fe },
{ "Conectiva 6 (apache-1.3.14)", 0x0809222c },
{ "Conectiva 7 (apache-1.3.12)", 0x0808f874 },
{ "Conectiva 7 (apache-1.3.19)", 0x08088aa0 },
{ "Conectiva 7/8 (apache-1.3.26)", 0x0808e628 },
{ "Conectiva 8 (apache-1.3.22)", 0x0808b2d0 },
{ "Debian GNU Linux 2.2 Potato (apache_1.3.9-14.1)", 0x08095264 },
{ "Debian GNU Linux (apache_1.3.19-1)", 0x080966fc },
{ "Debian GNU Linux (apache_1.3.22-2)", 0x08096aac },
{ "Debian GNU Linux (apache-1.3.22-2.1)", 0x08083828 },
{ "Debian GNU Linux (apache-1.3.22-5)", 0x08083728 },
{ "Debian GNU Linux (apache_1.3.23-1)", 0x08085de8 },
{ "Debian GNU Linux (apache_1.3.24-2.1)", 0x08087d08 },
{ "Debian Linux GNU Linux 2 (apache_1.3.24-2.1)", 0x080873ac },
{ "Debian GNU Linux (apache_1.3.24-3)", 0x08087d68 },
{ "Debian GNU Linux (apache-1.3.26-1)", 0x0080863c4 },
{ "Debian GNU Linux 3.0 Woody (apache-1.3.26-1)", 0x080863cc },
{ "Debian GNU Linux (apache-1.3.27)", 0x0080866a3 },
/* targets de BSD */
{ "FreeBSD (apache-1.3.9)", 0xbfbfde00 },
{ "FreeBSD (apache-1.3.11)", 0x080a2ea8 },
{ "FreeBSD (apache-1.3.12.1.40)", 0x080a7f58 },
{ "FreeBSD (apache-1.3.12.1.40)", 0x080a0ec0 },
{ "FreeBSD (apache-1.3.12.1.40)", 0x080a7e7c },
{ "FreeBSD (apache-1.3.12.1.40_1)", 0x080a7f18 },
{ "FreeBSD (apache-1.3.12)", 0x0809bd7c },
{ "FreeBSD (apache-1.3.14)", 0xbfbfdc00 },
{ "FreeBSD (apache-1.3.14)", 0x080ab68c },
{ "FreeBSD (apache-1.3.14)", 0x0808c76c },
{ "FreeBSD (apache-1.3.14)", 0x080a3fc8 },
{ "FreeBSD (apache-1.3.14)", 0x080ab6d8 },
...
{ "Slackware 8.0 (apache-1.3.22)", 0x08102b78 },
{ "Slackware 8.1 (apache-1.3.26)", 0x080b2100 },
};
```

FIG 1

Network Forensics

often, the owner of the machine starts to notice suspicious symptoms: the CPU is always at 100 percent usage, Internet access is suddenly slow, some programs don't seem to work "quite right," and so on. An example of this occurred many years ago when I was working in typesetting. Ventura Publisher, a very popular typesetting program of the '80s, suddenly stopped working on one machine. A few days later, another stopped. I was called in to investigate. I discovered the virus later named Bethlehem had infected the machines—but it had a bug. When it infected a .EXE file, its check for its own presence failed, and so it kept reinfecting them until they were too large to fit into memory.

Once this altered behavior has been noticed, the task is then to figure out what went wrong. Usually this is a messy affair, involving poking through logs, checking old e-mails, checking the disk for unexpected executables, and so forth. Sophisticated malware even covers its own tracks and modifies system binaries to hide its continued activities (this is what "rootkits" do), leading to great difficulty in tracking it down. Indeed, the standard advice is to take the machine offline, remove its disk and mount it read-only in another machine, and run all sorts of cunning diagnostics to try to reverse-engineer the malware (using, for example, The Coroner's Toolkit, created by Dan Farmer and Wietse Venema).[2]

Of course, most of the world hardly needs to worry about this—most of the world is neither an early victim nor the people who have to figure out what went wrong and how to fix it. Most of the world can sit around quite happily and wait for someone else to worry about it, figure out what's going on, and tell them; then they can start worrying about how they are going to defend against the problem, or start complaining about how ineffective XYZ's security is.

It is those who have to diagnose the problems, and how they can make their lives easier, that I am most interested in.

BEFORE-THE-FACT FORENSICS

The lazy and easy answer to this is, in general, logging. And, for the most part, most software will, if configured correctly, produce huge amounts of logs. But, as my friend and sometime collaborator Tina Bird has been pointing out for years, these logs are often close to useless for both detection and diagnosis. Why? Because, in order to diagnose logic faults and other bugs in the software, they are typically designed to be read manually by the programmer, or at least someone well acquainted with the code, in conjunction with a pretty firm knowledge of inputs and outputs from the program. They have not been designed to help with malicious attacks, nor have they been designed to be automatically analyzed. The problem with the former is obvious—but the problem with the latter may be less clear: one of the characteristics of attacks is that they are not always detected very soon after they occur; indeed, some are never detected. This may be either because the machine goes out of use before anyone notices (and this can be periods of years—the behavior of machines used for spam makes it quite clear that infected machines are ready in a pool, undetected, waiting to be used!) or, perhaps more interestingly, because the attack fails, so there's nothing obvious to notice.

How can this situation be improved? Interestingly, detailed logs of what is happening *inside* a program, which is what we usually get in logs, are probably not of much use; generally, successful attacks do not travel down the paths we expect inside the code, so the logging tends either not to occur or to be difficult to relate to the cause. Far more useful is knowing exactly what the inputs and outputs were, because then we can repeat the attack at our leisure, using debuggers, code instrumentation, or whatever to diagnose the vector of attacks.

BUT ISN'T LOGGING EVERYTHING A PROBLEM?

Yes, it is. The volumes involved are often huge, and most likely to be hugest in the places the attacks are most likely to occur. There is no magic bullet here, but there are some strategies that can help to mitigate the problem.

For example, I recently wrote a forensic logging module for Apache called mod_log_forensic. It logs twice: once at the start of a request and once at the end. Since most attacks on Apache fail (as discussed earlier), 99 percent of the usefulness of the module would be retained if we kept logs that related only to failed attacks. How do we spot those? That's easy: they're the ones that log at the beginning of the request, but not the end (because the server died in the middle). To reduce the log size of busy sites, I have considered having an external program filter the logs, holding the pre-request component for a time, and logging to disk only if the corresponding post-request entry does not arrive. This could be made more sophisticated by making the program aware of the death of the server and logging in that case, but this approach worries me—forensic logging should be lightweight and robust, lest it become a target for attacks itself.

Although there is an element of risk here, an attack that succeeds the first time, or in a short time frame, could kill the log-reducing filter and cause the log not to make it to disk at all; or attacks of interest might be directed at a CGI (common gateway interface) and thus not cause the death of the server itself. In some environments, particularly where usage is very high and there is no external CGI, this approach would make sense. Bear in mind that in-process CGI such as mod_perl or mod_php would still show up in the logs.

Mod_log_forensic also interacts with another popular module: mod_unique_id. If this module is in use, then mod_log_forensic will use its ID instead of its own; thus, the forensic log can be tied into the access log, as well as logs produced by CGIs. Mod_unique_id also handles log pooling (where a farm of machines all share a common log), which mod_log_forensic does not do on its own (there is a risk of ID collision).

Mod_log_forensic *doesn't* log any output from the server, because (in theory, at least) the output should be deterministic, or if not, irrelevant. Of course, other applications may have different requirements with respect to input and output.

Also recently added to Apache are forensic modules mod_log_backtrace and mod_whatkilledme. These modules log only when something goes drastically wrong, which means they are far lower volume, but again increase the risk that the event of interest will not be logged. Mod_log_backtrace also suffers from the major disadvantage that it tells you where you were when you died, but not what killed you. Mod_whatkilledme *may* tell you that, but only if what killed you didn't corrupt it (this can't happen in mod_log_forensic, because it logs before anything else happens, not after).

Another tactic that can be used to reduce volume is to log only a subset of the input. In fact, mod_log_forensic does this—it does not log data fed to CGIs (or modules) via HTTP POST requests. This is done for two reasons, the main one being pragmatic: POST requests tend to be large. The second reason is that Apache doesn't have a mechanism to allow arbitrary modules access to POST data, so such logging would have to be done by the module handling the request (if it is mod_cgi, then that can be configured to log the request data).

In the end, what to log, how to reduce volume, and the risk incurred by doing so are highly dependent on the application. But the good news is that it's usually largely in the control of the application authors, if they decide to do it, rather than being a random assortment of incidental information gathered more by luck than judgment.

MAKE SURE LOGS ARE USEFUL

A detail that is often missed until too late is that you must take care with forensic logs to ensure that they can be used to accurately reconstruct the attack. For example, astonishingly, until recently Apache didn't prevent carefully constructed requests from causing fake entries in the access logs.

Generally, **successful attacks** do not travel down the paths we expect inside the code.

It is also important that the logs can be analyzed automatically. When the time comes to look at a few dozen gigabytes of log, the last thing you want to have to do is to read them individually.

Luckily, these two details are easily handled, and at the risk of teaching my grandmother to suck eggs, I'll outline the general strategy.

LAYOUT

The layout should be easily parsable by tools such as grep, sed, awk, and perl. What this means in general is that it should:

1. Be entirely composed of readable ASCII.
2. Have unambiguous markup.
3. Be line-oriented.
4. Not have overly long lines (though I would rather use custom versions of grep than compromise log quality by artificially shortening lines).

Since in almost all cases, what is being logged is under the control of the attacker, you should not rely on any protocol rules being followed to enforce these constraints.

Network Forensics

This means that you should use escaping for non-ASCII characters (in most cases, control characters, too, to avoid attacks that are designed to exploit the terminal program used to display the log—yes, these attacks have been seen in the wild). To have unambiguous markup you need to choose a field separator that is also escaped when it occurs in the fields, and if fields are named rather than positional, then, ideally, use a second separator for those.

For the sake of illustration, here's what I did for mod_log_forensic. First, rather than logging the raw input, I log the HTTP request after initial parsing, so it has been decomposed into individual headers and the request line itself. This was a calculated risk—the downside being that bugs in the parsing code would be exploited before the log occurred, but the upside being that the log itself is far more easily parsed. It also happened that this approach disturbed the structure of the server much less than logging raw input would have (in fact, it disturbed it not at all; mod_log_forensic is a standard module). Fields are separated by vertical bars (hex 7c) and field components by colons (hex 3a). For the preprocessing log, the first field is a + followed by the unique ID; the second is the request; and the remaining fields are the headers, in the format <header name>:<header contents>. The entry is terminated by a newline (hex 0a). All characters below hex 20 and above hex 7e are URL-escaped (that is, replaced by %xx where "xx" is the hex value), as are the colon, vertical bar, and percent. In the case of a post-processing entry, the entire line is a — followed by the unique ID.

PROTECTING LOGS

As I said before, successful attackers often go to great lengths to remove logs that disclose their activities. Indeed, there are widely available tools for covering tracks after an attack. So, how do you ensure the logs do not go missing once attacked? The most obvious tactic is to log to a different machine, though this can be expensive *and* introduce another vulnerability, at least to a logging denial-of-service.

A less obvious tactic would be to write logs to write-once media; it's worth remembering, however, that not all write-once media is *really* write-once—for example, WORM (write once, read many) disks can be erased once written. In the more likely case of logging on a different machine, the obvious mechanism to use is the existing capability of syslog to send messages to another machine. Because you might well want to do sophisticated filtering (such as suggested for mod_log_forensic)—which should really be run on the logging machine and not the machine generating the logs, to protect the filters from the attacker—you should probably consider syslog-ng[3] as the logging platform. This easily allows an external filter to be plugged in. A useful resource for logging strategies in general can be found at the log analysis Web site.[4]

PLAN FOR THE INEVITABLE

Piecing together what happened to a system from distributed information not intended to be used for that purpose is a frustrating and unsatisfactory process. It is far better to plan for the inevitable attacks and ensure that necessary information is securely gathered, and remains available even if an attack is successful—and if you are writing code, make sure that necessary information is available in the first place! Q

REFERENCES

1. For more on the Slapper story, see my rant: Security: Why do I bother? O'Reilly Network; http://www.oreillynet.com/pub/wlg/2004.
2. The Coroner's Toolkit; see: http://www.porcupine.org/forensics/tct.html.
3. Scheidler, B. syslog-ng. http://www.balabit.com/products/syslog_ng/.
4. Bird, T., and Ranum, M. Loganalysis.org, http://www.loganalysis.org/.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BEN LAURIE is technical director of A. L. Digital and author of the Apache-SSL Web server. He is a founding director and head of security of the Apache Software Foundation. He is also a core member of the OpenSSL Project, the world's most widely used cryptographic library, and numerous other Internet projects. His main interests are security, cryptography, privacy, civil liberties, and beer.