

VIRUS ANALYSIS 1

Let free(dom) Ring!

Frédéric Perriot and Péter Ször
Symantec Security Response, USA

On 30 July, 2002 a security advisory from *A.L. Digital Ltd* and *The Bunker* disclosed four critical vulnerabilities in the *OpenSSL* package. *OpenSSL* is a free implementation of the Secure Socket Layer protocol used to secure network communications and it provides cryptographic primitives to many popular software packages, including the *Apache* web server. Less than two months later, the Linux/Slapper worm successfully exploited one of the buffer overflows described in the advisory and, in a matter of days, spread to thousands of machines around the world.

Linux/Slapper is one of the most significant outbreaks on *Linux* systems to date. Although the worm has the potential to infect many more machines, it skips private network classes such as 192.168.0.0/16 intentionally and thus it will not spread on local networks. Slapper shows many similarities with the FreeBSD/Scalper worm, hence the name.

Under Attack

Linux/Slapper spreads to *Linux* machines by exploiting the overlong SSL2 key argument buffer overflow in the *libssl* library that is used by the *mod_ssl* module of *Apache 1.3* web servers. When attacking a machine, the worm attempts to fingerprint the system by sending an invalid GET request to the http port (port 80), in anticipation that *Apache* will return its version number as well as the *Linux* distribution it was compiled on, along with an error status.

The worm contains a hard-coded list of 23 architectures upon which it was tested and compares the returned version number against this list. It uses this version information later to tune the attack parameters. If *Apache* is configured not to return its version number or the version is unknown to the worm, it will select a default architecture (*Apache 1.3.23* on RedHat) and the 'magic' value associated with it.

This 'magic' value is very important for the worm and is the address of the GOT (Global Offset Table) entry of the *free()* library function. GOT entries of ELF files are the equivalent of IAT (Import Address Table) entries of *Windows* PE files. They hold the addresses of the library functions to call. The address of each function is placed into the GOT entries when the system loader maps the image for execution. Slapper's aim is to hijack the *free()* library function calls in order to run its own shell code on the remote machine.

The Buffer Overflow

In the past, some worms have exploited stack-based buffer overflows. Stack-based overflows are the low-hanging fruits

compared to second-generation overflows exploiting heap structures. Since the *OpenSSL* vulnerability affected a heap-allocated structure, the worm's author had to deal with a lot of minor details in order to get the attack right for most systems. Thus, exploitation of the vulnerability was not a trivial process.

When *Apache* is compiled and configured to use SSL it listens on port https (port 443). Slapper opens a connection to this port and initiates an SSLv2 handshake. It sends a client 'hello' message advertising eight different ciphers (although the worm supports only one, namely RC4 128-bit with MD5) and gets the server's certificate in reply. Then it sends the client master key and the key argument, specifying a key argument length greater than the maximum allowed `SSL_MAX_KEY_ARG_LENGTH` (8 bytes).

When the packet data is parsed in the `get_client_master_key()` function of *libssl* on the server, the code does no boundary check on the key argument length and copies it to a fixed-length buffer `key_arg[]` of size `SSL_MAX_KEY_ARG_LENGTH`, in a heap-allocated `SSL_SESSION` structure. Thus anything following `key_arg[]` can be overwritten with arbitrary bytes. This includes both the elements after `key_arg[]` in the `SSL_SESSION` structure and the heap management data following the memory block containing the structure.

The manipulation of the elements in the `SSL_SESSION` structure is crucial to the success of the buffer overflow. The author of the exploit took great care to overwrite these fields in a way that does not affect the SSL handshake very much.

Double-take

Interestingly, instead of using this overflow mechanism just once, the worm uses it twice, first to locate the heap in the *Apache* process address space, and again to inject its attack buffer and shell code. There are two good reasons for splitting the exploit into two phases.

First, the attack buffer must contain the absolute address of the shell code, which is hardly predictable across all servers because the shell code is placed in memory allocated dynamically on the heap. To overcome this problem the worm causes the server to leak the address where the shell code will end up and then sends an attack buffer patched accordingly.

The second reason is that the exploit necessitates overwriting the cipher field of the `SSL_SESSION` structure located after the unchecked `key_arg[]` buffer. This field identifies the cipher to use during the secure communication and if its value were lost the session would come to an end too quickly. So the worm collects the value of this field

during the first phase and then injects it back at the right location in the `SSL_SESSION` structure during the second phase.

This two-phased approach requires two separate connections to the server and succeeds only because *Apache 1.3* is a process-based server (as opposed to a thread-based server). The children spawned by *Apache* to handle the two successive connections will inherit the same heap layout from their parent process. Thus, all other things being equal, the structures allocated on the heap will end up at the same addresses during both connections.

This assumes that two fresh ‘identical twin’ processes are spawned by *Apache* to handle the two connections. However, under normal conditions, this may not always be the case because *Apache* maintains a pool of servers already running, waiting for requests to handle. In order to force *Apache* to create two fresh processes, the worm exhausts *Apache*’s pool of servers before attacking by opening a succession of 20 connections at 100-millisecond intervals.

The first use of the buffer overflow by the worm causes *OpenSSL* to reveal the location of the heap. It does this by overflowing the `key_arg[]` buffer by 56 bytes, up to the `session_id_length` field in the `SSL_SESSION` structure. The `session_id_length` describes the length of the 32-byte-long `session_id[]` buffer located after it in the `SSL_SESSION` structure. The worm overwrites the `session_id_length` with the value 0x70 (112). Then the SSL conversation continues normally until the worm sends a ‘client finished’ message to the server, indicating it wants to terminate the connection.

Upon receipt of the ‘client finished’ message, the server replies with a ‘server finished’ message including the `session_id[]` data. Once again, no boundary check is performed on the `session_id_length` and the server sends not only the content of the `session_id[]` buffer but the whole 112 bytes of the `SSL_SESSION` structure starting at `session_id[]`. Among other things this includes a field called ‘ciphers’ that points to the structure allocated on the heap right after the `SSL_SESSION` structure, where the shell code will go, and a field called ‘cipher’ that identifies the encryption method to use.

The worm extracts the two heap addresses from the `session_id` data received from the server and places them in its attack buffer. The TCP port of the attacker’s end of the connection is also patched into the attack buffer for the shell code to use later. The worm then performs the second SSL handshake and triggers the buffer overflow again.

Abusing the glibc

The second use of the buffer overflow is much more subtle than the first. It can be seen as three steps leading to the execution of the shell code: (1) corrupting the heap management data, (2) abusing the `free()` library call to patch an arbitrary dword in memory, which is going to be the GOT entry of `free()` itself, and (3) causing `free()` to be called

again, this time to redirect control to the shell code location.

The attack buffer used in the second overflow is composed of three parts: the items to be placed in the `SSL_SESSION` structure after the `key_arg[]` buffer, 24 bytes of specially crafted data, and 124 bytes of shell code.

When the buffer overflow takes place, all members of the `SSL_SESSION` structure after the `key_arg[]` buffer are overwritten. The numeric fields are filled with ‘A’ bytes and the pointer fields are set to NULL, except the cipher field which is restored to the value that was leaked during the first phase.

The 24 bytes of memory following the `SSL_SESSION` structure are overwritten with fake heap management data. The glibc allocation routines maintain so-called ‘boundary tags’ in between memory blocks for management purposes. Each tag consists of the sizes of the memory blocks before and after it plus one bit indicating whether the block before it is in use or available (the `PREV_IN_USE` bit). Additionally, free blocks are kept in doubly linked lists formed by forward and backward pointers maintained in the free blocks themselves.

The fake heap management data injected by the worm after the `SSL_SESSION` structure poses as a minimal-sized unallocated block, containing just the forward and backward pointers set respectively to the address of the GOT entry of `free()` minus 12 and the address of the shell code. The address of the GOT entry is the ‘magic’ value determined by fingerprinting, and the address of the shell code is the value of the ciphers field leaked by *OpenSSL* in the first phase of the attack, plus 16 to account for the size of the fake block content and trailing boundary tag.

Once these conditions have been set up on the server, the worm sends a ‘client finished’ message specifying a bogus connection id. This causes the server to abort the session and attempt to free the memory associated with it. The `SSL_SESSION_free()` function of the *OpenSSL* library is invoked and this in turn calls the glibc `free()` function with a pointer to the modified `SSL_SESSION` structure as an argument.

One might think that freeing memory is a simple task. In fact, considerable book-keeping is performed by `free()` when a memory block is released. Among other tasks `free()` takes care of consolidating blocks, merging contiguous free blocks into one to avoid fragmentation. The consolidation operation uses the forward and backward pointers to manipulate the linked lists of free blocks, and trusts these to be pointing to heap memory (at least in the release build).

The exploit takes advantage of the forward consolidation of the `SSL_SESSION` memory block with the fake block created after it by setting the `PREV_IN_USE` bits of the boundary tags appropriately. The forward pointer in the fake block which points to the GOT is treated as a pointer to a block header, dereferenced, and the value of the backward pointer (the shell code address) is written to

offset 12 of the header. Thus the shell code address ends up in the GOT entry of free().

It is worth noting that the fake backward pointer is also dereferenced, so the beginning of the shell code is treated as a block header, and patched at offset 8 with the value of the fake forward pointer. To avoid corruption of the shell code during this operation, the shell code will start with a short jump followed by ten unused bytes filled with NOP-s.

Finally, on the next call to free() by the server, the modified address in the GOT entry of free() is used and the control flow is directed to the shell code.

Shell Code and Infection

When the shell code is executed it searches first for the socket of the TCP connection with the attacking machine. It does this by cycling through all file descriptors and issuing a getpeername() call on each until the call succeeds and indicates that the peer TCP port is the one that was patched into the shell code. Then it duplicates the socket descriptor to the standard input, output and error.

Next it attempts to gain root privilege by calling setresuid() with UIDs all set to zero. Apache usually starts running as root and then switches to the identity of an unprivileged user 'apache' using the setuid() function. Thus the setresuid() call will fail because, unlike the seteuid() function, setuid() is irreversible. The author of the shell code appears to have overlooked this fact, but the worm does not need root privileges to spread, since it writes only to the /tmp folder.

Finally, a standard shell '/bin/sh' is executed with an execve() system call. A few shell commands are issued by the attacker worm to upload itself to the server in uuencoded form, and to decode, compile and execute itself. The recompilation of the source on various platforms makes the identification of the worm in binary form a little more difficult. The operations are carried out in the /tmp folder where the worm files reside under the names .uubugtraq, .bugtraq.c and .bugtraq (notice the leading dots to hide the files from a simple 'ls' command).

Now you see me, Now you don't!

Since the worm hijacks an SSL connection to send itself, it is legitimate to wonder whether it travels on the network in encrypted form. This question is particularly crucial for authors of IDS systems that rely on detecting signatures in raw packets.

Fortunately, the buffer overflow occurs early enough in the SSL handshake, before the socket is used in encrypted mode, thus the attack buffer and the shell code are clear on the wire. Later, the same socket is used to transmit the shell commands and the uuencoded worm also in plain text. The 'server verify', 'client finished' and 'server finished' packets are the only encrypted traffic but they are not particularly relevant for detection purposes.

P2P

When an instance of the worm is executed on a new machine it binds to port 2002/UDP and becomes part of a peer-to-peer network. Notice that, although a vulnerable machine can be hit multiple times and exploited again, the binding to port 2002 prevents multiple copies of the worm from running at the same time.

The parent of the worm (on the attacking machine) sends to its offspring the list of all hosts on the peer-to-peer network and broadcasts the address of the new instance worm to the network. Then periodic updates to the hosts list are exchanged between the machines on the network. The new instance of the worm also starts scanning the network for other vulnerable machines, sweeping randomly chosen Class B-sized networks.

The protocol used in the peer-to-peer network is built on top of UDP and provides reliability through the use of checksums, sequence numbers and acknowledgement packets. The code has been taken from an earlier tool and each worm instance acts as a DDoS agent and a backdoor. Much has been written on this topic so we won't repeat information that is available elsewhere.

Conclusion

Linux/Slapper is an interesting patchwork of a DDoS agent, some functions taken straight from the OpenSSL source code and a shell code the author says is not his own. All this glued together results in a fair amount of code, not easy to figure out rapidly. Like FreeBSD/Scalper, most of the worm was probably already written when the exploit became available and, for the author, it was just a matter of integrating the exploit as an independent component.

And, as in Scalper, which exploited the BSD memcpy() implementation, the target of the exploit is not just an application but a combination of an application and the runtime library underneath it. One would expect memcpy() and free() to behave a certain way, consistent with one's everyday-programming experience. But when used in an unusual state or passed invalid parameters, they behave erratically.

Linux/Slapper shows that Linux machines can become the target of widespread worms just as easily as Windows machines do. For those with Slapper-infected Linux servers this will be a day to remember.

Name: Linux/Slapper

Alias:	Apache_mod_ssl.
Type:	C sources based worm that compiles itself to ELF; performs DDoS attacks; spreads via buffer overflow attacks against vulnerable versions of OpenSSL.