WHITE PAPER

Symantec Security Response

An Analysis of the Slapper Worm Exploit

by
Frédéric Perriot and Peter Szor
Symantec Security Response

INSIDE

Table of Contents

List of Figures

1.0 Introduction

On July 30, 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. It also provides cryptographic primitives to many popular software packages, one of which is the Apache Web server.

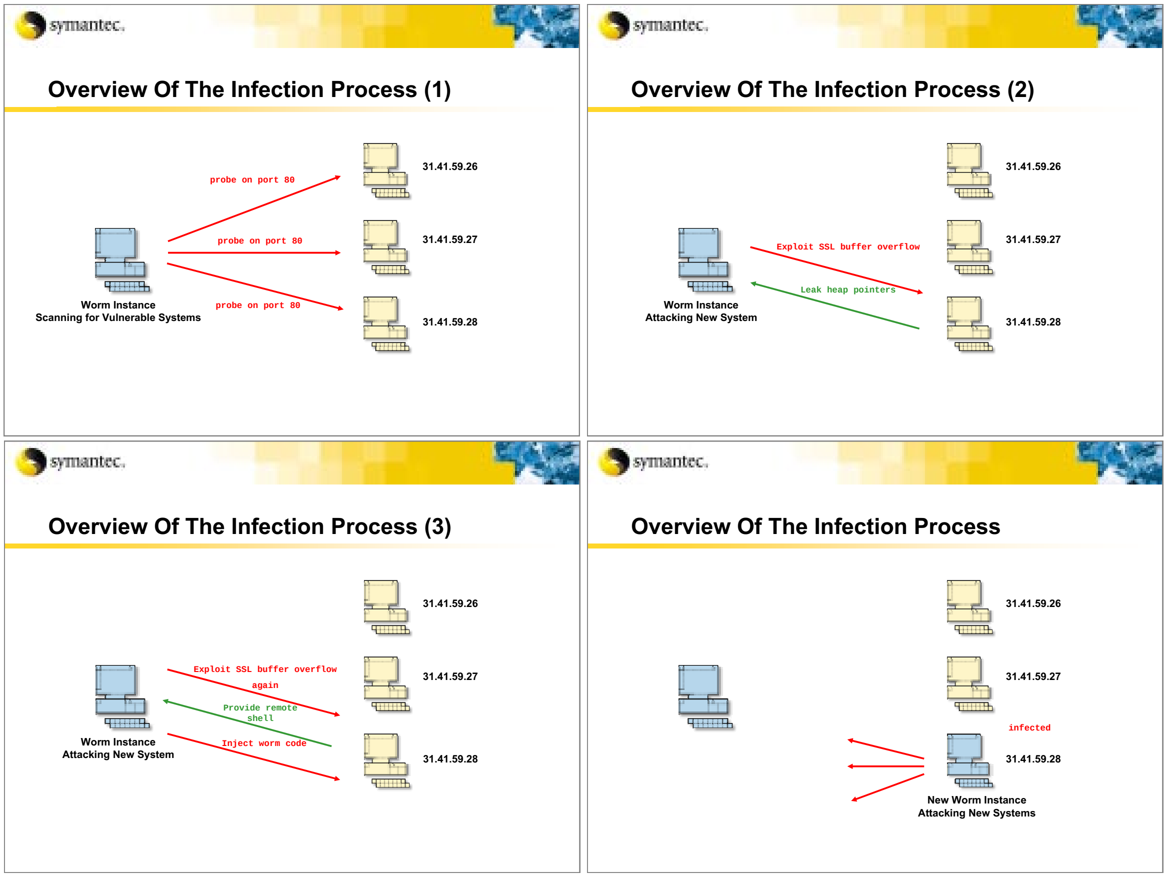
Less than two months later, the Linux/Slapper worm successfully exploited one of the buffer overflows described in the advisory and, in a matter of days, spread to thousands of machines around the world.

So far, Linux/Slapper is one of the most significant outbreaks on Linux systems. The worm could have infected many more machines, but it intentionally skipped private network classes, such as 192.168.0.0/16. As such, the outbreak will not spread on local networks. The Slapper worm is similar to the FreeBSD/Scalper worm, thus, the namesake.

2.0 Under attack

Linux/Slapper spreads to Linux machines by exploiting the long SSL2 key argument buffer overflow in the libssl library, which the mod_ssl module of the Apache 1.3 Web servers used. When attacking a machine, the worm attempts to fingerprint the system by first sending an invalid GET request to the http port—port 80—and expecting Apache to return its version number, as well as the Linux distribution on which it was compiled with an error status.

Refer to Figure 1 for illustrations of the “Overview of the infection process.”



Overview Of The Infection Process (1)

Overview Of The Infection Process (2)

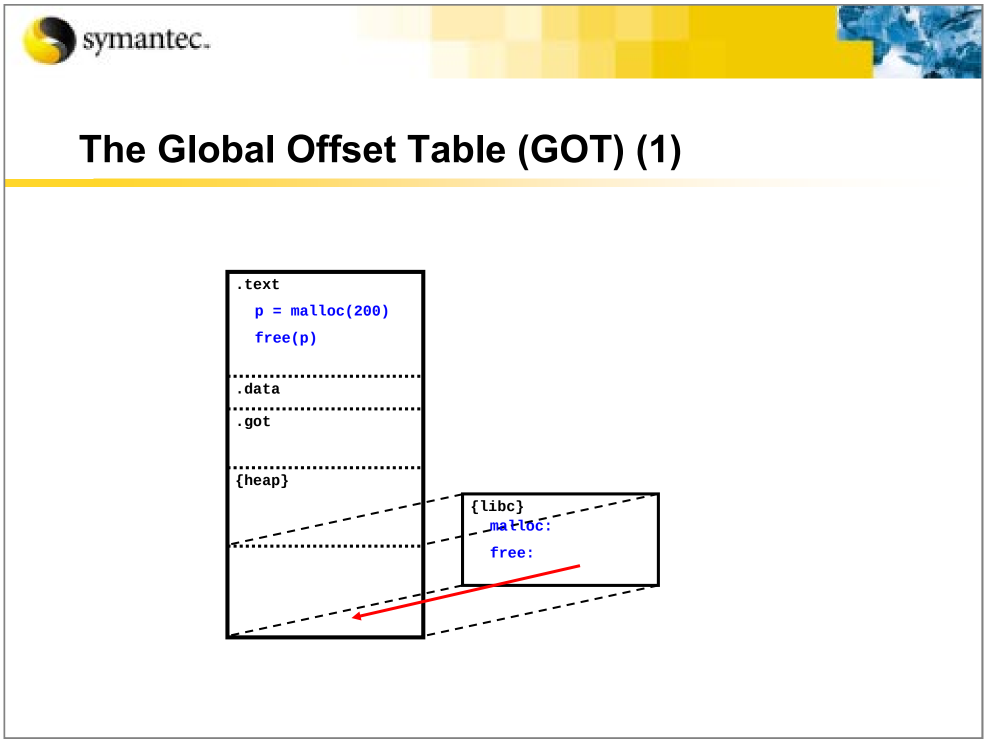
Overview Of The Infection Process (3)

Overview Of The Infection Process

Figure 1: Overview of the infection process

The worm contains a hard-coded list of 23 architectures, on which it was tested, and compares the returned version number to the list. Later, it uses this version information to tune the attack parameters. If Apache is configured to not return its version number, or if the worm does not know the version, it will select a default architecture (Apache 1.3.23 on Red Hat), as well as the "magic" value associated with it. This "magic" value is important for the worm, as it is the address of the Global Offset Table (GOT) entry of the free() library function. The GOT entries of the ELF files are the equivalent of the Import Address Table (IAT) entries of the Portable Executable (PE) files on Windows systems. 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.

Refer to Figure 2 and Figure 3 for illustrations of the "Global Offset Table (GOT) 1" and "Global Offset Table (GOT) 2," respectively.

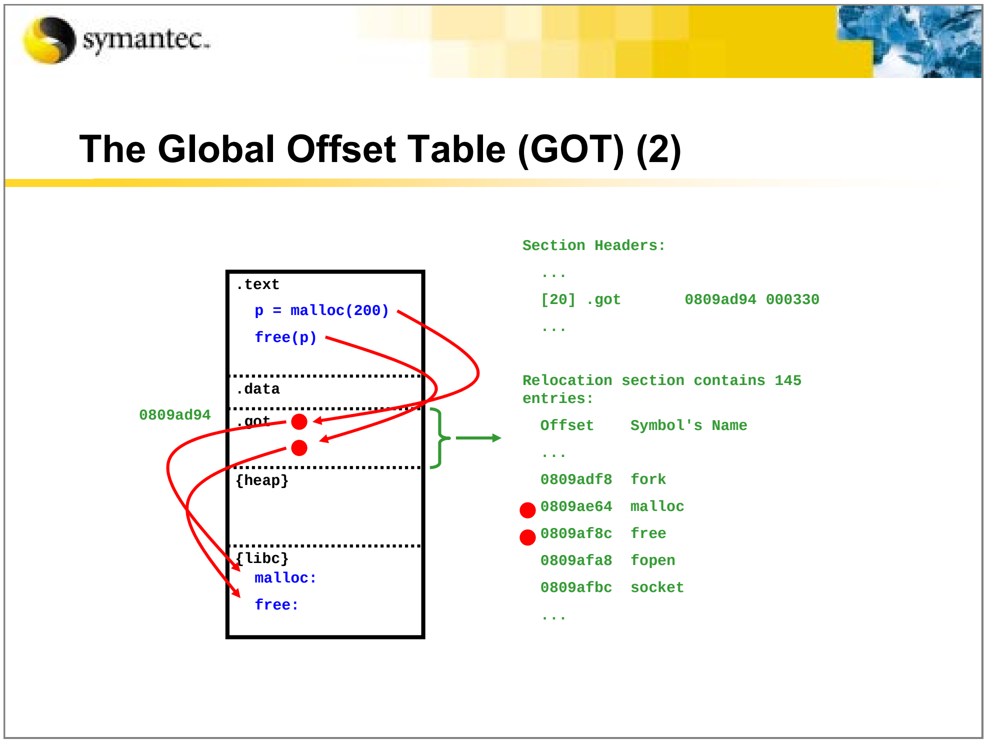


symantec.

The Global Offset Table (GOT) (1)

.text
p = malloc(200)
free(p)
.data
.got
{heap}
{libc}
malloc:
free:

Figure 2: Global Offset Table (GOT) 1



The Global Offset Table (GOT) (2)

```
Section Headers:
  ...
  [20] .got       0809ad94 000330
  ...

Relocation section contains 145
entries:
  Offset    Symbol's Name
  ...
  0809adf8  fork
● 0809ae64  malloc
● 0809af8c  free
  0809afa8  fopen
  0809afbc  socket
  ...
```

Figure 3: Global Offset Table (GOT) 2

Slapper wants to hijack the free() library function calls to run its own shell code on the remote machine.

3.0 The Buffer Overflow

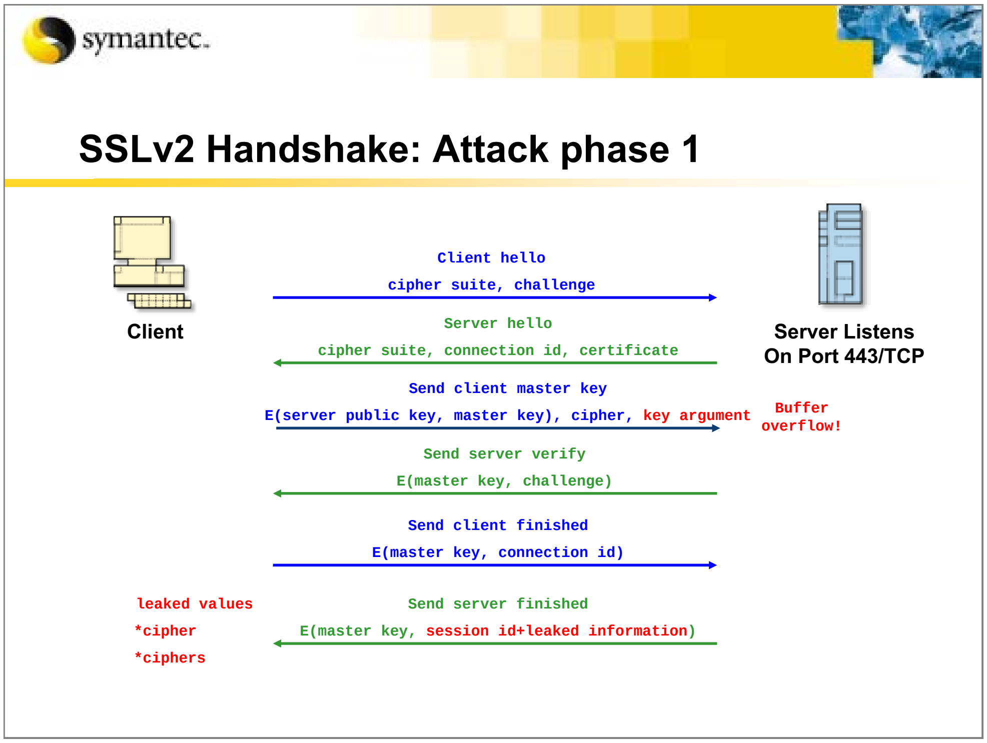
In the past, some worms have exploited stacked-based buffer overflows. Stack-based overflows are the low-hanging fruits compared to second-generation overflows, exploiting the heap structures. As the OpenSSL vulnerability affected a heap-allocated structure, the author of the worm encountered was not trivial and required a technically experienced individual.

When Apache is compiled and configured to use SSL, it listens on port 443.

The Slapper worm does the following:

- Opens a connection to this port and initiates an SSLv2 handshake.
- Sends a client “hello” message, advertising eight different ciphers (although, the worm sup ports only one; that is, the RC4 128-bit with MD5) and gets the server’s certificate in response.
- Sends the client master key and key argument, specifying a key argument length greater than the maximum allowed, SSL_MAX_KEY_ARG_LENGTH (8 bytes).

Refer to Figure 4 for an illustration of the "SSLv2 handshake: Attack phase 1."



symantec.

SSLv2 Handshake: Attack phase 1

Client

Server Listens On Port 443/TCP

Client hello
cipher suite, challenge

Server hello
cipher suite, connection id, certificate

Send client master key
E(server public key, master key), cipher, key argument

Buffer overflow!

Send server verify
E(master key, challenge)

Send client finished
E(master key, connection id)

Send server finished
E(master key, session id+leaked information)

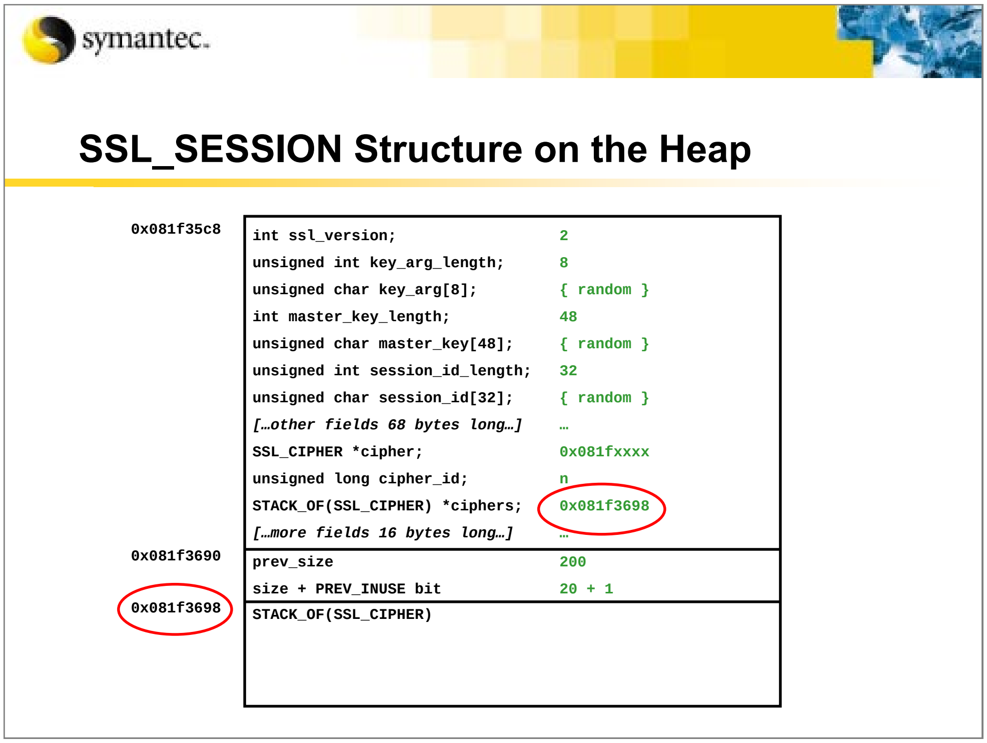
leaked values
*cipher
*ciphers

Figure 4: SSLv2 handshake: Attack phase 1

When the packet data is parsed in the get_client_master_key() function of libssl on the server, the code does not do a boundary check on the key argument length and copies the key argument from the packet to a fixed-length buffer key_arg[] of size SSL_MAX_KEY_ARG_LENGTH, in a heap-allocated SSL_SESSION structure.

As such, arbitrary bytes can overwrite any information following key_arg[]. This includes both the elements after key_arg[], in the SSL_SESSION structure, and the heap management data, which follows the memory block that contains the structure.

Refer to Figure 5 for an illustration of the “SSL_SESSION structure on the heap.”



SSL_SESSION Structure on the Heap

0x081f35c8	int ssl_version;	2
	unsigned int key_arg_length;	8
	unsigned char key_arg[8];	{ random }
	int master_key_length;	48
	unsigned char master_key[48];	{ random }
	unsigned int session_id_length;	32
	unsigned char session_id[32];	{ random }
	[…other fields 68 bytes long…]	…
	SSL_CIPHER *cipher;	0x081fxxxx
	unsigned long cipher_id;	n
	STACK_OF(SSL_CIPHER) *ciphers;	0x081f3698
	[…more fields 16 bytes long…]	…
0x081f3690	prev_size	200
	size + PREV_INUSE bit	20 + 1
0x081f3698	STACK_OF(SSL_CIPHER)	

Figure 5: SSL_SESSION structure on the heap

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 grossly affect the SSL handshake.

4.0 Double-Take

Interestingly, instead of using this overflow mechanism only once, the worm uses it twice:

- First, to locate the heap in the Apache process address space.
- Next, to inject its attack buffer and shell code.

There are two good reasons for splitting the exploit in two phases:

1. The attack buffer must contain the absolute address of the shell code, which is hardly predictable across all the servers, because the shell code is placed in memory that is dynamically allocated on the heap.

To overcome this problem, the worm causes the server to leak the address where the shell code will eventually reside, and then sends an attack buffer that is patched accordingly.

2. The exploit requires overwriting the cipher field of the SSL_SESSION structure located after the unchecked key_arg[] buffer. Refer to “Figure 5: SSL_SESSION structure on the heap” for an illustration.

This field identifies the cipher to use during the secure communication, and if its value were to become lost, the session would come to an abrupt end. So, the worm collects the value of this field during the first phase, and then re-injects it at the correct location within the SSL_SESSION structure, during the second phase.

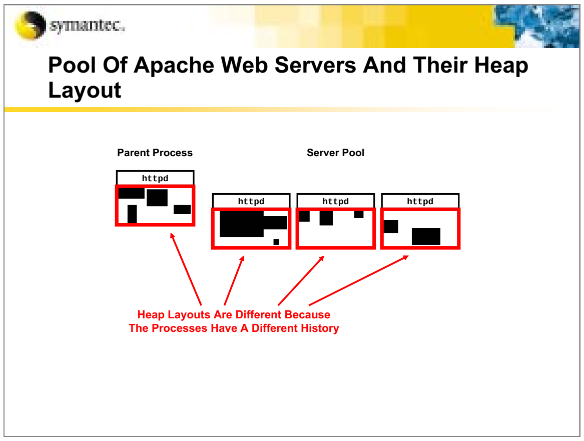
This two-phased approach requires two separate connections to the server and only succeeds because Apache 1.3 is a process-based server as opposed to a thread-based server.

The spawn of Apache that handles the two successive connections will inherit the same heap layout from their parent process. Therefore, all other things being equal, the structures allocated on the heap will end up at the same addresses during both connections.

This step assumes that Apache spawns two fresh “identical twin” processes to handle the two connections. However, under normal conditions, this may not always be the case, as Apache maintains a pool of running servers, which wait for the requests to handle.

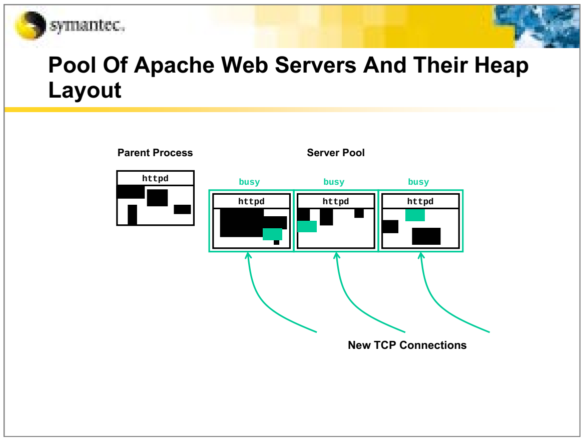
To force Apache to create two fresh processes, the worm exhausts Apache’s pool of servers before attacking, by opening a succession of 20 connections at 100 millisecond intervals.

Refer to Figure 6 for illustrated examples of the “Pool of Apache Web servers and their heap layout,” “Creation of a new server process,” and “Creation of multiple new server processes.”

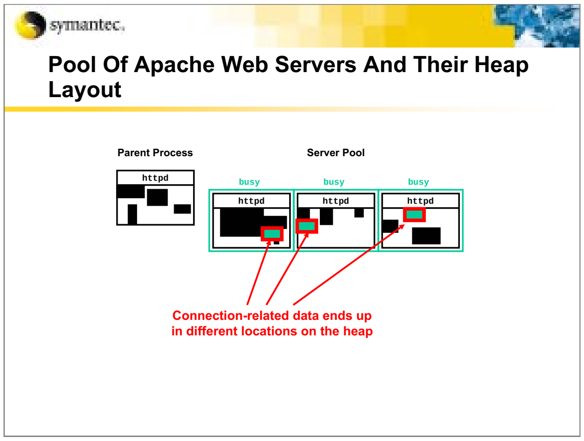


Pool Of Apache Web Servers And Their Heap Layout

Heap Layouts Are Different Because The Processes Have A Different History

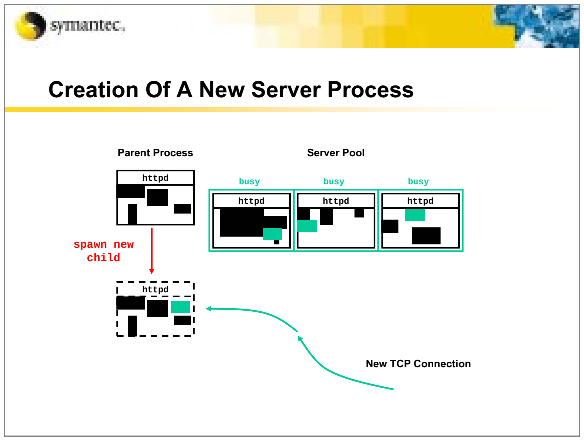


Pool Of Apache Web Servers And Their Heap Layout

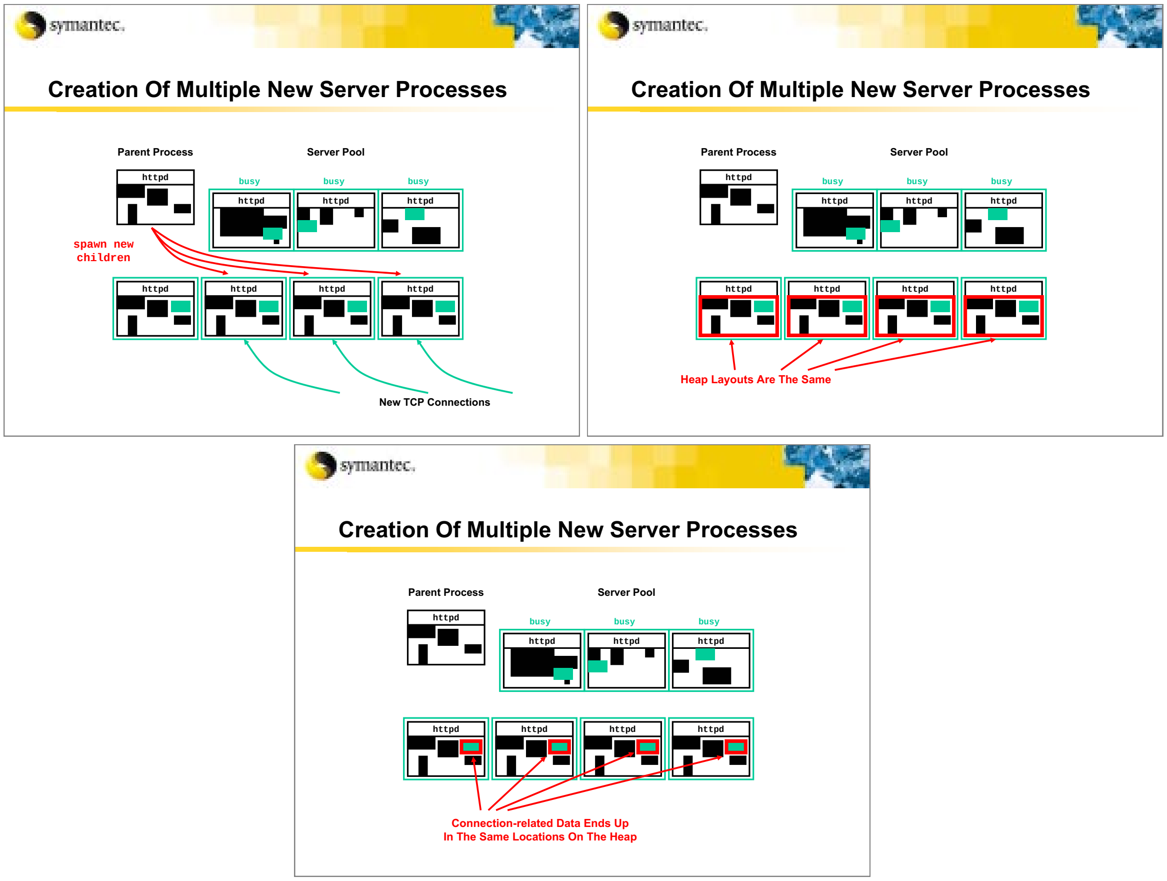


Pool Of Apache Web Servers And Their Heap Layout

Connection-related data ends up in different locations on the heap



Creation Of A New Server Process



Creation Of Multiple New Server Processes

Creation Of Multiple New Server Processes

Creation Of Multiple New Server Processes

symantec.
Parent Process
Server Pool
busy
httpd
Connection-related Data Ends Up In The Same Locations On The Heap

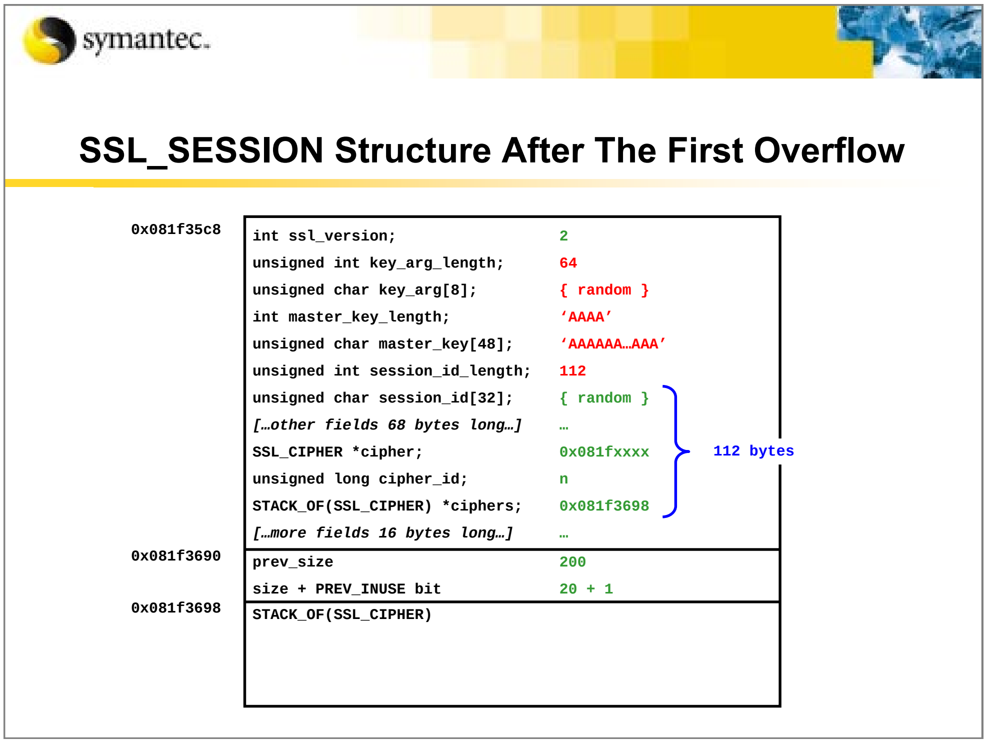
Figure 6: Pool of Apache Web servers and their heap layout, Creation of a new server process, Creation of multiple new server processes

5.0 Getting the Heap Address

The first use of the buffer overflow by the worm causes OpenSSL to reveal the location of the heap. The worm 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 field describes the length of the 32-bytes-long session_id[] buffer, which is the next field 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 that it wants to terminate the connection.

Refer to Figure 7 for an illustration of the “SSL_SESSION structure after the first overflow.”



SSL_SESSION Structure After The First Overflow

```
0x081f35c8   int ssl_version;                       2
             unsigned int key_arg_length;           64
             unsigned char key_arg[8];              { random }
             int master_key_length;                 ‘AAAA’
             unsigned char master_key[48];          ‘AAAAAA…AAA’
             unsigned int session_id_length;        112
             unsigned char session_id[32];          { random }  ⎫
             […other fields 68 bytes long…]         …           ⎪
             SSL_CIPHER *cipher;                    0x081fxxxx  ⎬ 112 bytes
             unsigned long cipher_id;               n           ⎪
             STACK_OF(SSL_CIPHER) *ciphers;         0x081f3698  ⎭
             […more fields 16 bytes long…]          …
0x081f3690   prev_size                              200
             size + PREV_INUSE bit                  20 + 1
0x081f3698   STACK_OF(SSL_CIPHER)
```

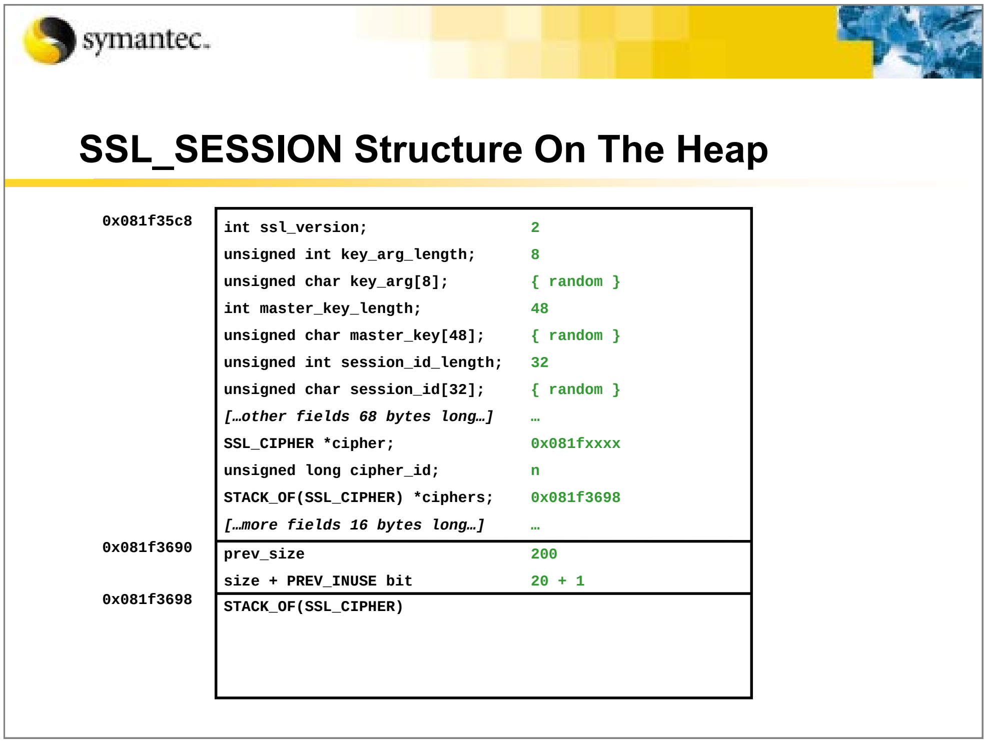
Figure 7: SSL_SESSION structure after the first overflow

Upon receiving the “client finished” message, the server replies with a “server finished” message, including the session_id[] data. Once again, boundary checking is not performed on the session_id_length, and the server sends not only the content of the session_id[] buffer, but the entire 112 bytes of the SSL_SESSION structure, starting at session_id[].

Among other things, a field called ciphers points to the structure allocated on the heap directly after the SSL_SESSION structure, where the shell code will go, as well as to a field called cipher, which identifies the encryption method to use.

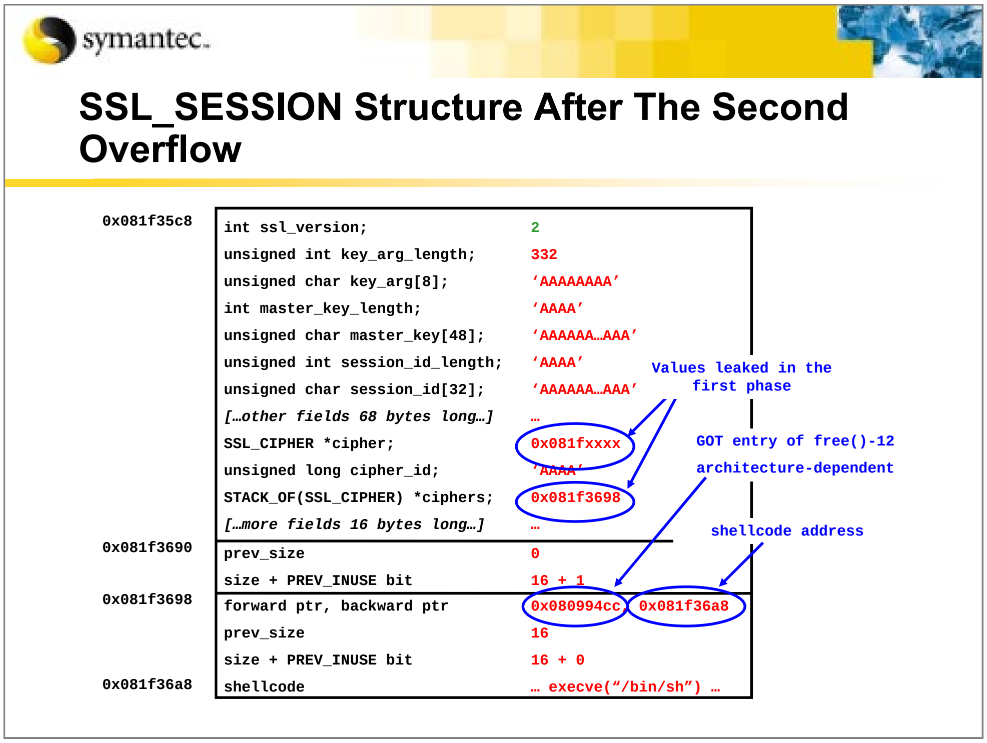
The worm extracts the two heap addresses from the session_id data received from the server and places them in its attack buffer. The TCP port of the attacker’s end of the connection is also patched into the attack buffer for the shell code to use later. Then, the worm performs the second SSL handshake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

Refer to Figure 8 for illustrations of the “SSL_SESSION structure on the heap” and “SSL_SESSION structure after the second overflow.”



SSL_SESSION Structure On The Heap

```
0x081f35c8   int ssl_version;                   2
             unsigned int key_arg_length;       8
             unsigned char key_arg[8];          { random }
             int master_key_length;             48
             unsigned char master_key[48];      { random }
             unsigned int session_id_length;    32
             unsigned char session_id[32];      { random }
             […other fields 68 bytes long…]     …
             SSL_CIPHER *cipher;                0x081fxxxx
             unsigned long cipher_id;           n
             STACK_OF(SSL_CIPHER) *ciphers;     0x081f3698
             […more fields 16 bytes long…]      …
0x081f3690   prev_size                          200
             size + PREV_INUSE bit              20 + 1
0x081f3698   STACK_OF(SSL_CIPHER)
```



SSL_SESSION Structure After The Second Overflow

```
0x081f35c8   int ssl_version;                   2
             unsigned int key_arg_length;       332
             unsigned char key_arg[8];          ‘AAAAAAAA’
             int master_key_length;             ‘AAAA’
             unsigned char master_key[48];      ‘AAAAAA…AAA’
             unsigned int session_id_length;    ‘AAAA’
             unsigned char session_id[32];      ‘AAAAAA…AAA’
             […other fields 68 bytes long…]     …
             SSL_CIPHER *cipher;                0x081fxxxx
             unsigned long cipher_id;           ‘AAAA’
             STACK_OF(SSL_CIPHER) *ciphers;     0x081f3698
             […more fields 16 bytes long…]      …
0x081f3690   prev_size                          0
             size + PREV_INUSE bit              16 + 1
0x081f3698   forward ptr, backward ptr          0x080994cc 0x081f36a8
             prev_size                          16
             size + PREV_INUSE bit              16 + 0
0x081f36a8   shellcode                          … execve(“/bin/sh”) …
```

Values leaked in the first phase
GOT entry of free()-12 architecture-dependent
shellcode address

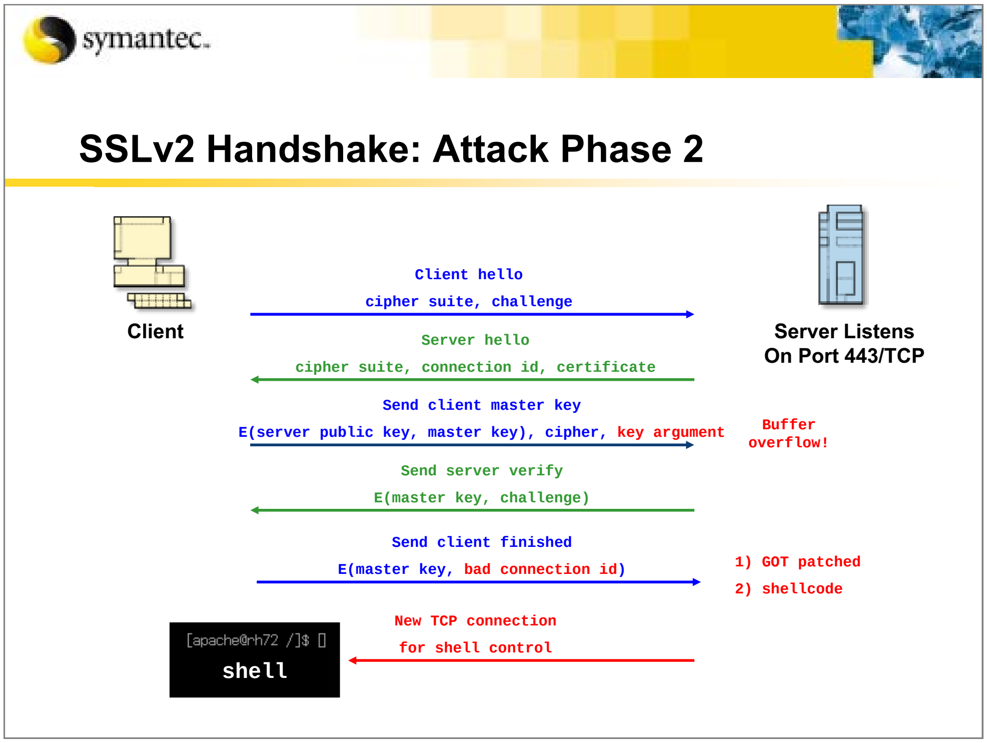
Figure 8: SSL_SESSION structure on the heap, SSL_SESSION structure after the second overflow

6.0 Abusing the Glibc

The second use of the buffer overflow is 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 will be the GOT entry of free() itself.
3. Causing free() to be called again, this time to redirect control to the shell code location.

Refer to Figure 9 for an illustration of the "SSLv2 Handshake: Attack phase 2."



symantec.

SSLv2 Handshake: Attack Phase 2

Client

Server Listens
On Port 443/TCP

Client hello
cipher suite, challenge

Server hello
cipher suite, connection id, certificate

Send client master key
E(server public key, master key), cipher, key argument

Buffer
overflow!

Send server verify
E(master key, challenge)

Send client finished
E(master key, bad connection id)

1) GOT patched
2) shellcode

New TCP connection
for shell control

[apache@rh72 /]$ ▯
shell

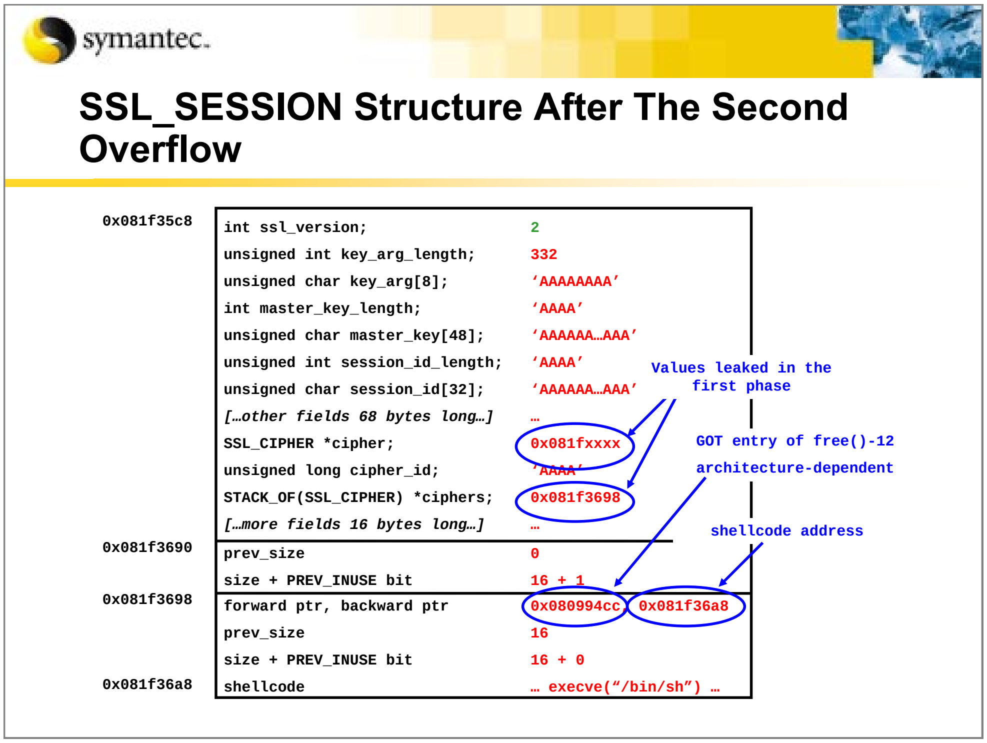
Figure 9: SSLv2 Handshake: Attack phase 2

The attack buffer used in the second overflow is composed of three parts:

1. The items to be placed in the SSL_SESSION structure after the key_arg[] buffer
2. 24 bytes of specially crafted data
3. 124 bytes of shell code

When the buffer overflow occurs, all the 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, with the exception of the cipher field. This field is restored to the same value that was leaked in the first phase.

Refer to Figure 10 for an illustration of the “SSL_SESSION structure after the second overflow.”



SSL_SESSION Structure After The Second Overflow

0x081f35c8	int ssl_version;	2
	unsigned int key_arg_length;	332
	unsigned char key_arg[8];	‘AAAAAAAA’
	int master_key_length;	‘AAAA’
	unsigned char master_key[48];	‘AAAAAA…AAA’
	unsigned int session_id_length;	‘AAAA’
	unsigned char session_id[32];	‘AAAAAA…AAA’
	[…other fields 68 bytes long…]	…
	SSL_CIPHER *cipher;	0x081fxxxx
	unsigned long cipher_id;	‘AAAA’
	STACK_OF(SSL_CIPHER) *ciphers;	0x081f3698
	[…more fields 16 bytes long…]	…
0x081f3690	prev_size	0
	size + PREV_INUSE bit	16 + 1
0x081f3698	forward ptr, backward ptr	0x080994cc 0x081f36a8
	prev_size	16
	size + PREV_INUSE bit	16 + 0
0x081f36a8	shellcode	… execve(“/bin/sh”) …

Values leaked in the first phase

GOT entry of free()-12

architecture-dependent

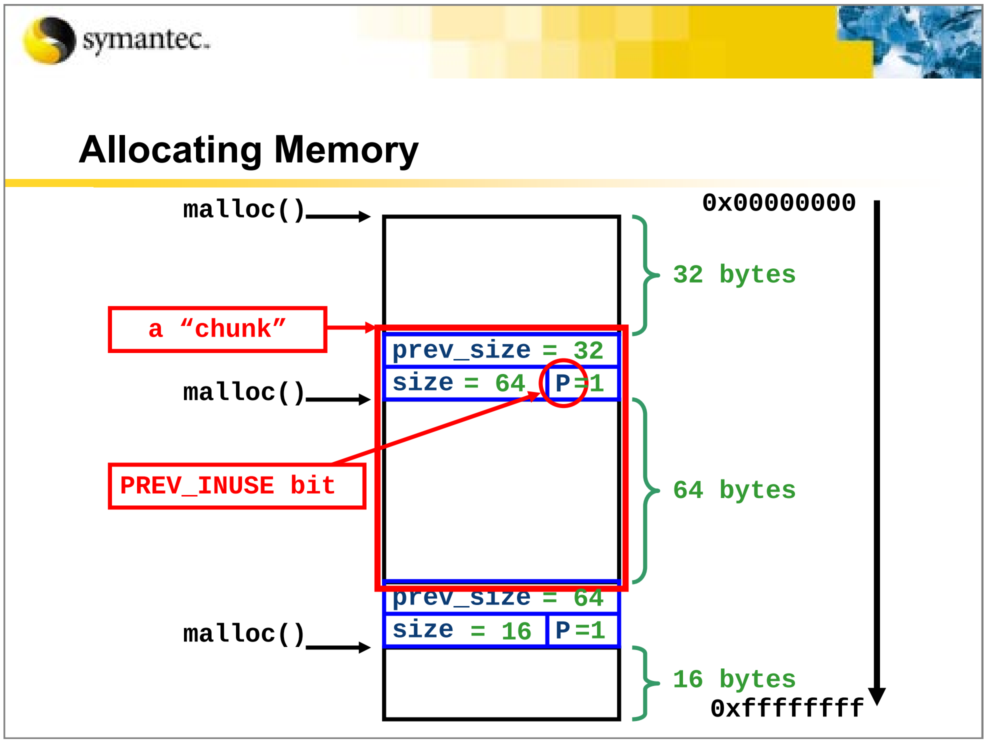
shellcode address

Figure 10: SSL_SESSION structure after the second overflow

Fake heap management data overwrites the 24 bytes of memory following the SSL_SESSION structure. (Refer to “Figure 10: SSL_SESSION structure after the second overflow” for an illustration.)

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).

Refer to Figure 11 for an illustration of memory allocation.

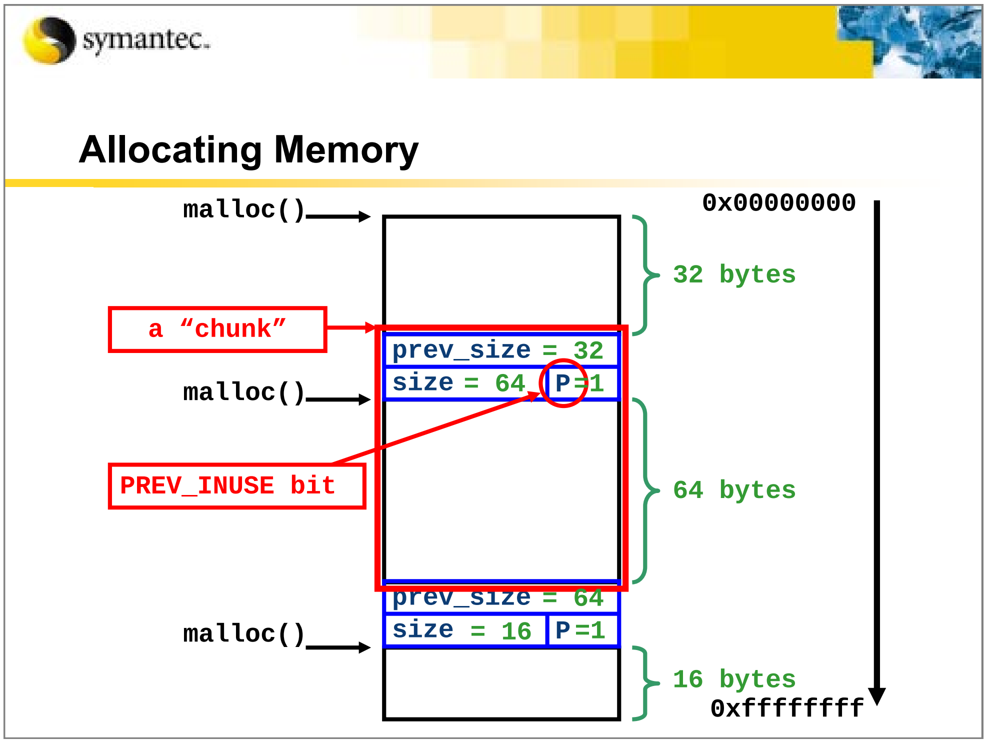


Allocating Memory

Figure 11: Allocating memory

Additionally, free blocks are kept in doubly linked lists formed by forward and backward pointers, which are maintained in the free blocks themselves.

Refer to Figure 12 for an illustration of “Freeing memory before a used block.”



Allocating Memory

malloc()

malloc()

malloc()

a “chunk”

PREV_INUSE bit

0x00000000

prev_size = 32

size = 64 | P=1

prev_size = 64

size = 16 | P=1

32 bytes

64 bytes

16 bytes

0xffffffff

Figure 12: Freeing memory before a used block

The fake heap management data injected by the worm after the SSL_SESSION structure poses a minimal-sized unallocated block, only containing the forward and backward pointers set respectively to the address of the GOT entry of free() minus 12, as well as the address of the shell code.

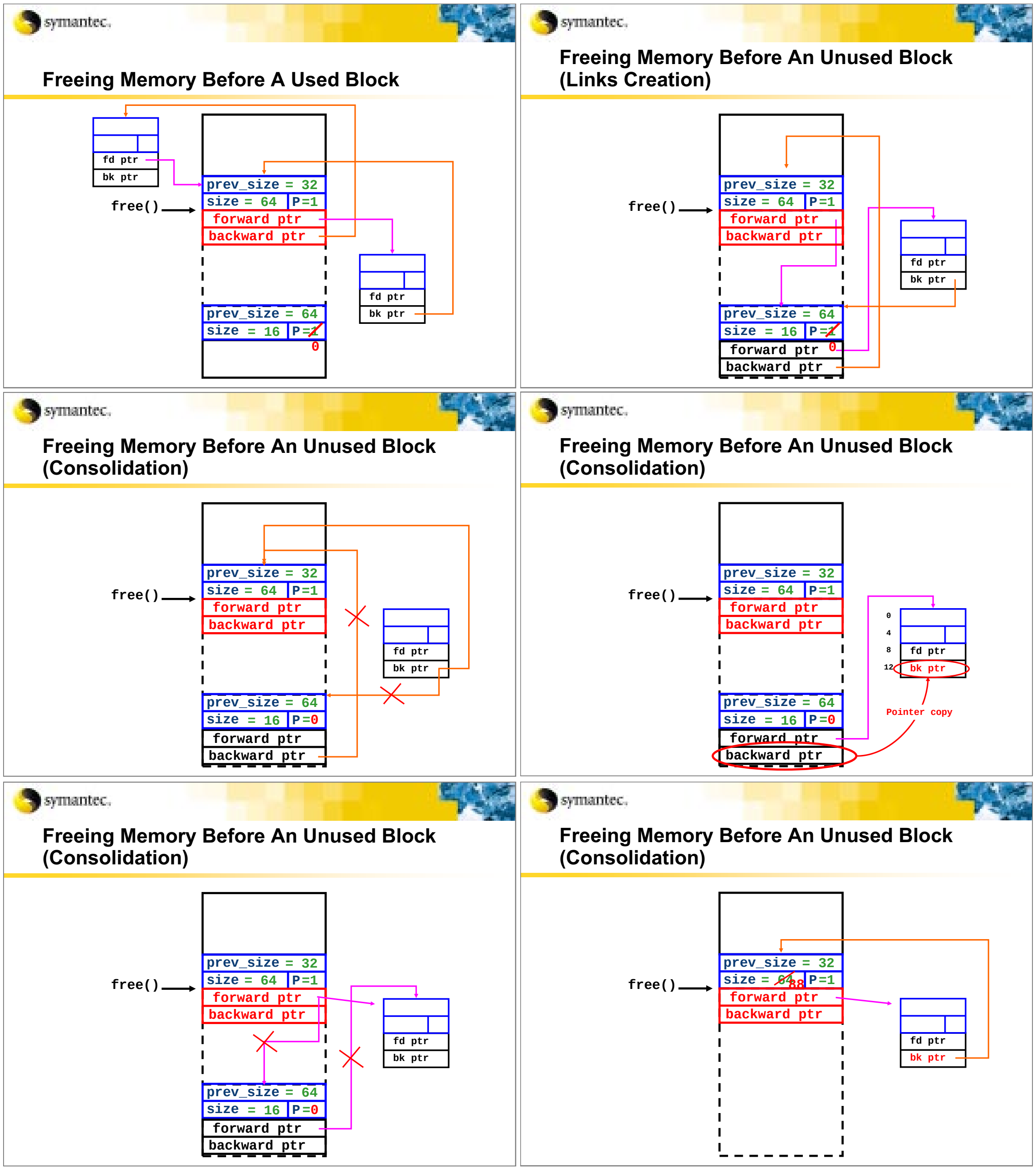
The address of the GOT entry is the “magic” value determined by fingerprinting. And, the address of the shell code is the value of the ciphers field leaked by OpenSSL in the first phase of the attack, plus 16 to account for the size of the fake block content and trailing boundary tag.

After the aforementioned conditions are 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, which in turn calls as an argument the glibc free() function with a pointer to the modified SSL_SESSION structure.

One may think that freeing memory is a simple task, but in fact, free() performs considerable bookkeeping when a memory block is released. Among other tasks, free() takes care of consolidating blocks; that is, 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. The operation trusts these to be pointing to heap memory, at least in the release build.

Refer to Figure 13 for illustrations on “Freeing memory before a used block” and “Freeing memory before an unused block.”



Freeing Memory Before A Used Block

Freeing Memory Before An Unused Block (Links Creation)

Freeing Memory Before An Unused Block (Consolidation)

Freeing Memory Before An Unused Block (Consolidation)

Freeing Memory Before An Unused Block (Consolidation)

Freeing Memory Before An Unused Block (Consolidation)

Figure 13: Freeing memory before a used block, Freeing memory before an unused block

The exploit takes advantage of the forward consolidation of the SSL_SESSION memory block, with the fake block created after it, by appropriately setting the PREV_IN_USE bits of the boundary tags. The forward pointer in the fake block, which points to the GOT, is treated as a pointer to a block header, de-referenced. Also, the value of the backward pointer (the shell code address) is written to offset 12 of the headers. Thus, the shell code address ends up in the GOT entry of free().

Note: The fake backward pointer is also de-referenced, so that the beginning of the shell code is treated as a block header as well. And, it is patched at offset 8 with the value of the fake forward pointer.

To avoid corrupting the shell code during this operation, the shell code will start with a short jump followed by 10 unused bytes filled with NOPs, so the shell code instructions are not corrupted during the consolidation.

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.

7.0 The Shell Code and Infection

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

Next, it attempts to gain root privileges by calling setresuid() with the 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, as setuid() is irreversible and contrary to the seteuid() function. The author of the shell code ostensibly overlooked this fact, however, the worm does not need root privileges to spread, because it only writes to the /tmp folder.

Finally, a standard shell “/bin/sh” is executed with an execve() system call. The worm issues a few shell commands to upload itself to the server in uu-encoded form, as well as to decode, compile, and execute itself. The recompilation of the source on the various platforms makes identifying the worm in binary form more difficult. The operations are performed in the /tmp folder, in which the worm files reside as the names, .uubugtraq, .bugtraq.c, and .bugtraq—notice the leading period marks, “.”, to hide the files from a simple “ls” command.

8.0 Now You See Me, Now You Don’t!

As the worm hijacks an SSL connection to send itself, a valid concern is whether it travels on the network in encrypted form. This issue is particularly crucial for authors of IDS systems who 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. Therefore, the attack buffer and the shell code are clear on the wire.

Later on, the same socket is used to transmit the shell commands, as well as the uu-encoded worm 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.

9.0 Peer-to Peer Network

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.

Note: 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 the hosts on the peer-to-peer network and broadcasts the address of the new instance worm to the network. Then, periodic updates to the host 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 the UDP and provides reliability through the use of checksums, sequence numbers, and acknowledgment packets. The code has been taken from an earlier tool and each worm instance acts as a Distributed Denial of Service (DDoS) agent and a backdoor.

10.0 Conclusion

Linux/Slapper is an interesting patchwork of a DDoS agent, with some functions taken straight from the OpenSSL source code and a shell code, which the author claims is not his own. This amalgamation results in a substantial amount of code, which is difficult to quickly comprehend.

Like FreeBSD/Scalper, most of the worm's code was probably previously written when the exploit became available. For the author, it was just a matter of integrating the exploit as an independent component.

And similar to Scalper, which exploited the BSD memcpy() implementation, the target of the exploit is not only an application, but rather a combination of an application and a run-time library underneath it. Perhaps one would expect memcpy() and free() to behave in a certain way, consistent with routine programming experience; however, when used in an unusual state or in passed invalid parameters, both memcpy() and free() behave erratically.

Linux/Slapper shows that Linux machines can become the target of worms that spread in the wild as easily as Windows machines do. For those with Slapper-infected Linux servers, it is a day to remember.

SYMANTEC, THE WORLD LEADER IN INTERNET SECURITY TECHNOLOGY, PROVIDES A BROAD RANGE OF CONTENT AND NETWORK SECURITY SOFTWARE AND APPLIANCE SOLUTIONS TO INDIVIDUALS, ENTERPRISES AND SERVICE PROVIDERS. THE COMPANY IS A LEADING PROVIDER OF VIRUS PROTECTION, FIREWALL AND VIRTUAL PRIVATE NETWORK, VULNERABILITY ASSESSMENT, INTRUSION PREVENTION, INTERNET CONTENT AND EMAIL FILTERING, AND REMOTE MANAGEMENT TECHNOLOGIES AND SECURITY SERVICES TO ENTERPRISES AND SERVICE PROVIDERS AROUND THE WORLD. SYMANTEC'S NORTON BRAND OF CONSUMER SECURITY PRODUCTS IS A LEADER IN WORLDWIDE RETAIL SALES AND INDUSTRY AWARDS. HEADQUARTERED IN CUPERTINO, CALIF., SYMANTEC HAS WORLDWIDE OPERATIONS IN 38 COUNTRIES.

FOR MORE INFORMATION, PLEASE VISIT WWW.SYMANTEC.COM

WORLD HEADQUARTERS

20330 Stevens Creek Blvd.
Cupertino, CA 95014 U.S.A.
408.517.8000
800.721.3934

www.symantec.com

For Product Information
In the U.S., call toll-free
800.745.6054

Symantec has worldwide
operations in 38 countries.
For specific country
offices and contact numbers
please visit our Web site.