Paper V: Attack on Sun's MIDP Reference Implementation of SSL

Attack on Sun's MIDP Reference Implementation of SSL

Kent Inge Simonsen, Vebjørn Moen, and Kjell Jørgen Hole

Abstract

Key generation on resource-constrained devices is a challenging task. This paper describes a proof-of-concept implementation of an attack on Sun's reference implementation of the Mobile Information Device Profile (MIDP). It is known that this implementation has a flaw in the generation of the premaster secret in SSL. The attack recovers the symmetric keys and plaintext from an SSL session.

1 Introduction

Running Java programs on resource-constrained devices like cellular phones and personal digital assistants require a specialized run-time environment. The Connected Limited Device Configuration (CLDC) [1] provides a set of Application Programming Interfaces (APIs) and a virtual machine for this environment. Together with a profile such as the Mobile Information Device Profile (MIDP) [2], it provides the possibility to develop Java applications to run on devices with limited memory, processing power, and graphical capabilities.

MIDP is a collection of APIs building on CLDC, providing some more advanced capabilities. Applications that comply with this standard are called MIDlets. Many companies have been involved in the development of MIDP, including Ericsson, NEC, Nokia, Palm Computing, Research In Motion (RIM), DoCoMo, LG TeleCom, Samsung, and Motorola.

MIDP has support for the Hyper Text Transfer Protocol (HTTP), where the information is sent in the clear, and secure HTTP, denoted HTTPS, which supports authentication, confidentiality, and integrity. The security of HTTPS is provided by Secure Socket Layer (SSL), or its successor Transport Layer Security (TLS).

As with many other cryptographic protocols, the security of SSL and TLS depends on generating secret key material. The randomness used in the process of generating the key material decides the strength of the resulting keys.

The first version of SSL in Netscape was shown to create key material using time [3] as input to a Pseudo-Random Number Generator (PRNG); this input is called a *seed*. Seeding with time is a common mistake, since it is difficult to get access to a good seed on a general purpose computer. Creating truly random numbers on a deterministic device such as a computer is impossible. We need to access a hardware source to get some randomness—strong sources of randomness include thermal noise and a radioactive decay source. Creating good random numbers in a constrained environment such as a cellular phone is
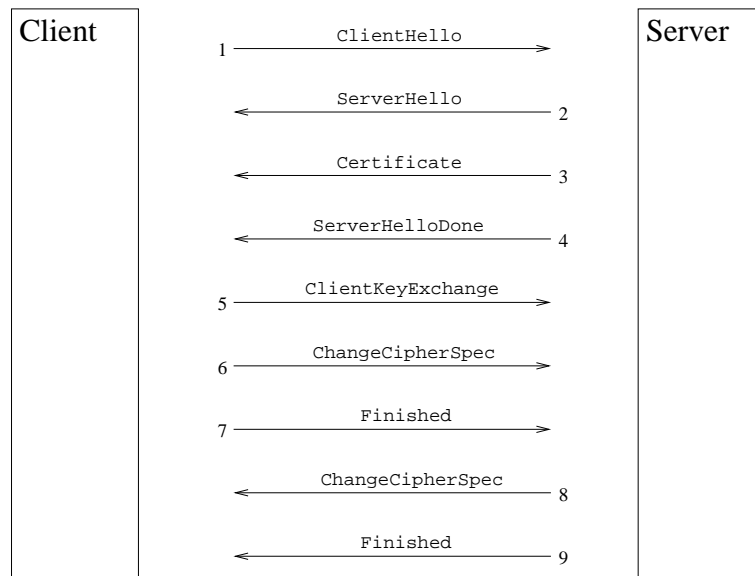
Figure 1: The 9 messages that SSL uses to establish an encrypted communication channel.

truly a challenge, but the security in SSL and most other crypto systems depend on a source for randomness.

It is known [4] that the reference implementation of MIDP provided by Sun has a flaw in the generation of the *premaster secret*, from which the message authentication and encryption keys in SSL are derived, due to seeding a PRNG with time. We describe an implementation of an attack on an SSL session between a server and a client using Sun's MIDP reference implementation which successfully recovers the SSL premaster secret, and consequently the authentication and encryption keys used in the SSL session.

In Section 2 we give a brief introduction to SSL, Section 3 considers randomness, Section 4 describes the attack on SSL in MIDP, as well as the implementation of the attack, and Section 5 concludes the paper.

2   SSL

This section is not meant to give a complete description of the SSL protocol; for a complete description [5] is recommended. We will consider the simplest case of SSL, namely establishing an encrypted communications channel.

The situation is that a client wants to establish a secure session with a server. To do this the client and server exchange SSL messages. Figure 1 shows the SSL handshake used to establish a share secret.

1. `ClientHello`: The client asks the server to begin the negotiation of the security services used by SSL. This message contains fields for a version number (3.0 for SSLv3 and 3.1 for TLS), and a 32-byte nonce used as seed in the generation of the premaster secret. The SSL specification suggests that 4 of these 32 bytes contain the time and date to avoid client reuse

of this 32-byte random number. A session ID to identify the specific SSL session, a list of cryptographic primitives that the client can support, and some more fields not mentioned here are also a part of the `ClientHello`.

2. `ServerHello`: The server responds to the `ClientHello`. This message contains fields for a version number, a 32-byte nonce where 4 bytes are used for time and date, a session ID number, a CipherSuite field which determines the cryptographic parameters, such as algorithms and key sizes. The `ServerHello` also contains some more fields not discussed here.

3. `Certificate`: The server sends a certificate containing the public key information.

4. `ServerHelloDone`: Tells the client that the server is finished with the initial negotiation messages.

5. `ClientKeyExchange`: The client generates the premaster secret, encrypts it with the public key received in the server certificate and sends the result to the server.

6. `ChangeCipherSpec`: This message tells the server that from now on any message received from the client will be encrypted with the agreed algorithm and key.

7. `Finished`: This message from the client to the server allows the server to verify that the negotiation has been successful. It contains a hash of key information, and contents of all previous SSL handshake messages exchanged by the client and server. Also notice that this message is encrypted.

8. `ChangeCipherSpec`: This message tells the client that from now on all messages from the server will be encrypted.

9. `Finished`: The client can now verify that the SSL negotiation has been successful. Just as for the finished message from the client it contains a hash of key information, and contents of all previous SSL handshake messages, and it is also encrypted.

After finishing the above protocol the client and the server share symmetric keys for message authentication and encryption, and using the certificate received from the server in message 3 the client can verify that it is talking to the correct server. Note however that the described SSL negotiation does not allow the server to authenticate the client. Observe also that "`Finished`" messages can be used by the server and the client to verify that the other part has the correct key.

3   Randomness and PRNGs

The security of SSL rests on the infeasibility of testing all possible keys used for encryption. If the key space is too large, then the brute-force attack will take too much time. But if an attacker can reduce the number of keys to be tested, she might be able to crack the key.

Many applications use easily available sources of randomness to create an initial value, or seed. This seed is then used as input to a PRNG. The PRNG expands the seed into a longer, random-looking bit stream. For a non-security application the seed only needs to change every time the program runs, but when we use it to generate cryptographic keys, the seed also needs to be as unpredictable and unguessable as the key itself for an attacker.

Consider a system using 128-bit keys. A brute-force attack on such a system would need to check on average 2^{127} keys, which is a huge number and clearly infeasible on a modern computer. What happens if these 128 bits are generated with a PRNG? Assuming that all the details about the PRNG are known to the attacker, the security of the cryptographic key now depends upon the seed. In other words, the number of possible seeds gives the number of possible cryptographic keys. If the PRNG is seeded with milliseconds since midnight, January 1, 1970 in the GMT timezone, and the attacker knows which year the seed is created, she only needs to check $365 \cdot 24 \cdot 3,600 \cdot 1,000 = 31,536,000,000 \approx 2^{35}$ different keys, which is a relatively small task for a modern computer.

Using PRNGs to create cryptographic keys requires that there exists at least as many equally likely seeds as possible keys, to avoid that the PRNG reduces the effective key length.

3.1   Creating a seed

The seed is essential for the security of the system. RFC 4086 [6] gives some recommendations for security in randomness. Essentially there are two strategies: either use a reliable hardware source of randomness or use a mixing function to combine several more or less random inputs to create a "pool" of random data, e.g. Yarrow [7] and */dev/random* in GNU/Linux.

Radioactivity decay source, Gaussian white noise and spinning disks [6, 8] are all examples of hardware sources of randomness. A small addition in hardware and software to access these sources, could solve the seed problem.

The */dev/random* in GNU/Linux is an RNG which collects environmental noise from devices and other sources into an entropy pool, and keeps an estimate of the number of available bits in the entropy pool. When random numbers are requested they are created from the pool. Gutmann [9] describes some pratical solutions of how to create random numbers for use in cryptographical protocols and for key material.

4   The Attack

The source code for Sun's reference implementation of MIDP is available for download from Sun, but it does not contain the source code for SSL and the PRNG. By decompiling the SSL.jar which comes with the compiled version of MIDP we obtained the Java byte code, and from that we discovered how the seeding of the PRNG is implemented.

The PRNG is seeded with the current time in milliseconds and 16 static bytes. The PRNG also allows manual seeding, but this is not used in the reference implementation. First, we give a brief overview of how the PRNG works and what the idea of the attack is, then more details are given in the remainder of the section.

The PRNG uses the MD5 hash function to mix input and the current state, and it is reseeded with current time and the previous seed for each block of data that is generated. The entire MD5 output is used, which gives a block size of 16 bytes.

During the SSL handshake a PRNG object is constructed on the client. The PRNG object generates a 32-byte nonce sent in the clear, as well as a 48-byte premaster secret which is sent encrypted. The first two bytes of the 48 bytes used for the premaster secret are discarded to make room for some version information.

The PRNG is seeded 5 times with time in milliseconds, and one can be certain that all the time seeds come in proximity of each other. Since the nonces are sent in the clear, it seems reasonable to split the process in two parts. First, the time seeds used to create the client nonce are found so that we can synchronize our clock with the clock on the device, and then we guess the next three time seeds that lead to the premaster secret.

For each suggestion for the premaster secret we need to generate the encryption/decryption and message authentication keys, decrypt a package and check the Message Authentication Code (MAC) value.

4.1   The PRNG

The handshake procedure uses the same PRNG object to create the nonce and the premaster secret. The pseudo code version of the decompiled PRNG from Sun's reference implementation of MIDP is shown in Figure 2.

When the PRNG is constructed it initializes the MD5 digest and the `update-Seed()` method is called, where a time seed together with a constant are used to create the first state. The `updateSeed()` method feeds the current state and the current time in milliseconds in that order and calls the `doFinal()` method whose output is the next state. The digest is reset after every `doFinal()`.

The `generateData()` method writes the pseudo random data to an array (which it takes as an argument). When it runs out of random data, every 16 bytes, it digests the current state and calls the `updateSeed()` method. The data resulting from hashing the current state is said to be the pseudo random data, and is written to the array until it is full, or more data is needed. Note that `randomBytes` is a global array.

The generation of the nonce and premaster in the MIDP SSL is illustrated in Figure 3. The client generates 5 different 16-byte values with this PRNG, the first two outputs are used for the known nonce and the three next outputs are used for the unknown premaster secret. To generate the first 16-byte, a 16-byte constant and current time in milliseconds are hashed and the output is the first state, which again is hashed to yield the first 16-byte of output. At the same time the state and current time in milliseconds are digested and the output is the next state. The next four outputs needed to create the nonce and premaster secret, are generated in a similar manner; digest the state to get the output, and digest the state together with current time to get the next state.

4.2   The attack step-by-step

1. Sniff an SSL session and record the starting time.

```
constructor() {
  initialize digest;
  updateSeed();
}
updateSeed() {
  digest.update(seed);
  digest.update(currentTimeMillis);
  seed = digest.doFinal();
}
generateData(byte[] buf, int off, int len) {
  int i = 0;
  int byteAvailable = 16;
  while(true) {
    if(bytesAvailable == 0)  {
      randomBytes = digest.doFinal(seed);
      updateSeed();
      bytesAvailable = 16;
    }
    while(bytesAvailable > 0) {
      if (i == len)
        return;
      buf[off+i] = randomBytes[--bytesAvailable];
      i++;
    }
  }
}
```

Figure 2: Pseudo code of the PRNG from Sun's reference implementation of MIDP.
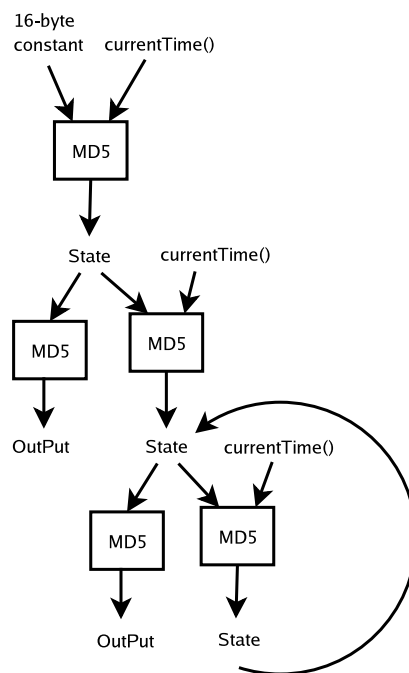
Figure 3: How MD5 is utilized to generate the pseudo random data used for nonce and premaster in the reference implementation of MIDP SSL. The 16-byte constant is known from the decompiled Java byte code.

2. Retrieve the client nonce and the server nonce. These are sent in the clear in the `ClientHello` and `ServerHello` messages.

3. Decide the start and stop time, i.e., in which time interval did the client seed the PRNG.

4. Since the client nonce is sent in clear, we know the first and second output of the PRNG. Find the value between start and stop time that was used to create the first 16 bytes of the client nonce by trying all possible values.

5. When the time seed that were used to generated the first 16 bytes is found, the PRNG can be set in the correct state. Then try all possible time seeds from the start time until the stop time, until the next 16 bytes of the nonce is found.

6. We now know exactly when the client's nonce was created according to the clients internal clock. Using this information we try to find the premaster secret which the client generates a short time after creating the nonce. Exactly how short this time is, is determined by the client device, its load, the speed of the network connection and many such factors. The amount of uncertainty about the time period in which the premaster secret is generated affects the complexity of the search for the premaster secret. Use the time seeds found in step 4 and 5 to set the state of the PRNG, then generate all possible values for the next three time seeds. Then use the suggested values together with the client nonce and server nonce to generate a candidate for the premaster secret and check if it is correct.

```
for each t1 in time interval
  for each t2 in time interval ≥ t1
    for each t3 in time interval ≥ t2
      premaster = generatePreMasterCandidate(
                    PRNG_state,t1,t2,t3)
      check(premaster)
```

4.3   Checking the premaster

There are several approaches to check if the suggested premaster secret is correct. One good suggestion is to create the keys used in SSL (encryption and message authentication keys) based on the premaster secret. Then we decrypt a package and attempt to verify the MAC. If the MAC verifies, we have a suggestion for the premaster secret. Any false positives can be eliminated by using more packets and MACs.

One other method is to use the `Finished` packets in the SSL handshake protocol, which contain a hash of the key material together with other known data. Yet another method could be a known plaintext attack on an SSL connection.

4.4   Time complexity

Given a start time t_{start} and finished time t_{stop} then $\Delta t = t_{stop} - t_{start}$ denotes how many milliseconds the SSL handshake takes on the device we are attacking. Using the client nonce and guessing the first time seed of the PRNG takes $\mathcal{O}(\Delta t)$ time, and guessing the second time seed also takes $\mathcal{O}(\Delta t)$ time. Notice

that this step allows us to synchronize with the device, i.e., we know the exact time on the device, which gives us an exact $\widehat{t}_{start}, \widehat{t}_{stop}$ for the generation and $\Delta\widehat{t} = \widehat{t}_{stop} - \widehat{t}_{start}$.

We need to guess three time seeds to generate a suggestion for the premaster secret, which have time complexity $\mathcal{O}\left(\left(\Delta\widehat{t}\right)^3\right)$. However, since the time seeds are generated sequentially with approximately the same amount of work between each generation, it is possible to implement the attack so that it divides $\Delta\widehat{t}$ into three time-slots and searches the first time-slot for the first time seed, and so on... Estimated time complexity for the search for the premaster secret is $\mathcal{O}\left(\left(1/3 \cdot \Delta\widehat{t}\right)^3\right) = 1/27 \cdot \mathcal{O}\left(\left(\Delta\widehat{t}\right)^3\right)$. Resulting in a total time complexity of:

$$2 \cdot \mathcal{O}\left(\Delta t\right) + \frac{1}{27} \cdot \mathcal{O}\left(\left(\Delta\widehat{t}\right)^3\right).$$

4.5 Implementation

The attack was tested with a simple SSL client MIDlet written in J2ME and a simple SSL server implemented in J2SE. We used Ethereal [10] to sniff the traffic between the two programs and recover one encrypted SSL package. The attack code guessed keys and decrypted the package and checked the MAC value, utilizing methods from TinySSL [11] for key generation, decryption and MAC calculation.

The MIDlet first ran on a Nokia 6600 and a SonyEricsson P900 over GPRS. However, we were unable to recover the time from the client nonce, which led to the conclusion that these phones do not use the same implementation of the PRNG as Sun's reference implementation.

The same MIDlet was then tested on the emulator in Sun J2ME Wireless Toolkit 2.1 over the loop back interface, where the attack successfully recovered the shared premaster secret.

4.5.1 How long to find the keys?

On average an SSL handshake took approximately 20–30 seconds over GPRS with both the SonyEricsson P900 and the Nokia 6600; the timings include the time it took to enter user input requested by the phones during an SSL connection.

When we tested the attack on the emulator we measured the handshake to take less than 200 milliseconds. To further simulate a proper phone we used $\Delta t = 40s$, and recovered the premaster secret in less than a second on a laptop with an Intel Pentium M processor running at 1600MHz. It is likely that the attack on an SSL connection between a real phone and a server will take more time, since all the seeds to the PRNG were created within 25 milliseconds on the emulator.

5 Conclusion

We have shown that Sun's reference implementation of SSL in MIDP is vulnerable to a key recovery attack because of a bad choice of seed to the PRNG. There

is a solution to this problem: find a better seed. However, this might prove difficult to implement on the software layer of resource constrained devices and the manufacturers of these devices should make hardware randomness available for software developers.

It is also unclear whether or not the developers of mobile phones have solved the problem with cryptographic randomness, history have shown how easy it is to do the generation of random data in an insecure manner.

References

[1] JSR 139 Expert Group, *Connected Limited Device Configuration, Version 1.1*. Sun microsystems, 2003.

[2] JSR 118 Expert Group, *Mobile Information Device Profile for Java 2 Micro Edition*. Java Community Process, 2002.

[3] I. Goldberg and D. Wagner, "Randomness and the Netscape browser," *Dr. Dobb's Journal*, pp. 66–70, January 1996.

[4] D. Povey, "Wireless Java security," Java Developer's Journal, last visited: May 26, 2006. [Online]. Available: http://java.sys-con.com/read/37377.htm

[5] S. Thomas, *SSL and TLS Essentials: Securing the Web*. Wiley Computer Publishing, 2000.

[6] D. Eastlake, J. Schiller, and S. Crocker, "Randomness Requirements for Security," RFC 4086 (Best Current Practice), June 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4086.txt

[7] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," in *Selected Areas in Cryptography*, no. Generators, 1999, pp. 13–33. [Online]. Available: citeseer.ist.psu.edu/kelsey99yarrow.html

[8] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic randomness from air turbulence in disk drives," in *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*. London, UK: Springer-Verlag, 1994, pp. 114–120.

[9] P. Gutmann, "Software generation of practical strong random numbers," in *Proceedings of the Seventh USENIX Security Symposium*, 1998, pp. 243–257.

[10] Ethereal network protocol analyzer, last visited: June 12th, 2006. [Online]. Available: http://www.ethereal.com/

[11] TinySSL, last visited: June 12th, 2006. [Online]. Available: http://www.xwt.org/javadoc/org/xwt/TinySSL.html