

J2ME Security



Thesis for the degree Master of Science

Kent Inge Fagerland Simonsen

Department of Informatics

University of Bergen

May 2005

The NoWires Research Group



<http://www.nowires.org/>

Preface

This thesis provides an introduction to Java 2 Micro Edition (J2ME) and in particular the security mechanisms in J2ME. The thesis also gives an understanding of the intricacies involved in creating a good random seed needed to encrypt data. Finally it provides an example of why good random seeds are important, and propose how to create such seeds using J2ME.

Acknowledgments

As with most works requiring a certain amount of effort this thesis could not have been completed, on time or otherwise, without the help of others. These people, even if they are not mentioned below due to a terrible oversight on my part, should consider themselves thanked.

I thank my supervisor Professor Kjell Jørgen Hole for the opportunity of working on this thesis, his support, guidance, and nitpicking on my seemingly endless stream of bad language constructs and misspellings.

I would also like to thank Vebjørn Moen for his help and cooperation on our paper dealing with an attack on a poor implementation of SSL which has become Chapter 5 in this thesis. Without his help and encouragement it would be little more than a footnote.

My gratitude goes out to every member of the Nowires Research Group, and the Selmer Center as a whole. Our discussions on all sorts of topics have been both useful and recreational.

Many of my fellow students have helped way to numerous to be mentioned here.

My friends and family have also been supporting and helpful. I want to especially thank my father for taking the time to proof read large portions of this thesis in the final days of its writing.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Issues covered	2
1.3	Work done	2
1.4	Difference from other works	3
1.5	The chapters	3
2	Introduction to Java 2 Micro Edition and Java Security	5
2.1	What is Java 2 Micro Edition?	5
2.2	Configurations	5
2.2.1	CLDC	6
2.2.2	CDC	7
2.3	Profiles	8
2.3.1	Mobile Information Device Profile	8
2.3.2	Foundation and Personal Profiles	8
2.4	J2SE security basics	9
2.4.1	The Java Sandbox	10
2.4.2	Memory Integrity	10
2.4.3	The bytecode verifier	10
3	Security Mechanisms in CLDC and MIDP	11
3.1	CLDC mechanisms	11
3.1.1	The life cycle of a CLDC application	12
3.1.2	The CLDC sandbox	12
3.1.3	The KVM byte code verifier	13
3.2	MIDP mechanisms	14
3.2.1	The MIDP sandbox	14
3.2.2	The Record Management System	15
3.2.3	SSL/TLS	15
3.3	Thoughts on CLDC and MIDP security	16

4	Random Numbers in J2ME	17
4.1	Why random numbers?	17
4.2	The dangers of <code>now()</code>	18
4.3	Randomness in J2ME	18
4.4	An audio based random number generating MIDlet	19
4.5	Attacks on this approach	25
4.6	Results from statistical analysis	25
4.7	Conclusion	27
5	Attack on Sun's MIDP Reference Implementation of SSL	29
5.1	Introduction	29
5.2	SSL	30
5.3	Randomness and PRNGs	32
5.3.1	Creating a seed	33
5.4	The Attack	33
5.4.1	The PRNG	34
5.4.2	The attack step-by-step	36
5.4.3	Checking the premaster	37
5.4.4	Time complexity	37
5.4.5	Implementation	38
5.5	Conclusion	39
6	Conclusions	41
6.1	Summary	41
6.2	Conclusions	41
6.3	Further work	42
A	A Portscanning MIDlet	43

List of Figures

2.1	How J2ME technologies relate to each other.	6
2.2	Using the Generic Connection Framework.	7
3.1	The operations CLDC code must undergo before it is allowed to execute.	13
5.1	The 9 messages that SSL uses to establish an encrypted communication channel.	31
5.2	Pseudo code of the PRNG from Sun's reference implementation of MIDP.	35
5.3	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.	36

Chapter 1

Introduction

1.1 Motivation

As of this writing Sun Microsystems lists 36 licensees [12] of their Java 2 Micro Edition (J2ME) technologies. A search for "j2me" on Google gives 1.5 million hits. And out of the 99329 projects registered on SourceForge.net [31], 218 are using or are somehow related to J2ME technology. And according to [9] more than 350 million Java enabled mobile devices have been made. So there should be little doubt that J2ME is a widespread technology with a large potential user base.

The most popular J2ME applications so far have probably been games. For a primary gaming platform many security issues are not as important as they are for business applications. This is particularly true for communications security. It can be extremely annoying, but in most cases not mission critical if someone cheats by submitting false high-score data or intercepts high-score data before it reaches the server. Still it may be very important for a gaming platform to be able to handle malicious code dressed up like a fun game in a secure manner and to stop such code from inflicting any damage to a pricey device. J2ME addresses this with a set of mechanisms for language security in much the same way Java 2 Standard Edition (J2SE) does.

One would expect to see more and more business and e-commerce applications emerge, which will create a larger need for security. In particular, both communication and data security are very important in this realm. Language security is no longer as important because business users can often be instructed not to put unknown (possibly malicious) software on their devices.

The growing market for J2ME programs makes it interesting to write about the underlying security mechanisms of J2ME and to look at problem

areas. One of the most fundamental and difficult things to do in a security system that utilizes cryptographic techniques is key generation which is in essence the same thing as generating good, truly random numbers.

Security in itself is a rather ambiguous word. Security is not an entity, and no language mechanisms can protect against a poor application programmer. However, how to write good application code falls outside the scope of this thesis. Instead, this thesis will look at some of the mechanisms embedded in parts of J2ME.

1.2 Issues covered

The aim of this thesis is to give the reader a rough understanding of the various security mechanisms in J2ME. It will attempt to point out some trouble areas concerning the security mechanisms in some popular J2ME technologies. The thesis will also try to find out what is needed to allow J2ME to take the big step into a secure platform for business applications. It will also discuss in some detail the problem of creating good random numbers on mobile devices.

Security will be used in this thesis to mean protection of the device against potentially harmful or disruptive code and confidentiality and integrity of data that is transmitted over an insecure channel. This thesis will not look at authentication or confidentiality/integrity of data that is stored on the device as no such mechanisms are in widespread use in J2ME.

The reader is expected to have a fair understanding of Java technologies and a good understanding of security related issues. The reader will also benefit from having a fair knowledge of cryptology and its applications.

1.3 Work done

This thesis will give an introduction to some important J2ME technologies, and Java security mechanisms. It will then try to evaluate whether J2ME technologies are ready for production use in business and commerce applications from a security standpoint. The thesis will look at the mechanisms that are embedded in some popular technologies, mainly Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP). The goal is to find out whether they are good enough for use. The thesis will also look at a key component of any cryptographic approach, namely random number generation, and show that the approach chosen for the reference implementation of MIDP 2.0 is clearly inadequate.

1.4 Difference from other works

Other works on J2ME security, such as [39] and [5], tend to focus on a goal and show how it can be done programatically. This thesis will mainly focus on the technologies that are used in a more general manner.

1.5 The chapters

Chapter 2 gives a brief overview of what J2ME is and some of its most popular parts. This includes a quick view of the most used technologies in J2ME and how they relate to each other. It also contains a very brief overview over some J2SE security mechanisms that are important for the rest of the thesis.

Chapter 3 describes the security mechanisms found in a popular variant of J2ME which utilizes the CLDC configuration and MIDP profile. It will look at how the device is protected from malicious code and how confidentiality and integrity can be conserved in communication.

Chapter 4 discusses the need for a good way to generate random numbers and proposes a way to do this. It also gives some analysis of the data generated using this approach.

Chapter 5 is a paper that was written together with Vebjørn Moen and Kjell Jørgen Hole. The paper describes an implementation of an attack on the Secure Socket Layer implementation in the reference implementation of MIDP 2.0. The paper has been submitted to the ACM Workshop on Wireless Security (WiSE 2005), Cologne, Germany, August 28—September 2, 2005.

Chapter 6 contains some conclusions concerning the topics discussed in this thesis.

Chapter 2

Introduction to Java 2 Micro Edition and Java Security

2.1 What is Java 2 Micro Edition?

Java 2 Micro Edition (J2ME) is a collection of specifications that defines ways to run a subset of Java 2 Standard Edition (J2SE) on a myriad of devices. These devices do not have the resources required to run J2SE. Since the devices are so unlike each other,¹ a different subset of J2SE must exist to accommodate each type of device. This is accomplished by splitting the specifications up into different configurations, profiles, and optional packages. Figure 2.1 shows how the most common configurations and profiles relate to each other.

2.2 Configurations

A configuration in J2ME provides an entire runtime environment for Java applications and supports a given subset of the features of J2SE. A configuration specifies its target devices in terms of a set of required capabilities such as memory requirements. Although a configuration is able to run several applications it is not necessarily very useful in itself since it is meant to give a least common denominator of features of an entire class of devices. To be useful it is often necessary to put a profile on top of the configuration. Profiles may put additional requirements on the device and can offer more services to the application layer.

¹A low end cellular phone can not be expected to do everything a high end Personal Digital Assistant (PDA) can do.

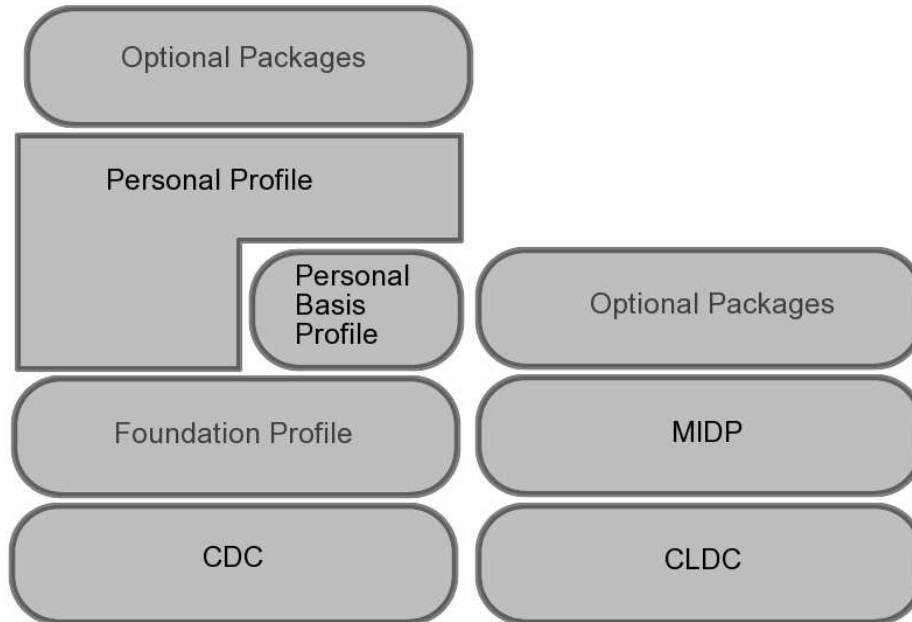


Figure 2.1: How J2ME technologies relate to each other.

Currently there are two configurations available for J2ME. These are the Connected Limited Device Configuration (CLDC)[14] and the Connected Device Configuration (CDC) [15].

2.2.1 CLDC

CLDC is a configuration for “limited” devices such as mobile phones, low end PDAs and home appliances. The characteristics of the target devices are:

- At least 160KB for CLDC 1.0 and 192KB for CLDC 1.1.
- A 16 or 32 bit processor.
- Low power consumption.
- Connection to a network with limited capabilities and reliability.

CLDC provides a minimal set of Application Program Interfaces (APIs) to the applications running on it. These include several packages that are

derived from J2SE such as system classes, Input/Output (I/O) classes, Exception and Error classes etc. CLDC also defines its own API for I/O and networking called the *Generic Connection Framework* (GCF). This is done because the I/O and networking APIs for J2SE are too large for CLDC and not really adapted to the multitude of link types that are available in CLDC compliant devices. The GCF provides more flexibility than the J2SE I/O libraries and is easy to map to any underlying implementation of the I/O services that are supported. It uses a controller class that takes a string argument on the form as shown in Figure 2.2.

```
Connector.Open("<protocol>:address;<parameters>")
```

Figure 2.2: Using the Generic Connection Framework.

This call should return an instance of the `Connection` interface which is suited to the type of link mandated by the protocol. The framework supports both stream and datagram (packet) based connections. Appendix A gives an example of use of GCF. CLDC does not provide or demand any specific protocols to be implemented.

CLDC applications can run on any virtual machine which is at least as capable as the Kilobyte Virtual Machine (KVM). The KVM is a modular virtual machine which is designed to have a small size and to be easily portable, while being as complete a Java Virtual Machine (JVM) as possible. Many features of JVM is said to be optional in KVM. These include class file verification, class loading, and Java Native Interface (JNI). The KVM, as defined by CLDC, does not provide any of these features in the same way as the JVM in J2SE.

2.2.2 CDC

CDC targets a broad range of devices that are network connected, have typically a 32-bit Central Processing Unit (CPU), and can make about 2MB RAM and 2.5 MB ROM available to Java. CDC's goal is to provide compatibility with J2SE, as well as being able to run on resource constrained devices. It provides full support for the JVM, many core library functions, and is compatible with CLDC, including support for GCF.

2.3 Profiles

2.3.1 Mobile Information Device Profile

Currently, the *Mobile Information Device Profile* (MIDP) is the only profile that fits on top of CLDC. See Figure 2.1. MIDP provides a set of APIs and a way of running applications. Applications run through MIDP are called *MIDlets*. MIDlets implement an interface that is defined in MIDP with methods that are called when the application is started, stopped, and paused.

In addition to the features of CLDC, MIDP provides APIs for the following features:

- Graphical user interfaces.
- I/O through implementation of several protocols in GCF. MIDP 2.0 also provides secure communication through the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols and Hypertext Transfer Protocol (HTTP) over SSL/TLS (HTTPS).
- A means of persistent data storage.
- Media support.

Appendix A contains the complete source code of a working MIDlet.

2.3.2 Foundation and Personal Profiles

There are three profiles for CDC that are in common use. See Figure 2.1. The most basic of these is the Foundation Profile [17]. The Foundation Profile provides complete support for the following packages from J2SE 1.3.1:

- `java.io`
- `java.lang`
- `java.lang.ref`
- `java.lang.reflect`
- `java.security`
- `java.security.acl`
- `java.security.cert`

- `java.security.interfaces`
- `java.security.spec`
- `java.text`
- `java.util`
- `java.jar`
- `java.zip`

This makes it possible for applications that build upon the Foundation Profile to be very much like J2SE applications. It is noteworthy that the Foundation Profile does not provide anything in the way of GUI. This is expected to be provided by profiles or optional packages that are building on the Foundation Profile.

The Personal Basis Profile builds on the Foundation Profile. It adds support for GUI by including lightweight components of the `Java.awt` packages. It also has limited support for Java Beans [13] and Remote Method Invocation (RMI), as well as support for a class of Java applications called Xlets. Xlets are a type of managed applications like MIDlets in MIDP and Applets in J2SE. The Personal Profile builds upon and include all the APIs of the Personal Basis Profile and adds new features such as full Abstract Windowing Toolkit compatibility and applet support.

2.4 J2SE security basics

In order to understand J2ME security mechanisms later in this thesis, it's necessary to first discuss some security mechanisms initially introduced in J2SE. J2SE uses different techniques to achieve several security goals [26]. These goals can be loosely split into two general groups: language and data security. Language security makes sure that code is not able to do more than it is granted permission to do from the user or administrator. This is especially important when running code of unknown or dubious origins. Data security consists mainly of making available APIs to perform operations such as encryption and digital signing of data in order to maintain confidentiality, integrity, and authentication.

2.4.1 The Java Sandbox

The Java sandbox provides a safe environment in which even untrusted Java programs can run without hurting other programs or the device the programs run on. The default sandbox which is used by most programs, is highly configurable by altering the policy files so that certain programs can run under a very specific set of permissions. The sandbox consists of permissions, code sources, protection domains, policy files, and keystores.

2.4.2 Memory Integrity

Java has several security features that aim to enforce memory integrity. The simplest of these is the lack of memory pointers in the Java language. This ensures that Java programmers cannot access arbitrary memory locations inside, or outside, the virtual machine. However, this is not enough to ensure memory integrity. Imagine what would happen if one could cast an empty object to an object that includes a large byte array. The result would be an array that points directly into somewhere in the JVM memory area, and would thereby compromise the memory integrity. To protect against these attacks Java enforces strict type checking at runtime. Java also protects against uninitialized variables, by assigning them default values when they are declared.

2.4.3 The bytecode verifier

The bytecode verifier is responsible for making sure that all classes loaded into the JVM are valid Java classes and for verifying that all code inside those classes follow the rules of the Java language. The bytecode verifier works closely with the class loader and inspects all classes being loaded. Much of the work that is done by the bytecode verifier is also done at compile-time. However, since not all necessary tests can be done at compile time, and one cannot really trust all compilers to have good intentions, much of the work is also done at load and runtime.

Chapter 3

Security Mechanisms in CLDC and MIDP

This chapter gives an overview of the security mechanisms in CLDC and MIDP. CLDC 1.0 [14] and 1.1 [20] contain security mechanisms to protect the user and the device from malicious code, which could do anything from crashing the device to being viral code spreading itself and wrecking general havoc and/or stealing information. In addition, the mechanisms in MIDP 2.0 [18] provides communication security, protecting the data from eavsdroppers.

3.1 CLDC mechanisms

A CLDC conformant virtual machine, such as the KVM, and implementations of the CLDC APIs must implement two layers of security mechanisms, low-level KVM security and application security. Low-level KVM security means that any application running on the virtual machine must be valid Java code¹ guaranteed by a *class file verifier*. The code must not be able to harm or crash the device it is running on, or access memory areas outside the Java object memory, regardless of whether the device itself can provide memory protection. The application-level security layer of CLDC includes class loading procedures, and controlling access to resources the device offers like a file-system or a Bluetooth subsystem. The CLDC specification also mentions that a profile building on CLDC should do something to ensure end-to-end communication security, this is done in MIDP 2.0.

¹In the same way all code in a Java applet must be validated before it is run.

3.1.1 The life cycle of a CLDC application

Before the application is ever loaded and can run on a device it must go through a few steps. This process is illustrated in FIGURE 3.1. One important step in this process is the preverification procedure of the byte code. This allows for much simpler and faster byte code verification in the device. When every class in an application is loaded it is verified by either a regular Java byte code verifier or by a special byte code verifier which uses the results from the preverification stage to verify the byte code. It has been shown [28] that this approach is as good as regular byte code verification, and that it is tamper proof. However, flaws have been discovered [8] in the implementation, which have led to a complete break of the device security provided by CLDC.²

The class loading restrictions also ensure that the search order of class loading is such that system classes (such as classes in `java.lang`) cannot be replaced. This is important since replacing some classes can be a quick and easy way to circumvent permission checks.

3.1.2 The CLDC sandbox

The CLDC sandbox ensures that all code has the following properties:

- All code is verified at the byte level and guaranteed to be valid Java byte code.
- The code is limited to a predefined set of APIs that are defined by CLDC, the available profiles and whatever classes that are made available by the manufacturer.
- User defined class loaders are not allowed. This ensures that no program can alter the class loading procedures or order. This protects all system libraries from being overridden, bypassed or replaced.
- Downloading and management of applications take place outside the virtual machine.
- The set of native function calls available to any application running on CLDC is closed.

These rules make sure that any application has limited functionality and cannot disrupt other applications or execute potentially harmful code other than that which is allowed by the manufacturer. It is of course up to the

²Execution of arbitrary native code.

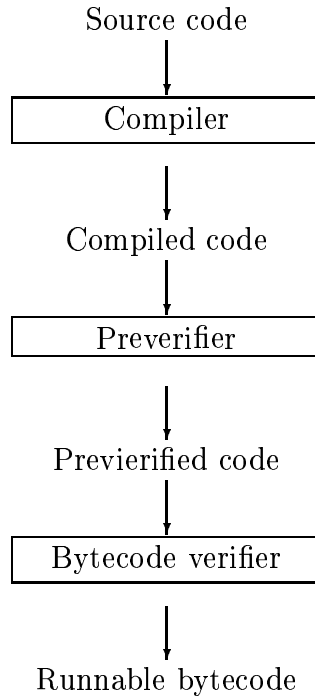


Figure 3.1: The operations CLDC code must undergo before it is allowed to execute.

manufacturer of a CLDC compliant device to ensure that all available APIs are securely implemented.

3.1.3 The KVM byte code verifier

The KVM byte code verifier uses a stack map to do less resource intensive byte code verification than the code verification used in a standard Java virtual machine. The stack map is an attribute to the executable byte code in a method. See the Java virtual machine specification [29] for details on the class file structure. The stack map, which is created in the preverification process, contains type information about all variables that should be on the stack at certain points in the code. This allows the KVM byte code verifier to essentially only have to verify the stack map instead of the whole code, which provides significant gains both in terms of CPU and memory consumption. The stack map attribute and the verification process are described in detail in Appendix 1 to the CLDC 1.1 specification [20].

3.2 MIDP mechanisms

MIDP provides several layers of security. It has its own sandbox model, a method for applications to store data such that other MIDlet suits can not reach it, and communication security mechanisms.

3.2.1 The MIDP sandbox

MIDP 2.0 provides a sandbox environment in which to run MIDlets. This sandbox prevents the MIDlets from accessing sensitive APIs and functions. MIDP 1.0 [16] does not specify any other features of the sandbox than that a security exception must be thrown when `System.exit` or `Runtime` functions are called. This is to ensure that no MIDlet can cause the entire virtual machine, which could be running other MIDlets, to exit or spawn native processes.

Permissions

In MIDP 2.0 the concepts of protection domains and permissions are introduced into MIDP. With this comes the concept of trusted applications, which are usually identified with digital signatures.³ The signatures are connected to a protection domain as a result of a policy. It is voluntary whether the end user is allowed to change or specify these policies, any “unauthorized parties” are not allowed to view or modify them.

Signed MIDlet suits

MIDP allows for signing MIDlet suites as a means of putting them in a trusted protection domain. A MIDlet suite may be placed in a protection domain according to the associated protection domain root certificate. A protection domain root certificate is a certificate that the device trusts to verify and authorize MIDlet suits to the extent of the associated protection domain. The signing process defined in the MIDP 2.0 specification is based on Public Key Infrastructure (PKI) and X.509 certificates, although additional signing mechanisms and certificate formats may be used.

The root certificates that are used to verify certificate chains must be available on the device. Root certificates may be stored on removable media. It also seems that the user may be able to manually insert and remove root certificates. These root certificates may only be viewed or modified by authorized parties although the MIDP specification does not mention who these

³The MIDP 2.0 specification allows other mechanisms to be used.

authorized parties are. In practice it seems that the set of root certificates is static in many devices, if signed MIDlet suites are supported at all.

3.2.2 The Record Management System

The Record Management System (RMS) is a system which allows MIDlets to store data on a device in a non-volatile and implementation independent manner. A Record Store (RS) is a collection of records in the RMS. A RS is available only to MIDlets in the same MIDlet suite as the creating MIDlet. MIDP 2.0 also allows explicit sharing of record stores between MIDlet suits. When a MIDlet suite is removed, any associated records must be removed. RSs must be stored at a location on the device which is unreachable by any other means than through RMS. This is to ensure that one MIDlet suite's RS is unavailable to other MIDlet suites.

3.2.3 SSL/TLS

MIDP 2.0 provides communication confidentiality through the use of the true and tested SSL/TLS and HTTPS protocols. Support for HTTPS is mandatory, while support for SSL/TLS seems to be optional. The features of SSL/TLS offered by MIDP 2.0 seems to be restricted to connecting to a server and communication over an encrypted channel. Although the server's certificate is attempted to be validated by the underlying software and the fact that an API to get some information from the server's certificate exists, there does not exist enough information available to the MIDlet so that it can validate the server certificate itself. This can be a problem to application developers that for some reason do not want to use one of the few certificate authorities that are supported by any given device.⁴ The client is not able to participate in a two-way authentication scheme either. It is of course possible for a application programmer to implement his own SSL/TLS suite although this might be hampered by the lack of cryptographic APIs and good sources for randomness in MIDP and CLDC. HTTPS may be implemented using one or more of the following protocols as its security layer:

- HTTP over TLS [27]
- SSL version 3.0 [30]
- Wireless Transport Layer Security (WTLS) [37]

⁴This can be because of mistrust, poor finances or lack of generality.

- Wireless Application Protocol (WAP) [35] TLS Profile and Tunneling [38]

If only WTLS is supported this can be a problem since WTLS decrypts packages in a gateway at the service provider before they are reencrypted and sent to the application server using SSL or TLS. Furthermore it is a problem for application developers since they must check which protocol is used, after the connection has been established and then determine whether the protocol is acceptable. The same protocols may be used for the SSL scheme except for WTLS. A problem arising with the use of SSL and TLS is that the application again must check which of these protocols is in use after the connection has been established.

Support for certificate revocation, a key component in any PKI enabled system, is not required by MIDP 2.0. However, if certificate revocation is to be included only support for the Online Certificate Status Protocol (OCSP) is required. Support for OCSP alone may, in some cases, be inadvisable since it would not allow the Certificate Authority to revoke the certificate of a OCSP server in the case of a compromise.

3.3 Thoughts on CLDC and MIDP security

The methods used for providing low level security in CLDC seem to do what they are supposed to do reasonably well. The methods, provided that they are correctly implemented, are probably good and only implementation problems have been found to attack CLDC. The communication security mechanisms in MIDP 2.0 lack some important features such as client authentication and explicit server authentication. However, implemented correctly, it does provide confidential exchange of data between the device and a server, provided that the server's private key has not been compromised and (in the event of HTTPS over WTLS) that the service provider can be trusted.

Chapter 4

Random Numbers in J2ME

This chapter describes a method of obtaining random numbers in J2ME. The chapter starts off by establishing the need for random numbers and discusses why using a clock's value at any given time is not good enough for cryptographic applications. The chapter then goes on to describe a way to obtain better random numbers and gives sample code that can accomplish this task. The chapter ends with some statistical results and a discussion on whether this method is good enough for cryptographic applications.

4.1 Why random numbers?

In order to safely transmit data over unsafe channels, such as GSM, Internet, etc, data must be encrypted in such a way that only the intended receiver can decrypt the data. To do this one needs a key. If an attacker knows the key, he can easily decrypt the message himself, therefore it is necessary to use a key which is very hard to guess. The hardest to guess key is one that is created completely at random. The second best thing is a pseudo number generator, which in itself is of course completely deterministic, seeded with a large true random value. The seed must be large, or at least have the potential to be large, such as to make it difficult to determine the seed value by some brute force technique.

4.2 The dangers of `now()`¹

The time right now, measured in milliseconds since 01.01.1970, is a very common seed used to initialize Pseudo Random Number Generators (PRNGs). In fact it is the default way to initialize PRNGs in several popular programming languages like Java² and Python. Even the SSL implementation in the MIDP reference implementation uses this approach to seed its random number generator by default.

The seed produced by `now()` is a rather large number, 1093852693 at the time of writing this, so why is it not good enough? If an attacker can determine to a small time interval when the key was created, he can use this information to find the key. If this time interval is an hour, the attacker will need to create at most $1000 * 60 * 60 = 3600000$ keys to find the right one. Which makes the key roughly as strong as a key of 22 bits. This observation applies to any encryption technique and requires that the attacker has only one known plain text, cipher text pair, or in the case of some public key encryption schemes including RSA—the public key. It is in most cases not likely that an attacker can narrow down the time the seed was created to such a small time interval as an hour. Still even if the attacker could only determine within a year when the seed was created, the the attacker would only have to create less than 2^{35} keys reducing the key length to 35 bits.

This attack only works when the attacker either knows that the PRNG is reseeded with `now()` just before the key is created, or can somehow guess how many times it has been used before, that is the offset in the sequence created by the PRNG. Even if the seed is completely random it should be at least as large as the key itself, otherwise the key would only be as strong as the size of the seed.

4.3 Randomness in J2ME

Sadly neither CLDC, MIDP or Security And Trust Services API (SATSA) [21] specify any means of getting truly random numbers to your application. One solution could be to measure the time between user actions several times and then only take into account the least significant bit every time. One drawback to this solution is that to create a sufficient large seed, the user have to do quite a lot of actions.

¹`now()` is here defined as a function that returns the number of milliseconds that have passed since 01.01.1970 00:00:00.000. This is a common way for computers to tell the time.

²Only the `java.util.Random` class does this, not `java.security.SecureRandom`

Another solution could be to get numbers from a third party source, like <http://www.random.org>, over an Internet link. One problem with this approach is that all network traffic may be eavesdropped upon, so it should be encrypted, possibly leading to a bootstrap problem. Another problem is that the third party source must be trusted.

A solution for mobile devices that support the Multi Media API (MMAPI) [19] is described at [11]. It just records sound through a microphone that stands close to an out of tune radio. It then takes the least significant bit of each 8 bit sample, which might be sensible depending on the encoding format, does some skew correction to ensure some statistical properties and pumps the corrected bits back out as random bits. Another approach that has shown good results from statistical tests is to take a parity bit over each sample instead of the least significant bit. The rest of this chapter will show how to do this in J2ME using CLDC, MIDP and MMAPI.

4.4 An audio based random number generating MIDlet

We need a nice little MIDlet that creates a Thread for recording sound, and an OutputStream where the sound is supposed to be stored. It could be implemented something like this, only methods important to the process are shown. The complete source code is available at <http://kenti.org/nowires/files/RandomMIDlet.java>.

```
public class RandomMIDlet extends MIDlet {

    RandomGettingOutputStream out;
    private ByteArrayOutputStream randomBits;
    private Display display;

    /*
    *Constructs the MIDlet. Creates an OutputStream and sets
    *the display.
    *@throws IllegalArgumentException
    */
    public RandomMIDlet() throws Exception {
        //we want 128 random longs and store them in randomBits.
        randomBits = new ByteArrayOutputStream();
        out = new RandomGettingOutputStream(128, file);
    }
}
```

```

        display = Display.getDisplay(this);
    }

    /*
    *Starts the MIDlet. Creates and starts the recording thread.
    *Then it shows something, exactly what it shows is not important.
    */
    public void startApp() {
        Thread t = new Thread(new Collector(out,this));
        t.start();
        showWaitScreen();
    }
}

```

We also need a recording Thread. The only place where this thread differs from the code example for recording audio with MMAPAPI is that it uses a custom made `OutputStream` and calls back the MIDlet when finished. Most of this code is taken from the API documentation of MMAPAPI.

```

public class Collector implements Runnable{

    RandomMIDlet rnd;
    RandomGettingOutputStream out;

    /*
    *Constructor that sets OutputStream and the object to be
    *called back.
    *@param out The OutputStream to which the sound bytes
    *are written
    *@param rnd The object to be called when finished.
    */
    public Collector(RandomGettingOutputStream out,
RandomMIDlet rnd){
        this.out = out;
        this.rnd = rnd;
    }

    public void run(){
        try {
            // Create a Player that captures live audio.

```

```
Player p = Manager.createPlayer("capture://audio?
    Encoding=pcm&rate=8000&bits=8&
    Channels=1");
p.realize();
// Get the RecordControl, set the record stream,
// start the Player and record for 5 seconds
RecordControl rc =
(RecordControl)p.getControl("RecordControl");
rc.setRecordSizeLimit(Integer.MAX_VALUE);
rc.setRecordStream(out);
rc.startRecord();
p.start();
while(!out.isFinished()){
    Thread.currentThread().sleep(5000);
    System.out.println("Committing "+
out.getNumSampledLongs());
    try{
        rc.commit();
    }
    catch(Exception u) {
        System.out.println(
            "committed and caught exception");
    }
    p.stop();
    rc.setRecordStream(out);
    rc.startRecord();
    p.start();
}

rc.commit();
p.close();
}
catch (IOException ioe) {
    System.err.println(ioe);
}
catch (MediaException me) {
    System.err.println(me);
}
catch (InterruptedException ie) {
    System.err.println(ie);
}
```

```

        rnd.callBack();
    }

}

```

The last class that is needed is a custom `OutputStream`. This is where the data is selected as to guaranty randomness in a larger degree than the audio encoding scheme provides. It can take only the lowest order bit of each 8-bit sample, or a parity bit over each sample. It always look at two bits at a time. If the bits are equal (00 or 11) it is disregarded. If they differ (10 or 01) the first bit is selected. This helps to prevent bursts of ones or zeros to taint the resulting bit stream and thereby giving a higher probability of a good random distribution of bits. The method, which is due to Von Neumann, is described among other places in [3].

```

public class RandomGettingOutputStream extends OutputStream {
    private boolean haveUncorrected = false;
    private boolean finished = false;
    int firstSample;
    int secondSample;
    int numSampledLongs = 0;
    int totalLongs;
    long bits = 0;
    int bitpointer = 63;
    int counter = 0;
    OutputStream out; //where the bits are eventually stored

    public RandomGettingOutputStream(int numLongs,
    OutputStream out) {
        totalLongs = numLongs;
        this.out = out;
    }

    /*
    *Get some random bits from a random array of bytes
    */
    public void write(byte[] b){
        for(int x = 0; x < b.length; x++){
            doWrite((int)b[x]);
        }
    }
}

```

```
/*
*Get some random bits from a random array of bytes
*/
public void write(byte[] b, int off, int len){
    for(int x = 0; x < len; x++){
        doWrite((int)b[off+x]);
    }
}

/*
*Write a byte
*/
public void write(int b){
    doWrite(b);
}

/*
*Process one byte
*/
public void doWrite(int b) {
    if(finished){
        return;
    }
    b = b & 0x01;//eradicate all but the least significant
                //bit.

    if(!haveUncorrected){
        firstSample = b;
        haveUncorrected = true;
        return;
    }
    else {
        secondSample = b;
        haveUncorrected = false;
        if(firstSample == secondSample)
            return;
    }

    long l = (long)firstSample;//firstSample;
    l = l << bitpointer;
```

```
bits = bits | 1;

if(bitpointer == 0){
    writeOut(bits);
    bits = 0;
    numSampledLongs++;

    bitpointer = 64;
}
bitpointer--;

if(numSampledLongs >= totalLongs){
    try{
        out.flush();
    }
    catch(IOException ioe){
        ioe.printStackTrace();
    }
    finished = true;
}
}

/*
 *Returns 1 if number of 1's are even, 0 otherwise
 */
private int parity(byte b){
    byte[] buff = new byte[8];
    buff[0] = (byte) (b & 0x01);
    buff[1] = (byte)((b >> 1) &0x01);
    buff[2] = (byte)((b >> 2) &0x01);
    buff[3] = (byte)((b >> 3) &0x01);
    buff[4] = (byte)((b >> 4) &0x01);
    buff[5] = (byte)((b >> 5) &0x01);
    buff[6] = (byte)((b >> 6) &0x01);
    //sign bit is sometimes a problem in Java
    if(b < 0)
        buff[7] = (byte)1;
    else
        buff[7] = (byte)0;

    //count ones
```

```
    int ones = 0;
    for(int x = 0; x < buff.length; x++){
        if(buff[x] == 1)
            ones++;
    }
    if((ones % 2) == 0)
        return 1;
    return 0;
}
}
```

4.5 Attacks on this approach

A few attacks are possible on this approach. One could broadcast prerecorded looped noise that sounds like an out of tune radio on the frequency that the user utilizes as his out of tune frequency and such be able to determine what seed is produced trying all likely offsets. Another way to break this approach is to record everything that goes on in the room where the seed is made.

4.6 Results from statistical analysis

A number of data sets were created by running the described program on Sun Microsystems' Wireless Toolkit 2.2 beta. The audio source was a radio connected to the computer which ran the program. The statistical properties of the data sets varied quite a bit depending on what frequency the radio was tuned to, which radio was used, and even from time to time on the same frequency and radio. The variations of the statistical properties suggest that this method is not ideal, especially not in the ever changing environment where wireless devices roam.

Using the a parity bit of each sample when constructing the random bit stream instead of just the least significant bit seemed to yield somewhat better statistical properties when using bad hardware or frequencies. It would be recommendable to do some statistical tests on any specific configuration before using this method. Provided that the device has enough resources available to do this, the code presented above could easily be altered to allow for such tests to be incorporated. There exists some tests implemented in pure Java [6] that should be able to run on a mobile device.

John Walker's [34] *ENT* tool gave the following results when applied a 13

MB sample. The sample was gathered from a good radio on a good frequency.

Entropy = 7.999986 bits per byte.

Optimum compression would reduce the size of this 13676120 byte file by 0 percent.

Chi square distribution for 13676120 samples is 257.19, and randomly would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 127.5460 (127.5 = random).

Monte Carlo value for Pi is 3.139855915 (error 0.06 percent).

Serial correlation coefficient is 0.000055

(totally uncorrelated = 0.0).

When the tool was asked to look at the same file as a sequence of bits instead of bytes these were the results.

Value	Char	Occurrences	Fraction
0		54699537	0.499955
1		54709423	0.500045

Total:		109408960	1.000000
--------	--	-----------	----------

Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size of this 109408960 bit file by 0 percent.

Chi square distribution for 109408960 samples is 0.89, and randomly would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).

Monte Carlo value for Pi is 3.139855915 (error 0.06 percent).

Serial correlation coefficient is -0.000163

(totally uncorrelated = 0.0).

The Entropy value here is a measure of information density.³ A com-

³Information density is the minimum mean number of bits per character needed to store the data in a retrievable fashion.

pletely random file will be expected to have an entropy value per character equal to the number of bits in the sample. The optimum compression is just another measure for information density which says how much a file containing the sequence can be compressed. With approximately 8 bits of information per byte and 1 bit of information per bit the sampled sequence passes these tests.

The chi-square distribution is one of the most common randomness tests. It is described in [23]. The percentage shown compares the chi-square distribution with the expected distribution. In [23] Knuth recommends that one considers anything between 10 percent and 95 percent as acceptable for a random sequence. These tests can be said to be passed for the sampled sequence.

The arithmetic mean is calculated as the sum of all samples divided by the number of samples. As the tests show, the calculated means are very close to the expected mean in each test, so these tests may be said to be passed.

The Monte Carlo value of Pi is the result of a Monte Carlo algorithm to find the value of pi. A truly random sequence should find Pi to high degree of accuracy.

The serial correlation coefficient is a measure of how much each sample depends on the previous sample. It is also described in [23].

The DIEHARD [24] battery of test was also applied on the set. The results of this test may be found at <http://www.kenti.org/nowires/files/result.txt>. The results from this test shows that this sequence is probably quite random in nature.

4.7 Conclusion

In this chapter it has been shown that a better way to generate random numbers than using `now()` to seed a PRNG is necessary for cryptographic applications. A method of generating random numbers using MIDP, CLDC and MMAPI has been demonstrated as a way to generate seeds which are probably harder to guess than `now()`. Some statistical properties of sequences generated using this method has been discussed. Although the statistical test results on some of the generated sequences suggest that the sequences are truly random, the results vary a bit too much to be comfortable with this solution.

Chapter 5

Attack on Sun's MIDP Reference Implementation of SSL¹

5.1 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) [25], 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

¹This is a manuscript authored by Kent Inge Fagerland Simonsen, Vebjørn Moen, and Kjell Jørgen Hole.

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

It is known [36] 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 5.2 we give a brief introduction to SSL, Section 5.3 considers randomness, Section 5.4 describes the attack on SSL in MIDP, as well as the implementation of the attack, and Section 5.5 concludes the paper.

5.2 SSL

This section is not meant to give a complete description of the SSL protocol; for a complete description [32] 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 5.1 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

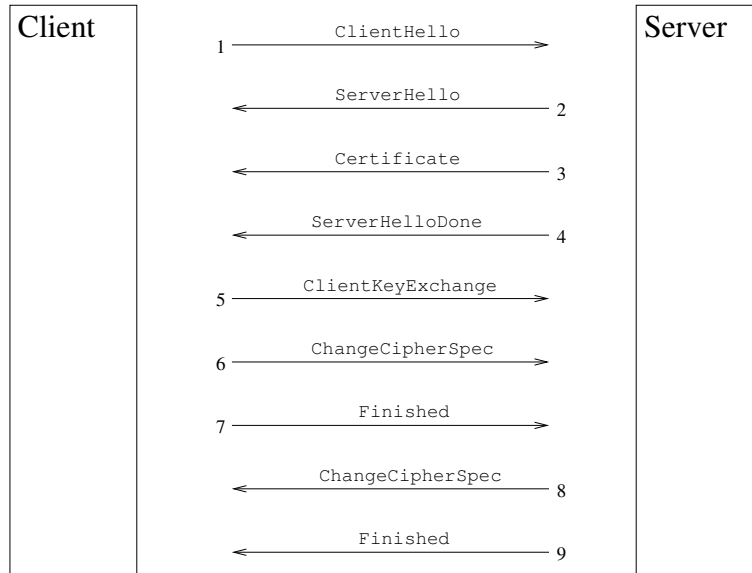


Figure 5.1: The 9 messages that SSL uses to establish an encrypted communication channel.

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.

5.3 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.

5.3.1 Creating a seed

The seed is essential for the security of the system. RFC1750 [3] 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 [22] and */dev/random* in GNU/Linux.

Radioactivity decay source, Gaussian white noise and spinning disks [3, 2] 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 [10] describes some practical solutions of how to create random numbers for use in cryptographical protocols and for key material.

5.4 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.

5.4.1 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 5.2. When the PRNG is constructed it initializes the MD5 digest and the `updateSeed()` 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 5.3. 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

```
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 bytesAvailable = 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 5.2: Pseudo code of the PRNG from Sun's reference implementation of MIDP.

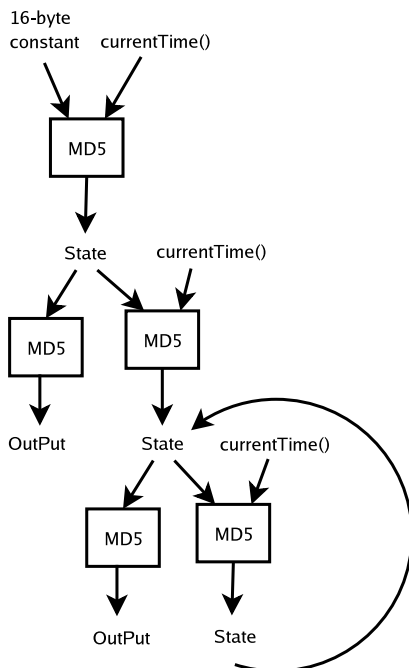


Figure 5.3: How MD5 is utilized to generate the pseudo random data used for nonce and premaster in the reference implementation of MIDP SSL. The 16-byte constant is known from the decompiled Java byte code.

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.

5.4.2 The attack step-by-step

1. Sniff an SSL session and record the starting time.
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  $\geq$  t1
    for each t3 in time interval  $\geq$  t2
      premaster = generatePreMasterCandidate(
                    PRNG_state,t1,t2,t3)
      check(premaster)

```

5.4.3 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.

5.4.4 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 attack-

ing. 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 \hat{t}_{start} , \hat{t}_{stop} for the generation and $\Delta\hat{t} = \hat{t}_{stop} - \hat{t}_{start}$.

We need to guess three time seeds to generate a suggestion for the pre-master secret, which have time complexity $\mathcal{O}\left((\Delta\hat{t})^3\right)$. 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 $\Delta\hat{t}$ 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 pre-master secret is $\mathcal{O}\left(\left(\frac{1}{3} \cdot \Delta\hat{t}\right)^3\right) = 1/27 \cdot \mathcal{O}\left((\Delta\hat{t})^3\right)$. Resulting in a total time complexity of:

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

5.4.5 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 [4] 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 [33] 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 pre-master secret.

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 a Intel Pentium M processor 1600MHz. It is likely that the attack on a 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.

5.5 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 for seed to the PRNG. There is an easy 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.

Chapter 6

Conclusions

6.1 Summary

This thesis has looked at various security aspects of J2ME. First, it gave some background on J2ME and Java security. Then, security mechanisms in CLDC and MIDP were looked at and discussed. The thesis then continued with a chapter on randomness in J2ME. It was explained that randomness is an essential part of many cryptographic techniques and that it is not necessarily trivial to obtain random numbers. The chapter also proposed a method to generate random numbers which had variable results. At the end came a chapter containing a paper detailing the implementation of an attack on Sun Microsystems' reference implementation of MIDP's SSL implementation.

6.2 Conclusions

As far as language security mechanisms go, CLDC and MIDP seem to be in good shape. Code is guaranteed to be valid Java code when it runs, and restrictions in CLDC and MIDP seem to restrict what unauthorized code may do and can thereby protect both the device and user against malicious code.

The confidentiality and integrity of data in transit are mainly provided by SSL/TLS and HTTPS. The MIDP 2.0 specification allows for sub-standard provisions in HTTPS which seriously endangers confidentiality and integrity of data transmitted over that protocol. The SSL/TLS does not allow for the usage of the worst of these protocols, so if implemented correctly it should be reasonable good, although the application programmer may find the APIs a bit limiting if he wants total control. Especially if custom certificate checking is required.

In this thesis it has been shown by example that good sources of randomness are essential on the client side for confidential communication over SSL. This is not news, but it is interesting to see how easy it is to break the encryption when the random source used to generate a seed is not good enough. The thesis has also proposed a way to gather random data using a common optional package of MIDP. This technique, although it may not provide adequate randomness in many cases, might be an improvement. It is the authors hope that vendors use better ways to gather randomness for their SSL implementations.

6.3 Further work

Further work along the lines of this thesis may include looking at some of the optional packages that are available and in the process of being specified for J2ME. In particular it would be interesting to look at the Security and Trust Services API package [21].

To better solve problems concerning randomness, it might be worth while to look at possible hardware solutions. It is the author's opinion that it is necessary to standardize some way of getting random data in J2ME, and that J2ME implementations must satisfy reasonable requirements when it comes to the quality of the source of randomness.

Appendix A

A Portscanning MIDlet

```
/* PorrtScanMidletMIDlet.java
 *
 * A midlet that can portscan a host over the Internet.
 *
 * Created on January 17, 2004, 3:10 PM
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;

public class PortScanMidletMIDlet extends MIDlet
implements CommandListener {

    private Command exitCommand; // The exit command
    private Command scan;
    private Command back;
    private Display display; // The display for this MIDlet
    private TextField hostField, portField1, portField2;
    private String scheme = "socket://";

    /**
     * The Constructor.
     */
    public PortScanMidletMIDlet() {
```

```
        display = Display.getDisplay(this);
        scan = new Command("scan", Command.SCREEN, 1);
        back = new Command("back", Command.SCREEN, 1);
        exitCommand = new Command("Exit", Command.SCREEN, 2);
    }

    /**
     * Start up the MIDlet.
     */
    public void startApp() {
        Form f = new Form("PortScanner");
        hostField = new TextField(
            "who do you want to scan today?",
            "localhost",
            255,
            TextField.ANY);
        portField1 = new TextField(
            "what port should be scanned from?",
            "630",
            30,
            TextField.NUMERIC);
        portField2 = new TextField(
            "what port should be scanned to?",
            "640",
            30,
            TextField.NUMERIC);
        f.addCommand(scan);
        f.addCommand(exitCommand);
        f.setCommandListener(this);

        f.append(hostField);
        f.append(portField1);
        f.append(portField2);
        display.setCurrent(f);
    }

    /**
     * Do the actual scanning.
     * This should really be in a separate Thread,
     * but it looks simpler this way.
     */
```

```

private void scan(String host, int from, int to){
    Form f = new Form("requested ports on "+host);
    List l = new List("",List.IMPLICIT);
    String report = "";

    String pstat;
    int tmp;
    if(from > to){
        tmp = from;
        from = to;
        to = tmp;
    }

    for(;from <= to;from++){
        try{

            StreamConnection sc =
                (StreamConnection)Connector.open(
                    scheme+host+": "+from);

            pstat = from+": open";
            sc.close();
        }
        catch(ConnectionNotFoundException cnfe){
            pstat = from+": closed";
        }
        catch(java.io.IOException ioe){
            pstat = from+": closed";
        }
        report += pstat+"\n";
    }

    StringItem si = new StringItem("",report);
    f.append(si);

    f.addCommand(back);
    f.addCommand(exitCommand);
    f.setCommandListener(this);
    display.setCurrent(f);
}

```

```
/**
 * Pause is a no-op since there are no background
 * activities or record stores that need to be closed.
 */
public void pauseApp() { }

/**
 * Destroy must cleanup everything not handled by
 * the garbage collector. In this case there is
 * nothing to cleanup.
 */
public void destroyApp(boolean unconditional) { }

/*
 * Respond to commands, including exit
 * On the exit command, cleanup and that the MIDlet
 * has been destroyed.
 */
public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
        destroyApp(false);
        notifyDestroyed();
    }
    if(c == scan){
        String host =
            hostField.getString().toLowerCase();
        int from =
            Integer.parseInt(portField1.getString());
        int to =
            Integer.parseInt(portField2.getString());
        scan(host, from, to);
    }
    if(c == back)
        startApp();
}
}
```

Bibliography

- [1] Connected Limited Device Configuration (CLDC),
<http://java.sun.com/products/cldc/>.
- [2] Don Davis , Ross Ihaka , Philip Fenstermacher, “Cryptographic Randomness from Air Turbulence in Disk Drives,” Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology, August 21-25, 1994, p.114–120.
- [3] D. E. Eastlake 3rd, S. Crocker, J. Schiller, “RFC 1750, Randomness Recommendations for Security,” December 1994,
<http://www.ietf.org/rfc/rfc1750.txt>.
- [4] Ethereal network protocol analyzer, <http://www.ethereal.com/>.
- [5] E. Giguere, “Data Encryption for J2ME Profiles,” Sun Developer Network, 2001,
<http://developers.sun.com/techttopics/mobility/midp/ttips/dataencryp/>.
- [6] The GNU Crypto project, <http://www.gnu.org/software/gnu-crypto/>.
- [7] Ian Goldberg and David Wagner, “How Secure is the World Wide Web?,” Dr. Dobbs’s Journal, January 1996, pp. 66–70.
- [8] A. Gowdiak, “Java 2 Micro Edition (J2ME) Security Vulnerabilities,” *Hack In The Box (HITB 2004)*, Kuala Lumpur, Malaysia, October 2004.
- [9] A. Gusev, “For the Second Year in a Row, Readers call the J2ME Wireless Toolkit Number One!,” developer.com, Jupitermedia Corporation, 2005,
<http://www.developer.com/java/web/article.php/3483051>.
- [10] P. Gutmann, “Software generation of practical strong random numbers,” Proceedings of the Seventh USENIX Security Symposium, 1998, pp. 243–257.
- [11] M. Haahr, *Introduction to Randomness and Random Numbers*, 1999,
<http://www.random.org/essay.html/index2.html>.

- [12] J2ME Licensees <http://java.sun.com/j2me/reference/licensees/index.html>.
- [13] Desktop Java, JavaBeans, <http://java.sun.com/products/javabeans>.
- [14] JSR 30, *J2ME Connected, Limited Device Configuration*, 2000,
<http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html>.
- [15] JSR 36, *Connected Device Configuration 1.0a*, 2002,
<http://jcp.org/aboutJava/communityprocess/final/jsr036>.
- [16] JSR 37, *Mobile Information Device Profile for the J2ME™ Platform*, 2000,
<http://www.jcp.org/en/jsr/detail?id=37>.
- [17] JSR 46, *J2ME Foundation Profile Specification*, 2002,
<http://jcp.org/aboutJava/communityprocess/final/jsr046/index.html>.
- [18] JSR 118, *Mobile Information Device Profile 2.0*, 2002,
<http://www.jcp.org/en/jsr/detail?id=118>.
- [19] JSR 135, *Mobile Media API*, 2003,
<http://jcp.org/aboutJava/communityprocess/final/jsr135/index2.html>.
- [20] JSR 139, *Connected Limited Device Configuration 1.1*, 2003,
<http://www.jcp.org/aboutJava/communityprocess/final/jsr139>.
- [21] JSR 177, *Security and Trust Services API for J2ME*, 2004,
<http://jcp.org/aboutJava/communityprocess/final/jsr177/index.html>.
- [22] J. Kelsey, B. Schneier, and N. Ferguson, “Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudo-Random Number Generator,” Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999, pp. 13–33.
- [23] D. Knuth, *The Art Of Computer Programming Vol. 2*, Third Edition, Addison-Wesley, 1998.
- [24] G. Marsaglia, *The Diehard Battery of Tests of Randomness*,
<http://www.cs.hku.hk/diehard/>.
- [25] Mobile Information Device Profile (MIDP),
<http://java.sun.com/products/midp/>.

- [26] S. Oaks, *Java Security*, Second Edition, O'Reilly & Associates, Inc., 2001.
- [27] E. Rescorla, *RFC 2818, HTTP Over TLS*, The Internet Society, 2000.
- [28] E. Rose, "Lightweight bytecode verification," in *Journal of automated reasoning*, vol. 31, pp. 303–334, 2004.

- [29] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Second Edition, Sun Microsystems 1999,
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>.

- [30] Secure Socket Layer Version 3, <http://home.netscape.com/eng/ssl3/draft302.txt>.
- [31] Sourceforge, <http://www.sourceforge.net>.
- [32] Stephen Thomas, "SSL and TLS Essentials: Securing the Web," Wiley Computer Publishing, 2000.
- [33] TinySSL, http://www.xwt.org/javadoc/javasrc/org/xwt/util/SSL_java.html.
- [34] J. Walker, <http://www.fourmilab.ch>.
- [35] Wireless Application Protocol,
<http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html>.
- [36] Wireless Java Security, <http://sys-con.com/story/?storyid=37377&DE=1>.
- [37] Wireless Transport Layer Security, http://www.wapforum.org/what/technical_1_2_1.htm.
- [38] WAP(TM) TLS Profile and Tunneling Specification,
<http://www.wapforum.com/what/technical.htm>.
- [39] M. J. Yuan, J. Long, "Securing wireless J2ME," developerWorks, 2002,
<http://www-128.ibm.com/developerworks/wireless/library/wi-secj2me.html>.