HTL St. Pölten edvo Linzer Straße

Diploma Thesis

Java™ Security Concepts Within 'Insecure Networks'

In the case of the SASII project

Author:	Jürgen Leitner Markt 328 3184 Türnitz
Examiner:	Prof. Mag. Otto Reichel
Period:	September 2001 – May 2002

Statement

With my signature, I assure, that this diploma thesis, with the topic "Java Security Concepts Within 'Insecure Networks'", was completely written by me. I also assure that I only used the mentioned sources, not others.

Any copies, partly or full, or any other reprint or use of this diploma thesis, for non educational purpose, is only allowed with the written permission of the author.

Juxi Leitner
juxi.leitner@aon.at

Dedicated to freedom and privacy

This diploma thesis is dedicated to freedom of speech, freedom of dreams and thoughts and the freedom of the internet. It is also dedicated to the privacy, and not military, purpose of cryptography and encryption.

Special thanks

I would like to thank the following persons for their help and contribution to this diploma thesis:

Prof. Mag. Otto Reichel: *for his guidance and supervision of the SAS II project and this diploma thesis, also for his education in programming and the Java language*
Andreas Lehrbaum: *for his inspiration, motivation and help for this diploma thesis and my whole life, his brilliant work on SAS II and his company*
Andreas Fellnhofer and David Rigler: *for their ideas in the SAS II technology team and their inspirations for life*
Florian Freistetter and Matthias Pölzinger: *for their awesome work on the SAS II project*

I also want to thank the rest of the SAS II team and all people that helped me to find bugs, typos and grammar faults in my first blueprints.

Table of Contents

Abstract, English Version

This diploma thesis deals with security concepts, designed to make "insecure network" more secure in handling and usage. My point of view is from Java's concepts for security, but also is based on the evaluation of security concepts for the SASII project.

Java has in the last years developed to an often used multimedia extension for HTML pages. It is said to be secure, but what does that mean? I am trying to give a look at what it could mean, for developers and for users with this diploma thesis.

This diploma thesis is separated into two main parts (Application Security and Servlet Security) and completed with an Introduction and a Future Outlook. Java, of course, consists of more then just Applications and Servlets, but those two main parts are resulting from the considerations during the planning phase of the SASII project

One section deals with Cryptography. At the moment that is a very often used word, especially in the computer business. Cryptography deals a lot with freedom and privacy. There are also a lot of political discussion about that at the moment and Cryptography is mostly linked with espionage, military and "Top Secret". I hope to show that Cryptography is only a tool, which can be used for everything, from allowing users to make secure bank transactions to military use.

Abstract, German Version

Diese Diplomarbeit beschäftigt sich hauptsächlich mit Sicherheitskonzepten, die dazu dienen „unsicher Netze" sichere zu machen, sowohl in Verwendung als auch im Umgang mit ihnen. Ich betrachte es vom Standpunkt der Programmiersprache Java aus, jedoch auch auf Basis der durchgeführten Studien bezüglich unseres Schulprojekts SAS II.

Java hat sich in den letzten Jahren zu einer oft genutzten Multimedia Erweiterung für statische HTML Seiten entwickelt. Es wird behauptet Java sei sicher, aber was ist damit gemeint? Ich versuche mit dieser Diplomarbeit einen Blick darauf zu geben, was sicher für Benutzer sowie Programmierer bedeuten kann.

Diese Diplomarbeit ist aufgeteilt in zwei Hauptpunkte (Application Security und Servlet Security), sowie Einleitung und Zukunftsaussichten, der Sicherheitskonzepte. Java bedeutet natürlich nicht nur Applikationen und Servlets, aber diese zwei Hauptpunkte haben sich bei der Evaluierung der Lösungen für das SAS II Projekt herauskristallisiert.

Auch ein Kapitel Kryptographie ist in dieser Diplomarbeit zu finden. Kryptographie ist im Moment ein ziemlich oft gebrauchtes Wort, nicht nur bei Programmierern und Computerfachleuten, da sie auch sehr stark mit Freiheit und Privatsphäre zusammenhängt. Es gibt auch viele Diskussion in der Politik, da Kryptographie meistens mit Spionage, Militär beziehungsweise „Top Secret" gleichgesetzt wird. Ich versuche mit dieser Diplomarbeit zu zeigen, dass Kryptographie nur ein

Hilfsmittel ist, welches sehr vielseitig verwendet werden kann, vom sicheren Einkaufen im Internet bis hin zu militärischen Zwecken.

1. Introduction

"Security is mostly a superstition. It does not exist in nature, nor do the children of men as a whole experience it. Avoiding danger is no safer in the long run than outright exposure. Life is either a daring adventure, or nothing. " - Helen Keller

1.1 Preface

With the raising number of internet users and the increasing use of "dynamic" web content (e.g. Java Applets, Servlets, Java Server Pages ...), a lot of webmasters try to make their servers and applications more secure. One of Java's main features is the ability to let code transfer over a network and then run on the local PC. This has pros and cons, it is positive for generating dynamic web content as used in Java Applets, but a disadvantage is that there could be security bugs. To keep them at a minimum Sun Microsystems, the designer of Java, included some security concepts. With Java Development Kit (JDK) 1.2 and 1.4 major changes to these concepts were introduced.

This diploma thesis covers the application side from the Java viewpoint. In this paper security concepts within JDK 1.2 and 1.4 are explained. Another chapter in this diploma thesis covers security concepts for Java Servlets, defined in the JSDK (Java Servlet Development Kit). These concepts are mainly for secure transfer and identifying users (and computers) and they are used in "secure" web-applications like web shops or net-banking.

Security is not a clearly defined term; therefore the first few pages are about how to define the word security and how to use security concepts within Java to generate more secure applications.

1.2 Java and Security

Java has a lot of security features but the problem is how to utilize them in applications. Some of these features are automatically components of all Java programs, many are not. Different ways are given to make them a part of a Java application.

1.3 What is security?[1]

As already mentioned security means different things to different people. Developers had different expectations of Java's security than the designers of Java. Different expectations would lead to expect that Java programs might be:

- No malevolent programs: *programs should not harm the users' computer environments. This includes viruses, Trojan horses and others.*
- Non intrusive: *programs should not be allowed to get private information on the host computer or host computers network*
- Authenticated: *The identity of parties involved in the programs processes should be verified*
- Encrypted: *Data transfers over unsecured connections should not be interpreted by third parties*
- Audited: *sensitive operations (or special users) should be logged*
- Well-defined: *strict regulations for security should have to be followed*

- Verified: *operation rules should be declared and checked*
- Well-behaved: *programs should not use too many system resources, or force computers to crash*

In the starting version of Java, Java 1.0 just 2 points were part of the security model, with later versions some others were added, but still not all of these points are available. With all updates the basic security model is designed to protect information on a computer and its network from being accessed or even modified while still allowing the program to run on that computer.

The founding of the Internet created new requirements for programs, like being free of viruses and Trojan horses. The introduction of Java as parts of dynamical web pages, could multiply the problems with wide spread viruses, because those Java programs were downloaded automatically, frequently and without the knowing of the user. Hence, the early security concepts were focused on these problems.

These requirements lead to the security concepts described in this diploma thesis, mainly in chapter 2. These security concepts include the Java Sandbox, signed classes and many more. Chapter 3 deals with Servlet security, which is connected to the standard Java security, but has some specialized concepts, especially for network data transfer.

1.4 The SAS II project & security

The SAS II (school/pupils' administration software) project allows teachers and other school staff the updating and viewing of specific pupil information. (E.g. The teachers can enter their pupils' marks and also print the school reports at the end of each semester.) The software is distributed, with the data saved in a database on one server for each school. The teachers are able to connect through the schools computer network (mostly Local Area Network - LAN, but also Wide Area Network -WAN). This project is realized with Java.

After defining the database system, the biggest question was how to make secure connections in these networks. Available options with Java are:

Server: (a Linux operated computer)

- A self-written server, either with HTTP or a self-made protocol
- The Apache HTTP Server
- The Tomcat Servlet Engine

Client: (mostly a MS Windows operated computer)

- A self-written client
- A commercial web browser (e.g. Opera, Netscape, …)

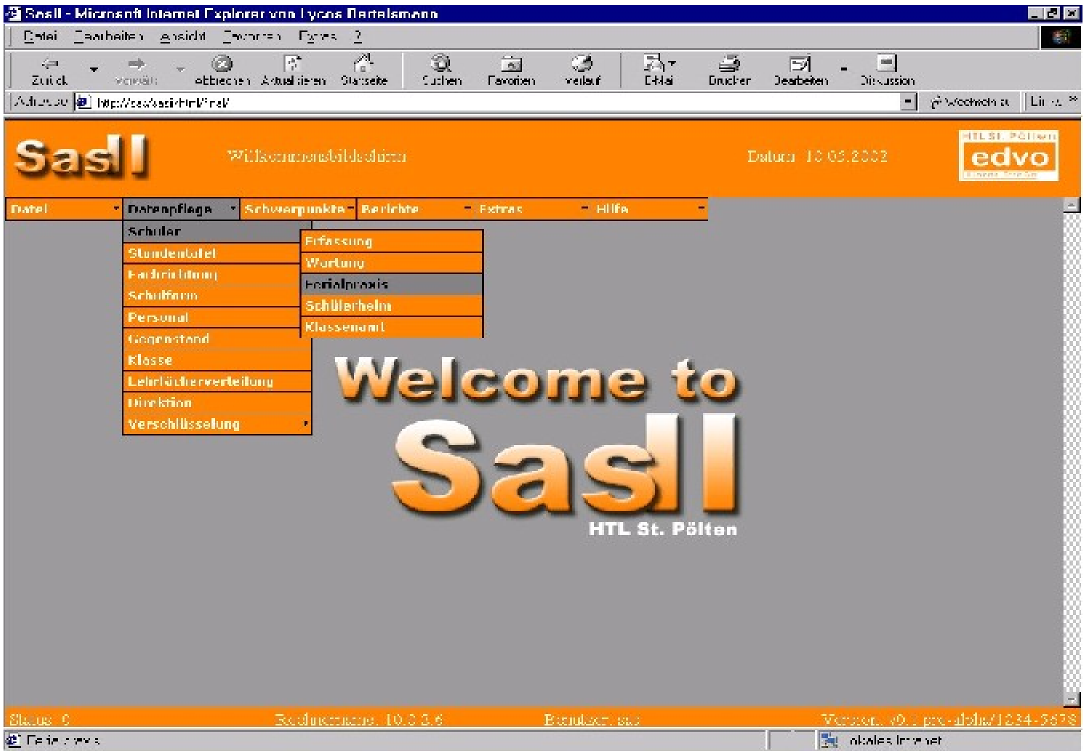
See picture 3.a

Our decision was to have an Apache/Tomcat combined server, and a commercial web browser client. More about the decision finding for server side technologies can be found in the diploma thesis "Evaluation of Server-Side Technologies In The Case of SAS II" by Andreas Lehrbaum, also written in 2001-2002, at the HTBLuVA St. Pölten, deptartment EDVO[10].

The transaction security concepts are defined within the Java Servlets. More about the security concepts in Java Servlets can be found in the respective chapters.

The current user and his rights are stored in the session data (an example can be seen in the Servlet security part of this diploma thesis) and on the database system (which is PostgreSQL). The database allows reducing the written lines of code for security in the Servlets, because checking whether a user is a valid one or not is completely done by the PostgreSQL database. Also the data transferred to the user, via JDBC (Java Database Connectivity) drivers, are retrieved by views. That is a database concept to restrict and/or combine the access to one or more tables. More information about the view-concepts in PostgreSQL can be found in the PostgreSQL documentation[6] and the SQL definition standard.

The connection security is defined as secure Hyper Text Transfer Protocol (HTTPS) - Secure Sockets Layer (SSL) connection, which encodes the whole data transfer between server and client. More information about SSL can be found in the Servlet chapter.



Picture 1.a: The SAS II project

2. Application Security

2.1 Introduction

2.1.1. What is an Application?

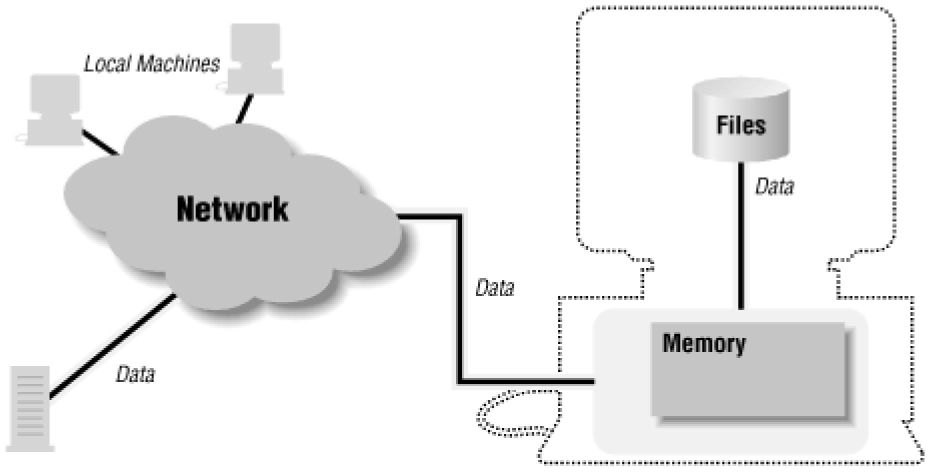
Applications are usually written with the JDK as a main interface. Mainly there are 2 bigger groups of Applications:

- Stand-Alone Applications: *programs for doing several tasks on the user's machine. Comparable to any other program running on the user's computer.*
- Applets: *Java programs that are implemented in web pages, to provide more functionality than the standard HTML code. They usually have no or very restricted access to files, hardware, network connections …*

Both of these types need the Java Runtime Environment (JRE) installed on the computer they should run. The security concepts are the same in all Java programs, though the programmer has the possibility to implement new ones. The basic security concepts are described in this chapter.

2.1.2. Java Sandbox

Most Java Security discussions concentrate on the idea of a sandbox model. Behind this model there is the idea that when programs are hosted on a computer, they get some, not all, abilities and functions provided by the system. This environment provided to the programs, varies with different computers, different users, and also different systems. Within this environment the program runs, (compared to a kid playing in his environment), but there are certain boundaries around it (like the boundaries around a sandbox). The environment may include some functions (i.e. toys for the kids) and system resources, but it is limited. This analogy works better, for close relative (not own) kids playing in a sandbox. They are safe in the sandbox (because it is a small, limited area), but also the things outside the sandbox are safe from the kids (like glass or computers, things kids should not touch, because there is a high risk of damaging).



Picture 2.a: Resources of a computer

The resources of a computer are protected by the Java Sandbox. It does so at a number of levels. There is access to:

- The internal, local memory (the Random Access Memory (RAM))
- The file system of the local machine
- Other computers connected over a LAN
- Applets also have access to the web server they are loaded from. This server could be in the LAN or WAN (the internet).

The data can flow through the whole model, from the user's machine through the network to the hard disk. Each of these resources needs to be protected, and these protections are the basis for Java's security concepts.

The Sandbox can have different sizes, from small (i.e. restrictive) to big (i.e. not restrictive) environments. The minimal Sandbox has access to:

- The Central Processing Unit (CPU)
- The screen
- The keyboard and mouse
- To its own memory

Without these a Java program is not able to run. A minimal sandbox consists of just enough resources to let a program run. The default state of a usual Applet sandbox includes:

- The Central Processing Unit (CPU)
- Its own memory
- Access to the web server, it was loaded from

Some applets might draw to the screen although they do not have the permissions to do so, but that is not a "real" screen resource, it is the web browser's screen and so not of interest to the Java sandbox.

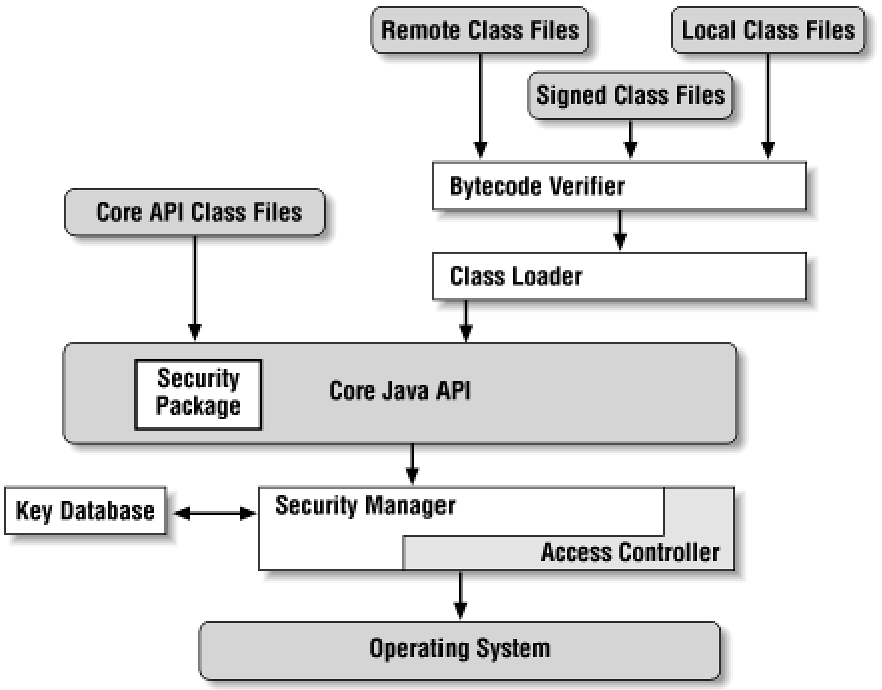
The sandbox, then, is not a one-size-fits-all model. Expanding the boundaries of the sandbox is always based on the element of trust: in some cases, programs are trusted to access the file system; in other cases, they are trusted to access only part of the file system; and in still other cases, they are not trusted to access the file system at all.

2.1.3. Anatomy of a Java Application

A simple Java application consists of just one class, writing to the display. The called HelloWorld example shows that:

```
/**
 * The HelloWorld class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorld {
    public static void main(String[] args) {
        //Display the string.
        System.out.println("Hello World!");
    }
}
```

In this example there is no need for security, although there is of course the Java Sandbox involved. Also the built-in Java Application Security is running when this application is executed. The built-in Java Application Security model is based on the information in the user's `CLASSPATH`. It contains classes that may be loaded; in particular it uses the access controller to provide that security.



Picture 2.b: Anatomy of Java applications

In this figure the typical anatomy of a Java application can be seen. This diploma thesis concentrates on security, so the rectangles play a role in the Java sandbox concept. These elements are in particular:

- The bytecode verifier
- The class loader
- The security manager
- The access controller
- The security package
- The key database (cryptography engine)

More about these elements can be found in their respective chapters.

2.2 Java Language Security

This chapter describes the Java language concepts for security, and how they are enforced. These security concepts consist of:

- Memory security: *It ensures that Java applications will not discern information in the user's memory. This also means that Java applications have their own memory to use and operate.*
- Environment security: *These are mainly the features of the bytecode verifier and the class loader. They usually enforce some of Java's security concepts.*

The following chapters do just apply if the language used is Java. If there is for example native C++ code used, this C++ code might be able to do nearly everything on the user's machine.

2.2.1. Java Memory Security

Within Java every entity – every object reference, every primitive data type – has an access level assigned. These access levels may be:

- private: *The entity can only be accessed by code, contained within the class that defines the entity.*
- default (or package): *The entity can be accessed by code, contained within the class that defines the entity, and also by code that is contained in a class that is within the same package as the class defining the entity.*
- protected: *The entity can be accessed by code, contained within the class that defines the entity, by code that is contained in a class that is within the same package as the class defining the entity, and also by code that is contained in a class that is a subclass, that means derived class, of the class defining the entity.*
- public: *The entity can be accessed by code in any class.*

Assigning access levels to entities is not an exclusive to Java; every object-oriented programming language has that concept implemented. Java, because it is very similar to C++, uses the C++ notations of access levels, though there are some slight differences in the meanings of these.

Java ensures that entities in memory can only be accessed, if permitted. This also prevents these entities from being somehow corrupted. For that reason Java always enforces the following rules:

- Access methods are strictly adhered to: *In Java,* `private` *entities may be treated like* `private` *entities; the intentions of the programmer must always be respected. The only exception is object serialization.*

- No access of arbitrary memory location: *Java does not have pointers, so this one is easy to ensure. For example: It is not allowed to convert (cast) an* `int` *entity into an* `Object` *entity.*
- Final entities: *Entities declared final are considered constant. The problems that could occur if a final entity was changed:*
    - If a `public final` variable could be changed, an Applet could for example change the variables `EAST` and `WEST` in the `GridBagConstraints` class could be changed, every new Applet would than look completely different. This example just shows what could happen; there are of course things that are much bigger security flaws.
    - A subclass could overwrite a `final` method, changing the behavior of it. For example if the `final setPriority()` method of the `Thread` class, would be overwritten, it would defeat that security mechanism.
    - A subclass of a `final` class could be created, with similar problems.
- Initializing of variables: Reading an uninitialized variable has the same effect as reading random memory locations. A malicious program could then declare a lot of big variables to read the data from the computer's memory to prevent that all local variables have to be initialized before use. All instance variables are automatically initialized to a default value.
- Array boundaries: The main effect of checking array boundaries leads Java programs open to fewer bugs and more robust programs. It also has security benefits: Writing out of the boundaries of an array will change the following variable (in the computers memory), which could lead to crashes or also attacks from intruders. (This security leak is not too far-fetched. Netscape once had problems with longer URLs put into

the "Go to" line, which then overwrote then other variables and caused the program to crash.)

- Casting into other objects: If casts were allowed from every class to every other class, a rogue program extension could just cast the secret information into a class it is allowed to read. There are 2 layers of checking those casts:
    - The Java Compiler: Java does not allow casting in other classes than its subclasses or superclasses. In the following example the compiler recognizes the wrong cast and complains about it:

      ```
      CreditCard cc = Wallet.getCreditCard();
      CreditCardSnoop ccsnoop = (CreditCardSnoop) cc;
      System.out.println("Your CC#: " +
      ccsnoop.ccnumber);
      ```

    - The Java Virtual Machine: If we would change the example above to:

      ```
      Object cc = Wallet.getCreditCard();
      CreditCardSnoop ccsnoop = (CreditCardSnoop) cc;
      System.out.println("Your CC#: " +
      ccsnoop.ccnumber);
      ```

      The compiler could not recognize the wrong cast, but the Java Virtual Machine, which knows the real class of the cc object, will throw a `ClassCastException` when the snoop object is assigned to the cc object.

2.2.2. The Bytecode Verifier

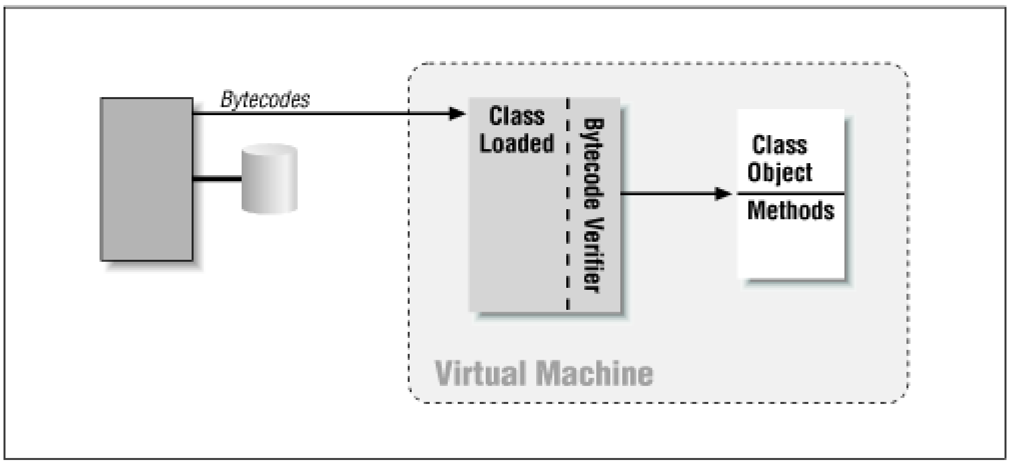
After the compiler generates a Java program from our source code, the Java bytecode of the program has to be run. This bytecode could be transferred over the Internet or a local network, like it does with Java

Applets. But how do we know that the bytecodes we received are actually legal?

Here comes in the need of the Java Bytecode Verifier. Normally it is the second chain (after the compiler) to check and enforce the Java Language Rules. If for example an evil compiler generated bytecode, that exploits security holes by bypassing Java Language Rules, it is still not allowed to run if the Bytecode Verifier did not verify the code. Such attacks are very difficult to achieve, because the attacker needs knowledge in writing a Java Compiler.

There are other ways that show why it makes Java necessary to have a Bytecode Verifier. For example if the `java.lang.String` source file (e.g. setting the array holding the string data public instead of private) is changed, compiled and put back in the JDK folder.

The Bytecode verifier is an internal part of the Java Virtual Machine and has no interface: programmers can not access it and users cannot interact with it. The verifier examines automatically most bytecodes as they are built into class objects by the class loader.



Picture 2.c: The Bytecode Verifier

The Bytecode Verifier, which Sun named in some of its papers "mini-theorem prover", can prove only one thing: Are the given series of Java bytecodes representing a legal set of Java instructions?

Specifically the Bytecode Verifier can prove the following things:[1]

- The class file has the correct format. The full definition of the class file format may be found in the Java virtual machine specification; the bytecode verifier is responsible for making sure that the class file has the right length and other characteristics, which define that this is a valid class.
- Final classes are not subclassed, and final methods are not overridden.
- Every class has a single superclass. (exception:`java.lang.Object`)
- There is no illegal data conversion of primitive data types (e.g., int to Object).
- No illegal data conversion of objects occurs. Because the casting of a superclass to its subclass may be a valid operation (depending on the actual type of the object being cast), the verifier cannot ensure that such casting is not attempted - it can only ensure that before each such attempt is made, the legality of the cast is tested.
- There are no operand stack overflows or underflows. In Java, there are two stacks for each thread. One stack holds a series of method frames, where each method frame holds the local variables and other storage for a particular method invocation. This stack is known as the data. The bytecode verifier cannot prevent overflow of this stack--an infinitely recursive method call will cause this stack to overflow. However, each method invocation requires a second stack (which itself is allocated on the data stack) that is referred to as the operand stack; the operand stack holds the values that the Java bytecodes

operate on. This secondary stack is the stack that the bytecode verifier can ensure will not overflow or underflow.

After the Bytecode Verifier completes his job the Java bytecode is correct, which means it follows the rules and constraints of the Java language. Of course some rules remain unchecked but they will be checked during runtime.

The Bytecode Verifier seems like a really great thing because it prevents malicious attackers from violating Java language rules. But it does not check every class! The Bytecode Verifier in Java 1.1 deemed classes located at the CLASSPATH for trusted classes and did not check them. In Java 1.2 it changed and all classes except the ones in Java's core API (Application Program Interface) are checked.

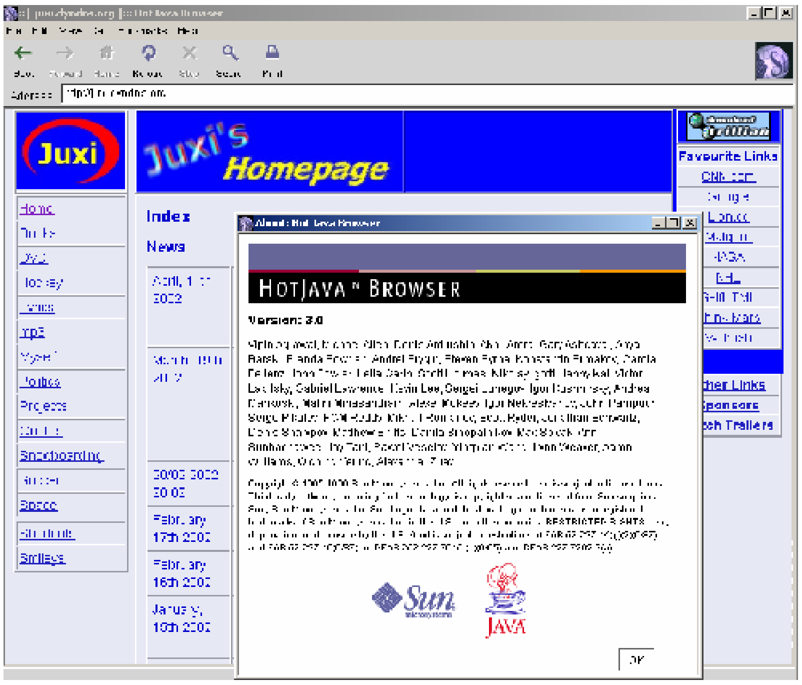
2.2.3. The class loader

The two main roles the class loader plays are:

- Coordinate with the Java Security Manager and/or the Access Controller
- Enforce Java language rules about the namespace Java classes use

2.2.3.1. Coordination with the Java Security Manager and the Access Controller

In order to determine the security policy of a Java program the class loader has to coordinate with the Java Security Manager and/or the access controller. For example if a Java Applet is run in a web browser such as the HotJava (HotJava is a good example because it is also written in Java), it is (normally) not permitted to read a file from the local file system.



Picture 2.d: The HotJava web browser

The browser, HotJava, is allowed to, though it uses the same classes from the Java API. So there has to be something that tells the `java.io` classes to fail in one try but to work in the other case. A by-product of the class loader is the differentiation between these two cases: the class loader gives information about the classes to the Java Security Manager, which is then able to apply the different security policies. More about these different security policies can be read in the Java Security Manager chapter.

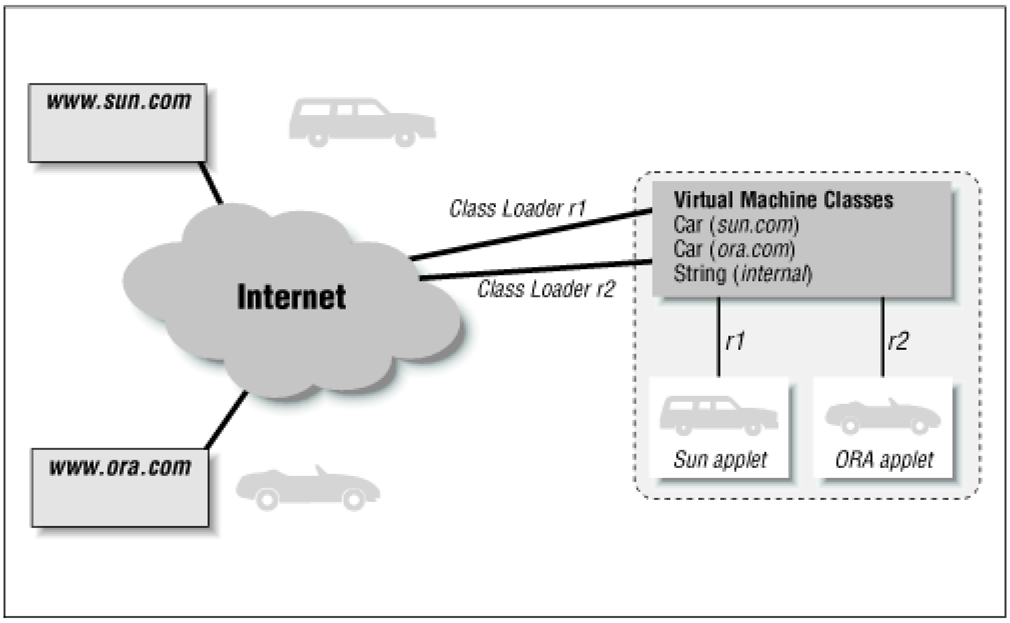
The class loader, since it loaded the class, knows where the class came from (network or local file system), if the class was delivered with a digital signature, and it also knows the exact location from which the class was loaded. All these facts may be used by the Java Security Manager and the access control to define the security policy.

2.2.3.2. Enforcing the Java Namespace Rules

The second role has to do with Java's namespace rules. These rules say that the full name of a Java class is qualified by the name of the package to which the class belongs. This means there is no class called `String` in the Java API, but there is a class called `java.lang.String`. But Java classes do not always have to belong to packages – these are mostly referred to as classes of the default package, though this name is a little bit misleading.

The problem is simple: If a user goes to a web page and loads a class file (e.g. for an applet), goes to the next site and loads a different class but with the same fully qualified name (which means that they are in no package (or maybe even the same)). How should the Java Virtual Machine know what class to execute?

The answer lies in the internal working of the class loader. The class loader is not just one class, which is instantiated and then run in the background of the Virtual Machine. Merely the class loader is every class that is derived from the `ClassLoader` class. When the Java Virtual Machine needs a particular class (e.g. the Car class from the sun site in the following picture) it makes a new instance of one of the class loader classes, which then loads the class and provides information.



Picture 2.e: Different instances of the class loader help to disambiguate the class names

This means that in the example above two instances of the class loader are working; each representing one of the `Car` classes.

Though the example above might be a good reason for class loaders, it could also be solved without them. For example if every company had to use its domain name in reverse order (i.e. com.sun for Sun Microsystems) for the start of their package names, there would not be such problems. There is another reason why Java uses a class loader in its security concepts: Classes that are members of a package have other privileges than classes without a package have.

This means that if all web pages would use "their" package (as shown above with com.sun), evil programs on the web pages could use the same package. Without the class loaders it would have the privilege to access data from other com.sun classes!!

Another idea would be to sign every class, with information that authenticated that it did in fact come from the right site. Though authenticated classes are used in Java, it would not be manageable to add authentication to all classes.

2.3 Java Security Manager

The Java Security Manager is what most people think of when they hear Java security. It is of course a big point in Java's security concept, but as shown here it is not the only one.

On one hand, the Java Security Manager is often summarized by saying that it is there to prevent Java (Applets) from accessing the user's local disc drives or local network connections. On the other hand, the story is more complicated, which leads to a misunderstanding of the Java security concept.

The Java Security Manager is responsible for determining whether many particular operations of a Java program should be permitted or rejected. In general Java applications do not have a security manager, unless the programmer added one, but Java Applets do have one! This leads to a very big misconception: Since Java is said to be secure, users may think every Java program is just as secure as any other, no matter if it is installed locally or running in a web browser. Nothing is further from the truth!

Beginning in JDK 1.2 it is much easier to provide security concepts from the programmer side, than it was in Java 1.1. A default, user-configurable Java Security Manager has been introduced. This default is started with a command line argument when starting the application.

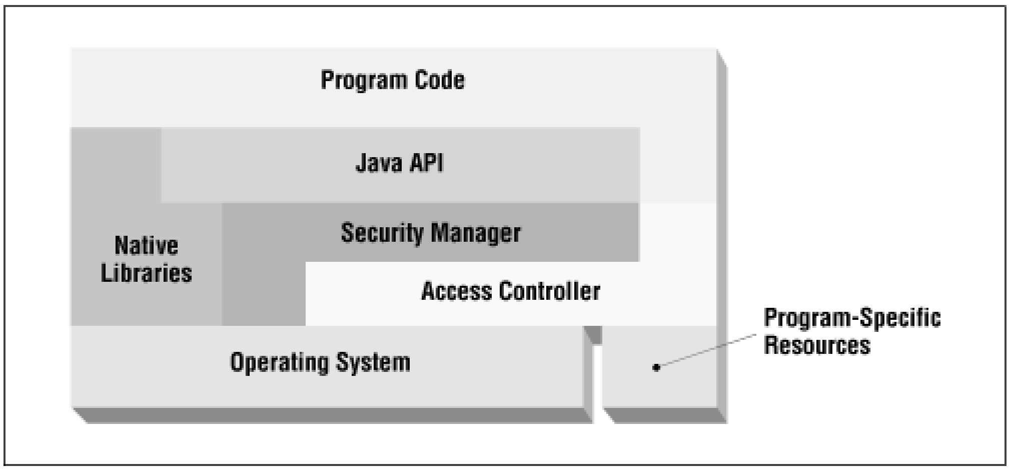
To make that point clear, take a look at the following example:

```
public class MaliciousApplet extends Applet {
   public void init() {
         try { Runtime.getRuntime().exex("rm –rf /"); }
         catch(Exception e) {}
   }
   public static void main(String args[]) {
         MaliciousApplet ma = new MaliciousApplet();
         ma.init();
   }
}
```

If this code is compiled, put on a web server as an Applet and visited by a user, an error will occur reflecting the security violation. If on the other hand it is run by for example the web admin locally on the (Linux or UNIX) web server all files will be deleted! This example shows that it is very crucial for the user to know which Security Manager is in place when a Java program is run.

2.4 The Access Controller

The Access Controller is the mechanism the Security Manager actually uses to enforce its protection policies. The Access Controller can do what the Security Manager can do, so the purpose of it is slightly redundant. This is because the Access Controller was first introduced in JDK 1.2. Before that the Security Manager had to enforce the rules alone. The Access Controller is a much more flexible mechanism for determining policies; it also gives a much simpler method of granting fine-grained, specific permissions to specific classes. That should have also been possible with the Security Manager, but it was just too hard to implement.



Picture 2.f: Relationship of the Java Security Manager and the Access Controller

The operation typically starts through the Program Code into the Java API, through the Security Manager and the Access Controller to finally reach the operating system. In some cases the Security Manager can bypass the Access Controller. The Java Native Libraries are outside the domain of the Security Manager or Access Controller.

The Access Controller is built upon four concepts:[1]

- Code Sources: *an encapsulation of the location from which Java classes were loaded*
- Permissions: *an encapsulation of a request to perform a particular operation*
- Policies: *an encapsulation of all specific permissions, granted to specific code sources*
- Protection domains: *an encapsulation of the code source and the permissions granted to it*

2.5 Cryptography

Cryptography is featured in the Java security package. These classes provide additional concepts and layers of security beyond the standard implementations. These classes play a role in the signing of Java classes. This signing of Java classes and with it the expanding of the Java Sandbox concept, which is the key goal of Java's security concepts, is just one role. These classes may play other roles in secure applications and in the Sandbox concept.

A digital signature, allows authentication of Java classes. A signed class has different security policy than "standard" classes. A signed class usually has bigger latitude in the operations it can perform. Digital Signatures do have another useful ability: executable code with a digital signature means that the company that produced the code (and of course the signature) "vouches for it" in some sense, they indicate a degree of insurance.

In order to use the classes of the security package, programmers do not need a deep understanding of cryptographic theory. On the other hand, one feature of the security package is that different implementations of different algorithms may be provided by third-party vendors, whose programmers do of course need to know the ideas behind the algorithms.

On the JavaOne Conference in 1997, the Java security architect, Li Gong, gave a list of five inequalities within Java's security concept, which might be of interest for programmers: [9]

- Security != cryptography: *Adding cryptography to an application will not make it secure, it is just a tool for building secure systems.*
- Correct security model != bug-free implementation: *Even with great security designs, bugs in the implementation can be exploited by attackers.*
- Testing != formal verification: *Testing is a great idea, but it will not prove that the system is secure.*
- Component security != overall system security: *The system security is a chain where every link can be broken.*
- Java security != applet containment: *A lot of people think about the "applet sandbox concept" when they hear Java security. In truth this is only a small part of the Java security.*

2.5.1. Java Security Package

Much of the Java security package is made up of a collection of engines. As a unit, these engines allow us primarily to create digital signatures--a useful notion that authenticates a particular piece of data. A digital signature can authenticate a Java class file, which provides the basis for a security manager to consider a class to be trusted, even though the class was loaded from the network.

The security package, like many Java APIs, is actually a fairly abstract interface that several implementations may be plugged into. Hence, another feature of the security package is its infrastructure to support these differing implementations.

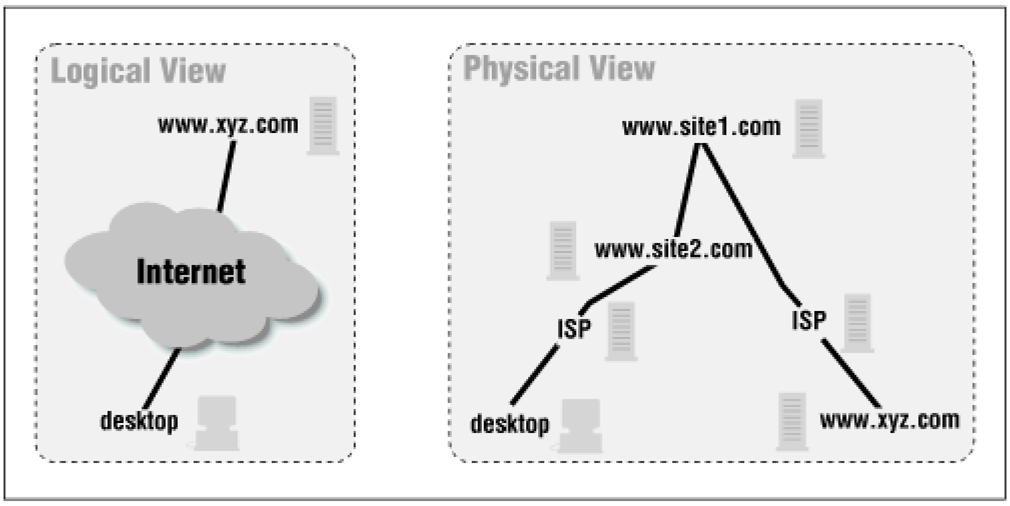
2.5.2. Authentication

The primary goal of the `java.security` classes is authentication. For example if a class is loaded over a network the Java API has to assure two things:

- The site the class was loaded from (author authentication)
- The class was not modified during the transfer (for example from a cracker) over the network (data authentication)

Java, by default, assumes that classes loaded from a network source are not trusted. Their permissions are set in correspondence to that assumption.

When a class is transferred over the internet, the Java application that loads the class (and any other application), does not know how the data got through the internet. This is why in nearly all network plans and other network figures, the internet is represented by a cloud. Every computer, router or other network components on the data's route might change it.



Picture 2.g: Data flow within the internet

Java does want to know if the data is still the same as it was when it was sent from the server. Therefore it has to verify it with the two types of authentication mentioned above.

2.5.2.1. Author Authentication

With Author Authentication the Java program tries to ensure that the data (and class) received was really sent from the server it claims to be. Intruders in Internet Service Providers (ISPs) could change the Domain Name Service (DNS) server/table to fool users. By changing this entry data, which should go to a specific server, goes to them, so they can send back data, where they can claim it is coming from the "original" server. This so called IP or DNS spoofing is an easy way for malicious users to send their data to fooled users.

This is just one reason why Java never trusts classes loaded from networks, by default. Classes, changed like that can with this security policy just annoy the users, because they are different than expected, but can not harm the computers they are running on.

In order to trust a class from a network source, it has to be verified by a digital signature that is sent with the class data, which just states that this class is truly coming from the "original" server.

2.5.2.2. Data Authentication

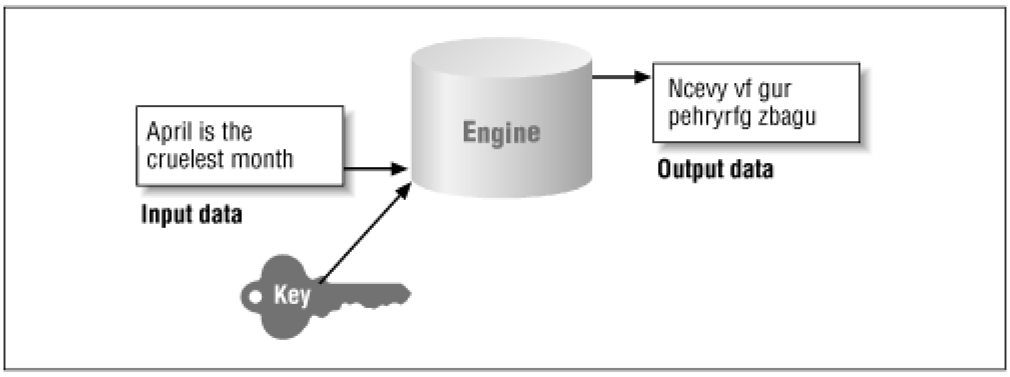
Other problems can also occur when data is transmitted over the internet. For example it could be changed by any network component on the data's route. If an Applet for a web shop sends information about the products a user will buy, the data that reaches the server should be the same as the one the user sent. Intruders at any point of the route could intercept, change and send the other data to the server (or back to the Applet). Therefore Java has to be sure that the data was not changed during the network transfer. This is usually done by a digital fingerprint sent with the data. This does not prevent snoopers from only reading the data transmitted; therefore Java has the ability to encrypt data. Hence, Data Authentication prevents writing of data, but not reading it, by third parties!

2.5.2.3. The Role of Authentication

Not all classes signed and fingerprinted are totally trustworthy. Everybody could sign a class. Hence, Java Authentication is not to allow special permissions to every signed class. It provides more information about the class for the user or administrator, so that they might then know which policy they use for the class. For example that any classes, that come from www.mycomp.com, have the right to read all files in the home directory of the user, but only the files in the home directory.

Authentication does NOT solve any problem; it is just a tool for pursuing solutions to those security problems.

2.5.3. Engine Classes



Picture 2.h: What is an engine?

To allow encryption, for authentication or any other purpose in a Java application, Java provides various engines. An engine is a class or part of the class, that gets input data, (and sometimes a key), encodes it with a cryptographic algorithm, and puts back the encoded output data.

Cryptographic engines can have the following features:

- Get input data
- Sometimes get a key
- Send back the output data
- Non symmetric output (which means that the output data can not be converted back to the input data, with the same engine)
- Differences in the size of input and output data

In the Java security package, there are two standard cryptographic engines: a message digest engine and a digital signature engine. In addition, for some users, an optional engine is available to perform encryption. Finally, because keys are central to the use of most of these engines, there is a wide set of classes that operate on keys, including engines that can be used to generate certain types of keys. The term "engine" is also used within the security package to refer to other classes that support these operations

2.5.3.1. Message Digests

Message Digests are the simplest of the standard engines, provided by Sun, and so a good start for examination. They also are the first link in creating and verifying a digital signature, but there are certain limitations on the security of a message digest that is transmitted along with the data it represents. Sun provided one single class for message digests in its API: `public abstract class MessageDigest extends MessageDigestSpi` (in Java 1.1 MessageDigest is just derived from Object), which implements operations to create and verify a Message Digest.

Like all engines in the java.security package, `MessageDigest` is abstract; it just defines an interface that all message digests must have. With that the concept programmers can make new engines with just this interface implemented, but the usage (from the application) will still be the same.

The message digest by itself gives some comfort about the state of the data it represents, but it does not provide a completely secure system. The message digest could be used with a pass phrase to make it more secure (that is also called Message Authentication Code - MAC).

Example:

```
// define the input data byte []
MessageDigest md = MessageDigest.getInstance("SHA");
md.update(inputData);
byte []digest = md.digest();
//after the digest call the md obj. is reset and can be used again
```

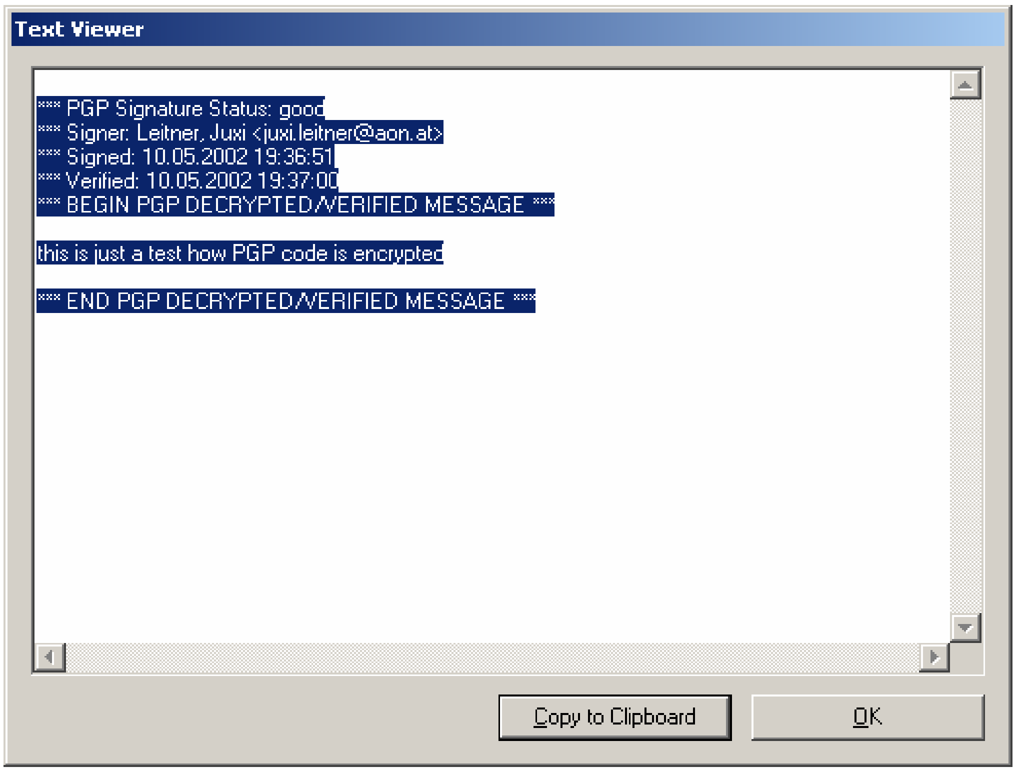
Message Digests could be used to transfer passwords from clients to servers, instead of sending the "plaintext" password over the network.

Another use of message digests is in the `DigestInputStream` and the `DigestOutputStream`. These high level streams allow the usage of one message digest for the whole data transfer.

2.5.3.2. Keys & Certificates

Keys are necessary components in many cryptographic algorithms – in fact keys are required to create and verify digital signatures. Today private and public keys are mostly used in a digital signature.

Certificates are used to authenticate keys; if keys are transferred electronically they are often embedded within certificates. In the following picture a PGP (Pretty Good Privacy) message is decrypted and verified. The first lines state the signer and the signature; this is the certificate.



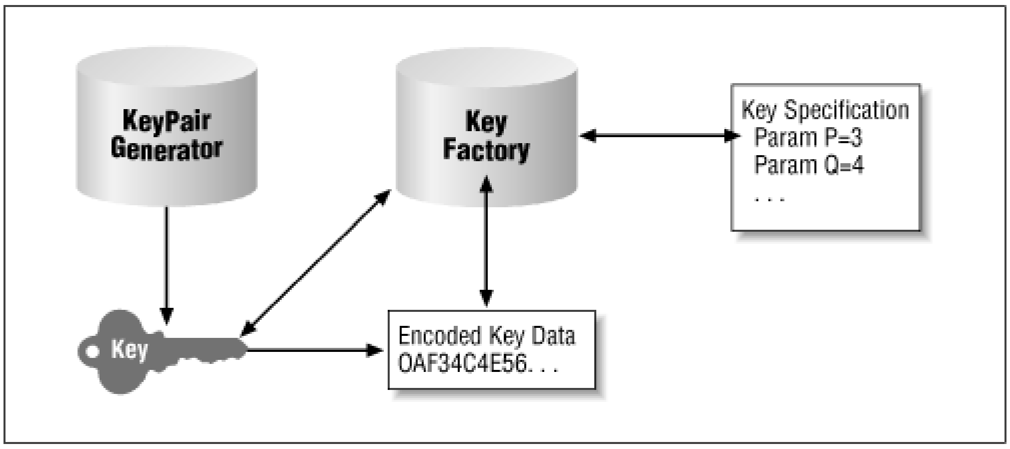
```
*** PGP Signature Status: good
*** Signer: Leitner, Juxi <juxi.leitner@aon.at>
*** Signed: 10.05.2002 19:36:51
*** Verified: 10.05.2002 19:37:00
*** BEGIN PGP DECRYPTED/VERIFIED MESSAGE ***

this is just a test how PGP code is encrypted

*** END PGP DECRYPTED/VERIFIED MESSAGE ***
```

Picture 2.i: Certificate of a PGP message

Keys and certificates are normally associated with a specific person or company, to authenticate them. How keys are stored, transmitted and shared is an important topic in Java's security package.

There are two engines that operate on keys in the Java API:

- The `KeyPairGenerator` class: can produce one or more pairs of keys
- The `KeyFactory` class: translates between key objects and their external representations.



Picture 2.j: The key classes in Java

Various classes support the notion of keys within Java:

- The `Key` Interface: represents the key model. It provides standard methods for getting information about the key, and extends the `Serializable` interface. There are two additional interfaces, `PrivatKey` and `PublicKey`, both extend `Key`. There is no difference; those classes are just for type convenience.
- The `KeyPair` class: contains a private and a public key. It is the only class in the Java standard API that extends the abstraction of `Key`. The simple constructor just gets a private and a public key. A key pair should always be initialized with both keys, but nothing prevents the programmer from sending `null` as parameter for one of the keys.
- The `KeyPairGenerator` class: Generation of key pairs (private and public key) is integrated in the standard API of Java. Because the generation is a very time-consuming operation, it is not performed very often; but it does not have to be done very often either. Initialize the key pair generator to generate keys of the defined strength. Typically that is the number of bits used for the key. Key pairs usually also require a "random seed" to assist them. In Java the standard is the `SecureRandom`

class, which provides those random numbers for the key generation.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance
("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair();
```

- The `KeyFactory` class: More often than generating own keys, programmers are confronted with obtaining them from the local file system or a network. This class provides functionality to convert a key object to a known external representation.

When public and private keys are generated, the public key needs to be published. If a document is digitally signed (using your private key), the recipient of the data needs your public key to verify the signature.

Certificates solve this problem by having a well-known entity (certificate authority, or CA) that is able to verify the public key sent over the network. A certificate can give assurance that the public key does indeed belong to the entity that the CA says it does. However, the certificate only validates the public key it contains. It does not say anything about how trustworthy the entity is!

A certificate contains three pieces of information: [1]

- The name of the entity
- The public key associated with the entity
- A digital signature (from the issuer of the certificate)

But how are the digital signatures of the CA's authenticated? This is a problem that has not been solved yet! The web browsers solve it with providing digital signatures with their installation, so that those digital signatures of CA's are already on the user's computer. The problem is that this solution is not airtight, what means if the data is changed when you download a browser, malicious users or programs could use that security leak. There are some proposals in solving this problem, but for now this is the only way.

In Java 1.1 Certificates were stored within an interface in the `java.security` package. With the release of Java 1.2 this interface became deprecated and a new class was introduced `java.security.cert.Certificates`, which is now the standard class for any certificates.

2.5.3.3. Key Management

The problem with key management turns out to be a hard one to solve: there is no universally accepted approach to key management. Java provides features to assist key management, but all key management techniques remain very much work in progress. Key management remains application specific.

The purpose of the key management is two-fold. When a program needs to sign data, the key management needs to provide the private key for the code that creates the data. When the application needs to verify a digital signature it needs a public key that will be used for authentication. A key management exists primarily to provide these services mentioned.

There are 3 elements in a key management system:

- Keys

- Certificates
- Entities: implemented by the `Identity` class

With Java 1.2 there is a tool called `keytool`, that allows to store individual private and public keys in a database, usually referred to as `keystore`.

Identity Class

The `Identity` class implements the interfaces `Principal`, an interface that only abstracts the idea that principals have a name, and the `Serializable` interface. It only holds public keys; private keys are stored in another object (`Signer` object). The information, `Identity` objects then hold, are:

- A name (from the `Principal` interface)
- A public key
- An information string (optional)
- An identity scope (optional, not used in Java 1.2)
- A list of certificates

Signers

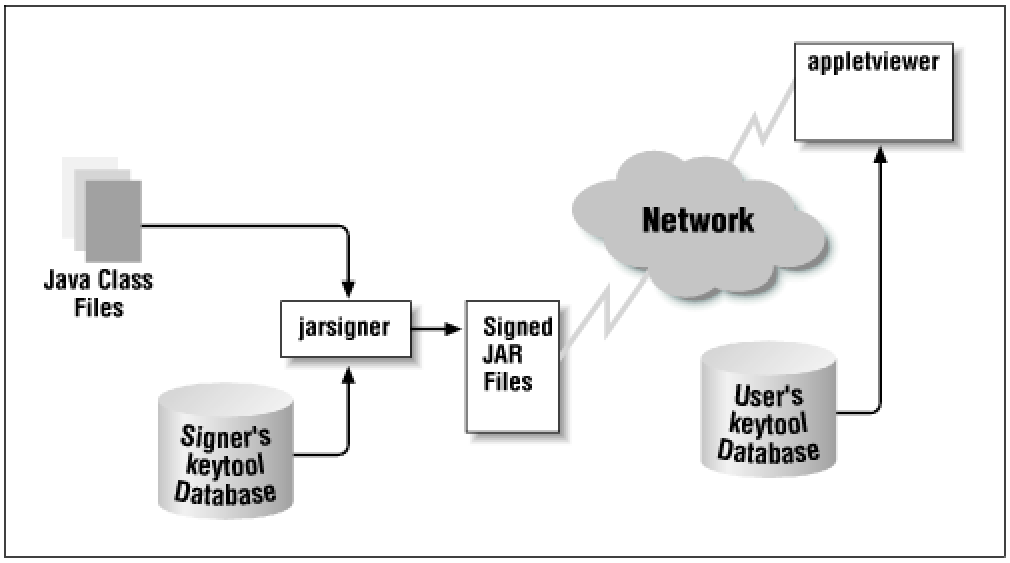
In order to create a digital signature the program needs a private key. An object that holds a private key is the `java.security.Signer` class, which extends the `Identity` class. It has just 2 more functions than an `Identity`:

- public PrivateKey getPrivateKey()
- public final void setKeyPair(KeyPair kp)

The KeyStore class

The `keytool` program operates upon a file (or other storage system) containing private keys and certificates for those keys. This file contains a set of entries, with the following attributes:

- An alias, for easier reference
- One or more certificates that vouch for the identity
- A private key (optional)



Picture 2.k: The role of the keytool database, in the creation and execution of a signed JAR file

The class in a Java program that represents the keytool database is the `KeyStore` class. The local keytool database is stored in the user's home directory, in the file `.keystore`. Though this is the standard, the `KeyStore` class does not automatically load it, when it is generated by the `getInstance()` method. A `KeyStore` object is created empty. To load data the programmer has to call the `load(InputStream is, String pwd)` method. To write it back to the hard disc a call of `store(OutputStream os, String pwd)` is necessary.

2.5.4. Encryption

Encryption is a tool to protect secrets and to provide more security. Java programs can encrypt data, or rather streams, that write to hard discs or network sockets. Java classes that provide methods and solutions are mostly implemented in Java's Cryptographic Engine (JCE), which is strictly limited by US export restrictions. Cryptography has always been part of military use, so the US government has made those restrictions for any encryption algorithm or anything that has to do with it.

One of the most used terms in encryption is cipher. A cipher encrypts or decrypts data. They are categorized in:

- Algorithm based Ciphers:
  - Symmetric or private key ciphers: data encrypted with this cipher can also be decrypted with it
  - Asymmetric or public key ciphers: data encrypted with this cipher cannot be decrypted with it
- Hybrid Ciphers: are protocol based. They use private and public key algorithms for en/decryption.

Ciphers are usually used as block ciphers or stream ciphers (or block ciphers used in stream cipher mode (CFB)). For example asymmetric ciphers are usually block ciphers.

2.5.4.1. Block Ciphers

A block cipher splits the data to en/decrypt up into blocks of fixed size (usually 64 bits). Problems occur because mostly the data will not fit in those blocks (the last block will not be filled completely). Therefore block ciphers need padding.

Padding schemes specify exactly how the last block of data is filled before it is encrypted. A standard for padding is for example the public key cryptography standard number 5 (PKCS#5).

Block ciphers have various modes, to determine how these blocks are en/decrypted. For example: the electronic code book (ECB) mode, the cipher block chaining (CBC) mode, the propagating cipher block chaining (PCBC) mode, …

2.5.4.2. Algorithms

With its class abstraction and encapsulation Java allows programmers to change the algorithms for en/decryption by just changing one parameter in their program. They do not have to completely rewrite the program. Sun's JCE comes with three predefined algorithms.

The class that encapsulates the cipher algorithms is `java.crypto.Cipher`, which implements the `Cloneable` interface. It can encapsulate both asymmetric and symmetric algorithms.

To use a cipher 3 steps have to be done:

- Get an instance of a `cipher` by calling the `getInstance()` method
- Initialize the `cipher` object for encryption or decryption, by calling the `init()` method
- Encrypt or decrypt the data by using the `update()` or `doFinal()` method

2.5.4.3. Cipher streams

Stream ciphers do not encode blocks, they encode every single character. They are used where it would be inconvenient or impossible to wait for the data to get into buffers to encode or decode it then. Cipher streams are realized in the classes `CipherOutputStream` and `CipherInputStream`. There a `cipher` object is associated with an input or output stream.

2.5.4.4. Hybrid Systems[9]

Hybrid systems combine the strengths of symmetric and asymmetric ciphers. In a hybrid system, an asymmetric cipher is generated for authentication and data integrity, and a symmetric cipher is used for confidentiality. Symmetric ciphers are faster then asymmetric ciphers, so it makes sense to mostly use symmetric ciphers for messages and/or conversations. Likewise, asymmetric ciphers are well suited to authentication and session key exchange.

The most widespread hybrid standards are:

- Pretty Good Privacy (PGP): PGP is software that tried to bring strong cryptography to the masses. It encrypts messages by using a combination of symmetric and asymmetric ciphers. It provides safe transport of those encrypted messages over an insecure network. PGP provides authentication, data integrity, and confidentiality.
- Secure/Multipurpose Internet Mail Extensions (S/MIME): S/MIME is a standard for cryptographically enhanced email. It places the message in a "digital envelope". The message itself is encrypted used a symmetric cipher, while the session key is created by an asymmetric cipher.
- Secure Sockets Layer (SSL): see chapter 3.5.4
- Secure Electronic Transaction (SET): The SET standard was developed by VISA and MasterCard to encourage e-commerce. In theory it works like SSL, while implementations differ a lot.

2.6 Signed Classes & Digital Signatures[1]

One of the primary applications of digital signatures in Java programs is to create and verify signed classes. Signed classes allow the expansion of the Java sandbox in 2 ways:

- The policy file can define, that classes coming from a specific source have to be signed by a particular entity, before the access controller will grant permissions to it. Such an entry in the policy file would look like: (grants permission to read and write any local files to classes coming from "http://juxi.dyndns.org/" only if the classes were signed by "JL")

```
grant signed by “JL”, codeBase “http://juxi.dyndns.org/”
{
      java.io.FilePermission “-“, “read,write”;
}
```

- The security manager can cooperate with the class loader, to determine whether the class is signed or not. The security manager is then free to grant permissions based on its internal policy.

There are three parts needed to expand the Java sandbox with signed classes:

- Functionality to create signed classes (e.g. `jarsigner`)
- A class loader that is able to deal with digital signatures (e.g. `URLClassLoader`)
- A security manager or access controller that grants the permissions based on the digital signature. (e.g. the default access controller)

2.7 SAS II: Jakarta-Tomcat security

In our project Application security was not a main problem. Of course a stable and not malevolent program would be the best, but the functionality played a much bigger role. We decided to use a tool named “Jakarta-Tomcat”, as our Servlet environment. It’s produced by the Apache Software Foundation (http://jakarta.apache.org/tomcat/). The main security concepts in this project were used for the connection, the Servlets and the database.

Reasons for selecting the Jakarta Tomcat Engine can be found in the diploma thesis „Evaluation of Server-Side Technologies In The Case of SAS II" by Andreas Lehrbaum, also written in 2001-2002 at the HTBLuVA St. Pölten, Abteilung EDVO[10].

3. Servlet Security

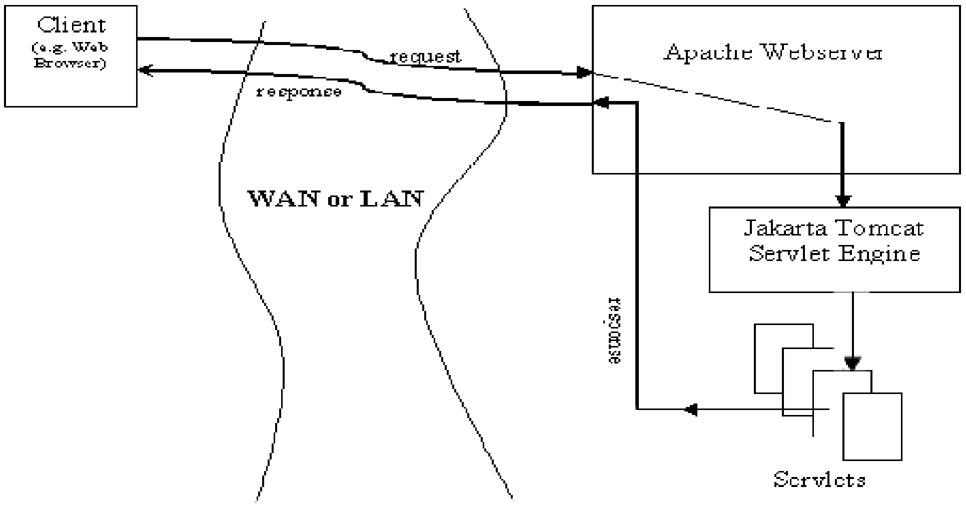
Servlet security is focused on authentication, confidentiality, and integrity. Those main points are split up in various implementation concepts. Those concepts are presented and explained on the following pages. How we use them in our project is also mentioned at the end of this chapter.

3.1 What is a Servlet?

The main question to be answered in this topic will be, "What is a Servlet?". The easy answer would be, "small Java programs running on a web-server", but that is not all Java Servlets can do. They are protocol- and platform-independent enabled components for servers. They have built in support for the request-response paradigm. Usually they are used for generating dynamic HTML content, data viewing and changing ability, and web page generation techniques.

Servlets run inside the server so they do not need a graphical user interface (GUI). But they are also known as counterpart to Java Applets on the server side. Those Applets are Java programs, which are downloaded, on demand, to the server part which needs them.

Clients for Java Servlets can range from simple HTML forms (like we used in our project) to highly-sophisticated Java Applets. Servlets usually use some kind of data storage, such as files or databases. We use both files and databases in our project.



Picture 3.a: Servlet Architecture

The files are in use for XSLT (the Style sheet Transformers for XML – Extended Markup Language) [more about that can be read in the diploma thesis "Evaluierung webbasierter Drucklösungen in Bezugnahme auf das Projekt SAS II" by Andreas Fellnhofer, also written at the HTBLuVA St. Pölten, Abt. EDVO, in 2002] and static HTML pages, whereas the PostgreSQL database is used for saving the data (about pupils, schools, teachers and grades)[10], and security purpose too, as already mentioned in chapter 1 they completely cover the user authentication process.

Servlet are flexible enough to provide standardized services such as static HTML files provided through HTTP (or HTTPS) protocols, proxy servicing, and customized multi user services (especially used at web shops or other costumer based services). They are also able to provide dynamic content to use in e.g. search engines, web based order entry or other database saved content viewing.

Java Servlets are used for middle tiers of distributed applications too. If so they turn into clients for other services, written in any language.

Servlets are for example regularly used in connection with JDBC^{TM} (Java Data Base Connectivity), to access data from relational databases. Communicating with other services may call for alternate software packages, as required by those applications. All those packages should be signed classes. Other security concepts like the Sandbox or the Class Loader also ensure that those connections will not "harm" the server, e.g. force to crash it or include security leaks, e.g. no user authentication.

The Java Servlet API (Application Program interface) is a Standard Java Extension API. This means that while it is not part of the core Java framework, which must always be part of all products bearing the Java brand, it will be made available with such products by their vendors as an add-on package. Sun has provided a package which may be used to embed Servlet support in other web servers, including Apache and Microsoft's IIS. Servlets were initially supported in the Java Web Server from Sun. [2]

Servlets can also be used in different modes such as (not all servers support those):

- Filter chains: *Servers can chain Servlets together*
- Specialization: *Servlets may be specialized to support protocols such as HTTP*
- HTTP based applications: *more efficient, portable, complete replacement for CGI*
- Server Side Includes: *Servlets may be used with HTML server side includes*

In all those modes security leaks can occur, but are mainly covered by the concepts already described in chapter 2.

3.1.1. Servlet Lifecycle

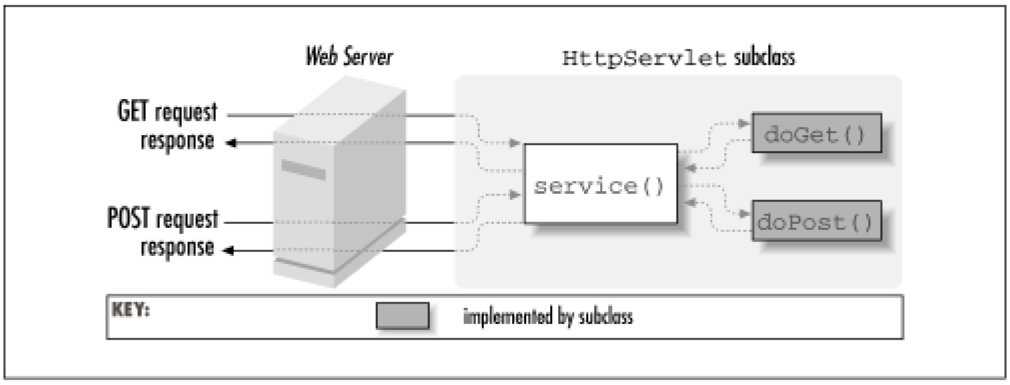
Servlets are dynamically loaded, although usually the servers provide administrative options to force the loading at the start of the server. The Servlets are loaded like any other Java classes, which may be loaded from trusted internet sources or the local file system. This allows increased flexibility and easier distribution of them in a network.

Servers vary in the way how and when they load Servlets. Usually following a request to the web server, the request will be mapped to a Servlet, which may be loaded first. The usual way to map is: [2]

- Server administrators might specify that some kinds of client requests always map to a particular Servlet. For example, one which talks to a particular database.
- Server administrators might specify that part of the client request is the name of the Servlet, as found in an administered Servlets directory. At many sites, that directory would be shared between servers which share the load of processing for the site's clients. At the SAS II project we use that "directory mapping", but not with the multi server feature.
- Some servers may be able to automatically invoke Servlets to filter the output of other Servlets, based on their administrative configuration. For example, particular types of Servlet output may trigger post processing by other Servlets, perhaps to perform format conversions.
- Properly authorized clients might specify the Servlet which is to be invoked, without administrative intervention.

After invoking a Servlet there are usually 3 methods called during the life cycle of the Servlet: [2]

- init: *Servlets are activated by the server through an init call. Servlets can, if needed, provide their own implementation of this call, to perform potentially costly (usually, I/O intensive) setup only once, rather than once per request. Examples of such setup include initializing sessions with other network services or getting access to their persistent data (stored in a database or file).*
- Requests: *After initialization, Servlets handle many requests. Each client request generates one service up call. These requests may be at the same time; this allows Servlets to coordinate activities among many clients. Class-static state may be used to share data between requests.*



Picture 3.b: Service calls to Servlets

- destroy: *Requests are processed until the Servlet is explicitly shut down by the web server, by calling the destroy method. The Servlet's class may then be removed from memory by the garbage collector.*

3.2 Servlet Output (HTML)

Usually Servlets generate HTML text directly, since it is easy to do so with standard Java Output classes such as `java.io.PrintWriter`. Normally there is no need to dynamically modify or generate HTML pages.

Other Java HTML generation approaches could also be used, such as Multi-Language Sites, which serve pages in more than one language (e.g. English and German) to localize the web page, or other dynamic web content generation packages, which could be implemented by themselves. Servlets may also be invoked by Servers with Server Side Includes (SSI) functionality. This means that the Servlet is called by a specific tag in the `.shtml` file. Parameters are provided by tags too, similar to the standard HTML4.0 `<embed>` tag. The output of the Servlet is then directly into the HTML file, which means that the ability to include tags is given but the standard HTML start tags are already set.

Using these facilities, HTML-aware Servlets can generate arbitrary dynamic web pages.

3.3 Servlet Input (Parameters) [2]

Typical Servlets will accept input parameters from various resources, such as:

- The input stream of a request, perhaps from an applet
- In the URI (Unified Resource Identification) of the request
- From some other Servlet or network service
- By using parameters passed from an HTML form *(like we did in the SAS II project)*

Those parameters will be used to generate HTML-formatted responses. The Servlet will often connect one or more databases, or other data with which the Servlet has been configured, when deciding what exact data to return with the response.

3.4 Cookies and Sessions

The HTTP is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext. One of the main problems with it is to provide authentication and the ability to have special user data. That is also a security problem, because "customers" do not want to transfer e.g. their credit card number over the internet, without security. Therefore the concepts of Cookies and Sessions were introduced to the Java Servlet Development Kit (JSDK). With these features, Java now has the ability to store user information at the server and identify a special user. Also security has been included to provide authentication and the ability to transfer data via HTTPS.

3.4.1. Cookies [3]

Cookies are a general mechanism which server side connections (such as Java Servlets, CGI scripts, etc.) can use to both store and retrieve information on the client side of the connection. The addition of a simple, persistent, client-side state significantly extends the capabilities of Web-based client/server applications. [4]

Cookies are just a mesh of data, transferred from the server to the local web browsing client. Once transferred to the client, it is used in every client request on the server. This concept allows the identifying and the logging the user's actions on the server. The basic buildup of Cookies exists of:

- name
- value
- optional parameters, like comments or the validation period

Cookies also have some disadvantages. They are very insecure, because they are just stored on the client computer, and so may be easy manipulated by the client or malicious programs. Furthermore they are restricted by amount and size by most of the clients. And they are usually just transferred with the HTTP GET statement, which is easy to intercept.

In Java Cookies can be generated with the `javax.servlet.http.Cookie` class. Names of Cookies are restricted by RFC 2109, a standard, defining that names can only be alpha-numeric, and may not start with a $-sign and may not include any white-space characters, like spaces or tabulators.

A much bigger disadvantage is that they were used to track user activities and also patterns. Those actions were officially attacked by organizations trying to protect the privacy of page visitors and customers. Therefore they should NOT be used for dynamic web content anymore.

Problems with Cookies can also occur if used with cached HTTP servers or Proxy Connections.

3.4.2. Sessions

When an HTTP client interacts with a Servlet, the state information associated with a series of client requests is represented as an HTTP session and identified by a session ID. The Session Manager, in our project implemented within the Tomcat Jakarta Servlet Runner, is responsible for managing HTTP sessions, providing storage for session data, allocating session IDs, and tracking the session ID associated with each client request through the use of cookies or URL rewriting techniques.

The Session Manager can store session-related information in memory in two ways:

- In application server memory (the default). This information cannot be shared with other application servers.
- In a database shared by all application servers. This is also known as persistent sessions or session clustering.

Persistent sessions are essential for using HTTP sessions with a load distribution facility. When an application server receives a request associated with a session ID that it currently does not have in memory, it can obtain the required session state by accessing the session database. If persistent sessions are not enabled, an application server cannot access session information for HTTP requests that are sent to servers other than the one where the session was originally created. The Session Manager implements caching optimizations to minimize the overhead of accessing the session database, especially when consecutive requests are routed to the same application server.

Storing session states in a persistent database also provides a degree of fault tolerance. If an application server goes offline, the state of its current sessions is still available in the session database. This enables other application servers to continue processing subsequent client requests associated with that session.[5]

A big advantage of sessions in comparison to Cookies is that big data and or information can be accessed and stored. There is no limit of data stored in a session. Also the data is NOT transferred, as with Cookies, it is stored at the server, so no "malicious" programs can access it on the user's computer.

A problem with sessions could be that the session objects are only terminated if the whole session is terminated. This means if users do not log out and destroy the session, the data is usually stored for half an hour (after that time unused sessions will be destroyed by the Servlet environment e.g. the Tomcat Servlet Engine), which could block the servers memory if there is much data stored and if there are a lot of different sessions opened. This problem is usually bigger during implementation and testing of the produced "program" (the term program might be a little misused for the dynamically generated web pages. Try to imagine an online banking system or an online shop), so if there are no problems during these phases, there should be none while the system is running.

To transfer the needed session ID from the server to the client and back two concepts are used: [3]

- Sending it with a cookie
- Sending it with URL rewriting

3.4.2.1. With Cookies

The session ID is stored in a Cookie at the Client computer. The name of the Cookie is JSESSIONID. With every client request to the server this Cookie is sent. If a new session is generated, because there is no old one or the old one is not valid anymore, it is only valid from the time where the client sends back (and implicit accepts) the generated, new ID to the server. The programmer is not allowed to generate this Cookie; it is an integrated mechanism in the Servlet concepts. To check if there is a session a programmer just has to call the function `getSession()` with the parameter `true`, which then creates a new one if none existed, with the parameter `false` it returns whether there is a valid session or not. Problems with this method occur when the client does not support cookies.

More about cookie generation and parsing can be found in the chapter Cookies (3.4.1).

3.4.2.2. With URL rewriting

At sessions the client and server have to transfer the session ID, which is just a "long", unique number and character key, which identifies the session on the server. This information has to be transferred from the server to the client and the other way round. I have just discussed the method based on Cookies. The methods for the URL rewriting are the same, but the session ID is not stored in a Cookie, it is just a part of the URL sent from the Client to the Server and back. Such an URL might then look like:

```
http://servername/servlet/servletname;jsessionid=12345a
```

For automatically generating such URL's Java provided some methods in its Java Servlet API. One of these methods is `encodeRedirectURL(String)`.

The reason why URL rewriting did not succeed in bigger projects, although it is the "cleaner" and "better" concept, is that it is very extravagant, and if just one programmer forgets to use encoded URLs the whole session is lost for the client. Another fact is that most of the newer browser generations understand and accept Cookies.

3.5 Servlet Security Concepts

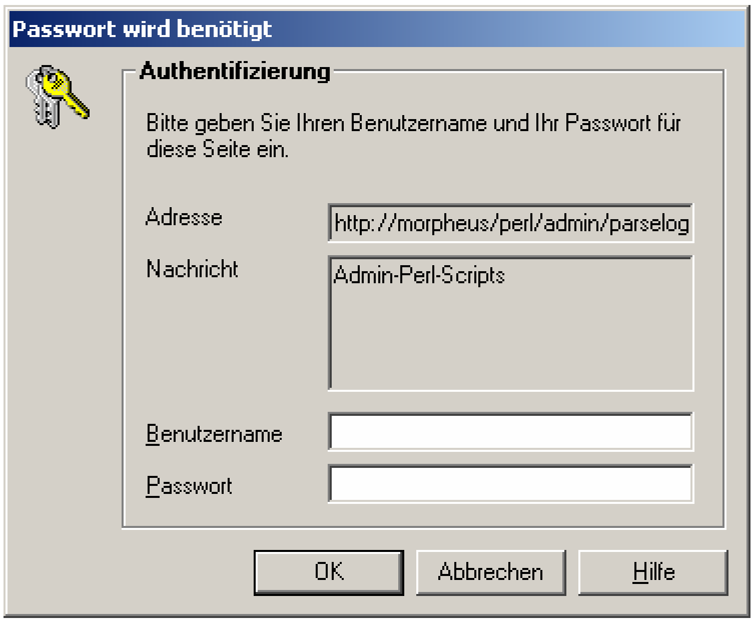
One of the main points of a Java Servlet is its access to user specific data. This data is NOT the data stored at the user's computer, examples for user specific data are the user name, used during login at a web page, or a basket or cart, at e-shopping web pages. Servlets usually do not have access to the user's computer and file system, the only exception might seem Cookies, but that is no real file system access by Servlets. The file system access for Cookies is controlled by web browsers.

One problem of the internet is that it has its share of "fiendish rogues". As companies place more and more emphasis on online commerce and begin to load their intranets with sensitive information, security has become one of the most important topics in web programming. Internet users, try to get "into" a company's network to get information about their information. This could "just" be company data like accounting or market data, but could also be customer data, like credit card numbers and solvency. One thing those "intruders" use to get data are just dynamically generated web pages, e.g. by Servlets, CGI scripts, active server pages (ASP), and various other concepts. They sometimes get data they want but should not be allowed to view by just changing the parameters they send to such programs. That is a very easy task, but it is also very easy to minimize that problem. Concepts for solving such problems are described in this chapter.

If Servlets are used with secure protocols such as SSL, the identifying of the peer is reliable. Servlets, based on the HTTP protocol, do also have the abilities to access various HTTP authentication types, but they do not have secure data traffic on the network.

3.5.1. HTTP Authentication

The HTTP protocol provides built-in authentication support — called basic authentication — based on a simple challenge/response, username/password model. With this technique, the web server maintains a database of usernames and passwords and identifies certain resources (files, directories, Servlets, etc.) as protected. When a user requests access to a protected resource, the server responds with a request for the client's username and password. At this point, the browser usually pops up a dialog box where the user enters the information, and that input is sent back to the server as part of a second authorized request. If the submitted username and password match the information in the server's database, access is granted. The whole authentication process is handled by the server itself.


Passwort wird benötigt

Authentifizierung

Bitte geben Sie Ihren Benutzername und Ihr Passwort für diese Seite ein.

Adresse: http://morpheus/perl/admin/parselog

Nachricht: Admin-Perl-Scripts

Benutzername:

Passwort:

Picture 3.c: A basic http authentication dialog (Opera 6.01)

Though the user has to authenticate to get the data, the data traffic is NOT secured, which means that any interceptor can read the data, because it is just "in plain text", but combined with the HTTPS techniques, which encode the whole data traffic it is a simple way to secure the access to data.

3.5.2. Form-Based Authentication

Servlets can also perform authentication without relying on HTTP authentication, by using HTML forms instead. Using this technique allows users to enter a site through a well-designed, descriptive and friendly login page. It has the same concepts as the HTTP (basic) authentication, but allows the web page designers to make the login procedure look like the corporate identity of the web page.

3.5.3. Custom Authentication

Normally, client authentication is handled by the web server. The deployment descriptor tells the server which resources are to be restricted to which roles, and the server somehow manages the user/group to role mapping. Custom Authentication allows the programmers to implement "their" authentication concepts in their Servlets, CGI-scripts …

3.5.4. Secure Sockets Layer (SSL) – secure Hyper Text Transfer Protocol (HTTPS)

Digital certificates encrypt data using Secure Sockets Layer (SSL) technology, the industry-standard method for protecting web communications developed by Netscape Communications Corporation. The SSL security protocol provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. Because SSL is built into all major browsers and web servers, simply installing a digital certificate turns on their SSL capabilities. The SSL standard is an extension to the standard HTTP protocol and uses the Private Key Infrastructure (PKI) to assure secure internet transactions. Read more about the PKI in the Cryptography chapter.

A measurement of the strength of the SSL encryption is the length of the key that is used. The longer the key the harder it is for "hackers", "crackers" or other interceptors to break the encryption. The two most used strengths are 40 bit and 128 bit. 128 bit encrypted transactions are trillions of times stronger (i.e. harder to break) than 40-bit sessions.

To ensure to customers that the server is from the company it claims to be, companies can use global server certificates. Those certificates are generated by those companies and can be verified from the customers at their web pages.

Installed digital certificates on a web server provide:

- Authentication of the web site: *A digital certificate on a server automatically communicates the site's authenticity to visitors' web browsers, confirming that the visitor is actually communicating with the right server, and not with a fraudulent site stealing credit card numbers or personal information.*
- Privacy of private communications: *Digital certificates encrypt the data visitors exchange with sites, to keep it safe from interception or tampering using SSL (Secure Sockets Layer) technology, the industry-standard method for protecting web communications.*

Nearly all web servers and all web browser are SSL capable, so it is an industry standard nowadays, which servers (companies) and clients (customers) can rely on.

3.5.5. Digital Certificates

More about them can be found in chapter 2.

3.5.6. Security features

Servlets have, like every other Java application, the following advantage: memory access violations and strong typing violations are not possible; this makes it nearly impossible for faulty Servlets to crash servers, like it is common in most C language server extension environments.

Unlike any other current server extension API, Java Servlets provide strong security policy support. This is because all Java environments provide a Security Manager which can be used to control whether actions such as network or file access are to be permitted. By default, all Servlets are "untrusted", and are not allowed to perform operations such as accessing network services or local files. [2]

"Built into" the server Servlets, or digitally signed Servlets, may be trusted and therefore granted more permission by the Security Manager. A digital signature on the Java code means that the organization that signed the code "vouches for it" in some sense. More about digital signature can be found in the chapter Cryptography.

More about it can be found in chapter 2.

3.6 SAS II: Servlet security

For our project we needed quite a lot of security. The concepts we used were the SSL technology for the whole data transferring between server and client. Therefore we configured the Apache web server application on our Linux server, to only allow SSL connections. The SSL key was generated by OpenSSL for Linux and used by mod_ssl for Apache. With SSL no data transfer to the server from any internet/intranet client was in plain text, which means that even if the data was intercepted, the interceptor could not "read" the data, without the generated keys, which are usually not transferred.

For Servlet security we used the session concepts. During login a session was created which then got an object assigned. The object was from a self-written class `SessionData`, which stored the main things needed for our application on the server. Those things were the username, the full XML menu structure and lots more.

The code of the SessionData class can be found in Appendix B.

All Servlets are inherited by SASIIServlet, which contains the main security checks, like checking if there is an active session, if the user is allowed to access that Servlet and so on. Only if the security check allows access, the overloaded `secureDoGet(…)` or `secureDoPost(…)` is called. The security check is implemented in the function `securityCheck()` of the class SASIIServlet:

A part of the SASIIServlet, where security is an issue:

```
import java.lang.*;
…

public abstract class SASIIServlet extends HttpServlet
{
  …

  /**
  * überprüft ob der aktuelle Zugriff(GET) erlaubt ist.
  * Ist er erlaubt wird secureDoGet aufgerufen.
  * Ist er NICHT erlaubt wird onError aufgerufen
  */
  public final void doGet(HttpServletRequest req,
 HttpServletResponse res)
  throws ServletException, IOException
  {
        //res.setHeader("Pragma","No-Cache");

        HttpSession session = req.getSession(false);
        …
        int err = securityCheck(session);
        try{
             if (err == SASIIException.ERR_NOTLOGGEDIN)
                  throw new SASIIException(err, "Sie sind nicht
                       eingeloggt");
             if (err == SASIIException.ERR_NOPERMISSION)
                  throw new SASIIException(err, "Sie haben
                       keine Berechtigung dieses Servlet
                       auszführen");
                       // Send Forbidden Error
             if (err == 0)
                  secureDoGet(req, res);
                  // session is OK, call derived secureDoGet
        }
        catch(SASIIException se){
             onError(req, res, se);
        }
   …
   }

   /**
   * überprüft ob der aktuelle Zugriff(POST) erlaubt ist.
   * Ist er erlaubt wird secureDoPost aufgerufen.
   * Ist er NICHT erlaubt wird onError aufgerufen
   */
   public final void doPost(HttpServletRequest req,
        HttpServletResponse res)
   throws ServletException, IOException
   {
        //res.setHeader("Pragma","No-Cache");
        HttpSession session = req.getSession(false);
        …
        int err = securityCheck(session);
        try{
             if (err == SASIIException.ERR_NOTLOGGEDIN)
                  throw new SASIIException(err,"Sie sind nicht
                       eingeloggt");
             if (err == SASIIException.ERR_NOPERMISSION)
                  throw new SASIIException(err,"Sie haben keine
                       Berechtigung dieses Servlet
                       auszführen");
                  // Send Forbidden Error
             if (err == 0)
                  secureDoPost(req, res);
                  // session is OK, call derived secureDoPost
        }
        catch(SASIIException se){
             onError(req, res, se);
        }
   …
   }

   …

   /**
  * überpruft die aktuelle Session auf Gültigkeit
  * und Zugriffsrechte */
   private int securityCheck (HttpSession session) {
        // Muß noch überprüfen
        if(session == null) {
             System.out.println ("session = null");
             return SASIIException.ERR_NOTLOGGEDIN;
        }
        SessionData sd = (SessionData) session.getAttribute
             ("SessionData");
        if(sd != null) {
             String cf = this.getClass().toString();
             if(sd.isClassFileAllowed(cf.substring(cf.indexOf
                  (" ")+1))) {
                  return SASIIException.ERR_NOERROR;
             }
             else {
                  return SASIIException.ERR_NOPERMISSION;
             }
        } else {
             System.out.println ("Session ungültig (" + sd);
             return SASIIException.ERR_NOTLOGGEDIN;         }
  }

   /**
   * sichere DoGet
   */

   public void secureDoGet (HttpServletRequest req,
   HttpServletResponse res)
   throws ServletException, IOException, SASIIException{
        throw new SASIIException (SASIIException. ERR_METHOD
             NOTSUPPORTED, "GET");
   }



   /**
   *    sichere DoPost
   */
   public void secureDoPost (HttpServletRequest req,
        HttpServletResponse res)
   throws ServletException, IOException, SASIIException{
        throw new SASIIException (SASIIException. ERR_METHODNOT
             SUPPORTED,"POST");
   }

   /**
   * zeigt Fehler an.
```

```
* Achtung !!! Hier darf kein getSessionData() verwendet
* werden. Es gibt also keine Möglichkeit auf die Session
* oder die DB zuzugreifen.
 */

 public void onError (HttpServletRequest req,
      HttpServletResponse res, SASIIException error)
 throws ServletException, IOException {
            …
 }


 …
}
```

4. Epilogue & Future Outlook

"The more you seek security, the less of it you have. But the more you seek opportunity, the more likely it is that you will achieve the security that you desire. " - Brian Tracy

There will never be one-hundred percent security, but security leaks can be minimized. When programs are published over the internet, millions of users can access them and contribute to them. Security leaks can be found by one of those, and the more people the higher is the chance to find those bugs or leaks. And the faster the leaks can be found the fast they can be eliminated. This tends to open source software. A change to open source would also a change the computer companies. They will not sell products anymore they would sell services, like maintenance of the software. This is similar to the change economy has gone through in the last century.

Sun with its open source Java API tends to go this way. A lot of people use and therefore check Sun's source codes. If bugs of leaks are found, programmers first try to implement solutions in their applications, and maybe one out of hundred mails Sun about the problem. Sun then adds this solution in the next release or the next update of their API, maybe they even add it to their concepts, so that every application is then "secured".

At the moment Sun is working at:

- Users, authentication, and credentials: this means that Java should extend its security concepts to include "running-on-behalf-of" concepts

- Resource consumption management: to prevent Denial of Service (DoS) attacks, and e.g. to reduce the crashes of computers, because too many windows are opened
- Grouping of permissions: e.g. generate "myPermission" which includes `FilePermissions` and `SocketPermissions`, for easier usage and management of the permissions.
- Object level protection: similar to the protection of `SignedObject` or `SealedObject` objects
- Signed content: How to ensure that multimedia content of an applet does not harm any policies? At the moment all images or sounds can be loaded.

There are approaches to make secure applications, but they will never be one-hundred percent secure!

Appendix

A) Sources, Bibliography

1 **Java Security** by Scott Oaks
ISBN: 1-56592-403-7

2 **The Java Servlet API**
from
http://java.sun.com/products/servlet/whitepaper.html
[Feb. 2002]

3 **Java Server Pages und Servlets**
by Brantner, Schmidt, Wabnitz, Wabnitz ISBN: 3-8158-2140-1

4 **Client Side State - HTTP Cookies**
from http://www.netscape.com/newsref/std/cookie_spec.html [March 2002]

5 **HTTP sessions, Servlets, and the session manager: WebSphere Application Server**
at http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/07010601.html [March 2002]

6 **PostgreSQL Database Documentation**
at http://www.at.postgresql.org/users-lounge/docs/7.2/postgres/ [April 2002]

7 **SSL Documentation – Technical Support**
at http://www.ssl.com/howsslworks.asp [April 2002]

8 **Secure Sockets Layer**
at http://www.netscape.com/security/techbriefs/ssl.html
[April 2002]

9 **Java Cryptography** by Jonathan Knudsen
ISBN: 1-56592-402-9

10 **"Evaluation of Server-Side Technologies In The Case of Sas II"** by Andreas Lehrbaum
Diploma thesis at the HTBLuVA St. Pölten, Abt. EDVO [2002]

B) Picture Index

C) The SessionData class in the SAS II project

```
import java.lang.*;
import java.sql.*;
import java.util.*;

/**
 * Diese Klasse stellt die Daten die in der Session gespeichert
 * werden dar.
 **/
public class SessionData extends Object {
	private Connection dbCon;
	private Menu menu;		// Objekt der XML Menüstruktur

	private String userLoginName;
	private String userName;
	private Vector userAmt;
	private String ip;
	protected Vector classFiles;

	/**
	 * Der Konstruktor erhält als Parameter die wichtigsten
	 * gespeicherten Werte.
	 * @param	uln	user Login Name
	 * @param	un		user real name
	 * @param	ua		user Amt
	 * @param	menu XML Menu Objekt
	 * @param	dbConDB Connection
	**/
	SessionData(String uln, String un, Vector ua, Menu menu,
	Connection dbCon, String ip) {
		userLoginName = uln;
		userName = un;
		userAmt = ua;
		this.menu = menu;
		this.dbCon = dbCon;
		this.ip = ip;
		//dbCon.setAutoCommit(false);
		classFiles = new Vector();
		System.out.println ("eine SessionData wurde erzeugt: "
			+ this);
	}

	/**
	 * Der Destruktor soll dazu dienen die Speicherauslastung zu
	 * minimieren.
	 **/
	public void finalize() {
		try{
		if(dbCon!=null)
			dbCon.close();
		}catch(SQLException e) {}
		System.out.println ("Eine Session wurde zerstört (" +
			ip + ")" + this.hashCode());
	}

	/**
	 * Zum hinzufügen dieser Java Klasse zum Vector der erlaubten
	 * Klassen.
	 * @param  cf   Klassenname
	 **/
	public void addClassFile(String cf) {
		if(cf != null)
			classFiles.add(cf);
	}

	/**
	 * Funktion zum überprüfen, ob das Aufrufen dieser Klasse,
	 * für diesen user erlaubt ist.
	 *
	 * @param       cf  Klassenname
	 * @retval Ob der Aufruf erlaubt ist(true) oder nicht(false)
	**/
	public boolean isClassFileAllowed(String cf) {
		if(cf == null)
			return false;

		boolean back = false;
		Enumeration e = classFiles.elements();
		while(e.hasMoreElements()){
			String st = (String) e.nextElement();
			if(st != null && st.equals(cf))
				back = true;
		}
		return back;
	}

	/**
	 * Zugriffsfunktion auf den User Login Name
	 * @retval der User Login Name
	**/
	public String getUserLoginName() {
		return userLoginName;
	}

	/**
	 * Zugriffsfunktion auf den User Name
	 * @retval der User Name
	 */
	public String getUserName() {
		return userName;
	}

	/**
	 * Zugriffsfunktion auf den User Login Name
	 * @retval der User Login Name
```

```
     **/
     public boolean hasUserAmt(String ua) {
          return userAmt.contains(ua);
     }

     /**
      * Zugriffsfunktion auf die DB Connection
      * @retval die DB Connection
     **/
     public Connection getdbCon() {
          return dbCon;
     }

     /**
      * Zugriffsfunktion auf das Menü Objekt
      *
      * @retval das Menü Objekt
      */
     public Menu getMenu() {
          return menu;
     }

     public String getComputerName() {
          return ip; //"sasii@sasii.sasii.sasii";
     }

     public String getLicenseNumber() {
          return "1234-5678";
     }

     public String getVersionNumber() {
          return "v 0.1 pre-alpha";
     }

     public String toString() {
          return "SessionData: " + userLoginName + " " + username
               + " IP:" + ip;
     }
}
```