Java Security

Bastiaan Bakker

Preliminary Investigation
Department of Computer Science
Technical University of Delft

Performed at NTEX Harbinger
Rotterdam

Table of Contents

1. INTRODUCTION

2. JAVA

2.1 The Java Runtime Environment
2.2 The Java Language

3. JAVA SECURITY

3.1 The Sandbox model
3.2 Restricting class visibility
3.3 Object visibility and access modifiers
3.4 Explicit authorization checks
3.5 Inter Applet Communication
3.6 Auditing and Accounting
3.7 Attack targets
3.8 Conclusions

4. CRYPTOGRAPHY

4.1 Introduction
4.2 Crypto building blocks
4.3 Certificate Based Authentication
4.4 The Secure Sockets Layer
4.5 The SSL Handshake Protocol
4.6 Crypto in Java
4.7 Key Management
4.8 Combining SSL and Java
4.9 Conclusions

5. VISUAL WEB

6. CONCLUSIONS

7. ACKNOWLEDGMENTS

APPENDIX A: PERILS OF THE SECURITY MANAGER

APPENDIX B: REPORT OF A SECURITY BUG IN HOTJAVA 1.0 PREBETA 2

APPENDIX C: AN HTML BASED ATTACK ON HOTJAVA

REFERENCES

ABBREVIATIONS

1 Introduction

In the summer of 1995 Sun MicroSystems introduced Java, a new technology for Client/Server based programming that circumvents disadvantages of traditional Client/Server systems. This paper investigates the security concerns raised by using Java and how Java tries to counter these.

In traditional Client/Server technology two approaches are commonly used:

1. In a *fat client* system the client performs a significant part of the functionality of the application. The server is relatively passive. An example is a WWW browser / HTTP server.
2. In a *thin client* system the client only provides an interface to the application that is running (remotely) at the server. A populair implementation of this strategy is the X-Windows system.

Fat clients can de taylored optimally to the application, since the functionality can be put wherever is the most efficient and the features of the client environment can be utilized maximally. But building specialized clients has its drawbacks too:

- the client has to be ported to any platform it should run on.
- the client has to be installed on all computers it has to run on.
- the client is specific for the application: for each new application a new client has to be written.
- care has to be taken to keep versions of the client and the server in sync, updating an application is difficult.

A thin client system like the X-Windows system avoids a lot of the fat client problems by offering an environment in which the server can dynamically build a user interface for the application. This reduces porting and installing of clients to porting and installing the X-Windows system at the client host once. Since all application software is stored at the server versioning problems are avoided too. However this solution also has disadvantages:

- the generic client is more complex than necessary for most applications
- scalability is limited as the server has to execute all code.

Both current fat and thin client approaches lack flexibility.
Java offers a new approach to Client/Server programming that combines the best of both worlds:
At the client system a runtime environment is installed, in which upon use clients are downloaded and executed. Since the client program is stored at the server updating software is simple. Moreover, the user only needs to install one program, the Java Runtime Evironment (JRE), just like with X-Windows, while retaining flexibility.

Sun has marketed Java as the new way of programming for the Internet and the WWW. Instead of building all kinds of plugins for webbrowsers programmers should write Java programs to handle new types of WWW content. To demonstrate this Sun published their HotJava WWW browser that was completely build with Java technology. As Netscape and Microsoft recognized the potential of Java they included it in their respective browsers. At this moment Java environments are available for most platforms, either as part of a browser, or standalone. The latest stable release of Java is Java Development Kit (JDK) 1.02, published May 1996. Currently a much improved and enhanced version, JDK 1.1, is in public beta testing.

The use of Java in an heterogenous, insecure environment like the Internet places certain demands on it:

1Portability

It should be portable: it should be able to run on different processors and under different operating systems. Java solves this problem by compiling Java programs to virtual machine code, called Java bytecode, that is interpreted by a part of the Java runtime environment, the Java Virtual Machine (JVM). To speed up the execution, bytecode may be translated to native code at the client by a Just In Time (JIT) compiler. Sun MicroElectronics even has pursued hardware based acceleration: CPUs are under development that directly execute bytecode. To couple Java programs to the local operating system, the Java runtime environment includes a set of core APIs. These provide an abstracted interface to the underlying system.

2Security

Secondly, Java should be secure. Java redefines the relation between software providers and users. Traditionally a user would buy an application from a software publisher, install it and use it. The user trusts the software not to do dangerous things that could compromise privacy, data integrity, etc., since the software provider is know and could be held accountable. Essentially the good reputation of the provider forms the protection of the user.
The situation is different in the case of Java programs: the user does not buy the application nor install it. Particulary when using applications on the Internet, the user does not have to know who the supplier is, or even that the application is running in some cases. Obviously the trust relationship between the software supplier and the user is entirely different or even nonexistent. A cautious user will not use Java if no measures are taken. This problem can be tackled with two methods:

1. The Java runtime environment denies all attempts by Java applications to perform dangerous operations.
2. The Java runtime environment keeps track of the source (provider) of all Java applications and of all their actions.

Java 1.0 follows the first method: downloaded Java software is prohibited from accessing the local user environment. This solution may provide sufficient security but at the same time it severely limits the usefulness of Java.
Therefore from Java 1.1 on, advances towards the second method are made: the sofware provider can digitally sign its Java programs and the user can configure the Java environment to grant applications from certain, trusted providers access to (parts of) the local system. Now data can be collected from the users file system, results stored to it, etc., etc.

2 Java

This chapter describes two parts of the Java technology: the Java Runtime Environment and the Java Language. Both are important in the discussion about Java security. The first, because it helps to understand the requirements on Java's security, the second because Java's security is based on its language.

2.1 The Java Runtime Environment

This paragraph will look into how the Java Runtime Environment works and where.
Although currently Java is best known as a part of WWW browsers by most people, it is not confined to them. Since it is small, modular and flexible it is well suited to run in other environments. Major network device companies have announced to incorporate Java in their devices. Network Computers will run Java and even chipcards are designed to run (a subset) of Java [SUN96-3].

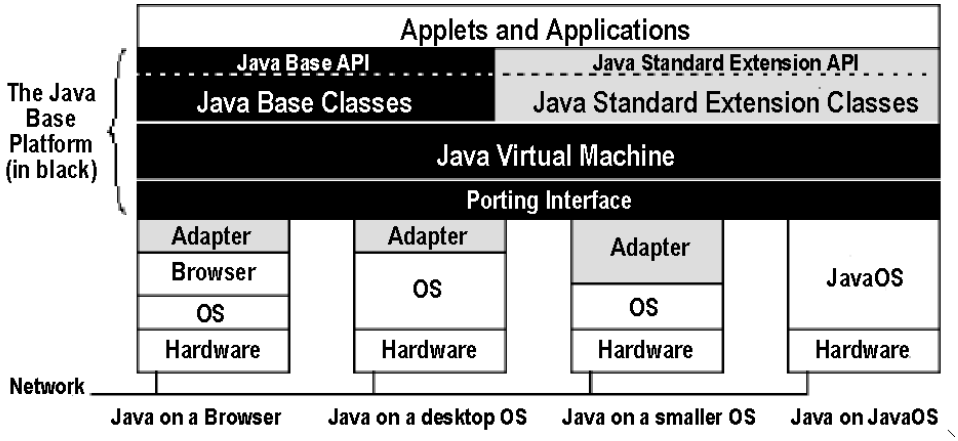


Figure 1*: Environments for running Java*

Java programs that run in a browser are called *applets*. Applets are small applications (hence the name) that are downloaded from a network. They run embedded in HTML pages just like browser plugins. A typical inclusion of an applet on a page looks like this:

```
<applet codebase="http://www.javasoft.com/applets"
code="DukeApplet" with=100 height=120>
<param name="someParameter" value="itsValue">
</applet>
```

Applets run within certain restrictions to prevent them to compromise the security of the user. They may not access the local environment of the user. Furthermore their networking capabilities are limited to connecting back to the server they are downloaded from. The latter restriction should ensure that applets cannot attack a third host, camouflaging the fact that the attack actually comes from the server. It should be safe to run applets that originate from outside a firewall.

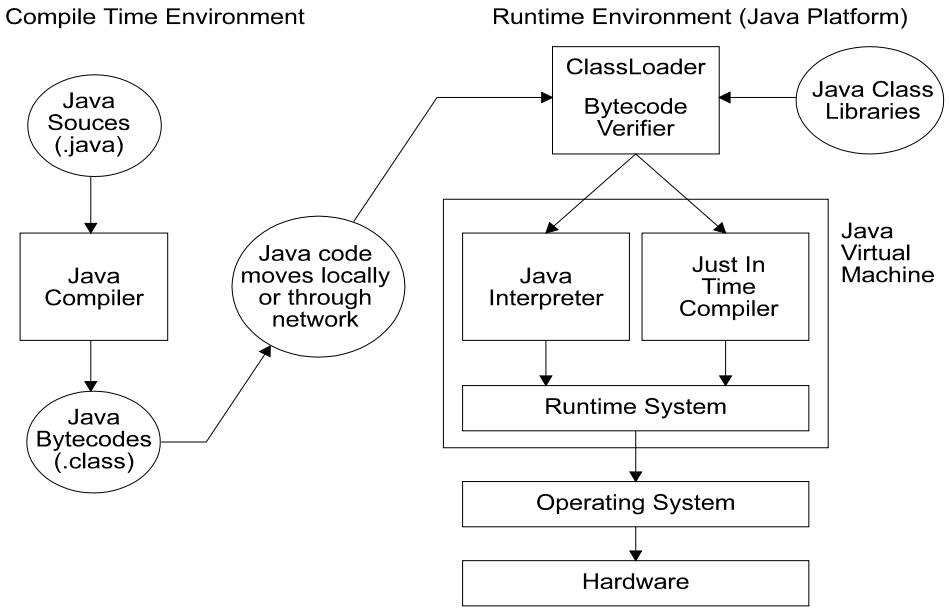


Figure 2*: execution of a Java applet or application*

Figure 2 shows how a Java applet is executed:

1. Java sources (.java files) are compiled by a compiler to bytecode. Every compiled class is put in a separate .class file. The '.class' files still contain all symbolic information needed for linking.
2. The bytecode is downloaded by the Java runtime (e.g. a browser) on a per class basis.
3. The ClassLoader (the part of the runtime responsible for loading classes) links the loaded class with system classes and other downloaded classes.
4. The ClassLoader invokes the bytecode verifier to check whether the class violates Java language rules.
5. Now the class is added to the runtime. It either will be interpreted by the Java interpreter or compiled to native code by a Just in Time Compiler.

It's also possible to create standalone Java applications. The JDK provides a runtime to use these. Java applications do not have the restrictions applets have. Suns HotJava browser is an application that runs in the JDK runtime environment.
JavaSoft has used Java for system development too. This has resulted in the JavaOS operating system. JavaOS is a small, portable, lightweight operating system [SUN96-5] written in Java. JavaOS is intended to run on the Network Computer and other Java enabled devices.

2.2 The Java Language

The Java language is an object oriented, class based language. Java was modeled closely after C++ to facilitate programmers. Java is much simpler and cleaner however, a lot of difficult, rarely used features have been removed.

1Object Referencing

In Java the memory model is abstracted. Pointers have been abolished, as has explicit memory handling. Instead opaque references to objects are used. The only ways to obtain references is either by creating a new object with the `new` operator or by assignment from another reference. Moreover it is impossible to forge a reference or to do an illegal type cast. Programmers cannot explicitly

deallocate an object, instead the runtime system has a garbage collector that removes objects once they are no longer needed. This way, the common problem of memory leakage is avoided. The forced type safe referencing of objects and the garbage collector enhances the robustness of Java software a lot.
References are limited to refer to objects inside the same JVM. To allow interaction between JVMs the Java 1.1 includes a Remote Method Invocation (RMI) API. With this API remote objects can be accessed through local stub objects.

2Inheritance

All complex types are classes derived from a single base class `java.lang.Object`. For performance reasons simple types such as integers are available both as Objects and as primitives.
Unlike C++, Java does not support multiple inheritance. It was felt that problems raised by supporting it (particularly name resolution ambiguities) outweighed the benefits. Instead in Java one can declare *interfaces*. An interface declaration is similar to a class definition, but it lacks instance data members and bodies for the methods it declares. A class can declare that it implements the interface, it should define all methods declared in the interface in that case. Declaring multiple interfaces for a class facilitates its use in different roles. The mechanism of interfaces resolves the naming ambiguities, but does not eliminate method name collisions between interfaces.

3Dynamic Linking

Whereas C++ programs are statically linked, Java uses dynamically linking. All compiled Java classes are stored in separate ".class" files. These files still contain all symbolic information needed for linking. When a Java program is executed class-files are loaded into the runtime system when they are first needed. Only then the symbolic names are resolved. Delaying the linking until execution has several benefits:

- Only the parts actually needed at runtime have to be downloaded.
- Header files are not needed anymore, as all information about a class is still available after compilation.
- The *fragile superclass* problem is avoided. In C++ if a class B derives from class A and class A is modified after class B is compiled, class B has to be recompiled too, else linking will fail. In Java linking and running will succeed as long as the methods and fields in A that B uses are still present.

4Packages

Classes that logically belong together can be grouped into a package. Classes in the same package are allowed access to each others member more freely. The JDK 1.02 VM maps package names onto directories, so all classes belonging to package java.awt.image can be found in subdirectory java/awt/image. Although Java's package names suggest a hierarchy, the current Java implementation does not support the notion of subpackages, i.e. java.awt.image is **not** a part of the java.awt package. The JDK 1.1 compiler supports *inner classes*, that facilitate a finer grained control on the scope of classes. However this is language based rather than runtime based: names of inner classes are simply mangled to avoid use outside scope. Outside their scope they are invisible at Java language level but appropriate bytecode can still reach them.

5Access protection

A programmer can protect the functionality of classes by using appropriate access modifiers on its members. Java provides the following access modifiers:

- *private* members may be accessed by methods of the same class only.
- *protected* members may be accessed by methods of derived classes as well and by any class that is defined in the same package.
- *public* members are accessible by anybody that can see their name.(Visibility will be explained when Java's use of name spaces is discussed).
- *package protected* members are accessible by any class that is defined in the same package. (Note: since *package protected* is the default access policy, there is no keyword associated with it.)

Classes themselves can be declared public or package protected. Package protected classes are inaccessible outside their package. In addition any method can be declared *final*, which means it cannot be overridden by subclasses. *Final* data members may not be changed after construction. Declaring a class to be final prohibits deriving from it.
Security sensitive methods should always be either private, package protected or final. Otherwise malicious programmers could override the method to circumvent security measures.
Data members should hardly ever be declared public even if they are final. Java handles objects by reference, so declaring an object as a final data member only means that the reference cannot be changed. The object itself still is muteable! Java does not have a 'const' specifier like C++, but this may change in the future: 'const' already is a reserved word in JDK 1.02.

6Multi-threading

Language based support for multi-treading is also included. Threads can be created by constructing `java.lang.Thread` objects to which runnable methods can be coupled. Threads can be assigned priorities ranging from 1 to 10. No guarantees are made about thread scheduling policies, except that higher priority threads get more CPU time. In particular one may not assume that threads are scheduled preemptively. For this reason `Thread` includes a `yield()` method to explicitly request that some other thread should be run.
The Java language supports safe concurrent object manipulation by providing recursive locks on all objects. With the `synchronized` keyword programmers may specify execution blocks only to be entered when the lock (called a *monitor* in the Java documentation) of a specified object has been obtained.
Since the Java runtime has a single shared address space, multithreading is the only method of concurrency: traditional UNIX processes are not needed anymore.

7Native environment interface

Java provides a simple interface to the native environment. Methods can be declared `native`, which means that the method body is implemented in native code. This code is not part of the Java class, but provided by native object libraries. Before a native method can be used the corresponding object library has to be loaded with the `java.lang.System.loadLibrary(String name)` method. As native code is not bound to Java's security restrictions only privileged code may directly use native methods. All applets and most applications access the native environment through the Java core APIs, as these provide a proper abstraction of the native system (thus ensuring portability).

3 Java Security

This chapter describes Java's security model with respect to applets. Also several basic protection techniques Java uses are explained:

- Language enforced restrictions, instead of traditional hardware based ones.
- Separate name spaces to protect different security domains
- Object hiding
- Use of a security manager for implementing specific security policies

3.1 The Sandbox model

Java's protection is based on the 'sandbox' model. This means that the applets are allowed to 'play' (run) in a restricted environment, the sandbox, in which they can do no harm to the local environment. Any attempt to access the local environment (for example to write a file to the harddisk) is supervised by a *security manager*. The attempt will fail if the applet does not have sufficient privileges. To ensure that all sensitive access is made through the security manager Java relies on type integrity: applets cannot forge pointers to objects and cannot access them in any other way than defined by the objects themselves.
Since the Java runtime does not download Java source code but the compiled bytecode instead, it has to verify that it does not violate security criteria. This is done by the byte code verifier. It checks for example whether access modifiers aren't violated, all constructors are properly called, if no stack under or overflows can occur and no illegal type casts are done. If a class has passed the verifier one may safely assume that it is the bytecode equivalent of a legal Java source. Earlier versions of the verifier contained bugs that allowed applets to violate type safety [DEAN96] and the current version still may too, since no proof has been given yet that it is secure. Since the Java language and bytecode specifications have not been subject to change for a relatively long time (nor will be probably) the byte code verifier is the Java part best suited for formal verification. A discussion about bytecode verifier security lies outside the scope of this paper.

3.2 Restricting class visibility

When a class is downloaded all other classes it uses are still referenced by their fully qualified name. The linker has to resolve these names to the actual classes (possibly causing these classes to be downloaded as well). Since different programs may use the same class names there have to be provisions to avoid naming collisions. Moreover different programs should not be able to extend eachothers packages by using identical package names, as this would void the usefulness of the package protected and protected access specifiers. Therefore (packages of) classes are grouped into *name spaces* inside which all classes have an unique fully qualified name. Classes with the same name but in different name spaces are considered to be different classes even if their definitions are completely identical.
By default just one namespace is present: the system name space. This name space contains all classes that reside on the local file system. The runtime looks for these system classes in all directories that are listed in the CLASSPATH environment variable. They are special the following respects:

1. They are trusted to do security sensitive things, such as calling native code
2. They are visible by all classes: the system name space is reachable from all other name spaces.

3. They cannot be shadowed by classes in other name spaces that have the same name: the system name space is always searched first. This rule is not enforced, Java system programmers should take care not to violate this.

All other name spaces are implicitly defined at runtime. These name spaces are maintained by ClassLoader objects, that take care of downloading classes and resolving class names. Every class has a `getClassLoader()`method that returns the ClassLoader by which that class is downloaded. The runtime will use that ClassLoader when a class name used in the class has to be resolved. The ClassLoader will first look whether the class name can be found in the system name space. If it can't be found there, the ClassLoader will try to download it from the remote 'source' of classes.
Each 'source' of classes is assigned a separate ClassLoader. Currently the 'source' means the location where the classes (or their packages) are stored on a server. Often this is identical to the place of the HTML page containing the applet but it may be entirely different if another CODEBASE is specified in the applet tag. Now that code can be digitally signed, future versions may redefine source to be the publisher of the classes. On the other hand, some documentation suggests that maybe every downloaded applet will be assigned its own name space in the future [JAVA97].
The vague rules concerning about name space separation proved to be a source of frustration for Java developers: during the development of Netscape Navigator 3.0, the policy regarding name space separation changed almost every beta release. In the last beta releases 3.0b6 and 3.0b7 every single applet was put in a separate name space. This infuriated many Java developers, who relied on shared name spaces to do Inter Applet Communication. A large stream of angry letters made Netscape reconsider: the final 3.0 version separates name spaces based on the CODEBASE again.

Class invisibility does not imply object invisibility! A class still can obtain references to instances of classes that are in non shared name spaces. In that case it will only see the part of the other object that is in a visible name space. Since all classes are derived from a system class (at least class Object in any case) this part is never empty.

Figure 3 shows an example of name space separation: two equally named classes (maybe even equally defined classes) are put in separate name spaces since they originate from different sources. Both derive from the system class `java.awt.Button`. Now if an instance of a class in name space 1 has obtained a reference to a `user.MyButton` in name space 2 it:

- may call `getParent()`, since this method is introduced in the (visible) system name space and is public.
- may call `action(Event e)`, since the method is introduced in the system name space and is public. The method body defined in `user.MyButton` in name space 2 will be executed since it overrides the one in `java.awt.Button`.
- may not access `label`, since it is package protected. An access attempt will cause an `IllegalAccessException` to be thrown.
- may not access `key`, since it is invisible from name space 1. An access attempt will cause an ClassCastExeception since the code will try to cast the name space 2 `user.MyButton` to a name space 1 `user.MyButton`.
- may not call `decode()` for the same reason it may not access `key`, but it may be possible to have it invoked if for example `action(Event e)` invokes it.

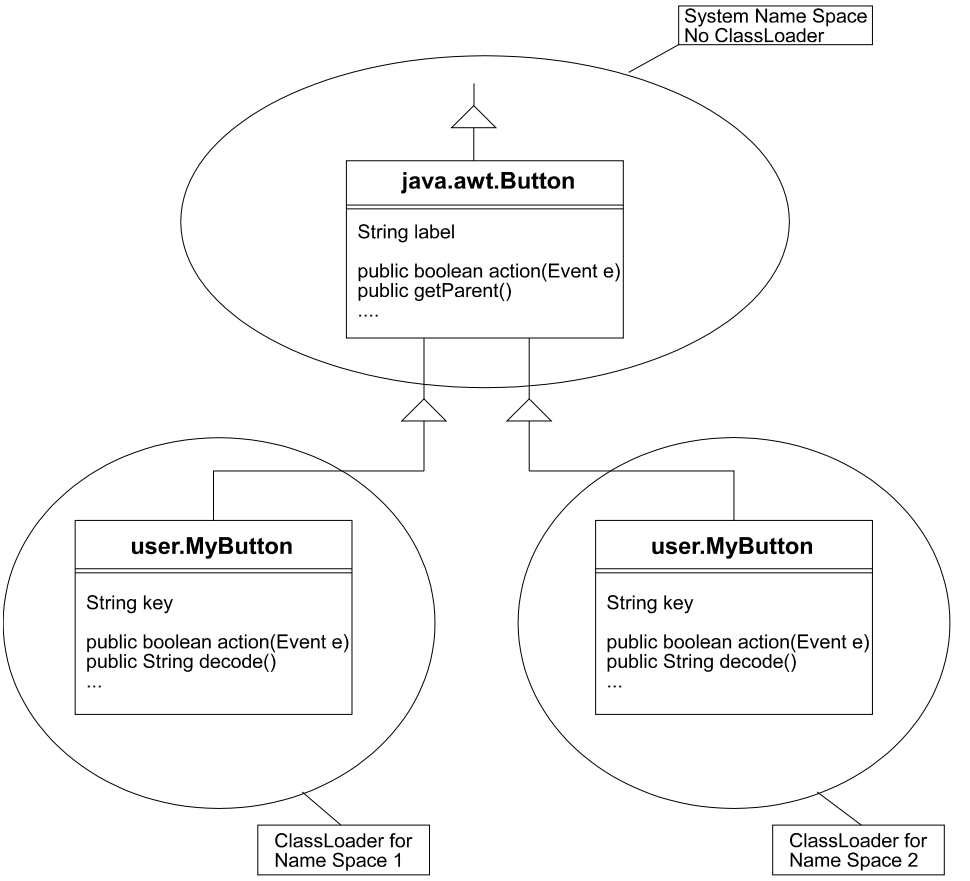


Figure 3: ***Two equally named classes in separate name spaces***

The Princeton Java security team [DEAN96] has demonstrated that applets can defeat type safety enforcement if they can instantiate their own ClassLoader. This is accomplished by writing a ClassLoader that on different occasions resolves the same class name to different classes. This way a system class can be treated as a customized permissive class that allows full access to all members. Early versions of Java (JDK 1.01 and Netscape 2.0) contained a bug in the bytecode verifier that allowed applets to instantiate a customized ClassLoader, effectively compromising all sandbox security restrictions.

3.3 Object visibility and access modifiers

Contrary to restricting name space visibility, hiding objects from possible malicious code does effectively prevent them from being tampered with. Now the problem remains how to keep ones object hidden from other applets. It turns out that several system classes are too permissive in returning objects. In the paragraph about Inter Applet Communication some examples will be given. Another difficulty is once a reference is obtained access to the full functionality of the corresponding object is granted too: a reference to some object that implements interface X may be used as a reference to an instance class Y if Y implements X. Even though a method only returns a reference to a class or interface with limited functionality, all other members are accessible as well (limited of course by name space visibility).
Even if a references cannot be kept secret, classes still can protect themselves by using proper access modifiers: the private, package protected and protected modifiers prevent classes in other packages to access the members they are

applied to. But again system classes may make this impossible in some cases: methods and fields that are declared public cannot be overridden to be (package) protected or private. Therefore these methods cannot be shielded by using access modifiers. The only option that remains for these methods is to include explicit access checks.

3.4 Explicit authorization checks

Explicit checks for permissions are centralized in a separate class, the SecurityManager. Applications (not applets) may instantiate one SecurityManager, to be used in the entire runtime from then on. By building different SecurityManager subclasses appropriate security policies for different situations may be defined. The JDK includes the `AppletSecurity` SecurityManager that restricts the capabilities of all downloaded applets.
The SecurityManager contains various `checkAccess`() methods that should be called by all methods that perform sensitive operations. The `checkAccess`() methods throw a `SecurityException` if access should be denied. Note that the calls to the SecurityManager are spread throughout all APIs that should be protected and that by default (if no call is made) access is granted. This violates the principle of denying access unless it is explicitly granted and makes it difficult to see whether all methods are sufficiently protected.

Contrary to UNIX Java uses two (logically) separate stacks for data and execution information. The latter, the call stack, can be used for security checking: every frame on this stack corresponds to a called method associated with a certain class. Since permissions in Java are linked to a classes ClassLoader, the list of ClassLoaders of classes on the stack provide enough information to be able to decide whether permission should be granted or not.
In Java 1.02 there was only one policy for all applets, simplifying the decision: use the applet policy if there is a ClassLoader on the stack, grant permission if not. Sometimes permission should be given even if there is a ClassLoader, for example a system class may want to invoke a protected method on behalf of an applet. In that case the SecurityManager looks at the depth of the ClassLoader on the stack to determine whether a system class made the call or the applet. This method is error prone however (see Appendix A for an example). Moreover, it makes independently developing SecurityManagers and the classes they should protect impossible.
In JDK 1.1 the situation has become more difficult: every ClassLoader may carry a different set of permissions. It has become unclear what policy a Security Manager has to implement: if class A calls a method in class B that calls a method that is protected by a security manager check, should the call fail or succeed? One can argue that it should succeed since it should be the responsibility of class B to protect itself from unprivileged class like A. However applets are not allowed to look at the call stack, therefore it has to revert to another mechanism than the SecurityManager uses. An option for B would be to base its security policy on whether the calling thread is one of its own, but this is not flexible. Besides that, having to include checks at every entry point (essentially every public method) is not a burden that should be placed on an application programmer.

The use of a SecurityManager allows JRE programmers to implement more flexible security policies. Building a secure SecurityManager has proven to be difficult however. Furthermore it violates the principle of denying access unless it is explicitly granted. Since calls to the SecurityManager are scattered throughout all APIs that have to be protected, security vulnerabilities easily are overseen. It

would be better to have more implicit language based protection mechanisms, that would reduce the size of the security manager.

3.5 Inter Applet Communication

Applets do not always run isolated on a Java Virtual Machine, but may want to interact with each other. A good Inter Applet Communication interface should ensure safe interaction between applets. The current IAC model is rather simple: first obtain a reference to (a class of) another applet and then invoke methods on it. This restricts IAC to applets running in the same JVM. The Java 1.1 RMI package is of limited use for IAC: since applets may connect only to their server they cannot use RMI for communication with other applets at the client.
The method `java.applet.AppletContext.getApplets()` was designed to obtain references to other applets: it returns an enumeration of all applets running on the same page[1]. Another method is to create a static variable containing references to all applets. Since a static variable is shared only within the same name space one can only reach applets with the same code base this way.
Other methods take advantage of (accidental) permissiveness of the system APIs or the environment. Here are some examples:

- java.lang.Thread.getParent() and java.lang.ThreadGroup.enumerate() let an applet obtain references to all running threads in the Java VM.
- java.awt.Component.getParent() and java.awt.Container.getComponents() let an applet obtain references to all windows components that contain the applet and all their subcomponents. In HotJava all pages in the browser frame are reachable this way. All applets in the browser can be found, circumventing the restrictions in getApplets().

Note that any two applets can be put on the same HTML page. The applets do not have to be reachable by the author: they can be behind a fire wall or in a restricted place as long as the viewer can access them. See appendix XXX for an example of an HTML page that takes advantage of this to mislead a user. Once a malicious applet has obtained a reference to another applet several attack schemes are possible.

It is clear that the current IAC scheme is inadequate: an applet doesn't have any control over who can access it and cannot adequately implement a security policy itself because it may not use the necessary methods of the SecurityManager. Implementation of cooperative IAC is hindered by name space separation and the limited visibility provided by `getApplets()`. Javasoft has indicated that it will introduce better IAC provisions in a future release.

3.6 Auditing and Accounting

Currently no auditing or accounting features are available in Java. It's possible to write a SecurityManager that logs all (failed) access checks but no one has implemented one yet.
Moreover Java has no notion of object ownership. A class is coupled to a security policy by its ClassLoader, which corresponds to a 'source'. System class instances have no ClassLoader and therefore cannot be linked to any source. In the case of AWT components, the object often even isn't referenced by its creator. Instead it is referenced by an AWT container class. Since this container or one of its (grand)parents ultimately is an applet, the component indirectly is

[1] In HotJava `getApplets()` is restricted to return only those applets that originate from the same host. This is a rather arbitrary decision: multiple users, that may not trust eachother may share the same host, a common practice on UNIX system. Netscape 3.0 does not inhibit this behaviour and returns all applets in the same HTML frame.

referenced by its ‘owner’. However one cannot determine this without knowledge of the semantics of the involved classes.

3.7 Attack targets

Attacks can be put in the following categories:

- denial of service: slowing down or blocking the Java VM or other applets.
- covert channels: using applets to circumvent firewall restrictions
- compromise of other applets.
- compromise of the host system.

Javasoft has focused almost exclusively on protection against direct host system compromise and covert channels. Denial of service attacks are considered low priority problems [DEAN96]. Applet compromise wasn’t given high priority either because up until JDK 1.1 applets were assumed not to do any sensitive things. This assumption is not valid however, for example the HotJava browser uses applets to perform security related tasks.
For the above reasons Java’s security and particularly the sandbox model has been compared to the Maginot defense line that France had built at its border with Germany before WO II [SUN96-6]: the line was a very strong defense against direct attacks, but the French did not protect itself against an attack through Belgium, since that was a trusted ally. The mistake they made was to confuse trusted with secure: Belgium was trustworthy but could not protect itself against an invasion by the Germans, which could then easily attack France. The same is happening in Java: trusted applets cannot protect themselves against malicious ones. In Java 1.1 or already in HotJava with JDK1.02 , where trusted applets may get they same priviliges as system code, this could lead to full compromise of the host environment.

3.8 Conclusions

Java uses several techniques to provide protection against malicious code. These techniques may sufficiently protect the local environment of the client, but this is not proven yet. Protection from applets against eachother currently is clearly insufficient: some Java runtime environments may provide enough security to safely run several applets simultaneously. However JavaSoft has not specified the requirements to such environments, so applets can not assume to be run in such a safe environment. Particularly the HotJava browser has been shown to provide an insecure environment.

4 Cryptography

4.1 Introduction

Secure communications over insecure channels would be impossible without the use of cryptography. In particular cryptography can be used to ensure the following properties:

- privacy: the message cannot be read by a third party.
- authentication: the source of the message is known.
- integrity: the message is not altered by a third party.
- non repudiation: the source cannot deny that it sent the message.

In this chapter introduces cryptographic methods to achieve the above properties and describes the application of these methods in (secured) WWW browsers and in Java.

4.2 Crypto building blocks

The basic building block in cryptography is a cipher: an algorithm that based on a key maps a message onto another message. Traditional ciphers are symmetric: both encryption and decryption are performed with same key. DES and RC4 are common symmetric key ciphers. Secure communication is possible only when the two parties share a secret that can be used as a key. In many situations this not feasible. The parties may not even know eachother for example. In those cases asymmetric ciphers provide a solution. Instead of one key, they use a pair of keys: one for encryption, the other for decryption. Now one can give the encryption key, the public key, to anyone as long as the corresponding decryption key is kept private. It is computationally infeasable to deduce the private key from knowledge of the public key. RSA, named after its inventors Rivest, Shamir and Adleman is an example of a public key algorithm.
Both keys are each others inverse: if a message is encrypted with the private key it can be decrypted with the public one and vice versa. This allows public key systems to be used for something else too: digital signatures. You simply uses our own private key to encrypt the message. Others can verify the message originates from you by decrypting it with your public key. Digital signatures have been standardized in the Digital Signature Standard (DSS).
One may ask why symmetric ciphers still are used, if asymmetric ciphers have so many benefits. The answer simply is: speed. Public key crypto is very computationally intensive. Therefore usually a session key is send via public key crypto and all further communications use a conventional cipher with this session key.

Ciphers alone do not protect against modification of the messages by a third party. To ensure message integrity secure hashes, also called message digests are used. These are functions that (variable length) blocks of data to a single, fixed length number. They have the unique property that changing any single bit in the data block completely changes the resulting hash code. It is computationally infeasable, given a hash code to calculate a data block that generates it. Therefore any alterations to a message can be detected easily if a message digest is attached to it. Ensuring the integrity of the message hereby is reduced to ensuring the integrity of the digest.
Messages digests also speed up digital signing: it suffices to sign the digest rather than the whole message. Commonly used hash functions are Message Digest 5 (MD5) and the Secure Hash Algorithm (SHA).

Figure 4a and 4b show how encryption, digital signatures and hashes are used to send a private and authenticated message, in this case an EDI file.

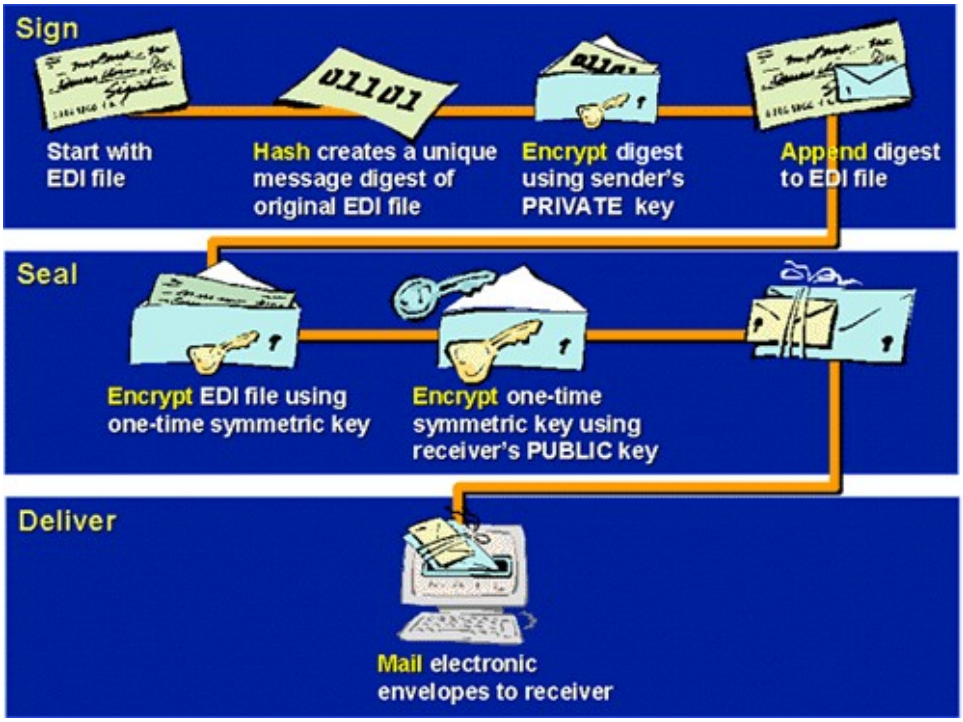


Figure 4a: ***private and authenticated delivery of a message***

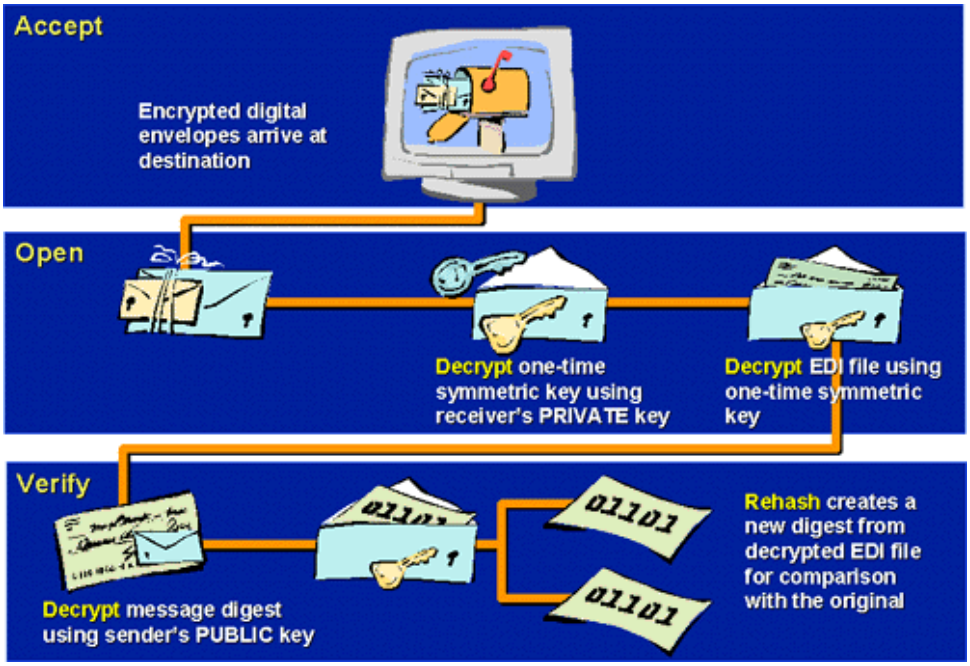


Figure 4b: ***receipt of the message***

The fourth property, non repudiation, builds on digital signing. If a message is digitally signed one can be assured that the owner of the signing key has sent it. Someone can confirm to have sent the message by creating a new message that can be decrypted with the same public key. However, when someone denies to have sent the message this does not work. The public key has to be tied to the

identity of its owner. A method of doing this will be discussed in the following paragraph about certificates.

In sum: symmetric ciphers provide privacy, asymmetric handle session key exchange and allow digital signing and message digests guarantee message integrity.

4.3 Certificate Based Authentication

Both The Secure Sockets Layer described in the next section and Java (as of version 1.1) can perform authentication based on public key cryptography and certificates. A certificate is a document attached to a public key that contains information about it. Generally it states who the owner (a person, institute, Internet site ,etc.) is, where the owner is located, from when to when the certificate is valid, etc. The format of certificates is standardized in ITU's X509 specification [X509 XXX]. A certificate is digitally signed by a so called Certificate Authority (CA). A Certificate Authority is a principal that guarantees the validity of the certificate. A CA introduces the identity described in the certificate, so to speak. A Certificate Authority can be introduced itself this way too, by having a certificate signed by another, higher level authority. This creates a chain of trust, on top of which resides a 'root', or 'top level' certificate authority. Now the problem still remains that the public key of this root certificate authority has to be authenticated. Preferably one would visit the authority in person and collect his or her key. Of course this is not feasible in all cases, therefore often the key is distributed over several independent insecure channels: if requesting someone's key by email, telephone and snail mail, yield the same key one may be convinced that key is genuine even though it is well known that any of them can be manipulated. Ultimately one has to trust someone or something. (Even in the case of personally visiting someone, you at least have to trust yourself to properly identify that person, and not her twin sister for example.)
In the future most likely governmental institutions will act as root certificate authorities. They will certify digital extensions to current identification documents such as pass ports and visa. At the moment a few commercial companies offer certification services. Their number is expected to grow rapidly as usage of digital identities becomes common practice.
In most countries digital certificates have no legal status yet. But some states in the USA have passed legislation regulating certificates and certificate authorities [UTAH95]. In anticipation of legislation most certificate authorities have written their own legal agreements and statements. The most notable of these is the Common Practice Statement of the Verisign company [VERI96]. This company, founded by the RSA inventors, was the first to sell certificates.

In sum: certificates are digitally signed statements that tie a public key to the identity of its owner. They do not prove the ownership, but are based on trust.

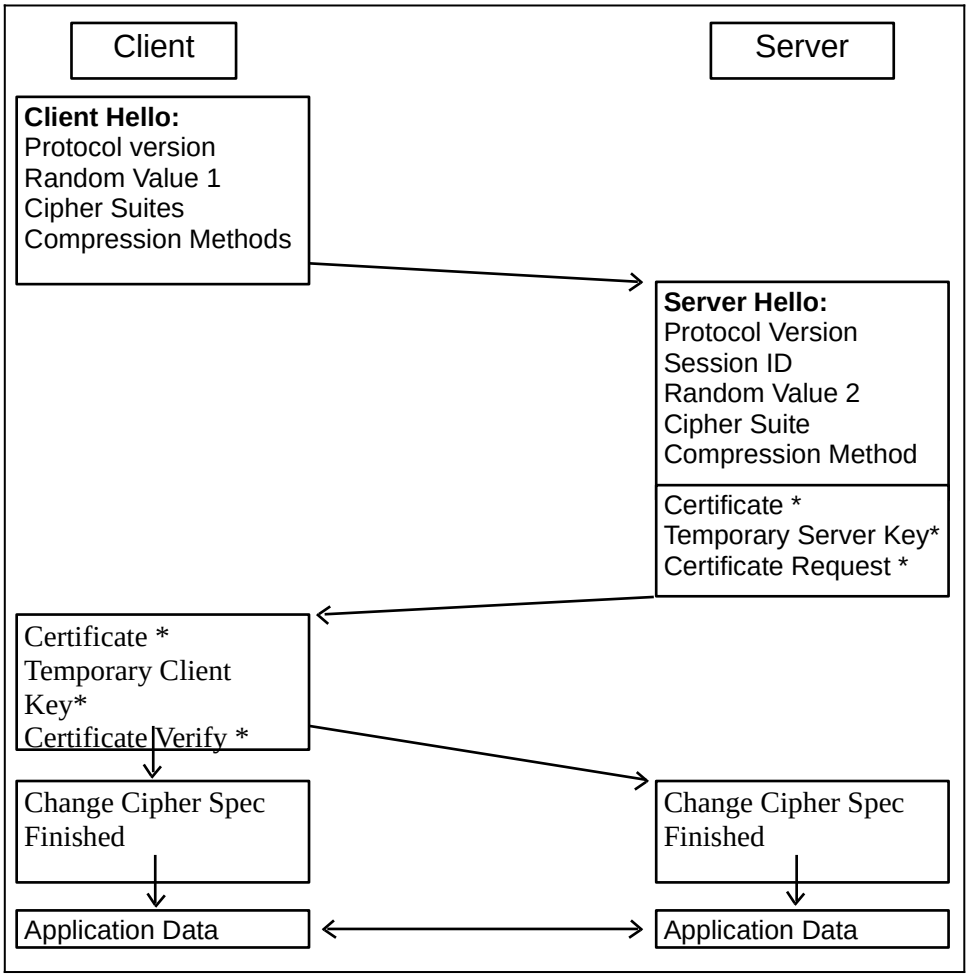
4.4 The Secure Sockets Layer

The Secure Sockets Layer protocol is currently the de facto standard for secure communications on webbrowsers. It is an application independent, transparent protocol that provides a secure connection between two applications. In webbrowser it is available as the HTTPS protocol, which actually simply is HTTP channeled through a SSL channel. HTTPS has been assigned well known port number 443 in order not to confuse the normal HTTP protocol on port 80.

SSL provides connections that are private and reliable [FREIER96]. Both ends of the connection may remain anonymous but can be authenticated by the protocol as well. SSL does not offer non repudiation as this requires application level message logging and bookkeeping.
To provide these services SSL relies on several cryptographic protocols. SSL itself does not prescribe any of these protocols, it is a framework in which they operate. Commonly supported are DES, RC4, RSA, MD2 and MD5.
The SSL protocol consists of two layers. The lower one, the SSL Record Layer is build upon a reliable communication protocol such as TCP. Its goal is to provide a private and reliable data stream between the two parties. On top of this run higher level protocols, that control the parameters in which the Record Layer has to operate. On top of these applications transparently can run their own protocols.

4.5 The SSL Handshake Protocol

One of the protocols in the upper layer is the SSL Handshake protocol. It is used to initialize a connection between two parties. Besides negotiating which ciphers should be used for the connection it also allows the parties authenticate eachother. A typical handshake performs the following steps:
The client sends a *client hello* to which the server has to reply with a *server hello*. These hello messages establish the protocol version, the session ID, the ciphers to use and the compression method. Additionally two random values are exchanged that will be used to make replay attacks impossible.
Next the server will send its certificate, if authentication is requested. If the server does not have a certificate or it is for signing only, a server key exchange message may be sent. In that case the server generates a temporary key pair and sends the public part of it. The connection will be anonymous then. An authenticated server may optionally request a certificate from the client.
Subsequently the server will send a *server hello done* to indicate the completion of the hello-message phase of the handshake. It will then wait for a response from the client.
If the server has requested a certificate, the client now has to send either its certificate or a *no certificate alert*. The client further has to send a client key exchange message, analogous to the server key exchange message. The content of that message depends on the public key algorithm selected during the hello phase. Finally the client sends a *change cipher spec* and switches to the new cipher, keys and secrets. With the new settings it sends a *finished* message. Now the server changes to the new settings to and notifies the client by replying with another *finished* message. This completes the handshake. Now the client and server may start to exchange application layer data.
An abbreviated version of the handshake is available to resume a previous session or to duplicate an existing one. The appropriate session is indicated using the session ID established during the full handshake. All information exchanged during that handshake is simply reused.



***** *indicates optional or situation dependent messages that are not always sent.*

Figure 5*: An SSL handshake*

4.6 Crypto in Java

Java version 1.0 does not provide support for cryptography. As a consequence several packages have been developed by third parties. The modularity of Java allows easy integration of these packages with existing applications. For example all flows of data, whether to network connections, to files, etc., are handled via subclasses of InputStream and OutputStream. To enable encryption one simply has to connect these streams to the appropriate encryption classes.

As of version 1.1 the JDK includes APIs for cryptography. Available are implementations of DSS, MD5 and SHA to allow authentication, integrity verification and non repudiation. Encryption algorithms will be added in future versions as part of an SSL implementation. (Actually JavaSoft already has written Java based encryption classes. They are used in the domestic version of their SSL enabled HTTP server, Jeeves. Since US export regulations do not permit the international distribution of these classes yet, they are omitted from the JDK).

For increased flexibility, JavaSoft introduced the concept of 'security package providers' (SPPs). It allows users to add or replace crypto algorithms by versions written by other vendors, in a way that is completely transparent to the applications using them. The default provider is the `sun.security.provider` package which contains an implementation of the earlier mentioned algorithms

written completely in Java. Other implementations could take advantage of optimized native code or special purpose hardware.
The 1.1 version JDK also has provisions for certificates. It is possible to create standard X509 certificates with help of the 'javakey' tool. One not only can use these to identify users or sites, but to sign applets as well. The HotJava browser can be configured to give trusted signed applets more privileges.

4.7 Key Management

Proper management of encryption keys is at least as important as the strength of the encryption algorithm. This paragraph looks at two aspects: key generation and key storage.
When a secure session is started a session key has to be generated. For best security this key should be a completely random number. There are several methods to obtain secure random numbers, ranging from hardware based solutions such as measuring radioactive decay to timing random key strokes. The latter was popularized by the Pretty Good Privacy (PGP) mail program and is considered one of the better options for PC software. Earlier versions of Netscape's SSL were hacked because of its weak key generation: it used a random number generator seeded with the current time, a value that contains not enough entropy.
Java 1.1 includes a random number generator that is claimed to be cryptographically secure by JavaSoft. However the JDK 1.1 documentation does not specify how it is implemented nor what algorithms or methods third party Java implementations should use. Therefore it is not clear yet whether this claim holds water.

We'll now have a look at key (and certificate) storage. Netscape stores certificates and keys encrypted on the local harddisk. When they are needed, the user has to type a password to unlock them. The JDK and HotJava currently lack this protection however. The identity database (containing both all present private and public keys) is simply stored as a serialized version of its runtime incarnation. Jean-Paul Billon has demonstrated that due to security bugs the complete database (including all private keys) can be stolen by an untrusted applet and send back to its server [BILL97]. It is not clear yet how the database will be protected in the final release of JDK 1.1.
A disadvantage of storing the certificate database on a harddisk, be it a local one or on the server of a diskless NetComputer is that a user is restricted to using a computer where it can be accessed. Therefore at NTEX Harbinger another storage medium is investigated: the smartcard. Smartcards are currently introduced in Holland by all banks to implement electronic cash. Within the next months devices to couple smartcards to PCs will become available for the price of a cheap mouse. A lot of Network Computers too will be equipped with smartcard sockets. The advantages of smartcards are clear: they provide a better mobility for the user and are more secure than storing (encrypted) keys on insecure harddisks. Moreover it helps to solve the problem of distributing top level CA certificates: a smartcard issued by the bank can be trusted to contain a valid root CA certificate. The user simply has to insert his or her smartcard in order to be able to use them.
In contrast Netscape currently doesn't have any scheme for validation of these certificates: some root level certificates are simply included in the browser, the user has to hope it does not download a modified browser. There are no provisions yet to add or renew CA certificates other than insecurely from the Internet. Even if provisions are added to add certificates supplied by floppy disk or another medium, the user friendliness wouldn't be near that of smartcards.

4.8 Combining SSL and Java

SSL can be a good solution for securing WWW interaction. Since it is transparent, it can be used to secure any HTTP based communication, including downloading of applets. However use of SSL in Java applets itself is hampered by the lack support for SSL in the Java APIs: it cannot simply use the browsers SSL implementation to do any other SSL based communication than sending HTTP requests. At least one commercial SSL implementation written completely in Java is available however [PHAOS96]. But there are other reasons as well why SSL may not be the best solution:

- Since SSL is a network layer protocol and not an application level one, it cannot base its encryption policy on the content of the messages: if some parts of the communication need to be private but others only need to be authenticated, everything still has to be encrypted.
- SSL's export is restricted by the US government: only 'crippled' versions that use short 40 bit encryption keys may be exported. It has been demonstrated that keys of this length can be cracked with reasonable effort: a 40 bit RC5 key that the RSA company challenged to crack was found by brute force in 3.5 hours by a graduate student using 250 computers.
- SSL's provisions for extensibility and interoparability add complexity to the protocol. Also it is very conservative from a security standpoint for best security. For example often two hash functions are used in tandem to keep the overall protocol secure in case one of them may contain a flaw. These provisions are necessary since the clients in which SSL is incorporated should be able to run securely without having to update them often. In a Java solution however both the SSL client and server are stored on the server. Therefore no provisions for version control are needed. Security fixes can be applied immediately without having to update each client at the system of the end user.
- SSL is optimized for bulk transfer of relatively large message blocks. Normal WWW communication normally consists of large data blocks (such as HTML pages, pictures, etc.) that are returned as reply of a small request message. High throughput of data is more important than low latency. For Java applets the converse may be true: typical thin client style applets may send lots of small user event messages to the server. A low response time is more important than high throughput: waiting some seconds before a requested web page is displayed may be acceptable to a user, but waiting 2 seconds every time a button is clicked on, is not acceptable. Unfortunately SSL imposes a relatively high overhead on the transfer of a message block. (It compresses, pads the block and calculates several hashes before encrypting.) Depending on the application, it may be possible to cut down a lot of this overhead by using a customized protocol. A point of concern with such an 'all Java' approach is encryption speed. Tests done with a non optimized Java implementation of the RC4 algorithm showed an encryption rate of approximately 75 KB/s on a Pentium 133 under the standard JDK 1.02 Java runtime. Just In Time compilers can improve this number dramtically. In the light of current transmission speeds on the Internet the achived rates can be qualified as sufficient. Future Java environments probably will include native encryption libraries, removing the speed concern completely.

This does not mean SSL isn't useful for Java encryption: it can be used for authentication and for transportation of session keys from the server to the client

(or the other way around). This simplifies the Java part to an implementation of a fast and simple symmetric cipher.

4.9 Conclusions

Strong cryptography is vital for the security of Client/Server systems such as Java. Besides the obvious utility of encryption for privacy, the possibility to sign code will enhance the security of Java based applications. The current Java Development Kit still lacks a lot of functionality however, and the parts that are implemented (such as applet signing) are either shown to be note secure yet or at least not tested thorougly enough.
The Secure Sockets Layer is a good effort to increase the security of WWW communications. Its integration in most webbrowser make it an attractive basis for building cryptographically secured Java applications. Its immediate appliance is hindered however by the lack of a proper interface between SSL and Java. Furthermore several of its properties are be not desirable for many Java applications. Therefore in many cases a hybrid solution is preffered, in which SSL is combined with JDK supplied parts and cryptography packages provided by third parties. Hardware based support for certificates in the form of smartcards may be an interesting option, as it offers several advantages over current software only schemes.

5 Visual Web

This chapter looks at an example of a Java based system developed at NTEX Harbinger and how it may be secured.

NTEX Harbinger focuses on research to leverage the use of the Internet for commerce. This has resulted in the development of the WebHost Application Manager (WHAM!) [ODEK96]. The WHAM introduces transaction oriented, session based operation to the WWW. Users no longer just download HTML pages from a HTTP server. Instead they log on to the server that keeps track of the session. Applications at server can interact with the users, based on their personal information. A good example of a WHAM improved website is that of the ANWB, a Dutch automobile club with approximately three million members. ANWB members can view information related to their situation, order insurances customized to their needs, etc.

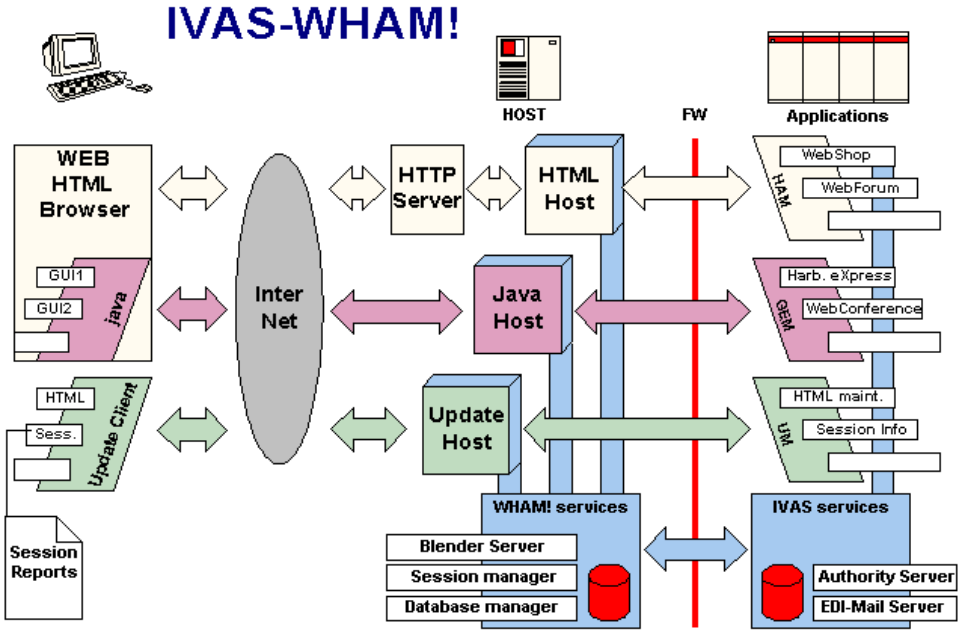


Figure 6*: Overview of the WHAM and Visual Web*

Although the WHAM enabled the availability of advanced applications to the WWW, these applications still had to use the standard HTML controls to interact with the user. It was felt that this was not sufficient for many applications. To remove this restriction a new technology was incorporated in the WHAM, named 'Visual Web' [HUET96]. Visual Web provides the user a generic, dynamic terminal that connects with the application on the server. This terminal, the Java GUI Terminal (JGT) is implemented as a Java applet that is automatically downloaded at first use. The user doesn't have to install any software at all. The application dynamically builds the user interface by putting all needed controls in the terminal. With Visual Web it has become possible to reuse existing applications in an internetworked environment. Instead of rewriting the application in Java, only the user interface of the application has to be coupled to the Visual Web server side system, a significantly simpler effort.
The Visual Web approach has a lot in common with X-Windows: at the user system runs a generic thin client environment in which the application running on

the server dynamically can build a user interface. However it is different in some respects:

- The JGT can be kept much simpler, since it does not have to include every possible control in advance: new controls can be added transparently to the user when needed.
- The JGT is more flexible: if necessary, specialized controls may be added that go beyond user interfacing alone.
- The JGT does not have to be installed by the user, since it is automatically downloaded by the Java runtime environment that is part of the browser.

The fact that the user does not have to install any software and uniformly can use applications from any computer, highly enhances the user friendliness of Visual Web based programs. The aim is to enable inexperienced computer users to utilize advanced network based applications.

The first application to be developed for Visual Web is Harbinger Exchange: a mail exchange that couples EDIFACT with the Internet. Its mail user agent is based on the JGT.

Applications like the Harbinger Exchange have serious security concerns. It should not be possible to read or modify the messages exchanged by it. Therefore all communication between the JGT and its server should be encrypted. When this investigation started, in July 1996, Java hadn't any provisions for cryptography at all. Though some browsers provided SSL, the proposed Network Computer did not. (It has become clear now that the NC will incorporate SSL [ORCL96]). The limitation that Java applets could not store information at the client complicated the situation. The following scheme was proposed to initialize a secure session:

1. The applet downloads the public key of the server.
2. The applet asks the user for user name and password for the usual logon.
3. As the name and password are typed by the user, the applet times the key strokes to generate a random number to be used as a session key.
4. The session key is transferred to the server by encryption with its public key.
5. The logon information and all further communication is encrypted with the session key.

This scheme has two possible weaknesses:

1. It is not resistant to 'man in the middle' attacks: a third party with access to the communication link between the client and the server could substitute the servers public key with its own, when it is downloaded by the applet. This could be avoided if the servers public key could be validated with help of a certificate authority. Network Computers have to be able to verify their runtime code as part of their bootstrap procedure. Therefore it was felt that validation of the server key would not impose significant problems.
2. The keystroke timing may not provide enough randomness. The Java API includes a method that returns the current time in milliseconds. Tests showed that it actually may be less accurate. Further regularity may occur due to the used time slicing algorithms for multi threading. These algorithms are not specified by the JDK and therefore can differ from platform to platform. As the real-time clock never was intended for secure random number generation the amount of bits taken from each measurement should be rather conservative. With an estimated number of 12 keystrokes for a login one may not expect more than 40 random bits, the absolute minimum for casual encryption.

With the introduction of cryptographic key management in JDK 1.1 and SSL in the NC reference the above scheme has become obsolete.
Now a hybrid solution has become possible: SSL can handle authentication of the user and the server, the login phase does not have to be performed in Java anymore. Since communication between the JGT client and the server are not HTTP based, SSL cannot be used for it. Moreover, as discussed earlier, SSL is not suited particularly well for encrypting communication of thin clients like the JGT. Instead this connection will be encrypted using Java, which has been proven to provide a sufficient encryption rate. When Java's cryptography capabilities have matured enough, SSL may be abandoned completely, since application level cryptography can be customized better.

6 Conclusions

Java is a promising technology for Client/Server applications. Its portability already has made it a de facto standard for Internet oriented programming. Contrary to many other contemporary systems, security has been an important aspect of its design. The Java language helps developers to avoid common programming mistakes that could have security repercussions. On the other hand, several parts of Java could have been designed better with respect to security. Java's approach of executing remote code is still a relatively new concept and its implementation has to undergo more maturation. The current beta release of the upcoming Java version (JDK 1.1) still contains a lot of security bugs. It is expected that these are fixed before JavaSoft releases the final version of JDK 1.1, since the success of Java depends on trust in its security. Current WWW browsers cannot be trusted yet to provide an environment in which trusted and untrusted applets safely can be run simultaneously.

Cryptography will play an important role in Java security. Encryption is a must for any Client/Server system in an insecure environment such as the Internet, not just for Java. Certificate based authentication and digital signatures improve Java's security perspective since it allows to selectively execute only code from known and trusted suppliers. Current software based provisions for handling certificates do not provide optimal security and are not user friendly. An alternative that is both more secure and more user friendly is storage of keys and certificates on smartcards. Since smartcards currently are introduced to the general public on a massive scale by the Dutch banks, availability of smart cards and support technology will increase dramtically. Since there is not much experience yet with using smartcards for this purpose, further research, including practical tests on an implementation is recommended.

7 Acknowledgments

This paper is the result of a preliminary investigation performed at the NTEX Harbinger Company in Rotterdam. It is part of the curriculum for my study in computer science at the technical university of Delft.
I would like to thank Mr. Heijnsdijk and Arthur Nederlof for their support during the investigation and writing of this paper. I also would like to thank Mr. Schoorl and Cees van Huet for proof reading the paper and their comments and suggestions. Im grateful to the people at JavaSoft that provided the full source to the Java Development Kit and answered my questions about Java security.

Appendix A: Perils of the security manager

The JDK 1.02 relies heavily on the contents of the stack to implement its security policies. Particularly it often counts how many calls deep a stack frame can be found that is associated with a class loader. The resulting number is called the ClassLoader depth.
An example of broken ClassLoader depth usage can be found in the protection of the `java.lang.Thread.stop()` method. The security policy with respect to this method is that applets only may invoke `stop()` on threads of (other) applets. Threads of applets are identified by the fact that they all are contained in an `AppletThreadGroup`.
Now let's look at the implementation:

```
public final stop() {
   stop(new ThreadDeath());
}
```

`Thread.stop()` simply calls `Thread.stop(Throwable o)` which is implemented as:

```
public final synchronized void stop(Throwable o) {
   checkAccess();
   resume(); // Wake up thread if it was suspended; no-op
otherwise
   stop0(o);
}
```

This seems secure, since `checkAccess()` is called before any attempt to stop the thread is made. `Thread.checkAccess()` simply calls the security manager:

```
public void checkAccess() {
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkAccess(this);
   }
}
```

Finally the `AppletSecurity` security manager performs the actual check:

```
checkAccess(Thread t) {
   if ((classLoaderDepth() == 3) && !(t.getParent()
instanceof
       AppletThreadGroup))
      throw(new SecurityException());
}
```

For an exception to be thrown, the ClassLoader should be 3 stack frames deep. This means that if an applet calls `Thread.stop(Throwable o)` on a system thread the attempt is blocked correctly. But Thread.stop() still is permitted since the ClassLoader lies 4 frames deep in that case.
The bug above demonstrates the weaknesses of the current security check method:

- A method (`stop()`) cannot rely on the access checking of another method (`stop(Throwable)`).
- The security manager cannot be implemented without detailed knowledge of the class it has to protect. Conversely modifying that class without detailed knowledge of the security manager may void its protection.

In effect independent development of SecurityManagers and system classes is not possible anymore.

Appendix B: Report of a security bug in HotJava 1.0 preBeta 2

The following bug report was send to the JavaSoft HotJava team. It describes some security holes that allow any applet to break the security sandbox. The HotJava team has not responded to the report yet.

Description:
Any applet can alter the security policies set in Edit->Preferences->Applet Security. Untrusted applet sources can be reconfigured to be trusted sources.

Impact:
Any applet can gain access to the local environment.

Affects:
HotJava 1.0 preBeta 2. Parts of the attack probably also can be used against earlier (JDK1.0x based) HotJava versions.

Discussion:
The applet HackHotJava included below demonstrates the vulnerability. It will set the security policy of the first identity in the list of software publishers to either High (strict security) or Low (minimal security). More sophisticated applets that go beyond proof of concept could add new identities to the lists as well. The functionality of the applet is simple: First it establishes access to the BasicSecurityPrefs applet. Then it simulates user interaction with this applet to set the desired policy settings. The HackHotJava applet uses the following weaknesses and bugs in the HotJava browser to accomplish this:

- `Component.getParent()` allows it to get access to the document panel containing both applets.
- `Container.getComponents()` will let it find the BasicSecurityPrefs applet that the `AppletContext.getApplets()` method kept hidden.
- `Object.getClass()` allows it to see the actual class the BasicSecurityPrefs applet rather than the java.applet.Applet base class.
- `Class.getMethod()` allows it to see members that are not in its name space.
- `Method.invoke()` allows it to access methods that are both outside its name space and in a restricted package.
- The BasicSecurityPrefs applet contains public methods that clearly are not for public use.
- The HotJava security manager has an empty `checkCaller()` method. This method should throw a `SecurityException` when the current context is not to be trusted. `CheckCaller()` is called by the SecurityGroups class to disallow tampering with security groups.

The attack used by the HackHotJava applet is but one of many attacks that could be used against the current HotJava implementation. A lot of HotJava objects contain public members that could be security sensitive. The same applies to several classes in the JDK 1.1 API.

Workarounds:
None.

Appendix C: An HTML based attack on HotJava

The current versions of the HotJava browser (1.0 preBeta 1 and 2) have been shown to use an insecure mechanism to handle the configuration of the browser. This can be used to build HTML pages that trick the user into configuring the browser to an insecure setting.

In HotJava the interface to its preferences is implemented as a collection of HTML pages containing several applets. Typically there are one or more PreferencesApplets that will modify the actual configuration of the browser. Other applets implement the actual controls, such as buttons or checkmark boxes. If such a button (for example an “apply” button) is pressed the applet sends a message (“apply”) to all the PreferencesApplets on the page. The weakness is that these applets trust any message send to them. Furthermore all control applets can be configured to display any text and send any text. Therefore a malicious HTML author can design a page that looks harmless, but in fact will modify the preferences when used.

The following demonstration applet will set the applet security setting of the first entry in the ‘publishers’ list to minimal security if all buttons are pressed from in left to right order. It essentially is a Trojan horse in HTML.

```
<html>
<head><meta name=type content="hotjava/utility"></head>
<center>
<applet code=sun.hotjava.applets.PrefsButtonApplet width=70
height=30>
<param name="label" value="First">
<param name="message" value="publishers">
</applet>
applet code=sun.hotjava.applets.PrefsButtonApplet width=70
height=30>
<param name="label" value="Next">
<param name="message" value="low">
</applet>
<applet code=sun.hotjava.applets.PrefsButtonApplet width=70
height=30>
<param name="label" value="Last">
<param name="message" value="apply">
</applet>
</center>
<applet code=sun.hotjava.security.BasicSecurityPrefs
width=350 height=0>
</applet>
<applet code=sun.hotjava.security.IdentityViewer width=200
height=0>
</applet>
<applet code=sun.hotjava.security.BasicPermissions width=540
height=0>
</applet>
</body>
</html>
```

References

[BILL97] Security Breaches in the JDK 1.1b2 Security API, J.P. Billon 1997 http://www.dyade.fr/actions/VIP/SecHole.html

[CRYP97] The Cryptix Crypto API V2, Phaos 1996 http://www.systemics.com/software/cryptix-java/alpha.html

[DEAN96] Java Security: From HotJava to Netscape and Beyond, Drew Dean, Ed Felten, Dan Wallach, Princeton University 1996 http://www.cs.princeton.edu/sip/pub/secure96.html

[DOD85] The Orange Book, US Department of Defense 1985 http://www.disa.mil/MLS/info/orange

[FREI96] The SSL Protocol Version 3.0 Internet Draft, Freier 1996 ftp://ietf.cnri.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt

[HUET96] Second Generation Internet, C. van Huët, NTEX Harbinger 1996

[JAVA97] JDK 1.1 beta 3 API documentation. JavaSoft 1997 http://www.javasoft.com/JDK1.1/docs

[ODEK96] Advanced applications for the WWW, R.G.J. Odekerken, NTEX Harbinger 1996

[ORCL96] The Network Computer Reference Profile, Oracle 1996 http://192.86.154.91/nc_ref_profile.html

[PHAOS96] The SSLava SSL implementation, Phaos Technology 1996 http://www.phaos.com

[SUN95-1] The Java Language Specification, Sun 1995 http://www.javasoft.com/doc/language_specification

[SUN95-2] The Java Virtual Machine Specification, Sun 1996 http://java.sun.com/java.sun.com/doc/vmspec/html/vmspec.html

[SUN96-3] The Java Card API FAQ, Sun 1996

http://www.javasoft.com/products/commerce/doc.javacard_faq.html

[SUN96-4] The Java Language Environment, J. Gosling & H. McGilton 1996 http://java.sun.com/doc/language_environment

[SUN96-5] JavaOS: A standalone Java Environment, Sun 1996 http://www.javasoft.com/nav/read/JavaOS.cover.ps

[SUN96-6] The JavaSoft Forum 1.1, Sun 1996 http://www.javasoft.com/forum/securityForum.html

[UTAH95] The Utah Digital Signature Law,
Utah Department of Commerce 1995
http://www.gvnfo.state.ut.us/ccjj/digsig

Abbreviations

ANWB	Algemene Nederlandse Wielrijders Bond
API	Application Programming Interface
AWT	Abstract Windows Toolkit
CA	Certificate Authority
DES	Data Encryption Standard
DSS	Digital Signature Standard
EDI	Electronic Data Interchange
EDIFACT	Electronic Data Interchange For Administration, Commerce and Transport
GEM	GUI Event Manager
GUI	Graphical User interface
HTML	Hyper Text Markup Langue
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IAC	Inter Applet Communication
JDK	Java Development Kit
JGT	Java GUI Terminal
JVM	Java Virtual Machine
JRE	Java Runtime Environment
MD5	Message Digest number 5
NC	Network Computer
RC4	Ron’s (or Rivest’s) Cipher (or Code) number 4
RFC	Request for Comments
RMI	Remote Method Invocation
RSA	Rivest, Aldeman, Shamir
SHA	Secure Hash Algorithm
SPP	Security Package Provider
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
URL	Uniform resource Locator
WHAM	Web Host Application Manager
WWW	World Wide Web