University of Helsinki
Department of Computer Science

Java and Internet Security

Antti Viljamaa
Jukka Viljamaa

Helsinki, May 1998

University of Helsinki
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

Java and Internet Security

Antti Viljamaa
Jukka Viljamaa

Department of Computer Science, University of Helsinki
May 1998
30 pages

Abstract

The Java programming language has significantly eased the development of Internet applications and networking software in general. At the same time it has also raised concerns about the already weak security procedures on the Internet. In this report we describe the main prevention mechanisms against the data communication threats and examine the Java security architecture and how it is prepared for possible security holes. We also describe Java's support for writing secure network applications that use cryptography mechanisms.

This report is a product of the FRED project (FRamework EDitor for Java) ongoing at the universities of Helsinki and Tampere. The main goal of the project is to enhance the construction of reusable object-oriented software by improving the tools and technology supporting *frameworks* and *design patterns*. Java has been selected as the implementation language for the tools and applications going to be developed, because of its object-oriented nature, platform neutrality, and the overall research interest towards it.

Keywords: Java, Security, Cryptography, Software Development

CR Classification: D.3, D.4.6, E.3

Contents

1. Introduction

Internet was originally designed to be a collegial environment used among the academic community where free availability of information was more significant than security. Not until recently has protection been an important consideration, and thus there are many fundamental flaws in the Internet security infrastructure [Opp97, Bhi96, GVV97]. As these shortcomings are combined with the growing popularity of the Java programming language [ArG98, Sun98d] — particularly its mechanism for almost automatic distribution of executable programs (applets) over the network — we really must reevaluate our current security measures.

When using Java, the traditional security procedures, such as building *firewalls*, obtaining software only from trusted sources, and using anti-virus programs, do not prevent users from inadvertently loading and executing malicious code. We must have additional means for selecting the applets we trust enough to give access to our local resources and for specifying the resources we can expose without compromising the overall security. Luckily, the Java technology itself tries to guarantee security to some extent and helps in defining security policies. There are also useful Java packages for writing secure network applications using cryptography.

In this report, we first discuss, on a general level, the security holes and threats on the Internet and also common means for protection. After that we examine Java's security architecture and basic language features enforcing security. We also explore Java's support for developing secure network applications, and finally, we examine some security flaws found in the Java implementations, view the criticism represented towards the Java security architecture in general, and discuss future directions of Java security.

2. Data Communication Security

The main threats against data communication security include *eavesdropping* (e.g., stealing credit card numbers), *password sniffing* (in order to gain access to sensitive information), *data integrity attacks* (e.g., modification of files), *Trojan horses* (masquerading as another party to access secure information), *denial of service attacks* (e.g., consuming computing power unnecessarily in order to prevent the system's normal functions), and the usage of *covert channels*[1] (exchanging information indirectly without an actual communication contact).

In this section we first take a look at the main prevention mechanisms against these threats: *authentication*, *access control*, and *cryptography*. Authentication means recognizing the entity that tries to access sensitive information or other resources. After authentication we can use access control mechanisms to determine which resources are to be revealed to the particular entity and which resources must be hidden. Cryptography can be used, e.g., when implementing authentication or guaranteeing confidential communication.

The current Internet infrastructure provides very little support for these security features, and several security protocols have been proposed to compensate its shortcomings. Some of them are described in chapter 2.3. In chapter 2.4 we represent general guidelines for evaluating security protocols.

2.1 Authentication and Access Control

Authentication can be considered as the most important security service, since all other services depend on it to some extent [Den82]. Together with access control it lays down the foundations for protecting important resources and information.

Various password schemes are by far the most widespread way to establish authentication in computer systems, although they are known to be vulnerable to many kinds of attacks [For94]. In an environment like Internet where eavesdropping is rather easy, one should use only *one-time passwords* to gain access to any resources. However, the use of such passwords is still fairly limited at the moment. The major problem with one-time passwords is their distribution: we have to know the utilized services beforehand and obtain a list of passwords from them. Cryptography provides means for implementing more robust and practical authentication methods (see chapter 2.2.2).

[1] Covert channels and denial of service attacks are generally unavoidable, because it is difficult to distinguish these actions from the normal system behavior.

2.1.1 The Trusted Computing Base

The placement of trust is a key issue in system security. To enforce authentication and access control one must have a trusted entity where dedicated system security code implements security protocols. This entity is often called a *trusted computing base* (TCB) [DoD85].

TCB-based security architectures for distributed object systems usually rely on a standard set of objects protecting the core system and preventing illegal access to resources. These *system security objects* function as a security shell just like reference monitors in OS memory management. They should be pluggable so that authorized management can easily configure the services the system provides and the algorithms used to implement these services. At the same time the architecture should make sure that as little trust as possible is needed in objects written by application developers. Examples of TCB-based security architectures for object systems include Java security architecture (see chapters 3.2 and 3.3) and CORBA security architecture [DBW97].

2.1.2 Capabilities and Access Control Lists

Once the user has been authenticated it is possible to control the resources that are accessible for her. The implementation of access control can be either *mandatory* or *discretionary*. Mandatory access control means enforcing fixed security policy on each security level for all clients, whereas discretionary access control allows to specify different policies for each client group [DBW97].

Any access control policy can be modeled as an *access matrix*. Under normal conditions the matrix becomes so huge and sparse that it is practical to implement it only either as *capabilities* or *access control lists*. In the first alternative the client has to represent the correct capability when accessing the server. This approach has several well-known shortcomings. It is, for example, difficult to revoke access rights or to confine access propagation. That is why the latter solution where the server stores lists of clients and their rights is usually better, although it may be more difficult to implement efficiently. [Gos91]

2.1.3 Firewalls

A more coarse-grained way to enforce access control is to use *firewalls* [Opp97, Jes98]. They are based on the observation that in an open networked environment all hosts must cooperate to achieve a uniformly high level of security. If the network is large enough this may become impossible, since the risk of having simple errors in configuration or inadequately chosen passwords increases as the number of hosts and users increases. This is why firewalls are used to segregate local network segments from one another and deny outside access from the Internet.

A firewall is an intermediate system plugged between the local network and the Internet (or between local network segments) to establish a controlled link where security and audit can be imposed.

Firewalls come in all shapes and sizes, ranging from relatively inexpensive public-domain software to expensive hardware/software systems. Modern firewalls are usually hybrid systems consisting of *screening routers* and *proxy servers* (see figure 2.1).

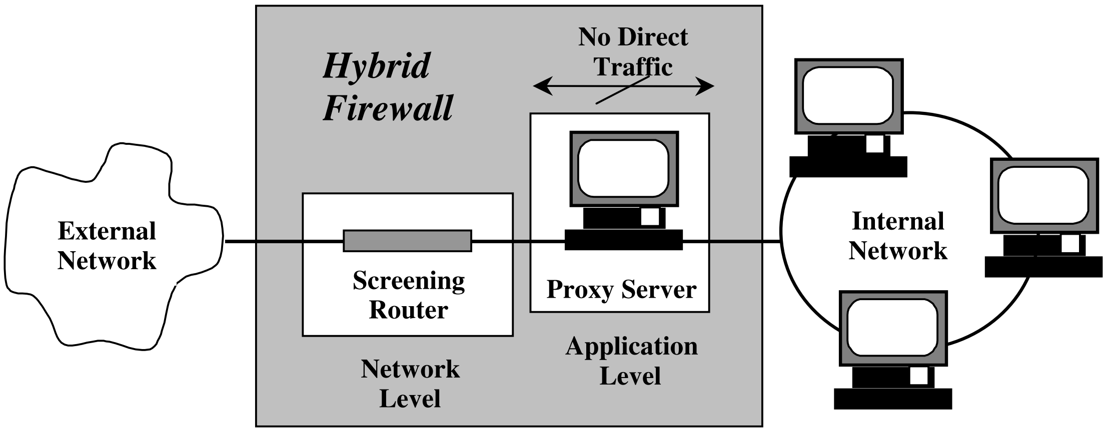

Figure 2.1: A firewall consisting of a screening router and a proxy server [Jes98]

A screening router applies a set of rules to each incoming IP packet to decide whether it is to be forwarded or not. Such packet filtering systems are appealing since they are generally fast and highly transparent to users. A proxy server, on the other hand, is a simplified, higher-security version of a common application such as TELNET or FTP. Since these proxies are application specific, a proxy is required for each protocol that is employed on the network. When a proxy is utilized, the system may perform complex logging and access control. It can provide extensive auditing, monitoring, and access restriction options that would be impossible with mere router-based security. This type of firewall may, however, be less transparent and slower than a screening router or packet filtering host.

Firewall technology is often neglected, because it is considered difficult to set up and because it affects the usability of services. Firewalls can also foster a false sense of security. Many attacks are in fact perpetrated by insiders and firewalls do not prevent problems with imported executable programs such as Java applets.

2.2 Cryptography

Even if an organization had chosen to employ one-time passwords and firewalls, these actions would still be generally insufficient to guarantee security. A more robust security solution should utilize cryptographic means to enforce (1) *authentication and access control* (with, e.g., digital signatures and certificates), (2) *confidentiality* (communication between parties should be restricted to the parties involved in the transaction), (3) *data integrity* (data should not be modifiable in transit or in storage), and (4) *nonrepudiation* (neither party should be able to deny having participated in a transaction) [Bhi96, AbN96].

2.2.1 Traditional Symmetric Cryptography

Traditional cryptography makes use of *symmetric algorithms* where two communicating entities share a common secret key that is used to encrypt and decrypt messages. The key includes information about possible place shifts and replacements that must be executed for each byte in order to encrypt or decrypt a message. If we choose a key whose length is equal to the length of the message and calculate the result (encryption or decryption) as a XOR (exclusive or) function[2] of the message and the key, we have a simple implementation of a classic *Vernam cipher* [DiH79] (see figure 2.2). A message encrypted with a Vernam cipher cannot be decrypted without the original secret key. Even if a possible attacker would try all possible byte combinations as the key, she would only result in all possible messages of the same length (including also apparently meaningful messages). A mathematical proof of Vernam algorithm's unbreakability was published by Shannon in 1949 [Sha49].

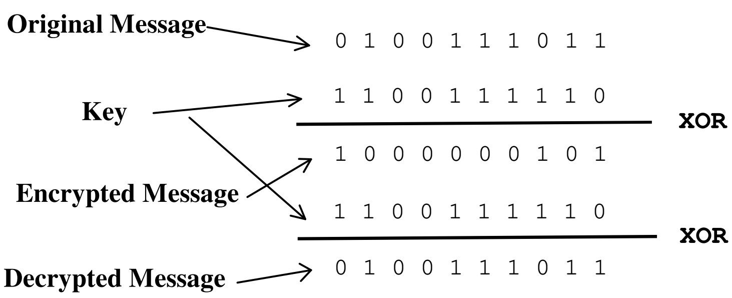

Figure 2.2: A classic Vernam cipher using a XOR function

The best known (and probably most widely used) traditional private key technology is the Data Encryption Standard (DES) defined in 1977 [NBS77]. It uses a 56-bit key and was originally intended to be implemented on hardware level. DES was developed by IBM, but is now endorsed by the U.S. Government's National Security Administration (NSA). This has lead to rumors that NSA might have deliberately left back doors and weaknesses in it.

2.2.2 Public-Key Cryptography

The main problem with classic encryption algorithms is key distribution: how do we distribute keys safely to communicating parties if the communication line itself is untrusted? If a communication network has a million users and billions of connections between them, the key distribution becomes unfeasible[3].

---

[2] Every resulting bit is calculated from the two corresponding original bits. A resulting bit is set to 1 only if one of the original bits (not both) is 1.

[3] Especially, if we use a Vernam cipher we need a key of the length of the actual message and a new key for every new message.

Public-key cryptography introduced by Diffie and Hellman in 1976 [DiH76] significantly simplifies the problem of key distribution. It also offers a possibility to sign a message with a digital signature that verifies the sender of the message. The public-key cryptography functionality implemented in Java is described in chapter 3.4.

With the public-key cryptography a sender and a receiver can communicate safely using a public communication line and commonly known methods. Whereas classic encryption algorithms have one secret key that is used to both encrypt and decrypt messages, public-key algorithms employ two keys: one for encryption and another for decryption. Although these two keys are theoretically closely related (they are each other's inverse functions) they do not reveal each other if the function used as the key has an inverse function that is practically impossible to calculate. As a basis for these *one directional functions* Diffie and Hellman suggested discrete logarithms, finding prime factors of a large number, and generally all NP complete problems [DiH76].

In public-key cryptography, the public key is kept visible for all users and the other (private) key is kept secret. By keeping the encryption key public and the decryption key private we can implement a confidential data transfer policy (see figure 2.3).

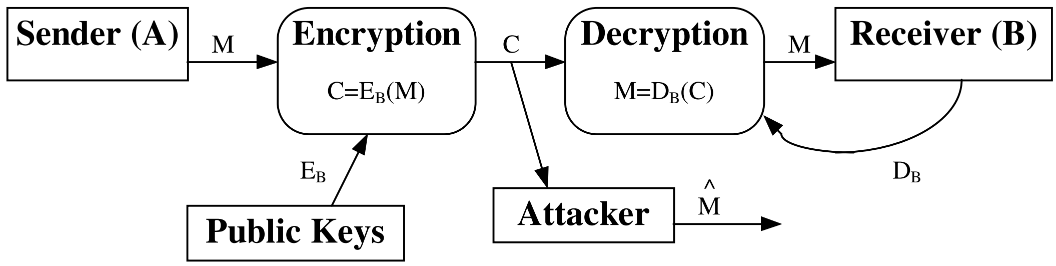

Figure 2.3: Safe data transfer with public-key cryptography

If the sender (*A*) wants to send a message (*M*) to the receiver (*B*), she encrypts the message with the publicly available key (E_B), which is used for encrypting all messages sent to *B*. *B* has the corresponding secret key (D_B) which she uses to decrypt the message. No attacker can reveal the message, because only *B* knows the decryption key[4].

If we want to employ public keys for digital signatures and message verification, the encryption keys are kept private and the decryption keys public [Omu90], see figure 2.4. It is also assumed that the system offers a publicly available hash function (*H*) that produces a string of a certain length from the given string[5]. The hash function must fulfill the requirement that it is practically impossible to find two strings that produce the same results.

4 Assuming that it is not possible to calculate the decryption key (D_B) from the encryption key (E_B).

5 The result of such a hash function is often called a *message digest*.

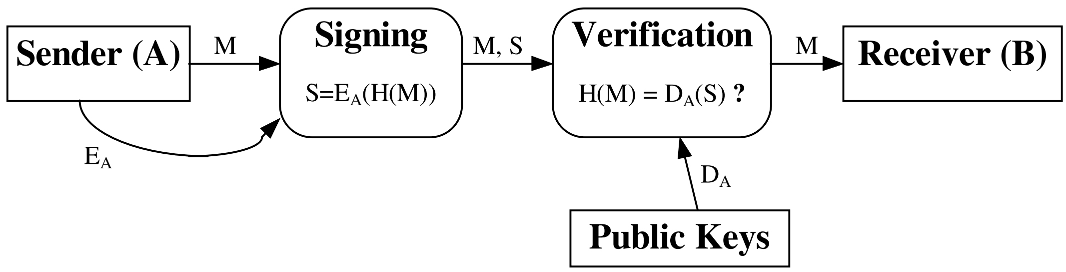

Figure 2.4: Digital signatures and message verification with public-key cryptography

Sender (*A*) produces her signature (*S*) with her private encryption key (E_A) and the public hash function. A sends both the original message (*M*) and the signature (*S*) to the receiver (*B*). *B* verifies the incoming package by checking that the hash function produces the same string from the message as the public key (D_A) produces from *A*'s signature. In this way the receiver can verify that the sender was really *A* (because nobody else knows the function E_A and thus no one else could have falsified *A*'s signature). In addition, *B* can be sure that the message is unchanged (otherwise $H(M)$ would not be equal to $D_A(S)$).

The most famous real-world implementation of the public-key cryptography is *RSA* developed by Rivest, Shamir, and Adleman [RSA78]. RSA is based on one directional functions that rely on the fact that calculating large primes (at most 200 digits) is easy, but finding the prime factors of the product of two numbers of that size in a feasible time is practically impossible.

Another well-known public-key crypto system for encrypting and decrypting files and for verifying email messages is *Pretty Good Privacy* (PGP) [Zim95]. PGP runs on many platforms and can be hooked into many news and mail readers to make secure online communication simple to users.

PGP applies a combination of RSA and the *IDEA* cipher algorithm [LLM92] to provide a high degree of security. It is widely believed, by experts in cryptography and computer security, that PGP-encrypted files cannot be broken even by sophisticated intelligence agencies.

In addition to cryptographic algorithms, proper functioning of a public-key system requires the ability to match specific keys to their owners. To that end, public-key certificates, such as those defined within X.509 [ITU93], are widely used. These certificates bind public keys to specific entities and allow for a third party to validate this binding.

2.3 Common Internet Security Protocols

There are various identified security problems related to individual Internet services, like the numerous security holes in the `sendmail` program and the default "trust" mechanism of BSD remote commands. Also difficult configuration of many security sensitive software packages has opened security holes.

These insecure applications may cause serious damage, but an even more sever fact is that there are no actual built-in security services in the Internet's TCP/IP protocol suite. The lack of these services can be summarized as follows [Bhi96]: (1) most lower-level protocols are based on broadcast messages, which makes it possible for any machine in the LAN to eavesdrop network traffic, (2) there is no authentication mechanism so it is straightforward for one system to impersonate another system, (3) the checksums and sequence numbers employed in TCP are so easy to predict that it is simple to modify contents of messages.

A number of cryptographic protocols on various levels in the TCP/IP stack have been proposed to provide the security services currently lacking in the Internet protocols [Bhi96]. In this section we describe some of the most popular security protocols in use today. They all utilize very similar cryptographic algorithms, but differ in the way the security services are provided and in their position (layer) in the protocol stack (see figure 2.5).

There has been a lot of discussion whether the security should be implemented lower in the network level or higher near to the application level. Those who say that security should be placed lower in the stack argue that in this way security could be implemented transparently to the end-users and application developers, effectively solving many problems at once. Their opponents, on the other hand, argue that lower-level solutions try to do too many things and are therefore impossible to implement efficiently. They also claim that application-specific security protocols allow more flexible configurations.

Starting from the bottom of the TCP/IP stack, the security techniques on the network layer include *AH* (Authentication Header [Atk95a]) and *ESP* (Encapsulating Security Payload [Atk95b]). AH deals with message headers and ESP with the actual message contents. While these methods are also incorporated into some existing IP implementations, their main target has been to make way for the next generation IP (the IPv6).

By far the most common session layer security solution is the *SSL* (Secure Sockets Layer [FKK96]). It is embedded, for example, in Netscape Navigator.

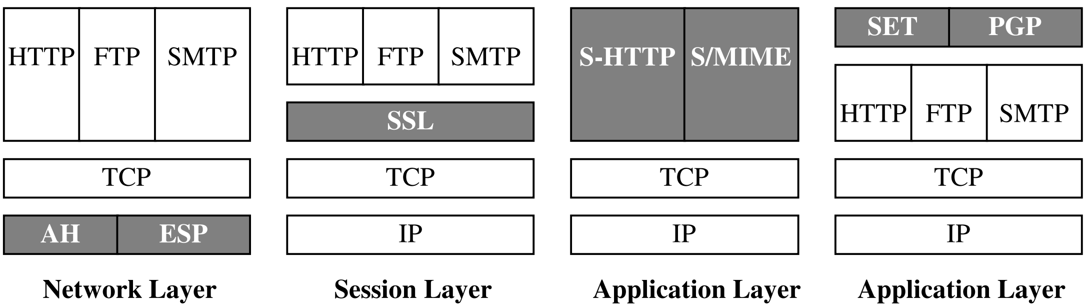

Figure 2.5: Common security services added to the TCP/IP stack [Bhi96]

When securing individual applications one should concentrate on three sets of applications that are most important, i.e. email, WWW browsers, and electronic payment mechanisms. For email the *PGP* (Pretty Good Privacy [Zim95]) protocol has been prevailing for some time now (see chapter 2.2). Also

S/MIME (Secure/Multipurpose Internet Mail Extensions [RSA98]) is widely used in email applications. It adds digital signatures and encryption to *Internet MIME* (Multipurpose Internet Mail Extensions) messages. For WWW there are two dominant systems: *HTTPS* which is implemented on top of the above mentioned SSL, and *S-HTTP* [ReS97] which is an extension to the HTTP protocol. Electronic payment systems are not yet very popular or established, but it seems that the *SET* (Secure Electronic Transaction [SET98]) protocol (supported by, e.g., Visa, Mastercard, and Microsoft) might be the winner in that field.

One problem related to all security enhanced applications is that modifications must be implemented individually for each application. It would be better to have a more generic approach. A step in that direction is *SSH* (Secure Shell [SSH98]), which allows users to securely log on to remote hosts, execute commands, and transfer files.

2.4 Evaluating Security Protocols

Security protocols used in distributed systems are prone to design errors. There have been numerous reports of security holes found in protocols widely used on the Internet. That is why it is important to be able to assess the properties of any security protocol before making a decision to employ it. In [AbN96] Abadi and Needham give some informal guidelines to eliminate the most obvious flaws while designing security protocols. Their guidelines provide also a good basis for evaluating the quality of existing security protocols.

2.4.1 Quality

First of all, the trust relations in any security protocol (e.g., which components of the system are assumed to be part of the trusted computing base) should be precisely stated so that discussion on why the dependence is necessary would be possible.

The conditions for a message to be acceptable should be clearly visible in the design of a security protocol. It should be possible to deduce that the message belongs to this protocol, and in fact to the current step of a particular execution of the protocol. The interpretation of every message should depend only on its content, and the sender's and receiver's identities must be mentioned if they are essential to the interpretation of the message. The same applies also for any encoding used to present the meaning of the message.

Many errors in security systems are due to unclear or false assumptions about timestamps or sequence numbers employed. For example, the use of counters can serve in guaranteeing newness, through a challenge-response change. However, the counter should be protected so that an intruder cannot simulate a challenge and later replay a response. Also, if timestamps are used as freshness guarantees by reference to absolute time, then the difference between local clocks at various machines must be less than the allowed age of a valid message. Furthermore, the time maintenance becomes part of the TCB everywhere. One additional common mistake is to think that the key's recent usage guarantees its freshness, while a key may well have been used recently, still being old and thus possibly compromised.

When encryption is concerned, one should be aware of why encryption is being done and avoid unnecessary encryption for efficiency reasons. It should be kept in mind that encryption is not a synonym to security.

Yet another common misconception about building secure systems is that combining secure subsystems automatically results in a secure compound system. Maintaining security in a composite systems is not, however, trivial task, and it has become a subject of intense research. Formal definitions of the conditions under which composition of systems is safe can be found, for example, in [HeT96] and [McL96].

2.4.2 Cost and Usability

The cost of implementing security mechanisms is a very crucial factor. If a new technology makes it easier or cheaper to obtain the same level of security than with conventional means, it would be very attractive. On the other hand, if new technology increases the security with a corresponding increase in cost, the organization must weigh the cost against the risks being averted [Sur96].

The addition of security layers can also have a negative effect on the performance or usability of systems [Bhi96]. If security mechanisms are too time-consuming or difficult to use, they can decrease the productivity. Users who find the policies difficult to follow may ignore the policies or implement them haphazardly. With languages such as Java with built-in security services it is easier to develop secure applications transparently to the end-user.

3. Java Security

Most of the network security research has concentrated on open distributed systems at the host-to-host level. Distributed object systems, such as CORBA and Java environments, are more dynamic and complex thus setting security requirements that are harder to achieve. Nevertheless, in secure object systems the same principles and methods for authentication and access control apply, although they must be implemented at the object level rather than at the host-to-host level [DBW97].

At the moment many computer and network systems try to maintain security by hiding the inner details and policies of the system. It is assumed that if the system is presented as a black box, then no one will expend the effort needed to discover the hidden vulnerabilities. The developers of the Java environment have chosen to publish the details of the Java security model and its full implementation. The obvious intention is to encourage security researchers to examine the model and report security flaws before the possible attacks cause problems.

In this chapter we examine how security features have been dealt with in the Java language and environment. On a general level, Java tries to guarantee four security aspects: (1) it should be possible to *authenticate* the source of the executed applets, (2) one should be certain about the *integrity* of the data received over the network, (3) the *encryption* mechanisms should enable confidential connections, and (4) the local resources on a client machine running an applet should be protected.

In the following, we first evaluate Java's core features that form a basis for the higher-level security features. After that we describe the basic *sandbox model* and the security objects it comprises. The concept of sandbox was practically the only security mechanism available in JDK 1.0. JDK 1.1 and 1.2 have introduced signed applets and extended sandboxes in order to separate trusted applets from untrusted ones and to facilitate the definition of different security policies for them. These features are examined in the third subchapter. The fourth subchapter describes Java's cryptography mechanisms. They can be employed, e.g., for signing applets, for managing encryption keys, and for creating Java programs with encryption capabilities. In the fifth subchapter we describe some weaknesses found in the Java implementations and summarize criticism represented towards the Java security architecture in general. The last subchapter is devoted to future directions of Java security.

3.1 The Core Security Features

Java's security architecture relies heavily on the core language features. Java is intended to be a simple and well-defined language — not only to enable the sandbox philosophy, but also to prevent programmers from making accidental mistakes. And, although Java's numerous extension packages may seem somewhat confusing at first, the core language has a relatively small number of language constructs and its syntax is familiar at least to all C++ programmers. Also, Java is strict in its definition of the language: all primitive data types in the language have a specific size and all operations are defined to be performed in a specific order.

Java is a strongly typed language. It enables extensive compile-time checking for potential type-mismatches and a number of run-time checks by the interpreter, including boundary checks for string and array accesses. In Java, all data attributes must be defined (with their type decided) before using them. The primitive type `boolean` is not type compatible with any other types. A Java compiler performs implicit primitive type conversions only for numeric types[6]. The `char` type is (a kind of strange) exception: it can be assigned to `int`, `long`, `float`, and `double` types without a need for an explicit cast. An object can only be assigned to a variable with a type of the variable's static class or one of its super classes. Objects cannot be casted to their subclasses without an explicit run-time check. Strict compilation checks make Java programs more robust, avoid run-time errors, and prevent security loop holes. Also, classes, methods, and variables can be declared as `final`. This means that they cannot be modified after the declaration, which also prevents the overriding of trusted methods by malicious code.

Besides the type system, also Java's memory allocation and reference model is essential for the whole security architecture. Memory management is completely opaque to the programmer and controlled entirely by the underlying run-time platform. Java defines no pointer arithmetic, and memory layout decisions are not made by the compiler, as they are, e.g., in C and C++. Instead, all references to methods and instance variables in compiled Java code (byte code) are done via symbolic names. The symbolic names are not resolved to real memory addresses until at run-time by the Java interpreter, and the memory layout will potentially differ depending on the characteristics of the hardware and software platforms on which the Java system executes. Thus, a programmer is not able to guess the actual memory layout of a program just by looking at its source code, and she cannot create memory references that intentionally point to sensitive areas and possibly trash system variables or give access to private information. Naturally, the lack of pointers also reduces the risk of accidentally corrupting memory.

Java uses *garbage collection* to automatically release memory once it is no longer needed. This prevents memory leaks and accidental release of the same piece of memory twice, which is a common bug in C and C++ programs. Garbage collection also eliminates some security holes. For instance, explicit release of memory might allow a programmer to successfully execute an illegal cast and thus, e.g., examine some private properties of a system object[7].

---

[6] Variables of a type for smaller numbers can be assigned to variables of a type for greater numbers.

[7] To achieve this, the programmer could first create her own object with similar attributes but weaker protection modifiers, allocate memory for her object, and then release the memory (however, keeping the reference). If she now allocates memory for the system object and the just released piece of memory is used for it, then also the original reference points to the system object making its private attributes visible.

3.2 Basic Sandbox Model

The JDK 1.0 version of Java introduced a sandbox model for securing the execution environment from malicious applets (or applications), see figure 3.1. The sandbox approach means executing remote code in a limited environment safely within the restrictions placed by the security model. In other words, an applet can do anything within the sandbox, but has only limited rights outside it. This implies, e.g., that applets are not allowed to read from or write to the local disk, or to make network connections to any other hosts but the applet source server.

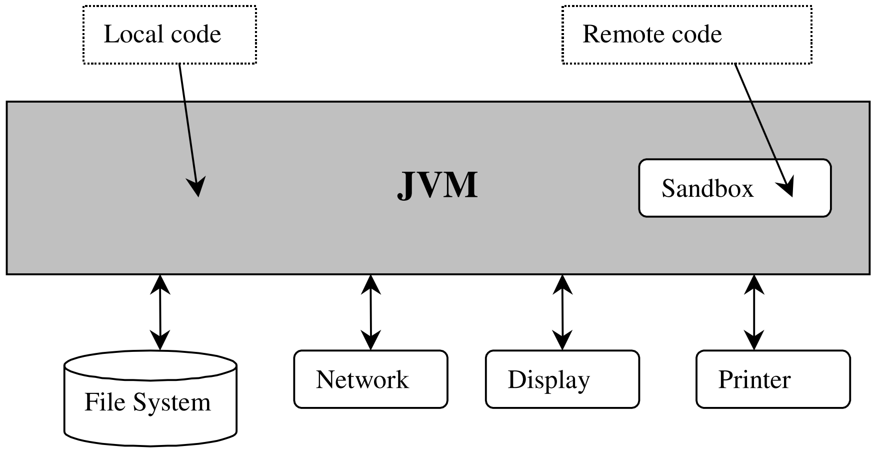

Figure 3.1: The sandbox security model

The sandbox ensures that even if a user downloads a malicious applet, it cannot damage the local machine. In JDK 1.0, all applets were executed in sandboxes and only local code was trusted to have full access to the vital system resources. This made it easy to protect the local host from any illegal actions, but at the same time it made applets somewhat useless, at least for any serious computing. The crucial improvements and additions to this security scheme — the ability to recognize trusted applets and to develop customized security policies — were introduced in JDK 1.1 and 1.2. These features are examined in the next chapter, but first we take a closer look at the foundations of the sandbox model.

The design of the sandbox model relies on the basic Java features described in the previous chapter: lack of pointers, automatic memory management, and run-time memory layout. These language properties make it impossible to reference memory outside the sandbox. In addition, there are three major system components that comprise the sandbox: *class loader*, *bytecode verifier*, and *security manager*[8].

Class loader is the first link in the security chain. It fetches class files from the network and converts them to internal data structures representing the classes. It also enforces the *name space hierarchy* ensuring that loaded classes are placed in a different scope than the standard Java classes. When a class is imported from across the network it is placed into the private name space associated with its origin.

8 JDK 1.2 has introduced a new important component called *access controller.* It is described later.

When a class references another class, it is first looked for in the name space for the local system classes, then in the name space of the referencing class. By this, imported classes cannot replace system classes. Also, system classes can never accidentally reference classes in imported name spaces — they can only reference such classes explicitly. Similarly, classes imported from different sources are separated from each other.

Although Java compilers perform thorough type checking, there is still the possibility of an attack via the use of a hostile compiler. Browsers that download class files do not know if the bytecode was produced by a trustworthy compiler. That is why the Java run-time system applies a byte code verification process to ensure that the code loaded from an arbitrary host does not violate Java's restrictions.

The byte code is in linear form, so type checking requires global data flow analysis as in the back-end of an optimizing compiler. The analysis is complicated further by the existence of exceptions and exception handlers. On the other hand, the byte code verifier also enhances the performance of the interpreter: some run-time checks that would otherwise have to be performed for each interpreted instruction can be eliminated.

Security manager enforces the boundary of the sandbox. Whenever an applet performs an action which is a potential violation, the security manager decides whether the action is approved or not. Also a Java application can have its own security manager object. The `SecurityManager` class in the `java.lang` package is an abstract class that provides the programming interface and partial implementation for all Java security managers[9].

3.3 Signed Applets and Extended Sandboxes

As described in the previous chapter, already the first version of Java (JDK 1.0) had a simple security design that forced downloaded applets to run in a restricted sandbox. JDK 1.1 introduced a mechanism for grouping Java files into Java Archives (JAR) that could be digitally signed. Based on the digital signature, it was now possible to *authenticate* the creator of the applet and *verify* that the applet is unaltered. System administrators were able to give full privileges for those applets that carried trusted signatures. The JDK 1.1 security architecture is represented in figure 3.2. Local and trusted remote code was executed without restrictions whereas untrusted remote code was forced into the sandbox.

JDK 1.1 did not include a common security API, and it was quite difficult to specify different levels of access privileges: applets had either full privileges or almost no privileges at all. Although it was possible to define different security policies by subclassing and customizing the `SecurityManager`

---

9 JDK 1.2 provides more flexible ways to define different security policies without actual coding (see chapter 3.3).

and `ClassLoader` classes[10], one basically had to start from scratch, which was rather error-prone and easily lead to security holes.

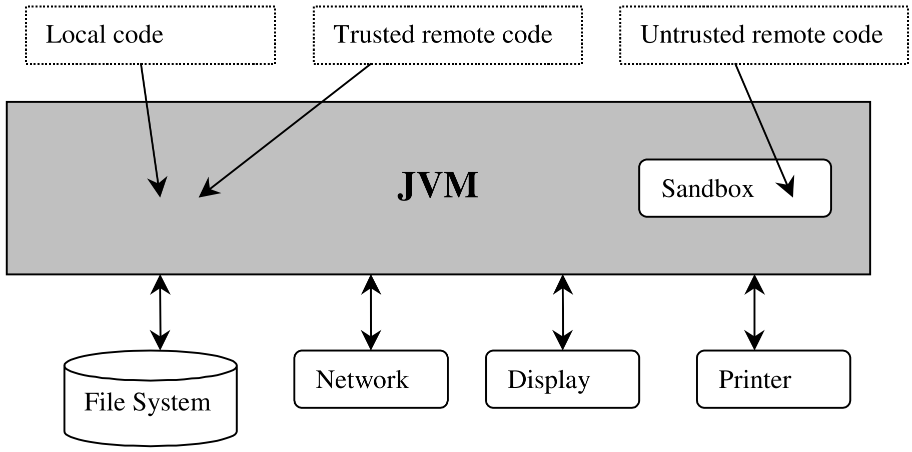

Figure 3.2: The JDK 1.1 security model

The JDK 1.2 [Sun98b] security architecture [Gon98a] offers more support for defining fine-grained access policies: loaded classes are placed into customized sandboxes at load time, based on the digital signature and the origin of the code [GMP97]. The user or administrator can usually define basic security policies without actual coding using an external policy file.

In JDK 1.2, Java code is executed with different permissions. Both applications and applets can now be executed under the same security restrictions. Prior to version 1.2 of JDK, applications were by default started without any security manager and all operations were allowed.

JDK 1.2 uses *protection domains* [Gon98b] for grouping and isolating individual units of protection. The two main domains are *system domain* which has access to the protected external resources, and *application domain* that accesses these resources via the services (APIs) provided by the system domain. There is a *system security policy* that specifies the protection domains and their *permissions*. A permission represents an *action* that is allowed to be performed against a system resource (*target*). For instance, a permission for reading the file `/foo.txt` could be created with the following Java statement: `perm = new FilePermission("/foo.txt", "read").`

Each class and its methods belong to some protection domain that is determined by the *code source* of the class. Code source includes the URL address of the class and a set of cryptographic public keys that correspond to the private keys that were used to sign the class. In the current implementation all classes loaded from the same URL and signed with the same keys are placed in the same protection domain.

A Java security object called *access controller* checks all method calls that ask the Java API to perform sensitive actions, and determines whether these methods are allowed to execute or not (see figure 3.3).

[10] As is done in, e.g., in the HotJava browser [DFW96].

To make this decision, the access controller examines the method call stack starting from the top and progressing downwards. For every method in the stack, the access controller verifies that the requested action is allowed in the method's protection domain. If all methods in the call stack pass the check, the requested action is permitted, otherwise a security exception is thrown and the request is denied.

A method may belong to a *privileged* domain. If the access controller encounters such a method, it aborts the checking and accepts the whole request immediately, even though further down the stack might be a method from an unreliable applet. This mechanism allows, e.g., untrusted applets to access local disk via restricted (privileged) Java API methods[11]. Although the method verifying mechanism includes lots of optimization strategies, it adds additional overhead to the execution of Java methods.

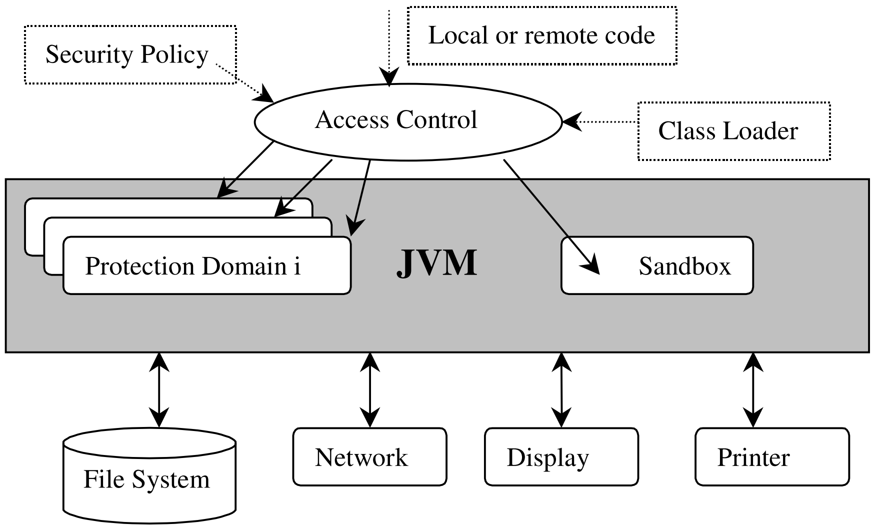

Figure 3.3: The JDK 1.2 security model

3.4 Java's Cryptography Functionality

The first release of Java Security in JDK 1.1 introduced *Java Cryptography Architecture* (JCA) [Sun98a], which refers to a framework for accessing and developing cryptographic functionality for the Java platform. In JDK 1.1, JCA included APIs for digital signatures and message digests. JDK 1.2 significantly extends JCA. It includes a certificate management infrastructure to support X.509 v3 certificates, and offers new configurable and extensible access control. *Java Cryptography Extension* (JCE) in turn extends the JCA API to include encryption and key exchange. JCA and JCE together provide a complete, platform-independent cryptography API.

3.4.1 Engines and Providers

Java Cryptography Architecture has two main objectives: (1) it allows users of the programming APIs to utilize cryptographic concepts, such as digital signatures and message digests, without concern for

[11] Such a method might, e.g., load a font file from the local disk.

the implementations or even the algorithms being used to implement these concepts, (2) it offers tools[12] for signing and authenticating applets, for managing key certificates used in those processes, and for specifying the security policy of the Java environment in use.

To meet JCA's first objective, one needs to have algorithm and implementation independence. In JCA *algorithm independence* is achieved by defining types of *cryptographic engines* (services) and classes that provide the functionality for them. An *engine class* defines a cryptographic service in an abstract fashion without a concrete implementation. A cryptographic service is always associated with a particular algorithm, and it either provides cryptographic operations (like those for digital signatures or message digests), or supplies the cryptographic material (keys or parameters) required for cryptographic operations.

JDK 1.2 includes engine classes for calculating message digests of specified data, signing data, verifying digital signatures, generating cryptographic keys, and managing the parameters of a particular algorithm.

These engine classes provide the interface to the functionality of a specific type of cryptographic service. For each engine class, a particular implementation is requested and instantiated by calling a factory method on the engine class. A factory method is a static method that returns an instance of a concrete implementation class.

These implementation classes are defined in terms of a *Service Provider Interface* (SPI). That is, for each engine class there is a corresponding abstract SPI class, which defines the methods that cryptographic service providers must implement. *Implementation independence* is achieved using these providers. In this context a *provider* is a package or a set of packages that implement one or more cryptography services.

Each JDK installation has one or more provider packages installed. New providers may be added statically or dynamically. JCA allows users to query which providers are installed and what services they support. Clients may configure their run-time environment with different providers, and specify a preference order for each of them.

The standard provider of JDK 1.2 (called "SUN") includes implementations of the *DSA* digital signature algorithm [NIS92] together with compatible key generation and management services, as well as *MD5* [Riv92] and *SHA-1* [NIS94] message digest algorithms.

[12] These tools are implemented in Java, and in fact they can be considered merely as scripts that use the low-level Java APIs.

3.4.2 Public-key Cryptography and Digital Signatures

In the following example of Java's public-key cryptography functionality we will first generate a public-private key pair for the DSA algorithm using a key pair generator object (we do not care which provider supplies the implementation):

```
import java.security.*;
import java.security.spec.*;
// ...

KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
keyGen.initialize(1024);
KeyPair pair = keyGen.generateKeyPair();
```

After that the private key of the created key pair can be used to digitally sign some data:

```
PrivateKey privKey = pair.getPrivate();
Signature sig = Signature.getInstance("DSA");
sig.initSign(privKey);
sig.update(someData);
byte[] signature = sig.sign();
```

If now somebody wants to use our signed data and to verify our signature, we need to send her three things: (1) the data, (2) the signature, and (3) the public key corresponding to the private key we used to sign the data.

For the public key, we must extract the encoded key bytes so that we can put them in a file and send it to our receiver along with the files containing the data and the signature:

```
PublicKey pubKey = pair.getPublic();
byte[] encKey = pubKey.getEncoded();
```

Now, assume the receiver has copied the data bytes from the files to byte arrays named `data`, `signature`, and `encodedPubKey`, respectively. She can now execute the following code to verify the signature. The code also illustrates how to use a key factory in order to instantiate a DSA public key from its encoding:

```
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encodedPubKey);
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
Signature sig = Signature.getInstance("DSA");
sig.initVerify(pubKey);
sig.update(data);
sig.verify(signature);
```

3.4.3 Security Policies and Key Management

The `Security` class manages installed security service providers and other system-wide security properties. Normally, when using JDK 1.2 one does not have to write any Java code in order to specify the security policy. Instead, the policy is defined using a *policy configuration file*. JDK 1.2 includes a graphical tool called `policytool` for editing the file. Policy configuration files contain statements granting specified code sources sets of permissions, specifying which actions are allowed. The contents of a sample policy configuration file appear below.

```
grant signedBy "sysadmin", codeBase "file:/home/sysadmin/" {
  permission java.security.SecurityPermission "Security.insertProvider.*";
  permission java.security.SecurityPermission "Security.removeProvider.*";
  permission java.security.SecurityPermission "Security.setProperty.*";
};
```

This specifies that only code that was loaded from a signed JAR file (whose signature can be verified using the public key referenced by the alias name `sysadmin`) from beneath the `/home/sysadmin/` directory on the local file system can add or remove providers or set security properties.

Java cryptography architecture employs a database called a *keystore* to manage a repository of keys and certificates. The keystore is available to applications that need it for authentication or signing purposes. It is by default implemented as a file storing private keys protected with a password.

Currently, there are two command-line tools that make use of the keystore: `keytool` and `jarsigner`. The keystore is also employed by the default security policy implementation when it processes policy files specifying the permissions to be granted to code from various sources. Since the keystore is publicly available, JDK users can write additional security applications that use it. Users can also write their own keystore implementations, e.g., to protect their private keys in a stronger way than is done by the default implementation, or to use a different (not file-based) keystore format.

`keytool` enables users to administer their own public/private key pairs and associated certificates for use in self-authentication or data integrity and authentication services, using digital signatures. It also allows users to cache the public keys (in the form of certificates) of their communicating peers.

As an example of `keytool` usage, suppose you want to create a keystore for managing your public/private key pair and certificates from entities you trust. The first thing you need to do is to create a keystore and generate the key pair:

```
keytool -genkey -dname "cn=Mark Jones, ou=JavaSoft, o=Sun, c=US"
        -alias mykey -keypass kpi135 -keystore \working\mykeystore
        -storepass ab987c -validity 180
```

This command creates the keystore named `mykeystore` in the `working` directory, and assigns it the password `ab987c`. It also generates a public/private key pair for the entity whose *distinguished name* has a common name of Mark Jones, organizational unit of JavaSoft, organization of Sun and two-letter country code of US. It uses the default DSA key generation algorithm to create the keys. The command also creates a *self-signed certificate* that includes the public key and the distinguished name information. This certificate will be valid for 180 days, and is associated with the private key in a keystore entry referred to by the alias `mykey`. The private key is assigned the password `kpi135`.

So far all we have got is a self-signed certificate. A certificate is more likely to be trusted by others if it is signed by a *Certification Authority* (CA). To get such a signature, you first generate a *Certificate Signing Request* (CSR):

```
keytool -csr -file MarkJ.csr
```

This creates a CSR and puts the request in the file named `MarkJ.csr`. If you submit this file to a CA, it will return a certificate authenticating your public key. You need to replace your self-signed certificate with the one from the CA. But first you need a *trusted certificate* entry in your keystore that authenticates the CA's public key.

A certificate from a CA is usually either self-signed, or signed by another CA (in which case you need a certificate authenticating that CA's public key). Suppose you obtain a file named `VeriSign.cer` that is a self-signed certificate from VeriSign, Inc., authenticating that CA's public key. If you trust that this certificate is valid, then you can add it to your keystore via the following:

```
keytool -import -alias vs -file VeriSign.cer
```

This creates a trusted certificate entry in the keystore, with the data from the file `VeriSign.cer`, and assigns the alias `vs` to the entry. Now you can replace your self-signed certificate with the one from VeriSign (`VSMarkJ.cer`):

```
keytool -import -file VSMarkJ.cer
```

To illustrate the usage of certificates, suppose you have signed a JAR file called `bundle.jar` with the `jarsigner` tool:

```
jarsigner bundle.jar mykey
```

Clients that want to use the file will want to authenticate your signature. In order to do so, they need your public key. You supply this to them by sending them a copy of the certificate authenticating your public key. Copy that certificate to a file named `MJ.cer` via the following:

```
keytool -export -alias mykey -file MJ.cer
```

Given that certificate, and the signed JAR file, a client can use the `jarsigner` tool to authenticate your signature with a command such as the following:

```
jarsigner -verify bundle.jar
```

3.4.4 Java Cryptography Extension

The basic JDK 1.2 does not include a mechanism for creating encrypted messages based on public-key algorithms. This functionality is introduced in a separate package called *Java Cryptography Extension*

(JCE). Due to the U.S. export regulations, the package is (at least at the moment) only available in the U.S. and Canada, and it is difficult to obtain accurate information about it. In general, JCE provides classes for encryption and decryption, key exchange, and message authentication. It includes interfaces and implementations of ciphers (symmetric, asymmetric, block, and stream ciphers), secure Java streams, key generation, and key agreement. JCE is designed to be extendible with new algorithms and interoperable with other cryptography libraries.

3.5 Java Security Weaknesses

While many experts agree that the Java security model is basically sound, it has not been examined in sufficient detail. Some security faults have been reported (for example in Netscape browsers). The security holes typically arise from implementation errors[13], unintended interactions between browser features, and differences between Java semantics and bytecode semantics [Sur96, Pri98]. It should be noted that very few holes are due to the language itself.

Many consider the lack of full-fledged formal treatment of the language semantics to be the main weakness of Java, because the security of any language relies on the soundness of its type system. A recent attempt to provide a formal basis for understanding systems like the Java environment and to explore some of the security-relevant interactions between static typing and dynamic linking can be found in [Dea97].

In this chapter we concentrate on informally examining some of the most important security threats and how the Java environment is prepared for them. We also take a look at some weak points in earlier Java implementations and in the general security design.

3.5.1 Applets and Security Threats

Integrity attacks are prevented fairly effectively by the Java sandbox approach. However, since nowadays all the popular WWW browsers implement their own (inflexible) security policies that are somewhat incompatible (e.g., applet signing procedures are very different), Java software developers cannot rely on Java's portability to different browsers, until the browsers properly support JDK 1.2.

To forestall denial of service attacks the user (or a system administrator) can define the amount of resources an applet can use, but this naturally inconveniences all unsigned applets. Furthermore, the source of such attack can be easily hidden, e.g., by programming the attack to occur after a time delay.

Also the interapplet security is weak. Since applets can persist after the Web browser leaves the page which contains them, it becomes important to separate applets from each other. Otherwise, a hostile applet could sabotage a third party's applet.

[13] For example, although Java systems are not supposed to reveal memory layout, some older virtual machines implemented the `hashCode` method (available for every object) so that it returned the address of the object's internal storage cast to an integer.

The main method for preventing disclosure attacks in Java applets is to restrict socket connections to other sites on the Internet. These attacks are not totally avoidable, because various covert channels exist in browsers allowing applets to communicate with untrusted third parties [Sur96]. One example often mentioned is to use a hostile DNS server as a two way communication channel to an arbitrary host. An applet might then reference a fictitious name in the attacker's domain. This transmits the name to the attacker's DNS server, which could interpret the name as a message and then send a list of arbitrary 32-bit IP numbers as a reply. Repeated DNS calls by the applet establish a channel between the applet and the attacker's DNS server.

An untrusted applet cannot create a socket connection to other hosts than the server it was loaded from. In some Java implementations this feature has been forced in the following way. First, all names of the applet host are fetched. These names are matched against the names of the host the applet is trying to contact. The problem is that an applet can ask to connect to any hostname on the Internet, and the matched IP addresses of the host are asked from its DNS server. So, the applet can control which DNS supplies the list of IP addresses. There is nothing that prevents an attacker from creating a DNS server that provides false information. Using the attacker's DNS server to provide a pair of addresses, the applet can connect to any host on the Internet.

3.5.2 Critical Views on the Java Security Architecture

The weak point of the Java security architecture is that it totally relies on the popular notion of trusted computing base[14]. The assumption is that it is possible to interpose the TCB (sandbox) between subjects (applets) and the resources they wish to use. The TCB consults some trusted data base and decides whether the access is to be allowed. Assuming that the TCB is correct (and not penetrated), security is enforced. However, even if breaking the TCB is difficult, once it is done Java's security architecture collapses totally and the attacker can do as she pleases. What most people seem not to realize is that the programming language, the compiler, the run-time libraries, file servers, network servers, dishonest or incompetent programmers, and malicious misuses can all be part of the TCB.

An alternative model for TCB could somehow simulate real-world practices where we do not depend on a single trusted gatekeeper — at the very least we should have multiple independent gatekeepers so that a breach of one perimeter does not gain access to all. To achieve this goal the Java security architecture should be augmented with domain-structured protection and capability architectures together with one-time authenticators as rapidly as possible [Sun96][15]. Java also clearly needs a more explicit system for specifying trust relationships[16].

---

[14] As do most of the today's computer systems.

[15] These requirements are, at least in part, met in the JDK 1.2's domain-based version of the Java security architecture.

[16] Consider, e.g., the fact that digital signatures can only proof the origin of a software module, but even if we believed that the software developer had implemented correctly working code, it says nothing about the behavior with modules from other software vendors.

The current Java security model makes it necessary for every client to perform its own safety checking. The verifier and the JIT compiler/interpreter are located in the same physical machine as the Java virtual machine. As a result, the trusted computing base is very large. TCB for an enterprise that uses the current Java implementations is on the order of verifier, compiler and interpreter size multiplied by the number of instances of Java virtual machines, such as Java enabled browsers.

The size and distribution of TCB leads to a variety of security problems. First of all, security verification and access control become discretionary with the current Java implementations. Since the verification process is under the control of end-users, any individual within an organization may disable some or all of the checks performed on imported code. Even in the absence of any malice within an organization, buggy Java implementations will degrade security level because we cannot assume bug fixes installed on every machine with users often aggressively resisting any change to their system software. Furthermore, the current structure of Java based systems makes security management at the level of an enterprise almost impossible. Since access control decisions are at the discretion of users, there is no way to impose a coherent, enterprise-wide security policy.

In the Kimera project at the University of Washington Sirer, McDirmid, and Bershad are implementing a new Java security architecture based on factored components for security, performance, and scalability [SMB97]. By locating crucial Java virtual machine services at trust-domain boundaries, such as Intranet firewalls, they claim they can make safety enforcement mandatory, ease security management and reduce the processing requirements of Java endpoints. Under such a centralized security architecture, TCB is minimal and consists of small and simple components whose security can be more readily assured, and security upgrades and bug fixes do not require end-user assistance or action.

Further, audit trails may be kept at the secure firewall, where they would be safe from any tampering by malicious code. Since the firewall itself does not execute any foreign code, a bug in the verifier that allowed a rogue application to execute on an end-point would not put the audit trails at risk. The audit trail may then be used to prohibit further entry of such code until a security patch can be applied.

Finally, transparent modifications of the Java bytecode can be performed at the firewall in order to perform fine-grained security checks. For instance, the centralized security policy may allow foreign code to touch only certain portions of the file system. The firewall could inject some Java bytecode into every downloaded class to check the arguments to the file system's open routine to see if that particular instance of the call should be allowed.

It should be noted that Java is an entirely new kind of an open, distributed execution platform, which has introduced many difficult security problems. Java developers have devoted themselves to solving these problems, and it can be said that — at least now with the JDK 1.2's domain-based architecture — Java is reasonably safe. There is lots of work ahead, though. That is why, in the next chapter, we discuss the future directions of Java security.

3.6 Future Directions of Java Security

Current Java code signing mechanisms allow classes within the same archive to be signed with different keys, a class to be unsigned, signed with one key, or signed with multiple keys. Other resources within the archive, such as audio clips and graphic images, can also be signed or unsigned, just like classes can. This flexibility brings about the issue of interpretation. Should images and audio be required to be signed with the same key if any class in the archive is signed? If images and audio are signed with different keys, can they be placed in the same browser page, or should they be sent to different viewers for processing?

These questions are not easy to answer, and require consistency across platforms and products to be the most effective. At the moment, all images and audio clips are forwarded to be processed within the same applet class loader, whether they are signed or not. This temporary solution will be improved once a consensus is reached.

3.6.1 Access Control, Secure Class Loading, and Resource Management

Developers of Java encourage the use of `AccessController` in application code, while customization of a security manager (via subclassing) should be the last resort and should be done with extreme care. Moreover, a customized security manager should utilize the algorithms provided by `AccessController` whenever appropriate.

Also class loaders are very delicate, and the current class loading architecture is still not perfectly secure. Applets and applications can create class loaders only if the system security policy is configured to allow this to happen. In the future the creation of class loaders should not be so strictly controlled (at the same time still maintaining system security, of course).

On another often-discussed topic, resource consumption management is already relatively easy to implement in some cases (e.g., to limit the number of windows any application can pop up at any one time) but hard in other cases (e.g., to limit memory or file system usage). Such issues will be addressed in the future versions of Java's access control and resource management.

3.6.2 Subdividing Protection Domains

A potentially useful concept not currently implemented is that of *subdomains*. A subdomain is one that is enclosed in another. A subdomain would not have more permissions or privileges than the domain of which it is a subpart. A domain could be created, for example, to selectively further limit what a program can do.

For convenience, at the moment the system domain is thought of as a single, big collection of all system code. For better protection, though, system code should be run in multiple system domains, where each domain protects a particular type of resource and is given a special set of rights. For example, if file system code and network system code run in separate domains, where the former has no rights to the networking resources and the latter has no rights to the file system resources, the risks and consequence of an error or security flaw in one system domain is more likely to be confined within its boundary.

3.6.3 Users and Authentication

Today the notion of a *principal* (e.g., user) in Java is implicit because each virtual machine is owned by one user. In the future, there will be a need to extend the existing concept of protection domain to include the notion of *running-on-behalf of a principal*. Therefore, Java developers are going to provide the following features in the near future:

- explicit principal concept and classes,
- user authentication primitives (both password-based and otherwise),
- cross-protection-domain principal authentication protocols, and
- general mechanisms for authorization and delegation.

3.6.4 Object-Level Protection

Given the object-oriented nature of the Java programming language, it is conceivable that developers will benefit from a set of appropriate object-level protection mechanisms that (1) goes beyond the natural protection provided by Java the language and that (2) supplements the thread-based access control mechanism.

One such mechanism is `SignedObject` already available in JDK 1.2. Parallel to this primitive, Java will provide `SealedObject` that uses encryption to hide the content of an object. Due to current U.S. export control regulations on the use of encryption, the `SealedObject` class will likely be provided separate from JDK.

4. Conclusion

In this report we have described issues concerning network security in general, and the Java programming language's security features in particular. First we introduced the concept of trusted computing base (TCB) and its responsibilities to enforce authentication and access control. As methods for implementing access control, we represented capabilities, access control lists, one-time passwords and firewalls.

We recognized that a more robust security system needs cryptographic means, e.g., public-key crypto systems, to guarantee authentication, access control, data integrity, confidentiality, and nonrepudiation. After that we looked at some existing security protocol implementations and discussed ways to evaluate them.

We also described Java's basic security-related language features, especially the type system and memory management. Then we examined the evolution of the Java security architecture as well as the implementation of cryptographic functionality in Java. Finally, we discussed applet security and some basic weaknesses in the Java security architecture. We also listed some improvements that have been proposed.

For further information we would like to refer to WWW pages devoted to general network security issues: COAST Homepage [Coa98], Secure Internet Programming page [Pri98], and the Forum of Incident Response and Security Teams [Ley98]. The main source of information concerning Java security is Sun's Java security page [FrM96]. Sun has also a page devoted to frequently asked questions about Java security [Sun98c]. It contains, e.g., information about reported bugs and their fixes. Other pages related to Java security include, e.g., General Internet's Java security page [Gen98].

References

AbN96 Abadi M., Needham R., Prudent Engineering Practice for Cryptographic Protocols. IEEE Transactions on Software Engineering 22, 1, 1996, 6-15.

ArG98 Arnold K., Gosling J., *The Java Programming Language, 2nd Edition*. Addison-Wesley, 1998.

Atk95a Atkinson R., IP Authentication Header. Internet RFC 1826, Proposed Standard, August, 1995.

Atk95b Atkinson R., IP Encapsulating Security Payload. Internet RFC 1827, Proposed Standard, August, 1995.

Bhi96 Bhimani A., Securing the Commercial Internet. *Communications of the ACM 39*, 6, 1996, 29-35.

Coa98 COAST Homepage. Purdue University, Indiana. Internet: http://www.cs.purdue.edu/coast/, 1998.

DBW97 Deng R., Bhonsle S., Wang W., Lazar A., Integrating Security in the CORBA Architecture. *Theory and Practice of Object Systems (TAPOS) 3*, 1, 1997, 3-13.

Dea97 Dean D., The Security of Static Typing with Dynamic Linking. *Proc. of the Fourth ACM Conference on Computer and Communications Security*, Zürich, Switzerland, April 1997, 18-27.

Den82 Denning D., *Cryptography and Data Security*. Addison-Wesley, 1982.

DFW96 Dean D., Felten E., Wallach D., Java Security: From HotJava to Netscape and Beyond. *Proc. of 1996 IEEE Symposium on Security and Privacy*, May 1996. Internet: http://www.cs.princeton.edu/sip/pub/secure96.html.

DiH76 Diffie W., Hellman M., New Directions in Cryptography. *IEEE Transactions on Information Theory 22*, 6, 1976, 644-654.

DiH79 Diffie W., Hellman M, Privacy and Authentication: an Introduction to Cryptography. *Proc. of the IEEE 67*, 3, March 1979, 397-427.

DoD85 Department of Defense, Trusted Computer System Evaluation Criteria. DOD 5200.28-STD, 1985.

FKK96 Freier A., Karlton P., Kocher P., The SSL Protocol Version 3.0. Internet: http://home.netscape.com/eng/ssl3/draft302.txt, 1996.

For94 Ford W., *Computer Communications Security: Principles, Standard Protocols and Techniques*. Prentice Hall, 1994.

FrM96 Fritzinger J., Mueller M., Java Security. Sun Microsystems White Paper. Internet: http://java.sun.com/security/index.html, 1996.

Gen98 General Internet Inc., Java Security. Internet: http://java.miningco.com/msubsecurity.htm, 1998.

GMP97 Gong L., Mueller M., Prafullchandra H., Schemers R., Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *In Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997. Internet: http://www.javasoft.com/people/gong/papers/jdk12arch.ps.gz.

Gon98a Gong L., Java Security Architecture (JDK 1.2). Internet: http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html, 1998.

Gon98b Gong L., Schemers R., Implementing Protection Domains in the Java Development Kit 1.2. *In Proc. of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998, 125-134. Internet: http://www.javasoft.com/people/gong/papers/jdk12impl.ps.gz.

Gos91 Goscinski A., *Distributed Operating Systems — The Logical Design*. Addison-Wesley, 1991.

GVV97 Guynes C., Vedder R., Vanecek M., Security Issues on the Internet. *ACM SIG Security Audit & Control Review 15*, 2, 1997, 8-12.

HeT96 Heintze N., Tygar J., A Model for Secure Protocols and Their Compositions. *IEEE Transactions on Software Engineering 22*, 1, 1996, 16-30.

ITU93 International Telecommunication Union, Recommendation X.509 (11/93). Internet: http://www.itu.int/itudoc/itu-t/rec/x/x500up/x509_27505.html, 1993.

Jes98 Jesiek B., Internet Security — Firewalls. Internet: http://www.ee.mtu.edu/courses/ee465/groupb/fwll.html, 1998.

Ley98 Ley W., Forum of Incident Response and Security Teams. Internet: http://www.first.org/, 1998.

LLM92 Lai X., Massey J., Murphy S., Markov Ciphers and Differential Cryptanalysis. *Advances in Cryptology - Eurocrypt '91*, Springer-Verlag, 1992, 17-38.

McL96 McLean J., A General Theory of Composition for a Class of "Possibilistic" Properties. *IEEE Transactions on Software Engineering 22*, 1, January 1996, 53-67.

NBS77 National Bureau of Standards, *Data Encryption Standard*. FIPS Pub. 46, January, 1977.

NIS92 National Institute of Standards and Technology (NIST), The Digital Signature Standard, Proposal and Discussion. *Communications of the ACM 35*, 7, July 1992, 36-54.

NIS94 National Institute of Standards and Technology (NIST), Announcement of Weakness in the Secure Hash Standard. May 1994.

Omu90 Omura, J. K., Novel Applications of Cryptography in Digital Communications. *IEEE Communications Magazine 28*, 5, 1990, 21-29.

Opp97 Oppliger R., Internet Security: Firewalls and Beyond. *Communications of the ACM 40*, 5, 1997, 92-102.

Pri98 Princeton Secure Internet Programming Team, Homepage. Princeton University, Department of Computer Science, Internet: http://www.cs.princeton.edu/sip/, 1998.

ReS97 Rescorla E., Schiffman A., The Secure HyperText Transfer Protocol. Internet-Draft, Internet: http://www.ietf.org/internet-drafts/draft-ietf-wts-shttp-05.txt, 1997.

[Riv92] Rivest R., RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, April 1992.

RSA78 Rivest, R. L., Shamir, A., Adleman, L., A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, 2, 1978, 120-126.

RSA98 RSA Data Security Inc., S/MIME Central, Internet: http://www.rsa.com/smime/, 1998.

SET98 SET Secure Electronic Transaction LLC, SET Technologies. Internet: http://www.mastercard.com/set/technologies.html, 1998.

Sha49 Shannon, C. E., Communication Theory of Secrecy Systems. Bell Syst. Tech. J. 28, 10, 1949, 656-715. *Proc. of the IEEE 67*, 3, 1979, 397-427.

SMB97 Sirer E., McDirmid S., Bershad B., Kimera: A Java System Architecture. Department of Computer Science and Engineering, University of Washington. Internet: http://kimera.cs.washington.edu/, 1997.

SSH98 SSH Communications Security Ltd., SSH 2.0 Protocol Specifications. Internet: http://www.ssh.fi/drafts/, 1998.

Sun96 Sun Microsystems, JavaSoft Forum 1.1. Internet: http://java.sun.com/forum/securityForum.html, 1996.

Sun98a Sun Microsystems, Java Cryptography Architecture — API Specification & Reference. Internet: http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html, 1998.

Sun98b Sun Microsystems, Java Development Kit, Version 1.2. Internet: http://java.sun.com/products/jdk/1.2/, 1998.

Sun98c Sun Microsystems, Frequently Asked Questions — Java Security. Internet: http://java.sun.com/sfaq/, 1998.

Sun98d Sun Microsystems, The Java Home Page. Internet: http://java.sun.com/, 1998.

Sur96 Sureshkumar V., WWW Beyond the Basics — Chapter 14: Java Security. Internet: http://ei.cs.vt.edu/~wwwbtb/book/chap14/index.html, 1996.

Yel95 Frank Yellin, Low-level Security in Java. WWW4 Conference, December, 1995. Internet: http://www.javasoft.com/sfaq/verifier.html.

Zim95 Zimmermann P., *The Official PGP User's Guide*. MIT-Press, Cambrigde, 1995.