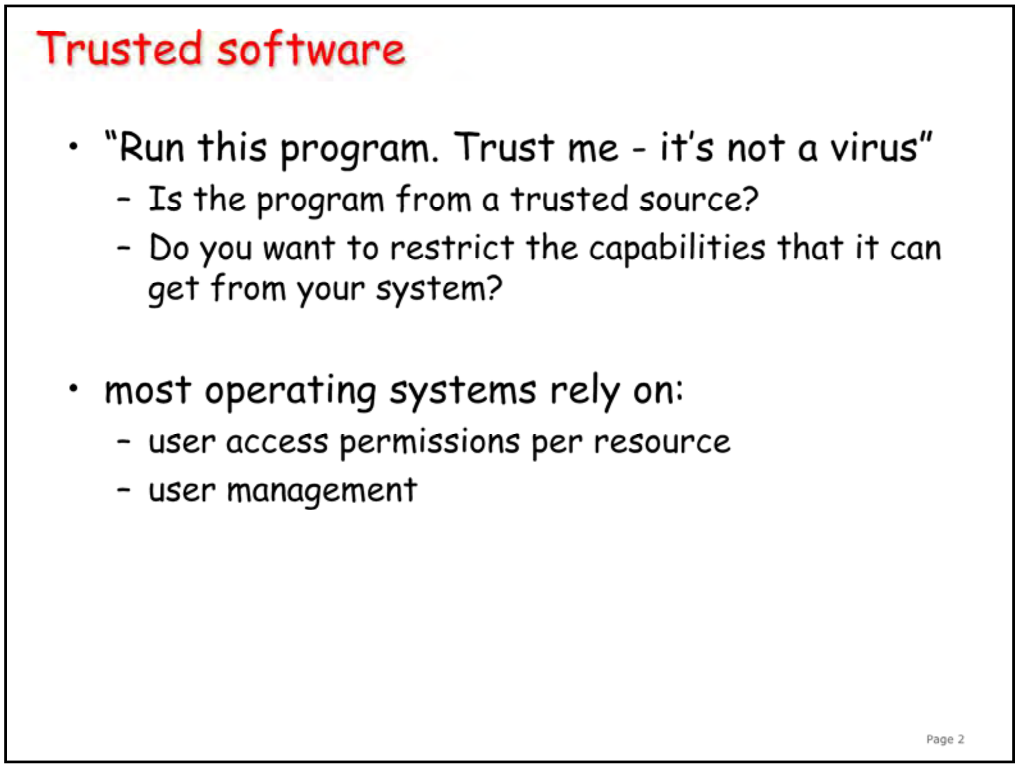
Sandboxing and Signed Software



Trusted software

- "Run this program. Trust me - it's not a virus"
  - Is the program from a trusted source?
  - Do you want to restrict the capabilities that it can get from your system?
- most operating systems rely on:
  - user access permissions per resource
  - user management

Page 2

One problem that is not addressed by firewalls, secure communications, and by the operating system itself is: what *safeguards can we impose when we want to run someone else's software?*

The general model today is that of trust: you trust the software that you install.

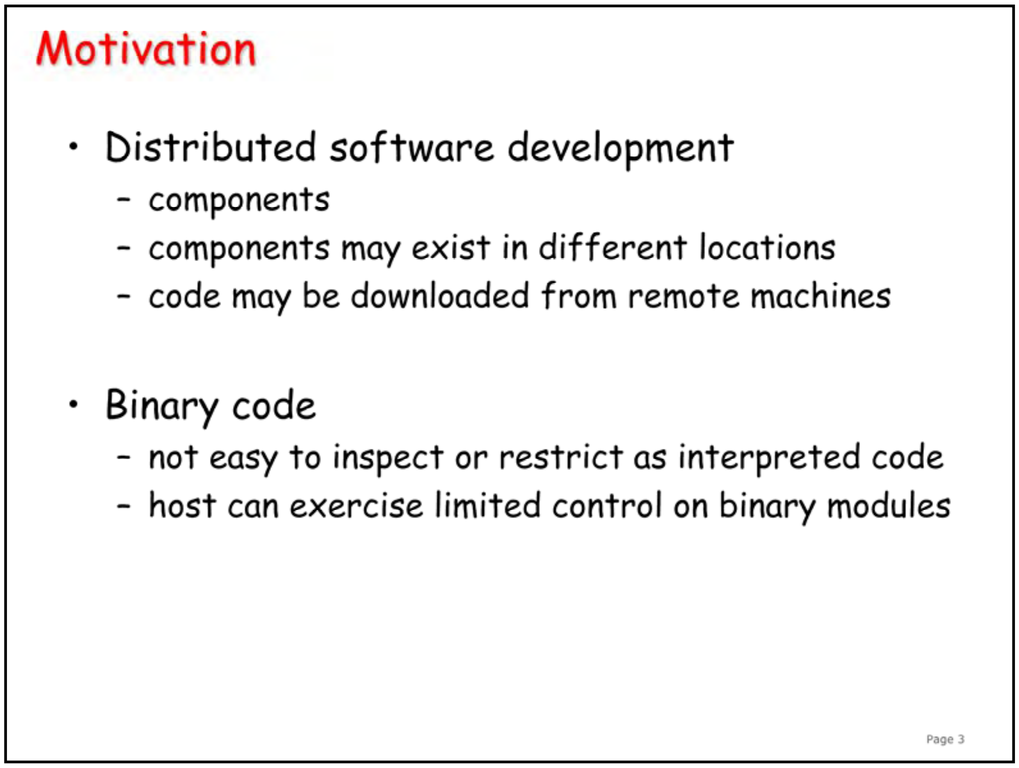
If you get it from a store, you trust the vendor not to do anything malicious and trust the integrity of the package because it is shrink wrapped. If you get the software from the web, you trust it because you downloaded it from a web site that you authenticated (with an X.509 digital certificate provided by the site during the SSL handshake).

You may rely on limiting the damage that a program can do by relying on the operating system to run it under a particular user ID (yours, most often) and having the operating system have appropriate access permissions set for the various resources it offers. This is particularly ineffective on most Microsoft Windows installations where users generally give themselves administartive privileges.

You may also choose to run a virus scan on the software to see if it has been modified with one of many known viruses. This, of course, does not ensure that the software doesn't contain new viruses or that it contains defects or malicious code that can impact your system or the integrity of its resources.

These days, this often isn't good enough. We would like to have software where we can validate that it was not tampered with and we may like to have certain software run in a more restrictive environment - so it cannot access certain resources.

Java is the first widely-used programming language/operating environment (we need both) to offer support for executing trusted and untrusted software, so we'll look at that as a case study.



Motivation

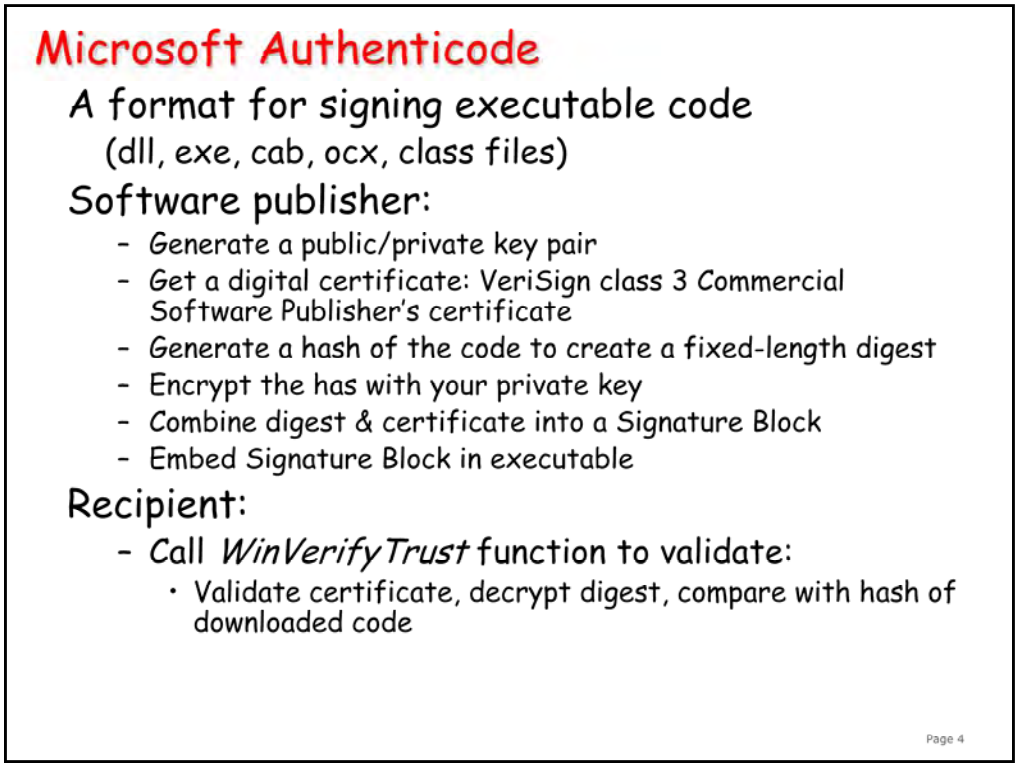
- Distributed software development
  - components
  - components may exist in different locations
  - code may be downloaded from remote machines
- Binary code
  - not easy to inspect or restrict as interpreted code
  - host can exercise limited control on binary modules

Page 3

One motivation for having trusted code is that software development has migrated to a distributed development environment. Applications are divided into pieces (components). These components may exist in different locations and code may be downloaded from these remote machines, often during run-time.

With binary code (compiled machine code), it is generally rather difficult to inspect or restrict operations as in an interpreted language like Java. The host can exercise only limited control on binary modules. Most operating systems only support the concept of software running with the permissions of a particular "user" rather than a broader model where you can have "sub-users" with permissions more restricted than those of the main user. Moreover, most operating systems will run the entire program with one set of permissions - dynamically linked libraries from different sources and the user's code are executed alike and have the same access to system resources.

--



Microsoft Authenticode

A format for signing executable code
(dll, exe, cab, ocx, class files)

Software publisher:
- Generate a public/private key pair
- Get a digital certificate: VeriSign class 3 Commercial Software Publisher's certificate
- Generate a hash of the code to create a fixed-length digest
- Encrypt the has with your private key
- Combine digest & certificate into a Signature Block
- Embed Signature Block in executable

Recipient:
- Call *WinVerifyTrust* function to validate:
  - Validate certificate, decrypt digest, compare with hash of downloaded code

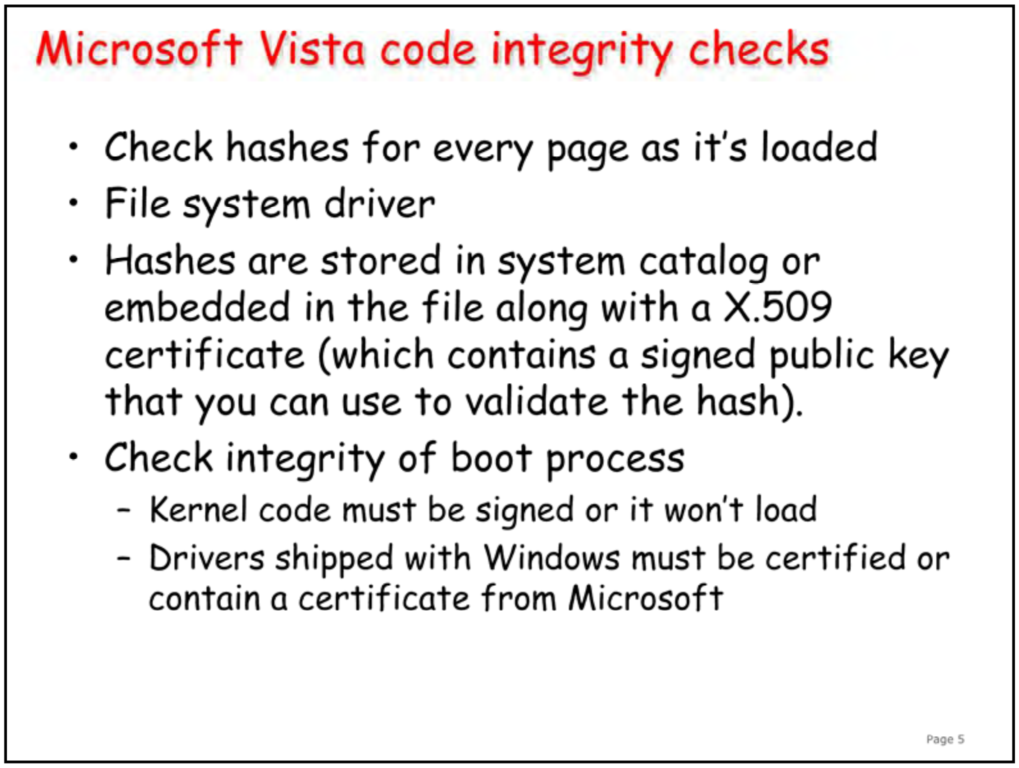
Page 4

Microsoft's Authenticode technology is simply a specification for affixing a digital signature to a block of code (that is typically downloaded over a network). The signature validates that the code was not modified since the signature was affixed and that it came from the signatory.

Authenticode works on various binary formats, such as dll, exe, cab, ocx, and class files.

The steps in creating a signed file are:
1. Generate a public/private key pair (this is something the organization does once)
2. Get a digital certificate. A digital certificate is just a public key + identification credentials, signed (has the data and encrypt it with a private key) by a trusted party. In this case, the trusted party is VeriSign - a class 3 Commercial Software Publisher's certificate (again, this is done once by the organization).
3. Generate a hash of the code to create a fixed-length digest.
4. Encrypt the digest with the private key.
5. Combine the encrypted digest with the certificate into a structure known as the Signature block.
6. Embed this in the executable.

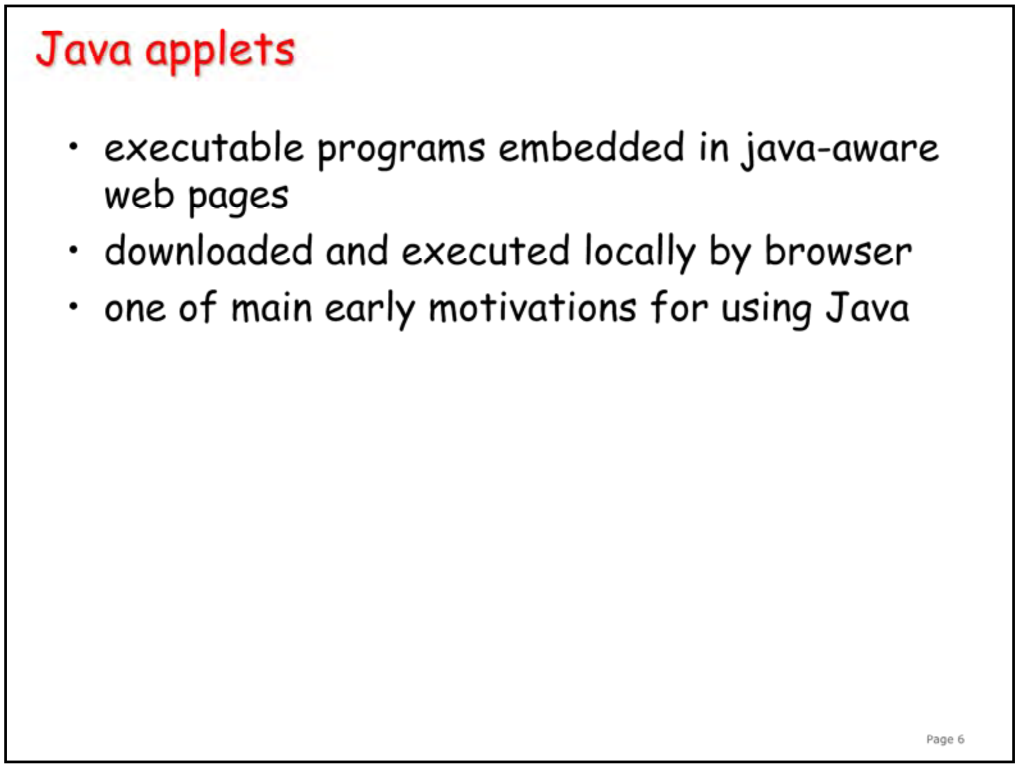
The recipient (client side) can call the Win32 function called WinVerifyTrust to validate the signature. This validates the certificate, decrypts the digest using the public key in the certificate and compares it with the hash of the downloaded code.



Microsoft Vista code integrity checks

- Check hashes for every page as it's loaded
- File system driver
- Hashes are stored in system catalog or embedded in the file along with a X.509 certificate (which contains a signed public key that you can use to validate the hash).
- Check integrity of boot process
  - Kernel code must be signed or it won't load
  - Drivers shipped with Windows must be certified or contain a certificate from Microsoft

Page 5



Java applets

- executable programs embedded in java-aware web pages
- downloaded and executed locally by browser
- one of main early motivations for using Java

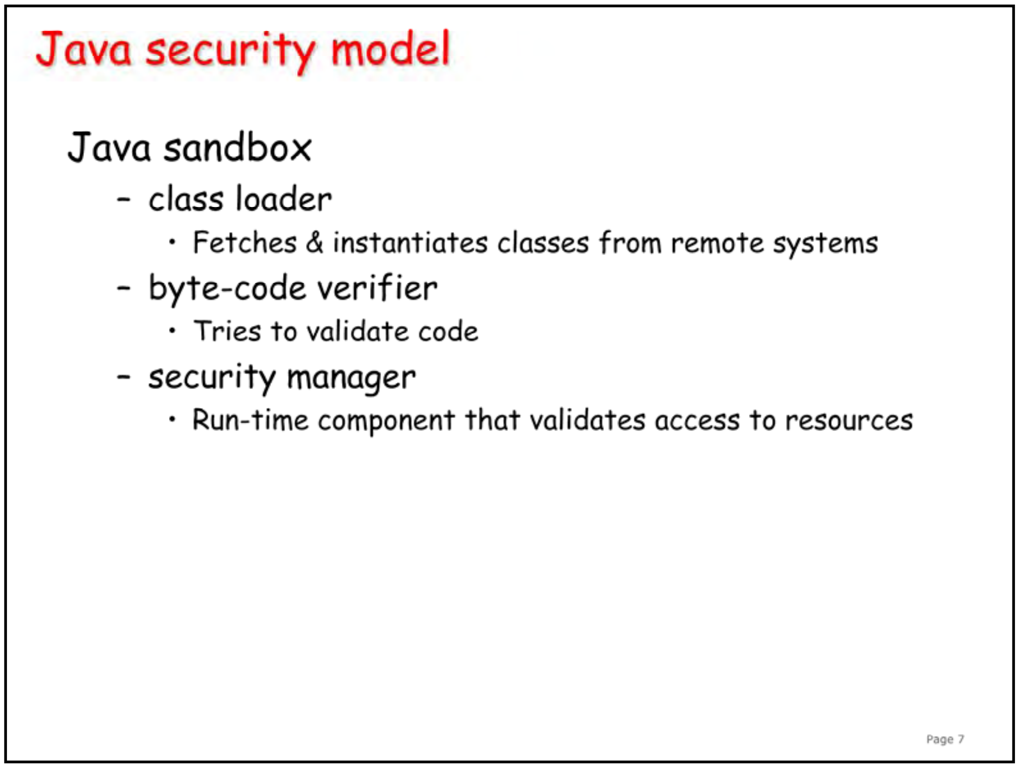
Page 6

Refresher: Java applets are (generally small) executable programs embedded in Java-aware web pages. They are downloaded and executed locally by the browser. This allows web publishers to provide applications as part of their web pages.

Java applets have been a major motivating factor for the early popularity of Java.

They are also a key area where you want to have code security: just by connecting to a web site, you may end up executing code on your machine!

--



Java security model

Java sandbox

- class loader
  - Fetches & instantiates classes from remote systems
- byte-code verifier
  - Tries to validate code
- security manager
  - Run-time component that validates access to resources

Page 7

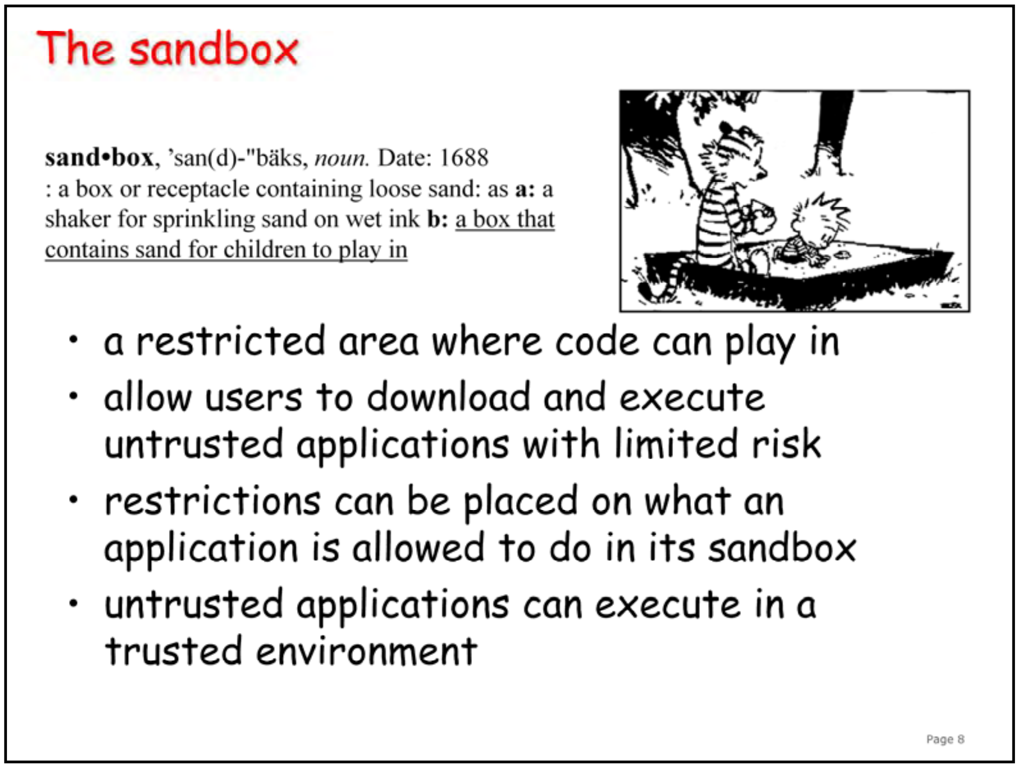
Java security is provided through a technique called “sandboxing” (we’ll get to that in the next slide). The main components of the Java sandbox are:

- class loader - this fetches and instantiates classes from remote systems

- byte-code verifier - this tries to validate the code to see that it conforms to the “rules of Java”

- security manager - this is the run-time component that validates access to system resources

--



The sandbox

sand•box, ’san(d)-"bäks, *noun.* Date: 1688
: a box or receptacle containing loose sand: as **a:** a shaker for sprinkling sand on wet ink **b:** a box that contains sand for children to play in

- a restricted area where code can play in
- allow users to download and execute untrusted applications with limited risk
- restrictions can be placed on what an application is allowed to do in its sandbox
- untrusted applications can execute in a trusted environment

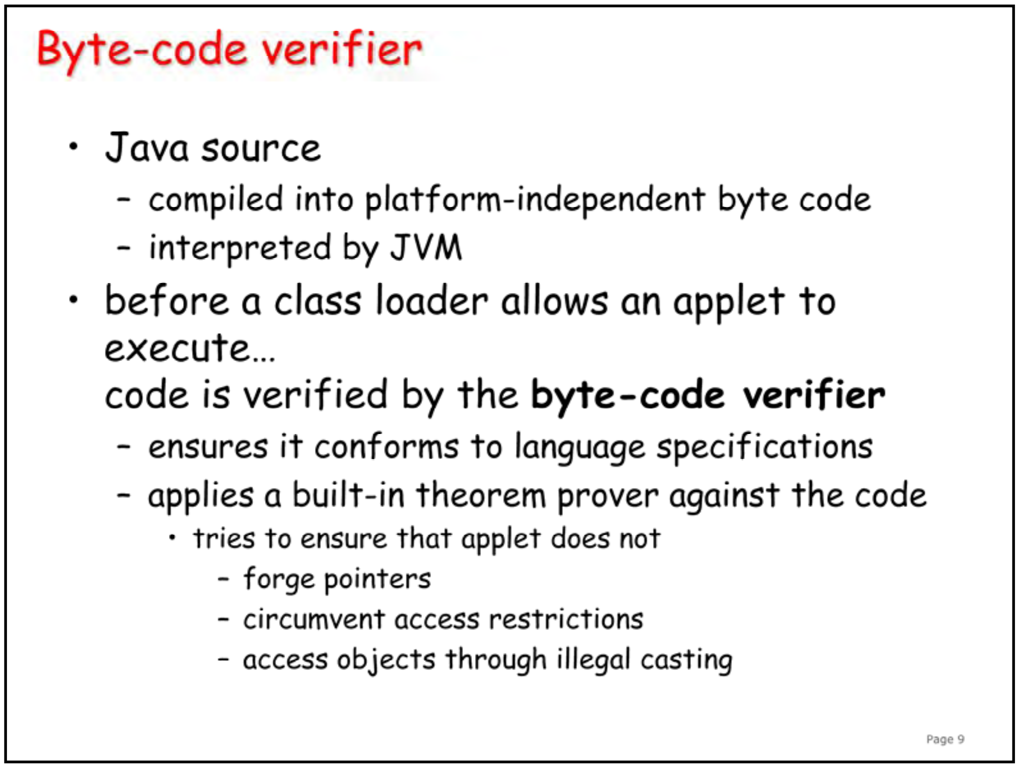
Page 8

Webster’s defines a *sandbox* as *a box that contains sand for children to play in*. In operating systems, a sandbox is a “box” where code can play in. A sandbox is a mechanism for providing restrictions on what software can and cannot do in terms of accessing memory, programs, threads, files, and other operating system resources.

Users can download and execute untrusted applications in a sandbox, limiting their risk since the sandbox will impose restrictions on what the application can do.

The sandbox allows us to bring untrusted applications into a trusted environment without compromising the environment.

--



Byte-code verifier

- Java source
  - compiled into platform-independent byte code
  - interpreted by JVM
- before a class loader allows an applet to execute...
  code is verified by the **byte-code verifier**
  - ensures it conforms to language specifications
  - applies a built-in theorem prover against the code
    - tries to ensure that applet does not
      - forge pointers
      - circumvent access restrictions
      - access objects through illegal casting

Page 9

Java source code is compiled into platform-independent byte-codes which are interpreted by the Java Virtual Machine (JVM) (instead of compiling to the machine's native instructions where they can be executed directly by the system processor).

The byte-code verifier is the first line of defense in the Java security model.

Before a class loader allows any applet to execute, the code is first verified by a **byte-code verifier**. It:

- ensures that the code adheres to the rules of the language - for example: only valid JVM opcodes are used with only the allowed arguments

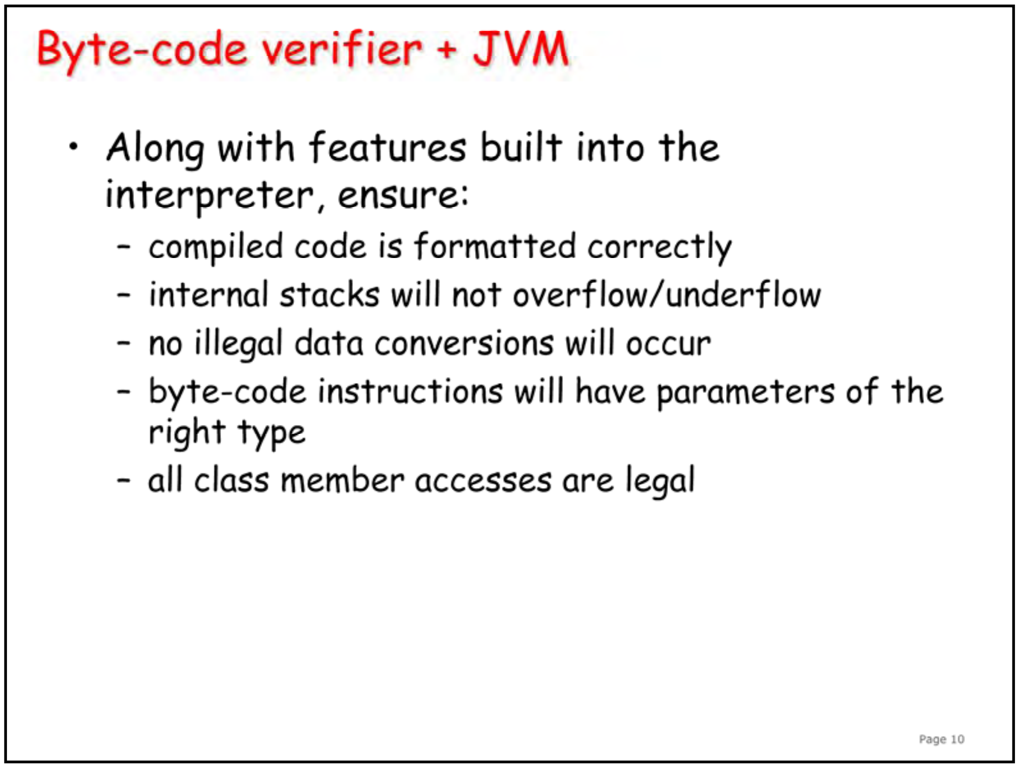
- applies a built-in theorem prover against the code. This tries to predict code execution paths and ensures that the software does not:

forge pointers

circumvent access restrictions

access objects through illegal casting

--



Byte-code verifier + JVM

- Along with features built into the interpreter, ensure:
  - compiled code is formatted correctly
  - internal stacks will not overflow/underflow
  - no illegal data conversions will occur
  - byte-code instructions will have parameters of the right type
  - all class member accesses are legal

Page 10

Between the byte-code verifier and the features built into the JVM interpreter, the system tries to ensure that:

- compiled code is in the right format and adheres to the language specification

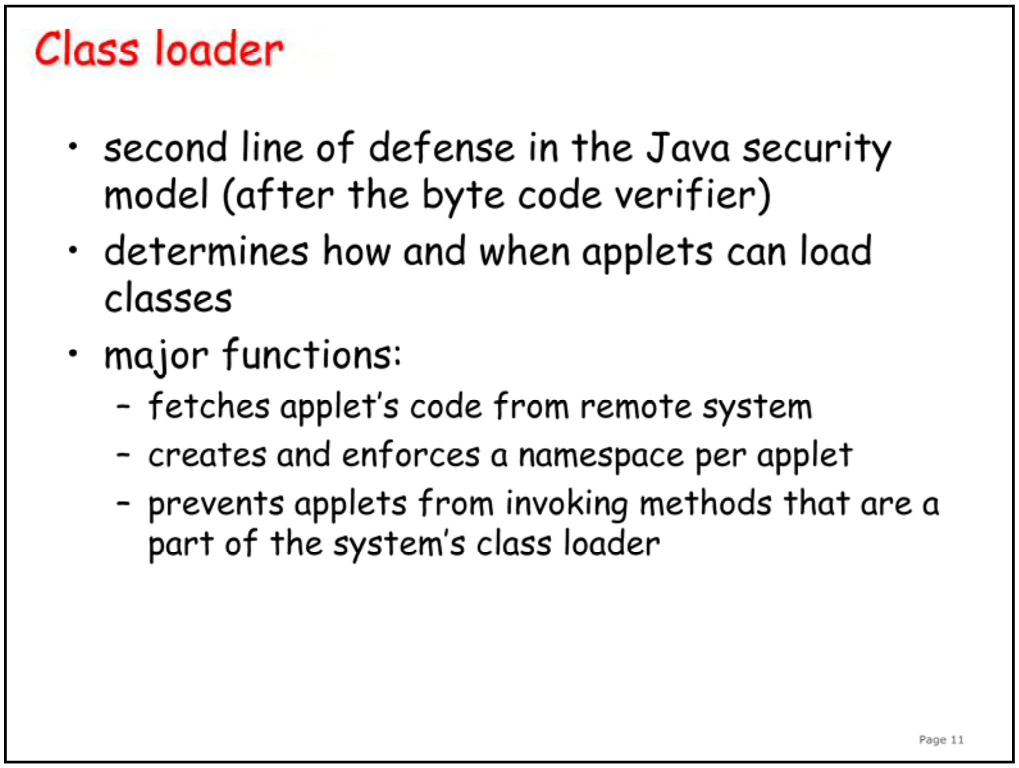
- internal stacks will not overflow/underflow

- no illegal data conversions will occur (e.g., integers cannot serve as pointers) -- it ensures that variables cannot access restricted memory areas.

- byte-code instructions will have parameters of the right type

- all class member accesses are legal - private data remains private

--



Class loader

- second line of defense in the Java security model (after the byte code verifier)
- determines how and when applets can load classes
- major functions:
  - fetches applet's code from remote system
  - creates and enforces a namespace per applet
  - prevents applets from invoking methods that are a part of the system's class loader

Page 11

To load an applet, the browser invokes the Java Applet Class Loader. It determines how and when applets can load classes.

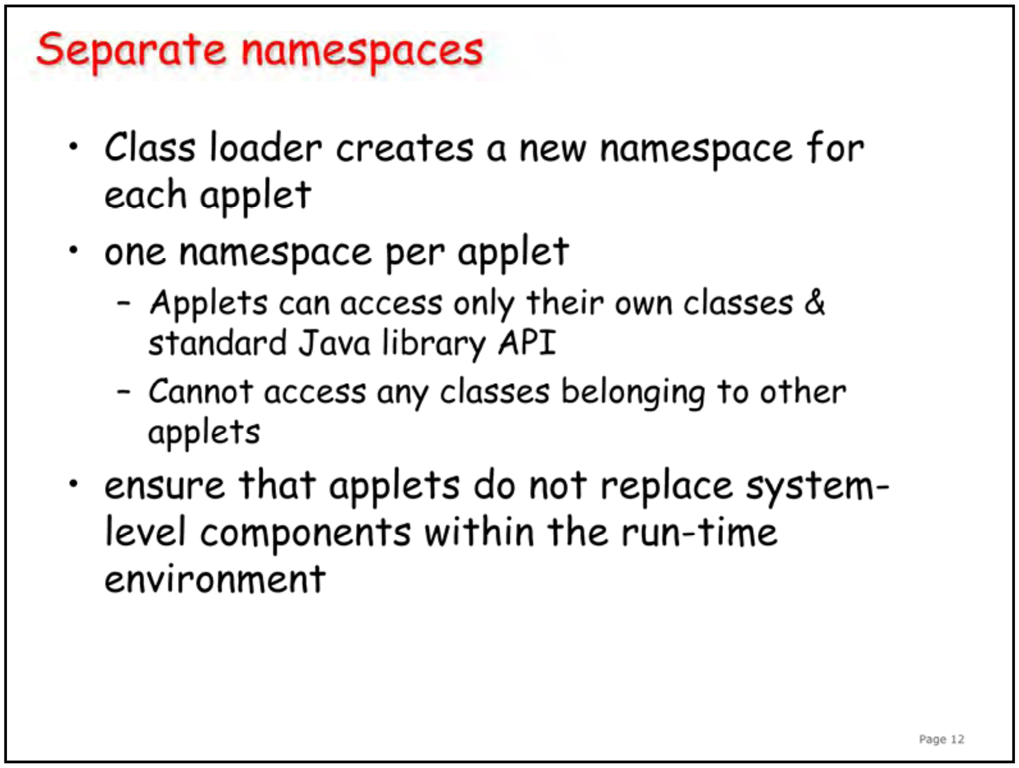
Its major functions are:

- it fetches an applet's code from the remote machine

- it creates and enforces a namespace (more on this later) for each applet

- it prevents applets from invoking methods that are part of the system's class loader (you don't want applets loading other applets)

--



Separate namespaces

- Class loader creates a new namespace for each applet
- one namespace per applet
  - Applets can access only their own classes & standard Java library API
  - Cannot access any classes belonging to other applets
- ensure that applets do not replace system-level components within the run-time environment

Page 12

The Applet Class Loader creates a new namespace for each applet. Hence, applets can access only their own classes and the standard Java library API.

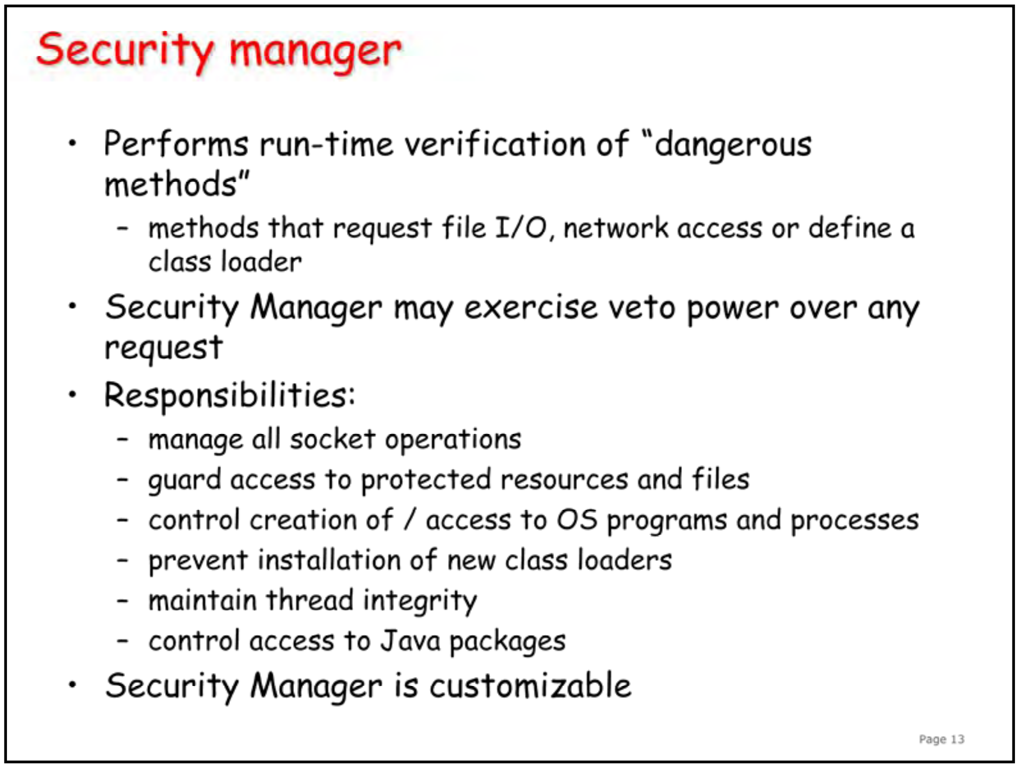
They cannot access any classes belonging to other applets.

The advantages of this are:

- separate namespaces make it difficult for applets to pool their resources to form a concerted attack.

- applet developers need not be concerned about name collisions

--



Security manager

- Performs run-time verification of "dangerous methods"
  - methods that request file I/O, network access or define a class loader
- Security Manager may exercise veto power over any request
- Responsibilities:
  - manage all socket operations
  - guard access to protected resources and files
  - control creation of / access to OS programs and processes
  - prevent installation of new class loaders
  - maintain thread integrity
  - control access to Java packages
- Security Manager is customizable

Page 13

The security manager is responsible for run-time verification of "dangerous methods" - methods that request file I/O, network access, or defining a new class loader.

It keeps track of who is allowed to do which dangerous operations.

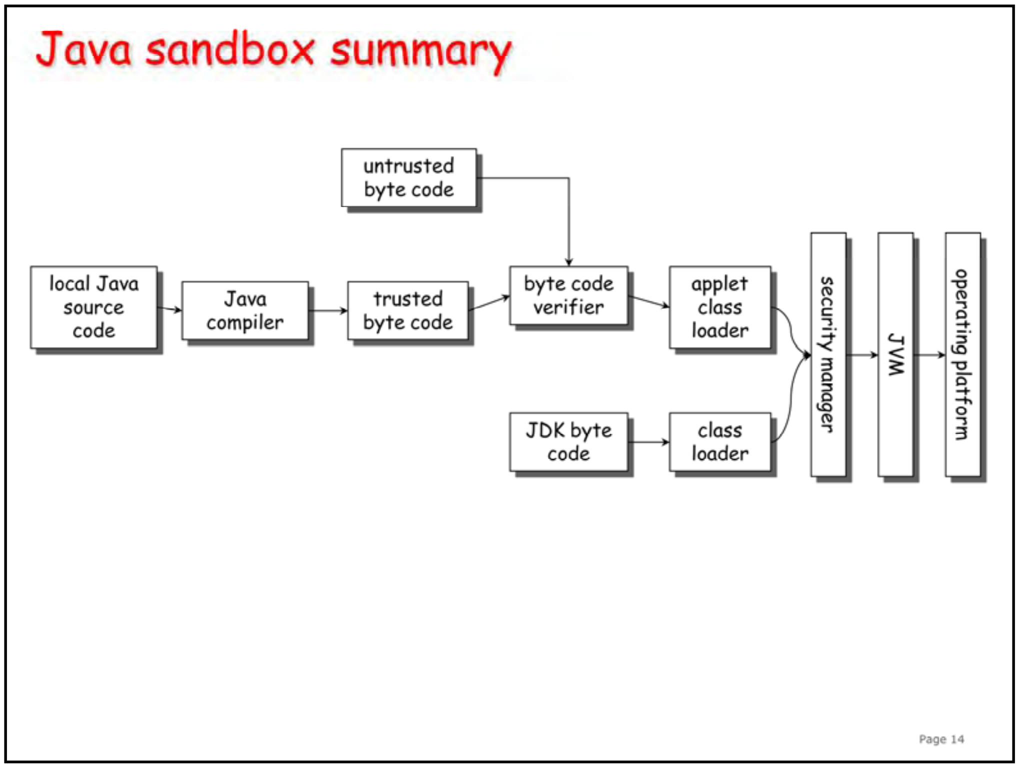
A security manager can choose what accesses are permitted and generate a SecurityException for those that it decides should not be permitted (look through the JDK API -- any methods that can throw a SecurityException are those where the Security Manager intervenes).

In general, the Security Manager can be a replaceable component that can be as complex as the authors want it to be.

Code in the Java library consults a security manager when a potentially dangerous operation is attempted.

Security checking code examines the run-time stack for frames executing untrusted code (each thread has its own stack). This process is known as **stack inspection**. All major Java vendors have adopted stack inspection. The stack frames are searched from newest to oldest. If an untrusted one is encountered, the security manager defaults to that level of trust (untrust).

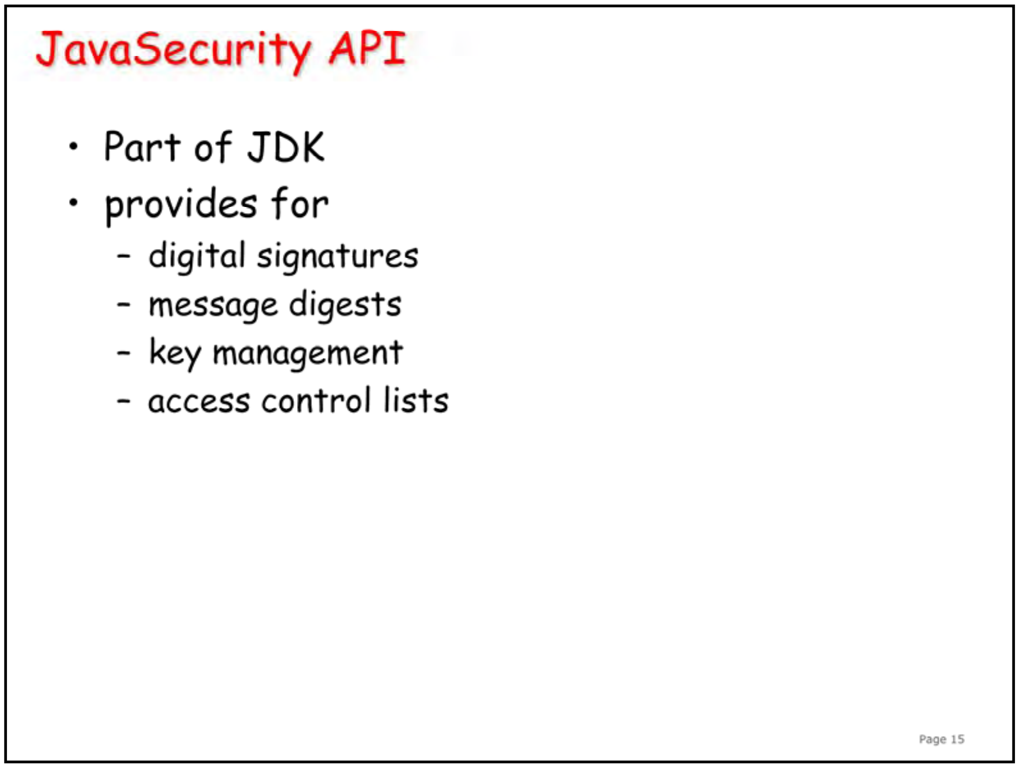
--



Java sandbox summary

This summarizes the operations in the Java sandbox. User-compiled code is considered “trusted code”. By default, remote code is considered “untrusted”. All Java byte code, except that from the JDK libraries is verified with the byte-code verifier. The class loader than allows the namespace of the class to be instantiated and for the class to be executable. At that time the security manager takes over for run-time intervention on operations that the code wants to perform.

--



JavaSecurity API

- Part of JDK
- provides for
  - digital signatures
  - message digests
  - key management
  - access control lists

Page 15

Since JDK 1.1, Java added a JavaSecurity API. This provides a broad set of methods for

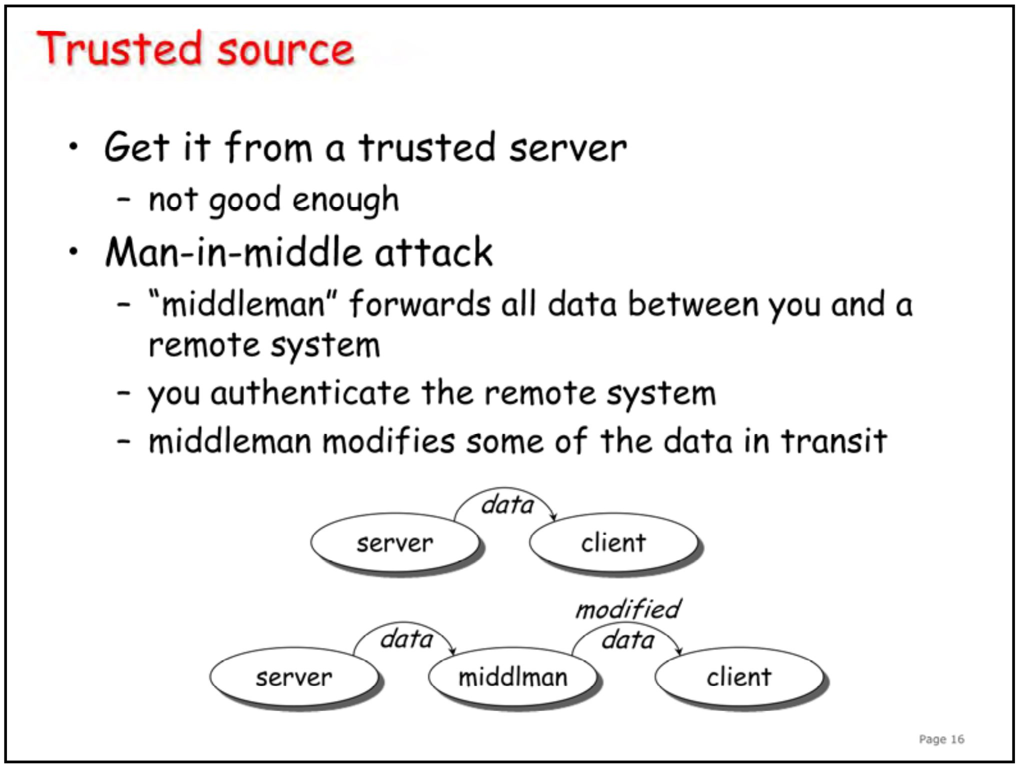
digital signatures

message digests

key management (support for X.509 digital certificates)

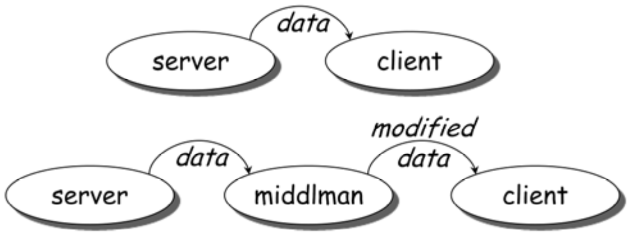
access control lists

--



Trusted source

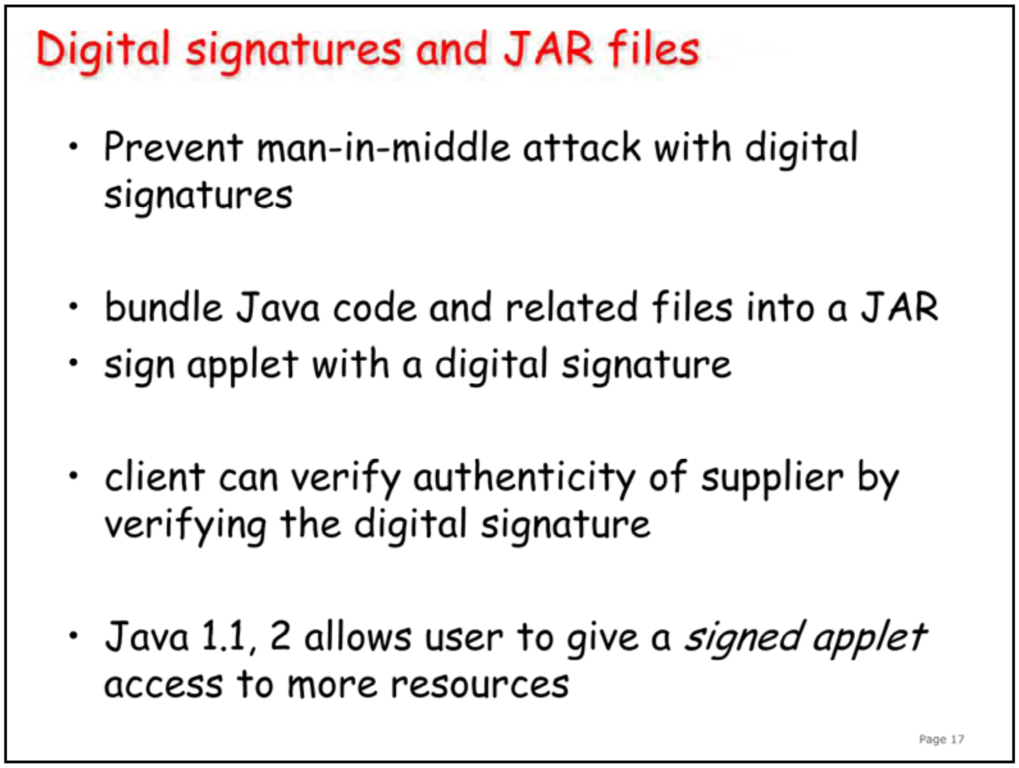
- Get it from a trusted server
  - not good enough
- Man-in-middle attack
  - "middleman" forwards all data between you and a remote system
  - you authenticate the remote system
  - middleman modifies some of the data in transit



Page 16

When we download software from a trusted source, we need to ensure that it has not been modified by some interceptor. Such a modification is called a "man-in-the-middle" attack. This middleman manages to get in the communication path and simply forwards data between client and server and vice versa. At times, however, he may choose to modify the data. In downloading code, the client may successfully authenticate with the server, even with a middleman, but the downloaded code may still be corrupted.

--



Digital signatures and JAR files

- Prevent man-in-middle attack with digital signatures
- bundle Java code and related files into a JAR
- sign applet with a digital signature
- client can verify authenticity of supplier by verifying the digital signature
- Java 1.1, 2 allows user to give a *signed applet* access to more resources

Page 17

To guard against this man-in-the-middle attack, digital signatures can be used.

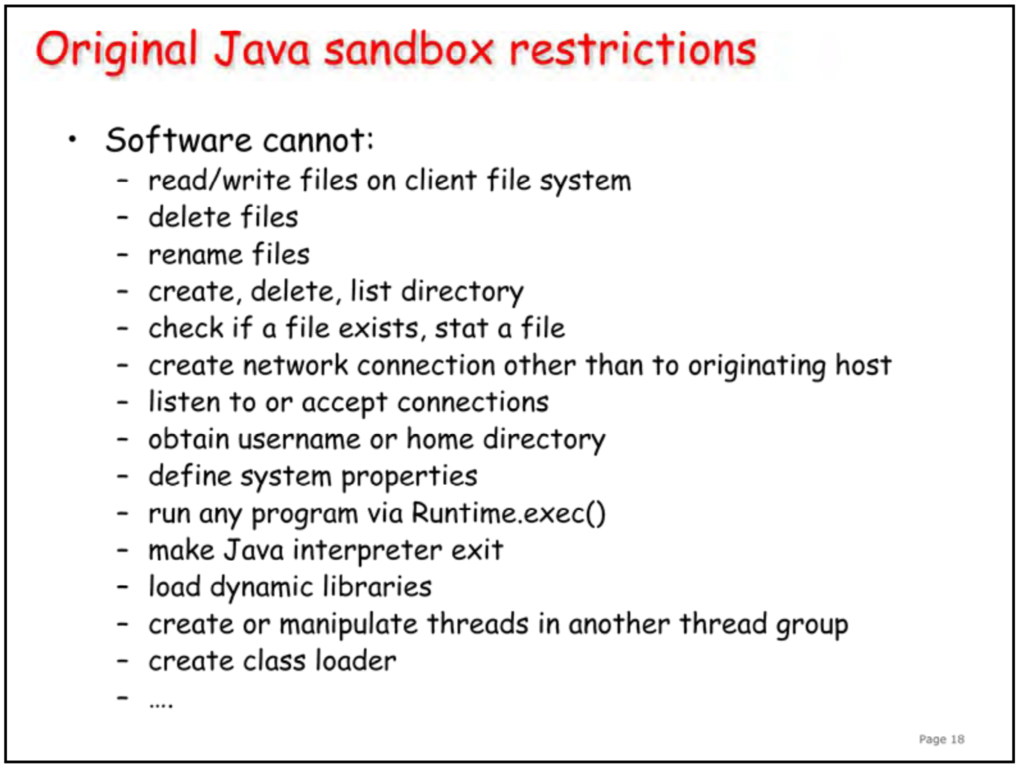
Java code and any related files can be bundled into a JAR (Java archive format). This resultant applet (jar file) is then digitally signed:

- add a hash encrypted by the supplier’s private key
- add identification information about the supplier

The client can verify the authenticity of the code by using the supplier’s digital certificate (it contains the public key that may be used to decrypt the hash and validate it).

Since Java 1.1, a signed applet can be considered “trusted” and be given access to more resources (on a per-supplier basis). For example, you may choose to trust all applets from the Microsoft corporation.

--



Original Java sandbox restrictions

- Software cannot:
  - read/write files on client file system
  - delete files
  - rename files
  - create, delete, list directory
  - check if a file exists, stat a file
  - create network connection other than to originating host
  - listen to or accept connections
  - obtain username or home directory
  - define system properties
  - run any program via Runtime.exec()
  - make Java interpreter exit
  - load dynamic libraries
  - create or manipulate threads in another thread group
  - create class loader
  - ….

Page 18

The original Java sandbox (the only one available before Java 1.1, and still the default for untrusted code) imposed a lot of restrictions on executing software:

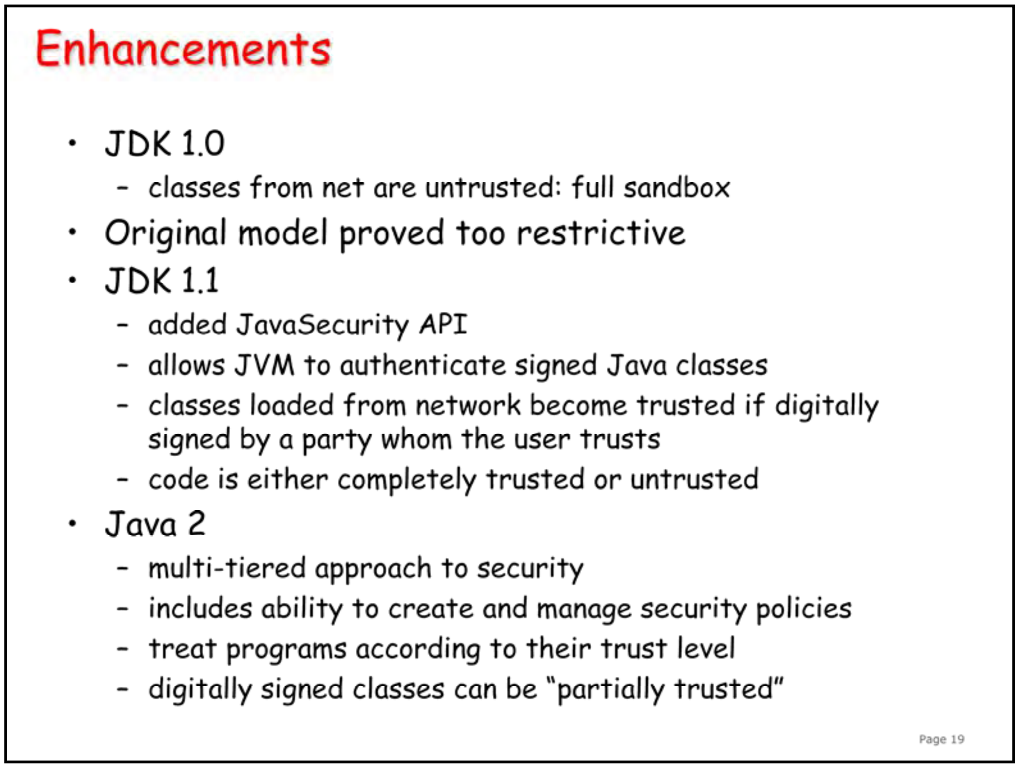
untrusted applets cannot read/write from/to the local disk.

All standalone windows created by applets are labeled as such so that users are aware of this when entering data (an applet cannot disguise itself as a terminal window, for example).

An applet was not allowed to establish network connections to any system other than the originating host.

Much more …. See the slide

--



Enhancements

- JDK 1.0
  - classes from net are untrusted: full sandbox
- Original model proved too restrictive
- JDK 1.1
  - added JavaSecurity API
  - allows JVM to authenticate signed Java classes
  - classes loaded from network become trusted if digitally signed by a party whom the user trusts
  - code is either completely trusted or untrusted
- Java 2
  - multi-tiered approach to security
  - includes ability to create and manage security policies
  - treat programs according to their trust level
  - digitally signed classes can be "partially trusted"

Page 19

This complete sandbox proved to be too restrictive for some applications.

With Java 1.1, the JavaSecurity API was provided to allow one to create and authenticate signed classes. A user can designate trusted parties. Classes that are signed and loaded from any of these trusted parties will become trusted (just as a user's own code). The model is still one of complete trust or complete mistrust.

With Java 2, a multi-tiered approach to security was adopted where users can create and manage security policies and treat programs according to their trust level. Some examples of items that can be controlled are:

- restricted access to file systems and network
- restricted access to browser internals
- use the byte-code verifier

Digitally signed classes can therefore be considered "partially trusted" (under user control). Privileges can be granted when they're needed. At other times, the code can operate with the minimum necessary privileges ("**principle of least privilege**").

Summary

Sandboxing proved to be a more elusive problem than originally anticipated. The Java sandbox was not (and is not) foolproof and countless attacks were found to penetrate it. It has been improved over time, but run-time security remains a problematic issue.

--

The end.