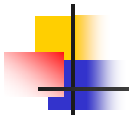
Secure Software

- Software is **secure** if it can handle intentionally malformed input; attacker picks (the probability distribution of) the inputs.
- Secure software: Protect integrity of runtime system.
- Secure software ≠ software with security features.
- Networking software is a popular target:
  - Intended to receive external input.
  - May construct instructions dynamically from input.
  - May involve low level manipulations of buffers.



Security & Reliability

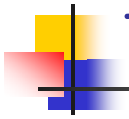
- In security, defender moves first; attacker picks inputs to exploit weak defences.
- To make software more secure, tested against "untypical" usage patterns (but there are typical attack patterns).
- On PC, in control of software components sending inputs to each other.
- On the Internet, hostile parties **can** provide input: Do not "trust" your inputs.

Abstraction

- When writing code, programmers use elementary concepts like character, variable, array, integer, data and program, address (resource locator), atomic transaction, …
- These concepts have abstract meanings.
- For example, integers are an infinite set with operations 'add', 'multiply', 'less or equal', …
- To execute a program, we need concrete implementations of these concepts.

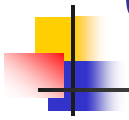
Benefits and Dangers of Abstraction

- Abstraction (hiding 'unnecessary' detail) helps in understanding complex systems.
- No need for the inner details of a computer to be able to use it: we can write software using high level languages and graphical methods.
- Software security problems typically arise when concrete implementation and the abstract do not match:
  - Address (location)
  - Character
  - Integer
  - Variable (buffer overflows)
  - Double-linked list
  - Atomic transaction



Input Validation

- Application wants to give users access only to files in directory `A/B/C/`.
- Users enter filename as `input`; full file name constructed as `A/B/C/input`.
- Attack: use `../` a few times to step up to root directory first; e.g. get password file with input `/../../../../etc/passwd`.
- Countermeasure: input validation, filter out `../` (not that easy).
- Do not trust any input.

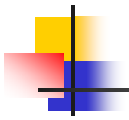


Unix *rlogin*

- Unix *login* command:
  - **`login [[-p] [-h<host>] [[-f]<user>]`**
  - **`-f`** option "forces" log in: user is not asked for password
- Unix *rlogin* command for remote login:
  - **`rlogin [-l<user>] <machine>`**
  - The *rlogin* daemon sends a login request for *<user>* to *<machine>*
- Attack (some versions of Linux, AIX):
  - % rlogin -l -froot <machine>
- Results in forced login as root at the designated machine
  - % login -froot <machine>

Unix *rlogin*

- Problem: Composition of two commands.
- Each command on its own is not vulnerable.
- However, *rlogin* does not check whether the "username" has special properties when passed to *login.*



What will happen here?

```
int i = 1;
while (i > 0)
{
i = i * 2;
}
```

NOT the apparent infinite loop because of the finite representation of integers

Programming with Integers

- In mathematics, integers form an infinite set.
- On computer systems, integers are represented in binary.
- representation of integer is a binary string of fixed length (precision), so only a **finite** number of "integers".
- Programming languages: signed and unsigned integers, short and long integers, …
- Unsigned 8-bit integers

  255 + 1 = 0 16 * 17 = 16 0 - 1 = 255
- Signed 8-bit integers

  127 + 1 = -128 -128/-1 = -1
- In mathematics: $a + b \geq a$ for $b \geq 0$ but in programming no longer true.

Code Example

- OS kernel system-call handler; checks string lengths to defend against buffer overruns.

```
char buf[128];
combine(char *s1, size_t len1,
        char *s2, size_t len2)
{
if (len1 + len2 + 1 <= sizeof(buf)) {
strncpy(buf, s1, len1);
strncat(buf, s2, len2);
}
}
```

`len1 < sizeof(buf)`

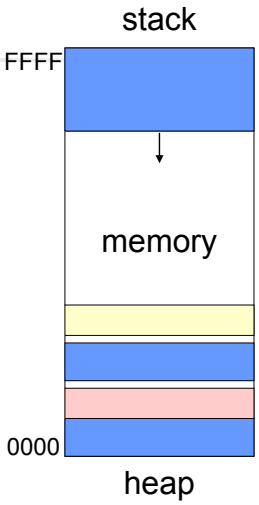
`len2 = 0xffffffff`

`len2 + 1 = 2^32-1 + 1 = 0 mod 2^32`

`strncat` will be executed

Memory configuration

- Stack: contains return address, local variables and function arguments; relatively easy to decide in advance where particular buffer placed on the stack.
- Heap: dynamically allocated memory; more difficult but not impossible to decide in advance where a particular buffer placed on the heap.



Variables

- Buffer: concrete implementation of a variable.
- If value assigned to variable exceeds size of allocated buffer, memory locations not allocated to this variable are **overwritten**.
- If memory location overwritten allocated to other variable, value of other variable changed.
- Depending on circumstances, attacker can change value of sensitive variable *A* by assigning deliberately malformed value to other variable B.

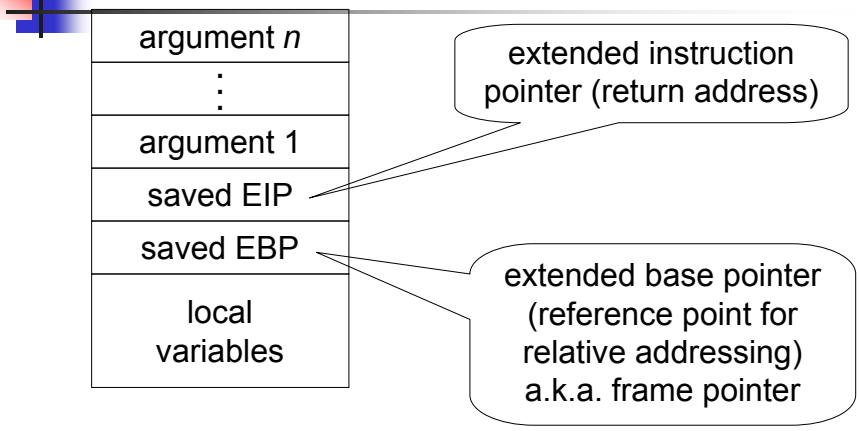
Buffer Overruns

- Unintentional buffer overruns crash software (focus for reliability testing)
- Intentional buffer overruns problematic if attacker can modify security relevant data.
- Attractive targets: return addresses (to next piece of code to be executed) and security settings.
- In languages like C the programmer allocates and de-allocates memory.
- Type-safe languages like Java guarantee that memory management is 'error-free'.

System Stack

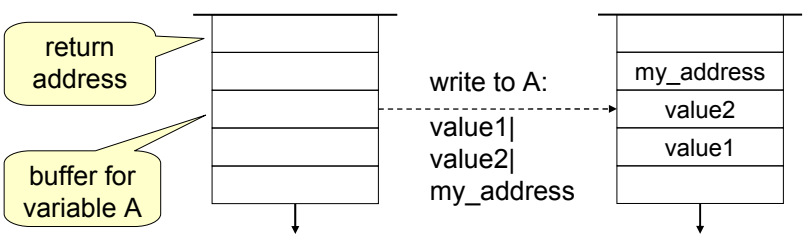
- Function call: stack frame (activation record) containing function arguments, return address and statically allocated buffers pushed on the stack.
- When call ends, execution continues at return address specified.
- Stack usually starts at top of memory and grows downwards.
- Layout of stack frames reasonably predictable.

Stack Frame – Layout



Stack-based Overflows

- Find buffer on runtime stack of privileged program to overflow return address.
- Overwrite return address with start address of code to execute (now privileged too).



Code Example

- Declare local short string variable

```
char buffer[80];
```

use standard C library routine call

```
gets(buffer);
```

to read single text line from standard input to save into buffer.

- it corrupts stack if input longer than 79 characters.
- Attacker loads malicious code into buffer and redirects return address to start of attack code.

Shellcode

- Overwrite return address so that execution jumps to attack code ('shellcode').
- Where to put the shellcode?
- Shellcode on the stack as part of the malicious input; a.k.a. argv[]-method.
  - To guess location, guess distance between return address and address of input containing shellcode.
- Details e.g. in Smashing the Stack for Fun and Profit.
- *return-to-libc* method: attack calls system library; change to control flow, but no shellcode inserted.

Race Conditions

- Multiple computations access shared data so that results depend on sequence of accesses.
  - Multiple processes accessing same variable.
  - Multiple threads in multi-threaded processes (as in Java servlets).
- An attacker try to change value after checked but before used.
- TOCTOU (time-to-check-to-time-of use) is a well-known security issue.

Prevention

- Hardware features can stop buffer overflow attacks from overwrite control information; Separate register for the return address in Intel's Itanium processor
- no need to rewrite or recompile programs; only some processor instructions modified.
- Drawback: existing software that uses multi-threading may no longer work.
- Non-executable stacks stops attack code from being executed from the stack.
- Memory management unit configured to disable code execution on the stack.
- Not trivial to implement if existing O/S routines are executing code on the stack, and problems with backwards compatibility
- Attackers may find ways of circumventing this protection mechanism

Prevention – Safer Functions

- C is infamous for its unsafe string handling functions: **`strcpy`**, **`sprintf`**, **`gets`**, …
- Example: **`strcpy`**

```
char *strcpy( char *strDest, const char *strSource );
```

- Replace unsafe string functions by functions where number of bytes/characters to be handled are specified (but not easy to compute correctly):

  `strncpy`, **`_snprintf`**, **`fgets`**, …

- Example: **`strncpy`**

```
char *strncpy( char *strDest, const char *strSource,
size_t count );
```

Prevention – Filtering

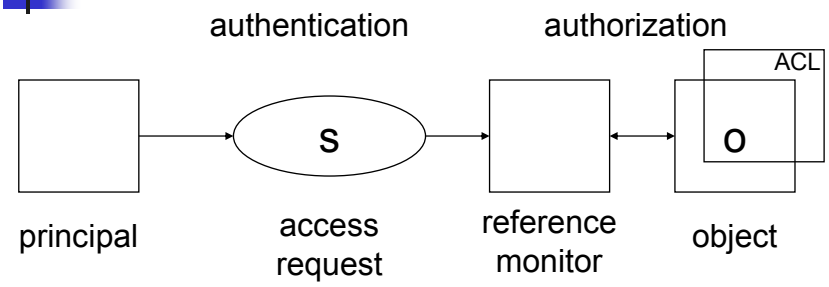
Filtering inputs recommended defence several times:

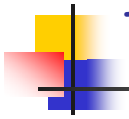
- Whitelisting: Specify legal inputs; accept legal inputs, block anything else.
  - Conservative, but if forget some specific legal inputs, legitimate action might be blocked.
- Blacklisting: Specify forbidden inputs; block forbidden inputs, accept anything else.
  - if forget some specific dangerous input, attack may get through.
- Taint analysis: Mark input from untrusted sources as tainted, stop execution if security critical function receives tainted input; sanitizing functions produce clean output from tainted input.

Summary

- Many of the problems listed may look trivial.
  - other ones not mentioned: scripting languages, XSS, SQL and code injection, format strings ...
- There is no silver bullet:
  - Code-inspection: better at catching known problems, may raise false alarms.
  - Black-box testing: better at catching known problems.
  - Type safety: guarantees from an abstract (partial) model need not carry over to the real system.
- Experience in high-level programming languages may be a disadvantage when writing low level network routines.

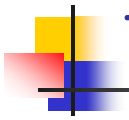
Authentication & Authorization





Identity-based Access Control

- Access control based on user identities.
- The kind of access control familiar from operating systems like Unix or Windows.
- Do not confuse the 'identity' of a person with a user identity (uid) in an operating systems; a uid is just a unique identifier that need not correspond to a real person (e.g. 'root').
- RBAC = IBAC + roles.



IBAC

- This model originated in 'closed' organisations ('enterprises') like universities, research labs.
- Organisation has authority over its members.
- Members (users) can be physically located.
- Access control policies refer naturally to user identities.
- Audit logs point to users who can be held accountable.
- Access control seems to require by definition that identities of persons are verified.
- Biometrics: strong identity-based access control?

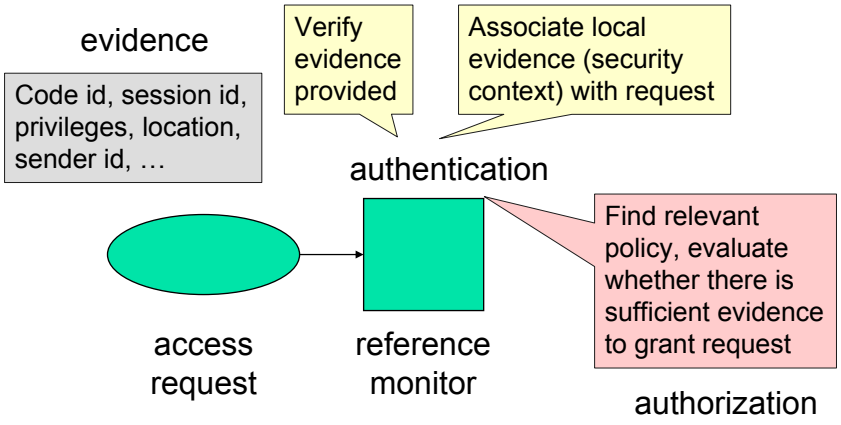
Other Aspects

- Access rules are local: no need to search for the rule that should be applied; the rule is stored as an ACL with the object.
- Enforcement of rules is centralized: reference monitor does not consult other parties when making a decision.
- Simple access operations: read, write, execute; single subject per rule; no rules based on object content.
- Homogeneity: the same organisation defines organizational and automated security policy.

Changes in the 1990s

- Internet connections to parties never met before:
  - 'identity' cannot be in our access rules.
  - not always able to hold them accountable.
- Java sandbox: it is not necessary to refer to users when describing or enforcing access control.
- Access controlled at the level of applets, not at the granularity of read/write/execute.
- Instead of asking who made the request, ask what to do with it.

Access Control in an 'open' World



What changed with the web?

- Separation of program and data is blurred; executable content (applets, scripts) embedded in interactive web pages that can process user input.
- Computation moved to the client who needs protection from rogue content providers.
  - Lesson of the early PC age: floppy disks from arbitrary sources were the route for computer virus infections.
- Client asked to make decisions on security policy and on enforcing security; end user becomes system administrator and policy maker.
- Browser becomes part of the TCB.

Changes in the Environment

- When organisations collaborate, access control can be based on more than one policy.
- Potential conflicts between policies have to be addressed.
- How to export security identifiers from one system into another system?
- Decisions on access requests may be made by an entity other than the one enforcing the decision.
- How does a user know which credentials to present?

Code-based Access Control

- If not possible to rely on principal who requests an access control decisions, look at the request itself.
- Requests can be programs, rather than elementary read/write instructions.
- Code-based access control: access control where permissions are assigned to code.
- Major examples: Java security model, .NET security framework (code access control).

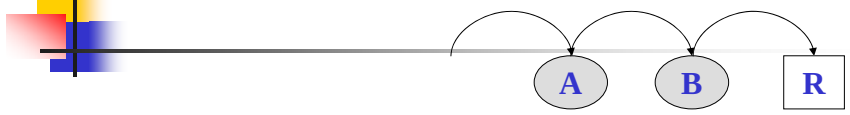
Access Control Parameters

- Security attributes of code could be:
  - Site of code origin: local or remote?
  - URL of code origin: intranet or Internet?
  - Code signature: signed by trusted author?
  - Code identity: approved ('trusted') code?
  - Code proof: code author provides proof of security properties;
  - Identity of sender: principal the code comes from;
  - …

Call Chains

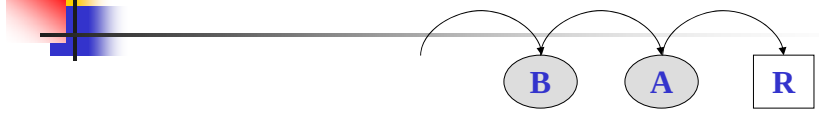
- In code-based access control, when a process calls another function, access decisions refer to access rights assigned to that function.
- Should the calling process also 'delegate' some of its access rights to the process executing the function being called?
- Should the calling process limit the access rights of the function executing the program being called?
- These questions central in code-based access control.

Call Chains



- Which privileges should be valid when one function calls another function?
- Example: function *A* has access right to resource R, B does not; *A* calls B, B requests access to R: Should access be granted?
- The conservative answer is 'no', but *A* could explicitly delegate the access right to B.

Call Chains



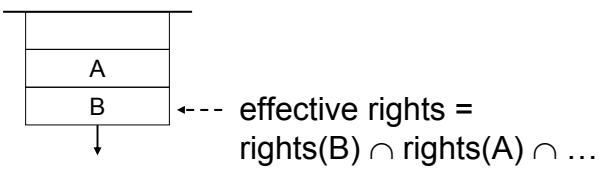
- Example: function *A* has access right to resource R, B does not; B calls *A*, *A* requests access to R: Should access be granted?
- The conservative answer is 'no', but *A* could explicitly assert its access right.

Enforcing Policies

- How to compute current permissions granted to code?
- Access decisions should know about entire call chain.
- Information about callers maintained on call stack used by Java VM for managing executions.
- Design decision: re-use call stack for policy evaluation.
- Lazy evaluation: evaluate granted permissions just when a permission is required to access a resource.

Dynamic Stack Inspection

- Record for each stack frame the security permissions of the function.
- Rights of final caller are computed as the intersection of the permissions for all entries on the call stack.



Limits of Stack Inspection

- Access control explained in terms of the runtime stack for implementation reasons (lazy evaluation).
  - Performance? Common optimizations are disabled.
  - Security: What is guaranteed by stack inspection? Hard to relate to high-level security policies.
- Two concerns for developers:
  - Untrusted component may take advantage of my code.
  - Permissions may be missing when running my code.
- Stack inspection is blind to many control and data flows:
  - Parameters, results, mutable data, objects, inheritance, callbacks, events, exceptions, concurrency…
- Each case requires a specific discipline or mechanism.

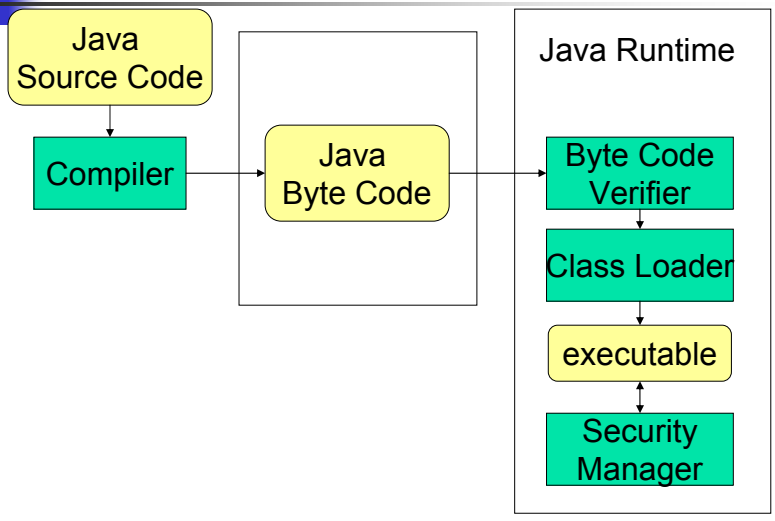
Java Security

- Java: strongly typed, object-oriented general purpose programming language.
- Java is type safe; the type of a Java object is indicated by the class tag stored with the object
- Static (and dynamic) type checking to examine whether the arguments received during execution are always of the correct type.
- Security advantage: no pointers arithmetic; memory access through pointers is one of the main causes for security flaws in C or C++.

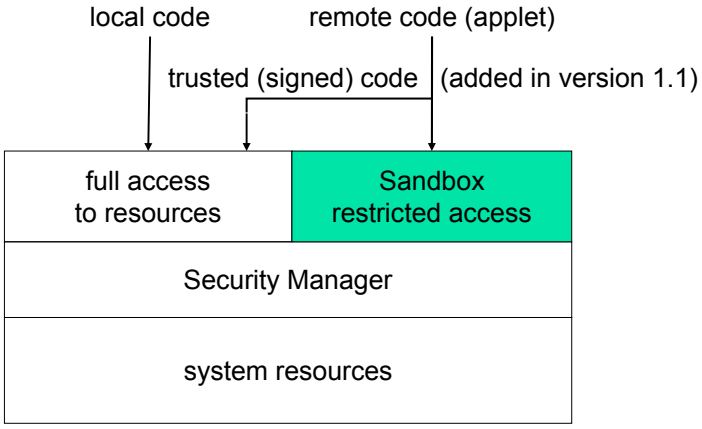
Java – Overview

- Java source code translated into machine independent byte code (similar to an assembly language) and stored in class files.
- *Platform specific* virtual machine interprets the byte code translating it into machine specific instructions.
- When running a program, a Class Loader loads any additional classes required.
- Security Manager enforces the given security policy.

Java Execution Model

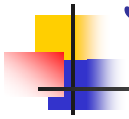


JDK 1.1 Security Model



Discussion

- Basic policy is quite inflexible:
  - Local/signed code is unrestricted.
  - Applet/unsigned code is restricted to sandbox.
- No intermediate level:
  - How to give some privileges to a home banking application?
- Local/remote is not a precise security indicator:
  - Parts of the local file system could reside on other machines;
  - Downloaded software becomes "trusted" once it is cached or installed on the local system.
- For more flexible security policies a customized security manager needed to be implemented.
  - Requires security AND programming skills.



Java 2 Security Model

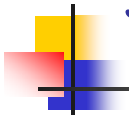
- Java 2 security model no longer based on the distinction between local code and applets.
- Applets and applications controlled by the same mechanisms.
- Reference monitor of the Java security model performs fine-grained access control based on security policies and permissions.
- Policy definition separated from policy enforcement.
- Single method checkPermissions() handles all security checks.

Byte Code Verifier

- Analyzes Java class files: performs syntactic checks, uses theorem-provers and data flow analysis for static type checking.
- There is still dynamic type checking at run time
- Verification guarantees properties like:
  - Class file is in the proper format.
  - Stacks will not overflow.
  - All operands have arguments of the correct type.
  - There will be no data conversion between types.
  - All references to other classes are legal.

Class Loaders

- Protect integrity of the run time environment; applets not allowed to create their own Class Loaders and to interfere with each other.
- Vulnerabilities in a class loader are particularly security critical (if exploited by attacker to insert rogue code).
- Each Class Loader has own name space; each class labeled with Class Loader that has installed it.

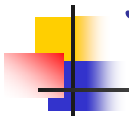


Security Policies

- Security policy: maps a set of properties that characterizes running code to a set of access permissions granted to the code.
- Code characterized by CodeSource:
  - origin (URL)
  - digital certificates
- Permissions contain target name and set of actions.
- Level of indirection: permissions granted to protection domains:
  - Classes and objects belong to protection domains and 'inherit' the granted permissions.
  - Each class belongs to one and only one domain.

Security Manager

- Security Manager: reference monitor in the JVM; security checks defined in AccessController class.
  - Uniform access decision algorithm for all permissions.
- Access (normally) only granted if all methods in the current sequence of invocations have the required permissions ('stack walk').
- Controlled invocation: privileged operations; doPrivileged() tells the Java runtime to ignore the status of the caller.



Summary

- Java 2 security model is flexible and feature-rich; it gives a framework but does not prescribe a fixed security policy.
- JAAS (Java Authentication and Authorization Service) adds user-centric access control.
- Sandbox enforces security at the service layer; security can be undermined by access to the layer below:
  - users running applications other than the web browser.
  - attacks by breaking the type system.