Software Security

Summary

Contents

Chapter 1

Language Based Security

1.1 Information Flow Security

1.1.1 Tracking Information Flow

Perl has a taint mode feature that allows the tracking of input. When active all forms of input to the programs are marked as "tainted". Tainted variables taint variables explicitly calculated from them and tainted data may not be used in any sensitive command (with some exceptions).

This mechanism implicits a set of security classes (tainted vs. untainted), as well as a classification of objects/information holders, a specification of when information can flow from onde security class to another and a way to determine security classes that safely represent the combination of two other.

1.1.2 Information Flow Policies

The goals of information security are confidentiality and integrity. Information flow policies specify how information should be allowed to flow between objects of each security class. To define one such policy we need:

- A set of security classes
- A can-flow relation between them
- An operator for combining them

Information Flow Policies For Confidentiality

Confidentiality classes determine who has the right to read and information can only flow towards confidentiality classes that are at least as secret.

Information that is derived from the combination of two security classes takes a confidentiality classes that are at least as secret as each of them.

Information Flow Policies For Integrity

Integrity classes determine who has the right to write and information can only flow towards integrity classes that are no more trustful.

Information that is derived from the combination of two integrity classes takes an integrity class that is no more trustful than each of them.

Formal Information Flow Policies

These policies can be described as a triple $(SC, \rightarrow, \oplus)$, where:

- SC is a set of security classes
- $\rightarrow \subseteq SC \times SC$ is a binary can-flow relation on SC
- $\oplus : SC \times SC \rightarrow SC$ is an associative and commutative binary class-combining or join operator on SC

Example high-low policy for confidentiality:

- $SC = \{H, L\}$
- $\rightarrow = \{(H, H), (L, L), (L, H)\}$
- $H \oplus H = H, L \oplus H = H, L \oplus L = L$

And for integrity:

- $SC = \{H, L\}$
- $\rightarrow = \{(H, H), (L, L), (H, L)\}$
- $H \oplus H = H, L \oplus H = L, L \oplus L = L$

Partial Order Policies

It often makes sense to assume that information can always flow within the same security level, security levels that are related to others in the same way are the same security level and, if information can flow from A to B and from B to C, it can flow from A to C. The flow relation $\rightarrow \subseteq SC \times SC$ is a partial order $(SC, \rightarrow)$ if it is:

- Reflexive: $\forall s \in SC, s \rightarrow s$
- Anti-symmetric: $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_1$ implies $s_1 = s_2$
- Transitive: $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$ implies $s_1 \rightarrow s_3$

When dealing with a partial order, the notation for $\rightarrow$ is $\leq$ and we can speak of security levels.

Hasse diagrams are convenient for representing information flow policies that are partial orders. They are directed graphs where security classes are nodes, the can-flow relation is represented by non-directed arrows, implicitly directed upward and reflexive/transitive edges are implicit.

1.1.3 Access Control to Information Flow Control

Information flow control focuses on how information is allowed to flow once an access control is granted. Access control is the control of interaction between subjects and objects, by validating access rights of subjects to resources of the system.

Discretionary Access Control (DAC)

Restricts access based on the identity of subjects and a set of access permissions that can be determined by subjects.

It has a limitation where access permissions might allow programs to, in effect, circumvent the policies. This can be done legally by means of information flows that are encoded in the program, or illegally, when vulnerabilities in programs and language implementations can be exploited by attackers.

Mandatory Access Control (MAC)

Restricts access based on security levels of subjects (their clearances) and objects (their sensitivity). Controls how information flows in a system based on whom is performing each access. It has limitations of restrictiveness and covert channels.

1.1.4 Encoding and Exploiting Information Flows

Objects may be classified as follows:

- Object - resource holding or transmitting information
- Security class/label - specifies who can access objects of that class
- Security labelling - assigns security classes to objects (statically or dynamically)

We use a standard imperative language where information containers are variables, where X_L denotes that a variable X has security level L. The information flow policy is as follows:

We want to ensure that propagation of information by programs respects information flow policies, i.e. there are no illegal flows. This means an attacker cannot infer secret input or affect critical output by inserting inputs into the system and observing its outputs.

1.2 Noninterference

1.2.1 Definition

A program is secure if, for every observational level L, for any two runs of the program that are given the same low inputs, if the program terminates in both cases, then it produces the same low outputs.

1.2.2 Attackers

Concurrent Attacker

An attacker program that is concurrently composed with the observed program does not depend on its termination. It has access to "low" outputs, and possibly non-termination (or even intermediate steps). Considering the following programs:

- p_L="$file_L$"; if RUID access to p_L then f=open(p_L);$f = 0$ (has RUID=L and EUID=H)
- $p_L = file_H$

Both are safe according to our notion of noninterference, but when composed concurrently, the program is insecure.

Possibilistic Input-Output Noninterference: is sensitive to whether the program is capable of terminating and producing certain final outputs.

Intermediate-Step Attacker

- $x_L := y : H; x_L := 1$

Possible low outcomes do not depend on y_H. However, the intermediate steps differ.

Intermediate-step-sensitive Noninterference: is sensitive to intermediate steps of computations.

Time-Sensitive Attacker

- $x_L := 0$; if y_H then skip else skip;skip;skip;skip ; $x_L := 1$

Possible outcomes and intermediate steps do not depend on y_H. However, the time it takes to change the value of x_L is different.

Temporal Noninterference: is sensitive to the time it takes to produce outputs.

Probabilistic Attacker

- $x_L := y_H \mid\mid x_L :=$ random(100)

Possible outcomes do not depend on y_H. However, the probability of the value of x_L revealing that of y_H is higher.

Probabilistic Noninterference: is sensitive to the likelihood of outputs.

1.2.3 Downgrading

Noninterference is simple and provides strong security guarantees. But sometimes we need to leak information in a controlled way.

- Declassification (for confidentiality) Example: flow declarations locally enable more flows

```
declassify password:L in
   if (password_H == attempt_L) {
     then print_L "Right!";
     else print_L "Wrong!";
   }
```

- Endorsement (for integrity) Example: pattern matching in Perl's taint mode

```
if ($filename_L =~ /^([-\@\w.]+)$/) {
  $filename_H = $1_H;
  open_H(FOO,"> $filename_H");
}
```

1.3 Formal Semantics

We will use two techniques to define the semantics of a programming language:

- Denotational semantics for expressions: defines mathematically what is the result of a computation.
- Operational semantics for instructions: describes how the effect of a computation is produced when executed on a machine

1.3.1 WHILE Language

This language has the following syntatic categories:

- c: constants
- x: variables
- a: arithmetic expressions
- t: tests
- S: statements

And follows the grammar:

- Operations: $op ::= + | - | \times | /$

- Comparisons: $cmp ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$
- Expressions: $a ::= c|x|a_1\ op\ a_2$
- Tests: $t ::= a_1\ cmp\ a_2$
- Statements: $S ::= x := a|skip|S_1; S_2|if\ t\ then\ S\ else\ S|while\ t\ do\ S$

The state/memory is represented as a function that maps variables to integers, ρ, for example, $\rho(x) = 1$. Now, we define the following semantic functions:

- $\mathcal{A}$: function that maps pairs of arithmetic expression and state to integers ($\mathcal{A}(x)_\rho = \rho(x)$)
- $\mathcal{B}$: function that maps pairs of test and state, to booleans ($\mathcal{B}(a_1\ cmp\ a_2)_\rho = \mathcal{A}(a_1)_\rho\ cmp\ \mathcal{A}(a_2)_\rho$)
- $\mathcal{S}$: partial function that maps pairs of statement and state to state ($< S, \rho > \rightarrow \rho'$: when executing program $\mathcal{S}$ on memory ρ we obtain the new memory ρ'. $< S, \rho > \rightarrow < \mathcal{S}', \rho' >$: Performing one step of program $\mathcal{S}$ on memory ρ leaves the continuation $\mathcal{S}'$ and produces new memory ρ')

The list of big-step axioms and rules is as follows:

Assignment: $< x{:=}a, \rho > \rightarrow \rho[x \mapsto \mathcal{A}[\![a]\!]\rho]$

Skip: $< skip, \rho > \rightarrow \rho$

Sequential composition: $$\frac{< S_1, \rho > \rightarrow \rho' \quad < S_2, \rho' > \rightarrow \rho''}{< S_1; S_2, \rho > \rightarrow \rho''}$$

Conditional test: $$\frac{< S_1, \rho > \rightarrow \rho'}{< \text{if t then } S_1 \text{ else } S_2, \rho > \rightarrow \rho'} \text{ if } \mathcal{B}[\![t]\!]\rho = true$$

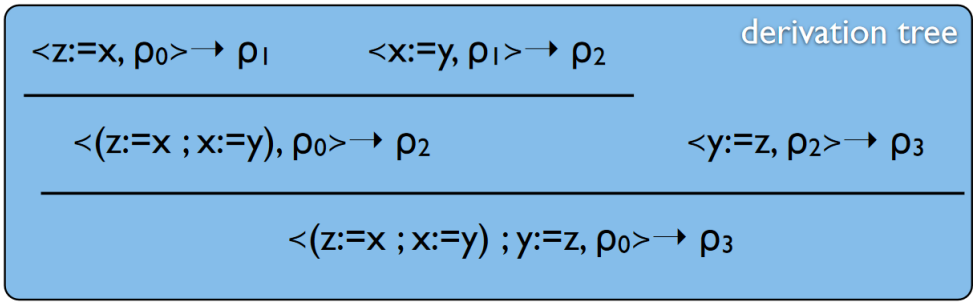
$$\frac{< S_2, \rho > \rightarrow \rho'}{< \text{if t then } S_1 \text{ else } S_2, \rho > \rightarrow \rho'} \text{ if } \mathcal{B}[\![t]\!]\rho = false$$

While loop: $$\frac{< S, \rho > \rightarrow \rho' \quad < \text{while t do S}, \rho' > \rightarrow \rho''}{< \text{while t do S}, \rho > \rightarrow \rho''} \text{ if } \mathcal{B}[\![t]\!]\rho = true$$

$$< \text{while t do S}, \rho > \rightarrow \rho \quad \text{if } \mathcal{B}[\![t]\!]\rho = false$$

For example, the sequential composition rule could be read as follows: When the first program starting on ρ produces ρ' and the second program starting on ρ' produces ρ", then the entire sequential composition starting on ρ produces ρ". The following is an example of a chaining of these rules:

- Evaluate (z:=x ; x:=y) ; y:=z, starting from a state ρ_0 that maps all variables except x and y to 0, and has $\rho_0(x) = 5$ and $\rho_0(y) = 7$.



$\rho_1 = \rho_0[z \mapsto 5]$ $\rho_2 = \rho_1[x \mapsto 7]$ $\rho_3 = \rho_2[y \mapsto 5]$

1.4 Security Properties and Enforcement Mechanisms

The definition of a security property is often not enough, since devs make mistakes and understanding whether a program satisfies the property is not always straightforward. At their core, security properties are about behaviour:

- Functional correctness
- Robustness
- Safety
- ...

Enforcement mechanisms were created to this end, automating the algorithm of preventing any given program from performing "unwanted" behaviors.

1.4.1 Program Analysis

Program analysis is the process of automatically analyzing the behavior of computer programs. The main aims of program analysis are:

- Optimization - about performance, to compute in a more efficient way
- Correctness - about assurance, to compute as intended

Automatic analysis can give stronger guarantees in less time, but is limited in scope and precision, which may come in the form of false positives or negatives.

Security properties typically talk about behavior of programs, and are often undecidable. On the other hand, enforcement mechanisms provide an automatic way of accepting/rejecting the behavior of programs, and are expected to be decidable.

Timing of Program Analysis

Program analysis may be done before program execution (static), during execution (dynamic) or using a combination of both, using the output of one to another (hybrid).

1.4.2 Static Analysis Mechanisms

Static analysis is closely related to compilation. Some tools used for static analysis include:

- String matcher: runs directly over source code. Simple tools like grep and findstr can do a very basic form of analysis.
- Lexical analyzer: runs over the tokens generated by the scanner. Can look for dangerous library/system calls.
- Semantic analyzers:
  - Control flow analysis: performs checks based on the possible control paths of a program; used to verify properties that depend on the sequencing of instructions.
  - Data-flow analysis: gathers information about the possible set of values calculated at various points of a program.Can determine where an actual value assigned to a variable might propagate.
  - Type checking: associate types to selected programs that fulfill certain requirements (eg. are considered correct with respect to a property)

Interactive Analysis

Verification of complex properties can be achieved with more human intervention, such as model checking or program verification.

- Model checking: checks a model (description) of a program, or the code itself. Enables to check its design.
- Program Verification: formally proves a property about a program. Uses a specification language (program logic) for expressing properties of a program and an associated logic for (dis)proving that programs meet specifications.

1.5 Static Analysis for Information Flow

We till not develop a static type-based analysis for our WHILE language. The general steps for an IFlow analysis are:

1. Definition of the language
2. Information flow policy of security levels
3. Classification of objects into security levels

4. Security property of programs
5. Mechanism for selecting secure programs
6. Guarantees about the mechanism

1.5.1 Definition of a Language

In order to define a language, its syntax and semantics must be defined. For this purpose we will use the WHILE language defined previously.

1.5.2 Information Flow Policy of Security Levels

In this step, we choose a set of security classes, an flow relation between them, and an operator for combining them. These security levels can speak for instance of confidentiality, integrity, or both. For this purpose, a lattice of security levels will be used.

Lattice Policies

A class of common information flow policies have some convenient ingredients:

- Security levels form a partial order (allowed flows are transitive, anti-symmetric a)
- Two security levels can always be combined
- There is a highest and a lowest level

A flow relation has a lattice structure:

$$L = (L, \leq, \vee, \wedge, \top, \bot)$$

- L: set of security levels
- $\leq$: partial order on L
- $\vee$: Join - least upper bound operation
- $\wedge$: Meet - greatest lower bound operation
- $\top$: top security level
- $\bot$: bottom security level

The operation $\oplus$ is given by $\vee$ or $\wedge$. Some more definitions:

- Upper and lower bounds:
  - Upper-Bound: given a partially ordered set $(S, \leq)$, element s is an upper-bound of two elements l_1 and l_2 if it satisfies $l_1 \leq s$ and $l_2 \leq s$
  - Lower-Bound: given a partially ordered set $(S, \leq)$, element s is a lower-bound of two elements l_1 and l_2 if it satisfies $s \leq l_1$ and $s \leq l_2$
- Join and meet:

- Join ($\vee$): defined for any two $l_1, l_2 \in L$, we have that $l_1 \vee l_2$ is the least upper-bound of l_1 and l_2 if all other upper-bounds s of l_1 and l_2 are greater: $l_1 \vee l_2 \leq s$
- Meet ($\wedge$): defined for any two $l_1, l_2 \in L$, we have that $l_1 \wedge l_2$ is the greatest lower-bound of l_1 and l_2 if all other lower-bounds s of l_1 and l_2 are lower: $s \leq l_1 \wedge l_2$

1.5.3 Classification of Objects into Security Levels

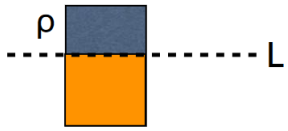
Objects, or information containers (variables, channels, files, etc) are given a security level. For this example, we will use a security labeling, i.e. amapping from variables to security levels:

$$(\Gamma : Var \to L)$$

1.5.4 Security Property of Programs

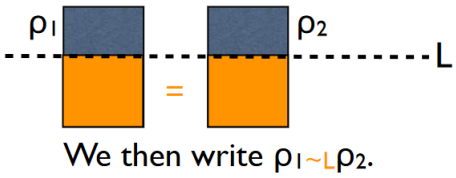
Consists of a formal specification of what it means for a program to be secure. While taking into consideration the language expressivity and execution context, specifies the power of the attacker. For this purpose, deterministic input-output noninterference will be used.

The part of a memory ρ that is observable at level L corresponds to the set of variables that are lower or equal to L.



(Omitting the parameter Γ for simplicity)

Two memories ρ_1 and ρ_2 are indistinguishable with respect to a security level L, if ρ_1 and ρ_2 agree on the values of variables that are observable at level L That is, for all x such that $\Gamma(x) \leq L$, then $\rho_1(x) = \rho_2(x)$.



(Omitting the parameter Γ for simplicity)

Deterministic Input-Output Noninterference -
A program S is secure if for every security level L and for all pairs of memories ρ_1 and ρ_2 such that $\rho_1 \sim_L \rho_2$, we have that
$\langle S, \rho_1 \rangle \rightarrow \rho_1'$ and $\langle S, \rho_2 \rangle \rightarrow \rho_2'$ implies $\rho_1' \sim_L \rho_2'$.

1.5.5 Mechanism for Selecting Secure Programs

For this goal, a type system must be built. A type system can be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute. This system should analyze programs written in the language that was defined (WHILE in our case) and only accept programs that follow the chosen security property (Noninterference, for this example). A type system is composed of:

- Types
- Typing environment (Γ): maps variables to types. Used to know what is the type of a variable
- Judgements ($\Gamma \vdash program : type$): partially map programs to types. Used for stating that a (part of a) program is typable and what is its type
- Rules ($\frac{assumptions}{conclusions}$): relate valid judgments with assumptions. Used for establishing syntactic conditions for judgments to hold

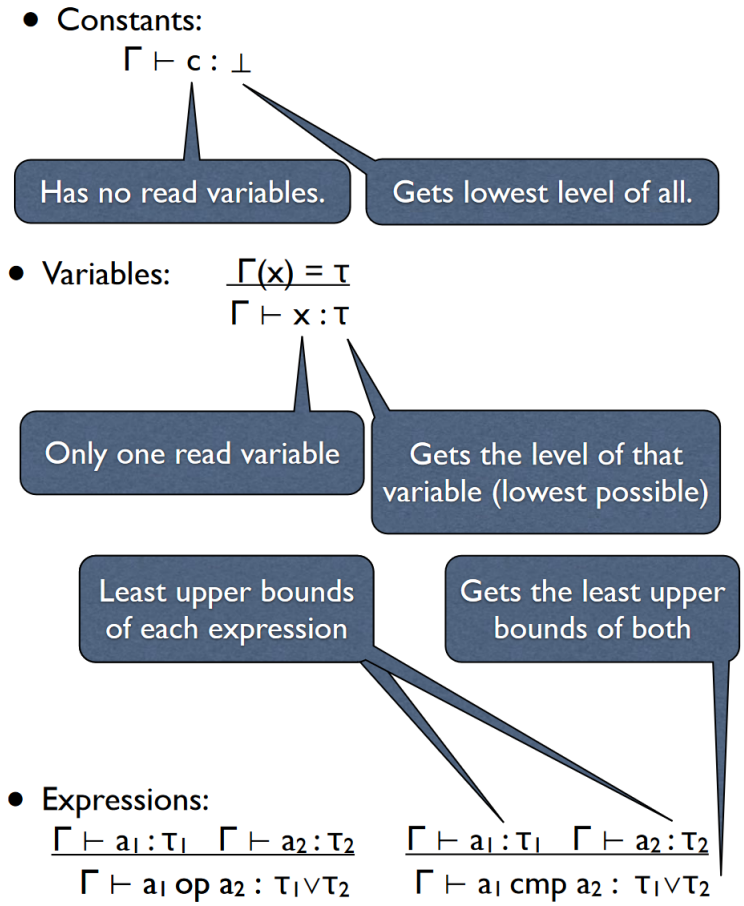
The idea is to choose a level that represents those of the variables that are read in an expression or test. Only type statements where written variables only depend on read variables (expressions or test) of lower or equal level.

- **Reading level for typing expressions**
  - Types: τ
  - Judgments: $\Gamma \vdash a : \tau$

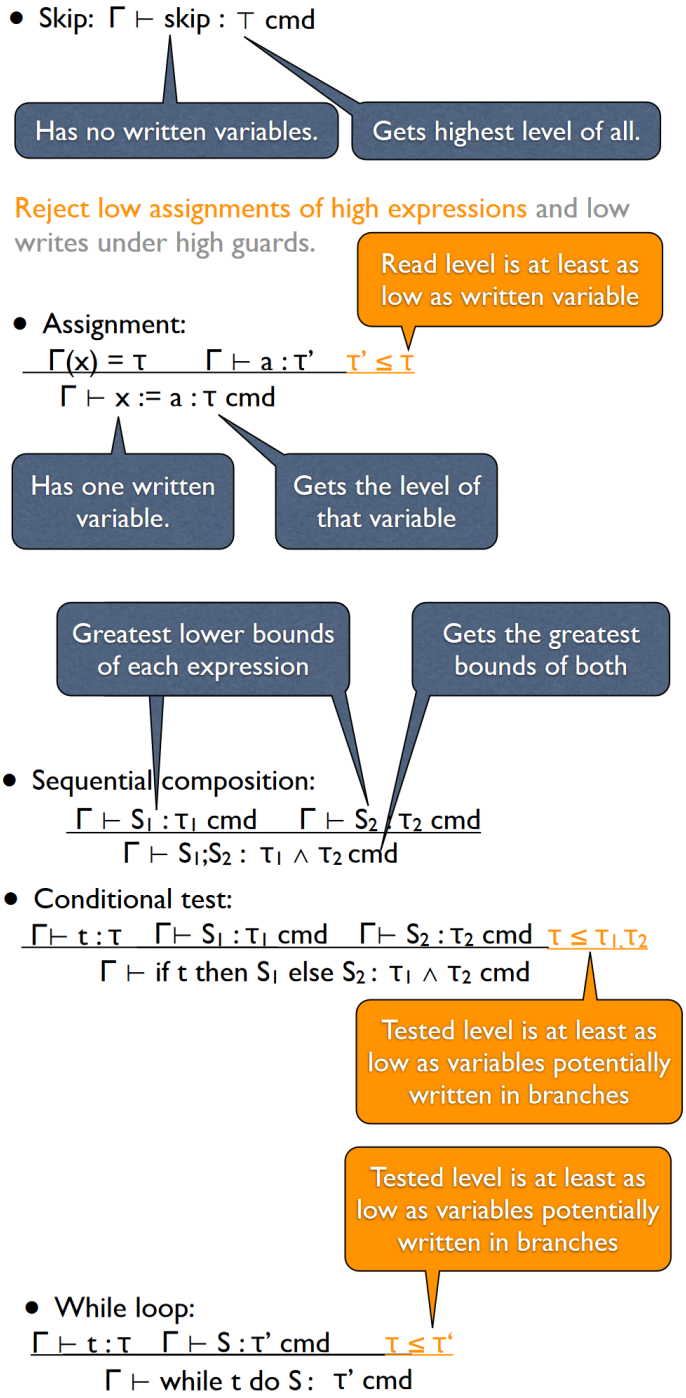
  the least upper bound to read variables
- **Writing level for typing statements**
  - Types: τ cmd
  - Judgments: $\Gamma \vdash S : \tau$ cmd

  the greatest lower bound to written variables

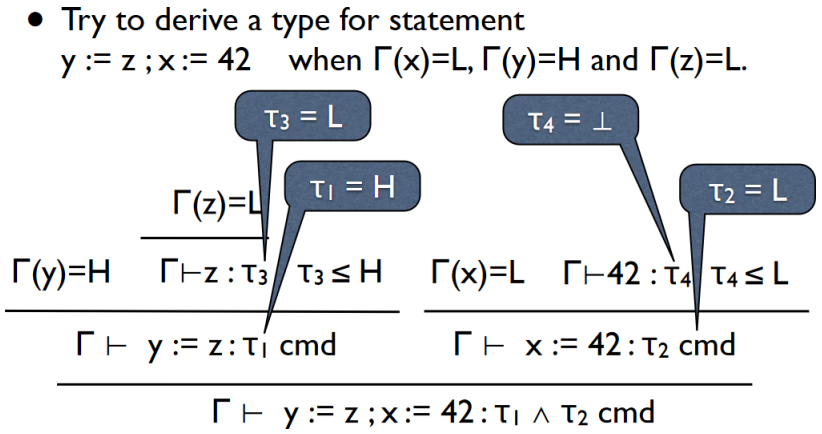
Types of expressions are at least as high as that of its sub-expressions (remaining an upper bound!). For example:



Types of statements are at least as low as those of its sub-statements (remaining a lower bound!).



Typing Derivation Tree



1.5.6 Guarantees About the Mechanism

There are two lemmas:

- Simple security: if $\Gamma \vdash a : \tau$ then expression a contains only variables of level τ or lower.

  Types given to expressions are obtained from types of its variables, and can only be raised.

- Confinement: if $\Gamma \vdash a : \tau cmd$ hen statement S assigns only to variables of level τ or higher.

  Types given to statements are obtained from types of its assigned variables, and can only be lowered.

Finally, there is the theorem of type soundness: if program S is typable then it satisfies deterministic input-output noninterference.

1.6 Dynamic Analysis

As more information is available at runtime, it is possible to use it and accept secure runs of programs that might be otherwise rejected by static analysis.

1.6.1 Accepting vs. Transforming

Our (previous) type system checks whether programs should be accepted: either accepts or rejects a program (there are also type systems that transform programs). Monitors change the behaviour of programs, so that the result is acceptable.

1.6.2 Web Security and Dynamic Languages

Most client-side web apps consist of (JavaScript) mashups - a combination of scripts, possibly from external untrusted origins (gadgets). Gadgets are highly dynamic - they can be loaded at runtime and can depend on the input given by the user.

Browsers follow a Same Origin Policy (SOP): a script loaded from one origin is not allowed to access or modify resources obtained from another origin. Access is granted based on the origin, and not on what will be done with that access.

Dynamic Code Evaluation

In the context of web applications, dynamic code evaluation is a popular feature. Nearly a quarter of the pages with embedded JavaScript use the eval primitive. There are two main dynamic approaches to control information flow in Javascript:

- Modify a JavaScript engine so that it additionally implements the security monitor. Lock-step monitor:
  - Define monitored semantics for the composition of the program and monitor, which can only perform safe executions
  - Each step of the monitor and program must be synchronized
- Inline the monitor into the original program, which has the advantage of being browser-independent. Inlined monitor:
  - Define a source-to-source compiler that transforms programs into "programs that monitor themselves".
  - The monitor is inlined in the program and transformed programs should be safe to execute

1.6.3 A Monitor for Information Flow Analysis

We want to be able to talk about each individual step performed by programs and monitors. We will need:

- To define a small-step labeled semantics for the programs ($\Rightarrow^{\alpha}$)
  - This will be divided in two parts: basic and labeled small-step semantics
- To define a small-step labeled semantics for the monitor ($\Rightarrow_m^{\alpha}$)

Basic Small-Step Semantics

We will need to make some additions to the syntax of WHILE, in order to obtain a dynamic WHILE. Namely, add to the syntatic categories:

- Syntactic categories:

s - strings
c ∈ Z - constants (integers) } values
x ∈ *Var* - variables
a ∈ *Aexp* - arithmetic expressions
t ∈ *Bexp* - tests
S ∈ *Stm* - statements

- Grammar (BNF notation):

op ::= + | - | * | / | ++ cmp ::= < | ≤ | = | ≠ | ≥ | >
a ::= s | c | x | a_1 op a_2 t ::= a_1 cmp a_2
S ::= x:=a |skip | S_1 ; S_2 | if t then S else S | while t do S |
| eval a

The small-step semantics are mostly unchanged:

- Configurations:
  - intermediate <Statement S, state ρ>
  - terminal ρ
- Transitions: <S, ρ>⇒ Υ where Υ is either <S', ρ'> or ρ'
- Rules: $\frac{<S_1, \rho_1>\Rightarrow \Upsilon_1 \ldots <S_n, \rho_n>\Rightarrow \Upsilon_n}{<S, \rho>\Rightarrow \Upsilon_n}$ if ....

And there is one addition to the small step axioms and rules:

Assignment: < x:=a, ρ>⇒ ρ[x↦ $\mathcal{A}[\![a]\!]\rho$]

Skip: < skip, ρ>⇒ ρ

Sequential comp.: $\frac{<S_1, \rho>\Rightarrow \rho'}{< S_1;S_2, \rho>\Rightarrow <S_2, \rho'>}$ $\frac{<S_1, \rho>\Rightarrow <S_1', \rho' >}{< S_1;S_2, \rho>\Rightarrow <S_1';S_2, \rho'>}$

Conditional test: <if t then S_1 else S_2, ρ>⇒<S_1, ρ> if $\mathcal{B}[\![t]\!]\rho$ = true
<if t then S_1 else S_2, ρ>⇒<S_2, ρ> if $\mathcal{B}[\![t]\!]\rho$ = false

While loop: <while t do S, ρ>⇒ <S ; while t do S, ρ> if $\mathcal{B}[\![t]\!]\rho$ = true
<while t do S, ρ>⇒ ρ if $\mathcal{B}[\![t]\!]\rho$ = false

Eval: $\frac{\mathcal{A}[a]\rho=s \qquad parse(s) = S}{< eval(a), \rho>\Rightarrow \ < S, \rho>}$

To evaluate statement S, starting from an initial state ρ, a derivation sequence can be constructed:

- Try to find an axiom or rule whose left side matches $< S, \rho >$, whose side conditions are satisfied, and for which you can build a derivation tree showing that it can be applied.
  - If it is an axiom: determine the final state and terminate.
  - If it is a rule: determine the intermediate configuration that is the right side of the rule, and try to find the derivation sequence starting form it.

An example follows:

- Evaluate (z:=x ; x:=y) ; y:=z, starting from a state ρ_0 that maps all variables except x and y to 0, and has $\rho_0(x) = 5$ and $\rho_0(y) = 7$.

$<(z:=x ; x:=y) ; y:=z, \rho_0> \Rightarrow <x:=y ; y:=z, \rho_0[z \mapsto 5]>$

$\Rightarrow <y:=z, \rho_0[z \mapsto 5, x \mapsto 7]>$

$\Rightarrow \rho_0[z \mapsto 5, x \mapsto 7, y \mapsto 5]$

derivation sequence

Labelled Small-Step Semantics

The runtime syntax of dynamic WHILE sees one more change:

- Syntactic categories:

s - strings
c ∈ Z - constants (integers) } values
x ∈ *Var* - variables
a ∈ *Aexp* - arithmetic expressions
t ∈ *Bexp* - tests
S ∈ *Stm* - statements

runtime construct: end

- Grammar (BNF notation):

op ::= + | - | * | / | ++ cmp ::= < | ≤ | = | ≠ | ≥ | >
a ::= s | c | x | a_1 op a_2 t ::= a_1 cmp a_2
S ::= x:=a |skip | S_1 ; S_2 | if t then S else S | while t do S |
| eval a | end

Also on the small-step semantics:

- Configurations cfg:
  - intermediate <Statement S, state ρ>
  - terminal ρ
- Transitions: $\langle S, \rho \rangle \Rightarrow^{label} \Upsilon$ where Υ is either $\langle S', \rho' \rangle$ or ρ'
- Rules: $\dfrac{\langle S_1, \rho_1 \rangle \Rightarrow^{label1} \Upsilon_1 \quad \ldots \langle S_n, \rho_n \rangle \Rightarrow^{labeln} \Upsilon_n}{\langle S, \rho \rangle \Rightarrow^{label} \Upsilon_n}$ if ....

And the rules/axioms:

- Skip: $\langle skip, \rho \rangle \Rightarrow^{nop} \rho$
- Assignment: $\langle x:=a, \rho \rangle \Rightarrow^{(x,a)} \rho[x \mapsto \mathcal{A}[\![a]\!]\rho]$
- End: $\langle end, \rho \rangle \Rightarrow^{f} \rho$
- Sequential composition:
  $\dfrac{\langle S_1, \rho \rangle \Rightarrow^{\alpha} \langle S_1', \rho' \rangle}{\langle S_1;S_2, \rho \rangle \Rightarrow^{\alpha} \langle S_1';S_2, \rho' \rangle} \qquad \dfrac{\langle S_1, \rho \rangle \Rightarrow^{\alpha} \rho'}{\langle S_1;S_2, \rho \rangle \Rightarrow^{\alpha} \langle S_2, \rho' \rangle}$
- Conditional test:
  $\langle$if t then S_1 else $S_2, \rho \rangle \Rightarrow^{b(t)} \langle S_1;end, \rho \rangle$ if $\mathcal{B}[\![t]\!]\rho$ = true
  $\langle$if t then S_1 else $S_2, \rho \rangle \Rightarrow^{b(t)} \langle S_2;end, \rho \rangle$ if $\mathcal{B}[\![t]\!]\rho$ = false
- While loop:
  $\langle$while t do S, $\rho \rangle \Rightarrow^{b(t)} \langle S;end;$while t do S, $\rho \rangle$ if $\mathcal{B}[\![t]\!]\rho$ =true
  $\langle$while t do S, $\rho \rangle \Rightarrow^{b(t)} \langle end, \rho \rangle$ if $\mathcal{B}[\![t]\!]\rho$ =false
- Eval: $\dfrac{\mathcal{A}[a]\rho = s \qquad parse(s) = S}{\langle eval(a), \rho \rangle \Rightarrow^{b(a)} \langle S ; end, \rho \rangle}$

Note: $b(t)$ serves the purpose of indicating to the monitor that the program is entering a region guarded by t.
The runtime instruction "end" is only for sending a message to the monitor.

An example follows:

Program execution:
$\langle$ (if x_H then $y_L := 1$ else $x_H := 0$); $z_L := 1, \rho \rangle$
$\Rightarrow^{b(xH)} \langle y_L := 1; end ; z_L := 1, \rho \rangle$
$\Rightarrow^{(yL,1)} \langle end ; z_L := 1, \rho[y_L \mapsto 1] \rangle$ $\rho(x_H)$=true
$\Rightarrow^{f} \langle z_L := 1, \rho[y_L \mapsto 1] \rangle$
$\Rightarrow^{(zL,1)} \rho[y_L \mapsto 1, z_L \mapsto 1]$

Monitor Small-Step Semantics

The monitor either accepts an event generated by the program or blocks it by getting stuck. It operates on stacks of security levels (st). (The empty stack is represented by ε.) Transitions are labeled. The rules are as follows:

- nop: $st \Rightarrow_m^{nop} st$
- branching: $st \Rightarrow_m^{b(t)} lev(t) :: st$
- end: $l :: st \Rightarrow_m^{f} st$
- assignment: $$\frac{lev(a) \leq \Gamma(x) \quad lev(st) \leq \Gamma(x)}{st \Rightarrow_m^{(x,a)} st}$$

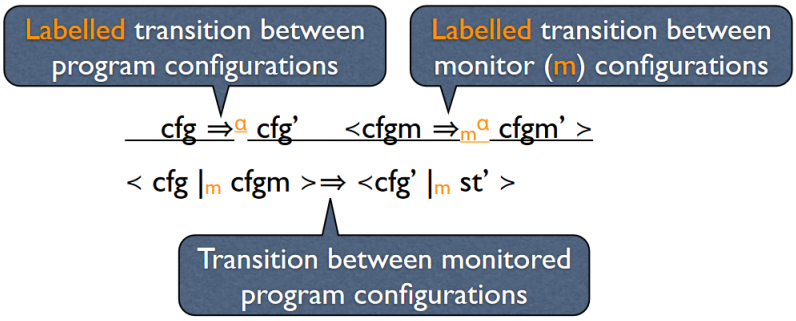
$lev(e) = \bigvee$ levels of the variables in e

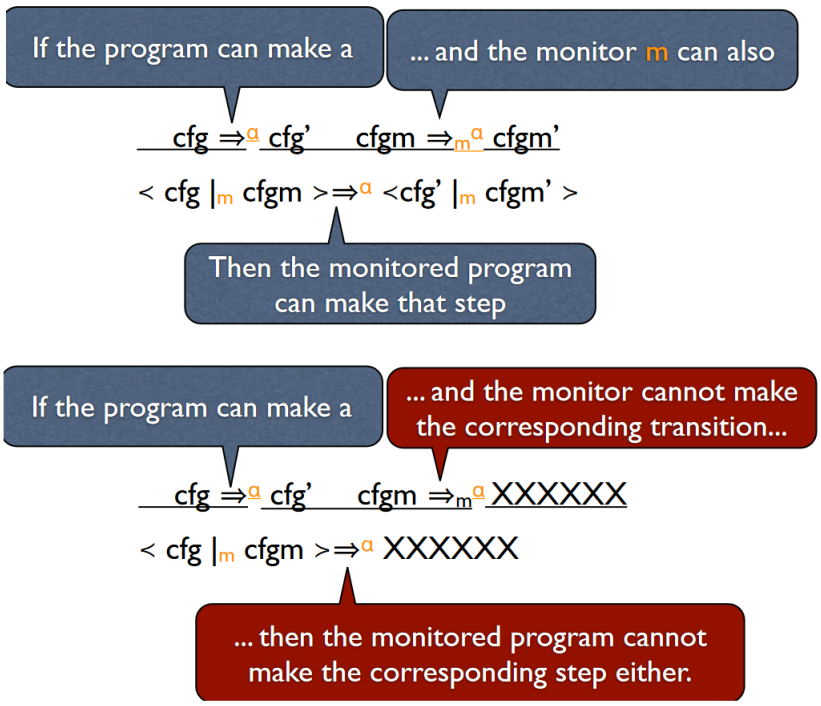
$lev(st) = \bigvee$ levels in st

An example follows:

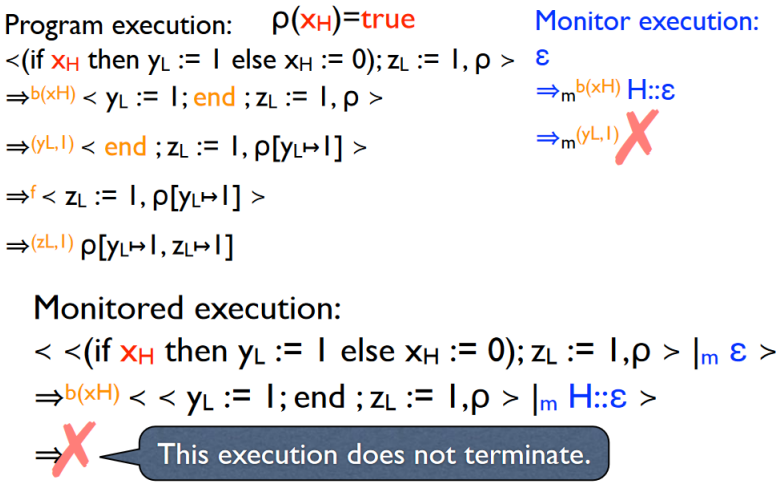
Program execution:	Monitor execution:
$< (\text{if } x_H \neq x_H \text{ then } y_L := l \text{ else } x_H := 0); z_L := l, \rho >$	ε
$\Rightarrow^{b(x_H \neq x_H)} < x_H := 0; \text{end} ; z_L := l, \rho >$	$\Rightarrow_m^{b(x_H \neq x_H)} H::\varepsilon$
$\Rightarrow^{(x_H,0)} < \text{end} ; z_L := l, \rho[x_H \mapsto 0] >$	$\Rightarrow_m^{(x_H,0)} H::\varepsilon$
$\Rightarrow^{f} < z_L := l, \rho[x_H \mapsto 0] >$	$\Rightarrow_m^{f} \varepsilon$
$\Rightarrow^{(z_L,l)} \rho[x_H \mapsto 0, z_L \mapsto l]$	$\Rightarrow_m^{(z_L,l)} \varepsilon$

Lock-step monitored semantics compose ($|_m$) the executions of the program and the monitor:





For example:



1.7 Security Verification and Bug-Finding

Testing and code reviews are widely used to find bugs in code, but each test only explores one possible execution, i.e. some bugs are still missed.

1.7.1 Program Properties and Noninterference

A program *Property* is a set of behaviours. The behaviours of a program S are denoted $\{[S]\}$.

- Verification of a program S with respect to *Property* means proving that all the behaviors of S are contained in *Property*:

$$\{[S]\} \subseteq Property$$

- Bug Finding in a program S with respect to *Property* means finding a behavior of S that is not in *Property*:

$$\{[S]\} \cap \neg Property \neq \emptyset$$

Properties may be categorized as satety properties (something bad never happens, e.g. type/memory safety) or liveness properties (something good will eventually happen, e.g. termination). Depending on how we define the set of allowed behaviors, we get different classes of properties:

- Trace properties: behaviors are traces
- 2-trace properties: behaviors are 2-traces

Trace

Represents a complete trace of a program: a sequence of configurations (cfg) that represents a complete step-by-step execution of a program (according to the small-step semantics).

These can be finite or infinite sets $[cfg_0, ...]$, such that $cfg_0 =< S_0, \rho_0 >$ and $\forall 0 \leq i \leq n, cfg_i \Rightarrow cfg_{i+1}$. $\{[S_0]\}$ is the set of trace-behaviors of S_0; it includes all the (infinite and finite) traces $(\{[cfg_0, ..., cfg_n]\}, \{[cfg_0, ...]\})$.

Additional small-step-semantics rules for representing errors:

Assignment: $$\frac{\mathcal{A}[\![a]\!]\rho = \lightning}{< x{:=}a, \rho> \Rightarrow \lightning}$$

Sequential composition: $$\frac{<S_1, \rho> \Rightarrow \lightning}{< S_1;S_2, \rho> \Rightarrow \lightning}$$

Conditional test: $$\frac{\mathcal{B}[\![t]\!]\rho = \lightning}{<\text{if } t \text{ then } S_1 \text{ else } S_2, \rho> \rightarrow \lightning}$$

While loop: $$\frac{\mathcal{B}[\![t]\!]\rho = \lightning}{<\text{while } t \text{ do } S, \rho> \Rightarrow \lightning}$$

We can then define $NoDivZero$ as a safety property and $Termination$ as a liveness property:

"No program execution performs a division by zero"

$\mathcal{NoDivZero}$ = {[cfg$_0$, …, cfg$_n$] | cfg$_n \neq \lightning$} ∪ {[cfg$_0$, …]}

"All program executions eventually terminate correctly"

$\mathcal{Termination}$ = {[cfg$_0$, …, cfg$_n$] | cfg$_n$=ρ}

2-Trace

$\{[S_0]\}$, the set of "behaviours of S_0" is the set of pairs of traces, which includes all pairs of infinite or finite traces.

$$\{[cfg_0, ..., cfg_n], [cfg'_0, ..., cfg'_m]\} \tag{1.1}$$
$$\cup \{[cfg_0, ..., cfg_n], [cfg'_0, ...]\} \tag{1.2}$$
$$\cup \{[cfg_0, ...], [cfg'_0, ..., cfg'_m]\} \tag{1.3}$$
$$\cup ... \tag{1.4}$$

Noninterference is a 2-trace property: "no pair of program executions reveals an information leak".

1.7.2 Verification and Bug-finding for Noninterference

Existing verification tools are mainly for trace properties. If we can formulate Noninterference as a trace property, we can use an existing analysis to check it instead of building a new one from scratch. The idea is to verify noninterferece by self-composition:

$$\{[S]\} \subseteq Noninterference(2 - trace\ property) \tag{1.5}$$
$$iff \tag{1.6}$$
$$\{[SelfC(S)]\} \subseteq SafetyNoninterference(trace\ property) \tag{1.7}$$

Self-composition consists of performing 2 runs of S at once by creating a copy of S using fresh variables from Var, then executing $S; S$. To assume and assert same "low" variables in the encoded "two runs" we use a memory $\rho + \overline{\rho}$, where ρ and $\overline{\rho}$ don't overlap but between which we can map variables.

To prepare a program S for verification/bug finding for Noninterference by symbolic execution:

- Produce a program $SelfC(S) = S; S$ by self-composition
- For each low variable x that appears in S:
  - Add assumptions and assertions $x = x$, at the beginning and end of the program, respectively.

- $x_L := y_H$

```
assume(x = x);
x := y;
x := y;
assert(x = x)
```

- $z_L := l$;
  if (h_H) then $x_L := l_L$ else skip;
  if (!h_H) then $x_L := z_L$ else skip;
  $l_L := x_L + y_L$

```
assume(l=l ∧ x=x ∧ y=y ∧ z=z)
z:=1;
if (h) then x:=1 else skip;
if (!h) then x:=z else skip;
l:=x+y;
z:=1;
if (h) then x:=1 else skip;
if (!h) then x:=z else skip;
l:=x+y;
assert(l=l ∧ x=x ∧ y=y ∧ z=z)
```

Bug-finding for Noninterference

To search for a Noninterference bug in S:

- Prepare S for trace-based verification/bug finding
- Perform a symbolic execution of the new program
  - Collect all assumptions over the symbolic variables in the path condition
- See if path condition implies assertions for all possible symbolic values
  - (equivalently) see if negation of the implication is satisfiable for any symbolic values

```
assume(x = x);
x := y;
x := y;
assert(x = x)
```

```
True,  [ x→#x, y→#y, x→#x, y→#y ]   assume(x = x)
#x=#x, [ x→#x, y→#y, x→#x, y→#y ]   x := y
#x=#x, [ x→#y, y→#y, x→#x, y→#y ]   x := y
#x=#x, [ x→#y, y→#y, x→#y, y→#y ]   assert(x = x)
```

(#x = #x) ⇒ (#y = #y) **Valid? No!** ✗

(#x = #x) ∧ (#y ≠ #y) **Is its negation SAT? Yes!**

Verification for Noninterference

To verify that S satisfies Noninterference, prove the absence of Noninterference bugs in S:

- Prepare S for verification/bug finding for Noninterference by symbolic execution
- For each symbolic execution path of the new program:
  - Search for a Noninterference bug
- S satisfies Noninterference iff no bug is found

Chapter 2

Vulnerabilites And Secure Software Design

In secure software design, there are 3 main security attributes: confidentiality, integrity and avaliability.

A vulnerability is a system defect relevant security-wise, which may be exploited by an attacker to subvert security policy. Vulnerabilities may be classified as:

- Design vulnerabilities
- Coding vulnerabilities
- Operational vulnerabilities

2.1 Attacks

Attacks enter through interfaces, the attack surfaces. Attacks can be techincal or through social engineering, directed or not, manual or automated.

2.1.1 Manual Attacks

Some examples of manual attacks include:

- Footprinting
- Scanning
- Enumeration
- Discovering vulnerabilites
- ...

2.1.2 Automated Attacks

Worm

A worm is composed of a target selector, a scanning motor, a warhead (exploit code), a load and a propagation motor.

Drive-by Download

Performed by web pages with malware. When user accesses one with a vulnerable browser, the malware exploits the vulnerability.

Viruses and Trojans

Viruses are similar to worms but propagate with physical contact (usb drives, disks, ...). Trojans are also similar but requires the user to run an infected program (e.g. emails with attachments).

2.1.3 Torpig

Torpig is a sophisticated malware. It infects bots with drive-by download. Attackers modify legitimate but vulnerable server for some webpages to request JavaScript code from the attacker's web server:

1 The victim's browser accesses the vulnerable server

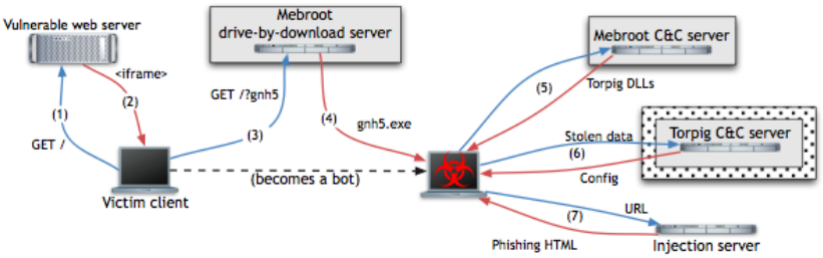
2 JavaScript code exploits the browser/plugins/etc.

3-4 If an exploit is successful, the script downloads and installs the Mebroot rootkit (replaces Master Boot Record) – victim becomes a bot. Mebroot has no attack capacity.

5 Contacts C&C server to obtain malicious modules and stores them encrypted in directory system32 and changes the names and timestamps to avoid suspicions.
Every 2h contacts C&C server: sends its configuration (type/version of modules); gets updates; communication is encrypted over HTTP

6 Every 20 minutes contacts C&C to upload stolen data

7 When victim visits domain from a list (e.g., a bank), the bot contacts an injection server. Injection server returns attack data: URL of trigger page in the legitimate domain (typ. the login page), where to send results, etc. When user visits trigger page, Torpig asks injection server for another page (e.g., that asks for credit card number)



2.2 Race Conditions

Race conditions violate the assumption of atomicity. The vulnerability lies in a problem of concurrency or lack of proper synchronization. There are several sources of races:

- Shared data (files and memory)
- Preemptive routines (signal handlers)
- Multi-threaded programs

There are 3 main kinds of race conditions, discussed next.

2.2.1 TOCTOU

TOCTOU stands for "time-of-check to time-of-use". It is often perpetrated through the use of symbolic links (special files that reference other files/folders).

The *access* function is specially vulnerable as it was designed for *setuid* programs. It does privilege check using the process' real UID instead of the effective UID.

When a call with a pathname is done ($open, access, stat, lstat, ...$), the pathname is resolved until the inode is found, so if two calls are made one after the other the path can lead to different inodes. The solution is to avoid the two sequential resolutions by avoiding using filenames inside the program. Functions such as $fstat, fchmod$ and $fchown$ are safe.

2.2.2 Temporary Files

Temporary files have the added problem of being in a shared directory. The typical attack is as follows:

- Privileged program checks that there is no file X in /tmp
- Attacker races to create a link called X to some file, say/etc/passwd
- Privileged program attempts to create X and opens the attacker's file doing something undesirable that its privileges allow

One may use the *mkstemp* function to create a unique, currently unused, filename from template (*mkdtemp* for directories).

Possible solutions include setting *unmask* appropriately and using $fopen$ instead of *open*.

2.2.3 Concurrency and Reentrant Functions

In the previous cases, concurrency is created by the attacker, with malicious intention. In many cases in which there is normal concurrency of operations on objects, operations may have to be executed atomically (using mutual exclusion mechanisms).

A function is reentrant if it works correctly even if its thread is interrupted by another thread that calls the same function. Such functions cannot use static variables, global variables, other shared resources like libraries (i.e and can only call other reentrant functions.

In some cases, unix signals may be useful in indicating asynchronous events to a process.

2.3 Buffer Overflows

A buffer is a memory space with contiguous chunks of the same data type (typically bytes or chars).We have a buffer overflow when a program writes after the end of a buffer.

When a buffer overflow occours, the program becomes unstable, and may crash or proceed. Side effects depend on how much data was written (or overwritten), if the program tries to read such data, etc.

2.3.1 Stack Overflows

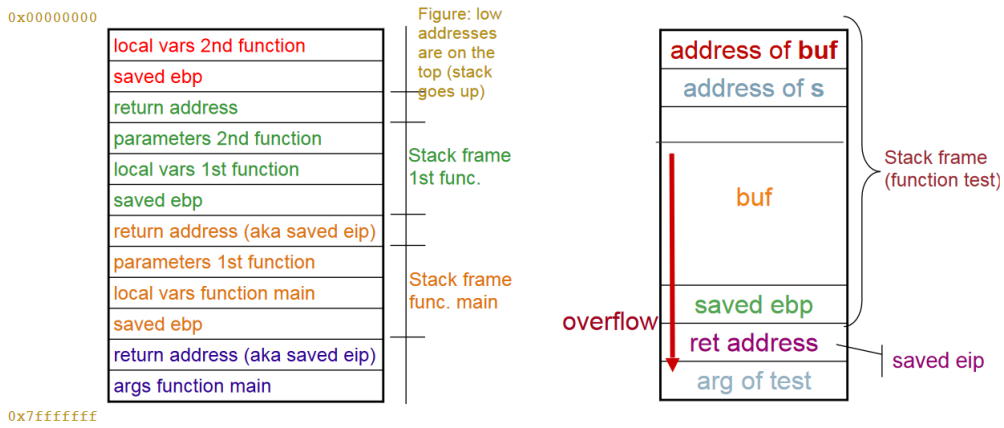
Stack Smashing

Stack smashing is the classic stack overflow attack. For example, the following code is vulnerable, since is inserts untrusted input in *buf* without checking:

```
void test(char *s) {
  char buf[10];          // gcc stores extra space
  strcpy(buf, s);        // does not check buffer's limit
  printf("&s = %p\n&buf[0] = %p\n\n", &s, buf);
}

main(int argc, char **argv) {
  test(argv[1]);
}
```

Such attacks can overflow local vars or the saved EIP, and may crash or modify the state of the program, as well as executing code. The effect on the stack can be visualized on the following image:



Code Injection

Consists of inputting arbitrary code to be executed. For this attack, the stack has to be executable and the address where the code is injected has to be discover. It is often difficult because of the reduced space and the inclusion of null bytes (functions like *strcpy* stop in the first zero).

Arc Injection / return-to-libc

Consists of inserting a new arc in the program control-flow graph (stack-smashing includes a new node in the graph). For example, overrun the return address to point to code already in the program.

Pointer Subterfuge

Pointer subterfuge is a general term for exploits that involve modifying a pointer. The objective can be to circumvent protections against BOs when the return address is protected. There are four types:

- Function-pointer clobbering: modify a function pointer to point to attacker supplied code
- Data-pointer modification: modify address used to assign data
- Exception-handler hijacking: modify pointer to an exception handler function
- Virtual pointer smashing modify the C++ virtual function table associated with a class

2.3.2 Integer Overflows

The semantics of integer-handling are complex and programmers often don't know the details.

Integer overflows are usually caused due to insufficient memory allocation, causing a buffer overflow, or excessive memory allocation/infinite loop, leading to

DoS. On rarer occasions they may originate from signedness errors (signed integer is interpreted as unsigned or vice-versa) or truncation (assigning an integer with a longer width to another shorter)

2.3.3 Heap Overflows

Similar to a stack overflow, a heap overflow occurs when a program writes more data to a dynamically allocated region in the heap memory than it can hold.

2.3.4 Return-Oriented Programming (ROP)

ROP provides a "language" which an attacker might use to make a machine do something useful. It does not involve code injection; generalizes return-to-libc (since it does not work well in 64 bit CPUs, as the first function parameters are put in registers).

This type of attack uses gadgets, which are sequences of instructions ending with ret(C3). In order to find a gadget, search for C3 and see what instructions can be found before it. Gadgets aren't necessarily based on instructions of the original code.

The attack consists of overflowing the stack with:

- Addresses of gadgets
- Other data the gadgets may pick from the stack

This constructs a ROP chain where each gadget performs a specific operation and ends with a "return" instruction, transferring control to the next gadget in the chain. By carefully selecting gadgets and their addresses, the attacker can build a sequence that achieves arbitrary actions, such as disabling security features or executing malicious code.

2.3.5 Prevention

The prevention method is simple: always do bounds checking; problems might arise only when you cannot control input. Some C functions should be avoided:

- *gets*
- *strcpy*
- *sprintf*
- *scanf*
- *streadd*
- ...

There is also a risk of internal BOs, when the overflow is not in the program's buffers but inside a library function.

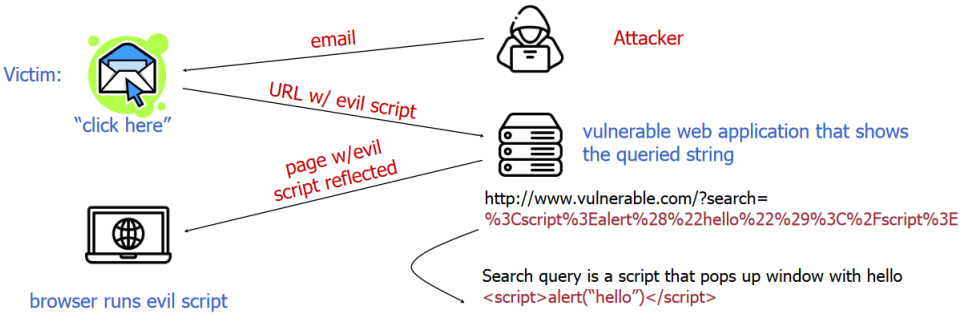
2.4 Web Application Vulnerabilities

Web suffers from all the 3 causes of trouble: complexity, extensibility, connectivity.

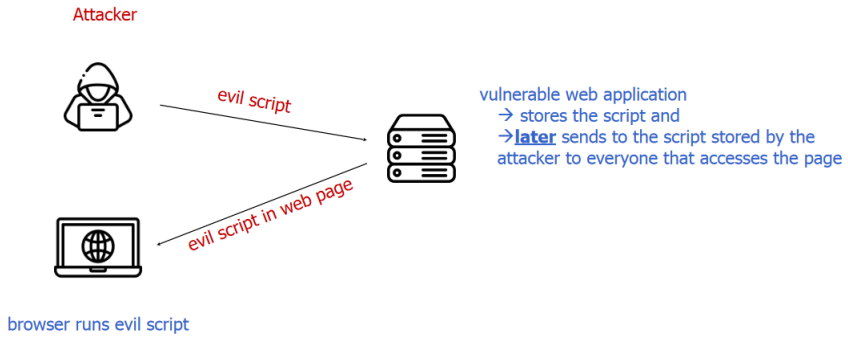
2.4.1 Cross Site Scripting (XSS)

XSS allows attacker to execute script in the victim's browser. There are different types of XSS:

- Reflected XSS (or non-persistent): page reflects user supplied data directed to the user's browser



- Stored XSS (or persistent): hostile data (script) is stored in a database, file, etc., and is later sent to user's browser



- DOM based XSS (Document Object Model): manipulates JavaScript code and attributes instead of HMTL

In reflected and stored XSS the server injects the script. In DOM based XSS, the client injects the script.

Some protection mechanisms are:

- Server-side input validation
- Strong output encoding

- Content Security Policy (level 2)
- State tracking mechanisms

2.4.2 CRLF Injection

It is similar to reflected XSS but the injection is in the response header, while in reflected XSS the injection is in the responde body.

The attacker inserts a carriage return (CR) and a life feed (LF) creating a new field in the header, or a second response (HTTP response splitting).

Just as in the reflected XSS, the attacker sends the victim a URL of a vulnerable website.

2.4.3 Direct Object Reference (DOR)

Occours when a site exposes a reference to an internal object and no proper access control (e.g. files, database records, keys, ...).

Should be prevented by forbidding the exposure of object references and by doing proper access control.

2.4.4 Local File Inclusion (LFI)

LFI is a vulnerability that occurs when an application allows an attacker to include files on the server through user-controllable input. LFI attacks can expose sensitive information, and in severe cases, they can lead to cross-site scripting (XSS) and remote code execution.

DOR and LFI can be related in scenarios where a DOR vulnerability allows an attacker to determine the location of sensitive files, and an LFI vulnerability then allows the attacker to include or access those files.

2.4.5 Cross Site Request Forgery (CSRF)

Many sites do certain actions based on an automatically submitted, fixed, ID, typically a session cookie. The attack consists of forcing the user to execute unwanted actions in a vulnerable site in which it is authenticated. Can be done by sending a link by email or chat.

The solution is similar to XSS, but may also consist on using nonces or requiring re-authentication for critical actions.

2.4.6 Failure to Restrict URL Access

Pages that are "protected" simply by being inaccessible from the "normal" web tree. The attack consists of forced browsing: guess links and brute force to find unprotected pages.

Protection relies in having good access control, no "hidden" pages as a form of protection.

2.4.7 Unvalidated Redirects and Forwards

Applications frequently redirect users to other pages. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page.

2.4.8 Others

Insecure Cryptographic Storage

Can be caused by sensitive data not being encrypted, use of home-grown or weak algorithms, ...

Insecure Transport Layer Protection

Sensitive traffic or authenticated sessions may be unencrypted. HTTPS should be used.

2.5 Database Vulnerabilities

SQL injection is the most common vulnerability in modern database systems. It may occour when user input is pasted into SQL commands; the concrete syntax is dependent on the specific DBMS and server-side language used. The steps of an attack include:

- Identifying parameters vulnerable to injection
- Database fingerprinting (discover type and version using queries)
- Discovering DB schema
- Extracting/adding/modifying data from the DB
- Denial of service
- Evading detection

2.5.1 Injection Methods

Tautologies

Inject code in 1 or more conditional statements so that it always evaluate to true (a tautology). It is commonly used to bypass authentication and extract data.

For example: $'\ or\ 1=1\ --$. Such an input may be evaluated as:

$SELECT\ accounts\ FROM\ users\ WHERE\ login=''\ or\ 1=1\ --\ '\ AND\ pass=''\ AND\ pin=$

Commenting the last part and breaking the query.

Union Query

Trick the app into returning additional data by injecting $UNION\ SELECT\ <\ rest\ >$. The attacker uses $<\ rest\ >$ to extract data from another table, since the query returns the union of data.

Piggy-Backed Queries

It does not modify a query, instead adds more queries; requires the DB to be configured to accept multiple statements in a single string.

For example: $';DROP\ TABLE\ users\ --$.

Stored Procedures

The injection could instead target a stored procedure.

Example: variation of the example code above that calls the following stored procedure (that checks credentials):

```
CREATE PROCEDURE DBO.isAuthenticated
    @username varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users WHERE login='" + @username + "' and
  pass='" + @password + "' and pin=" + @pin);
GO
```

The procedure is vulnerable to the same attacks seen before

Illegal/Incorrect Queries

Find injectable parameters (DB type/version, schema) by causing errors:

- Syntax errors: identify injectable parameters
- Type errors: deduce data types of certain columns or extract data
- Logical errors: reveal names of tables and columns that caused error

Inference

Attack attempts to infer if a certain condition on the state of the DB is true or false. The objective is similar to using illegal/incorrect queries. There are two techniques:

- Blind injection: Information is inferred by asking true/false questions

Example: If these 2 queries return different outputs, then user legalUser exists

```
SELECT accounts FROM users WHERE login='legalUser' and 1=0
  -- ' AND pass='' AND pin=0        (always false)
```

```
SELECT accounts FROM users WHERE login='legalUser' and 1=1
  -- ' AND pass='' AND pin=0        (true if legalUser exists)
```

- Timing attack: Information is inferred from delay in the response, usually with a branch that executes a $WAITFOR\ DELAY$

  Example: extract the name of a table from the DB

  SELECT accounts FROM users WHERE login='legalUser' and ASCII(SUBSTRING((select top 1 name from sysobjects), 1,1)) > X WAITFOR DELAY 5 -- ' AND pass='' AND pin=0

  – If field is injectable and 1st character from the 1st table is greater (in ASCII) than X then delay of 5 seconds

Alternate Encodings

Not a full attack but a trick to evade detection. The idea is to encode input in an unusual format, such as hex, ascii or unicode. For example:

$SELECT\ a\ FROM\ u\ WHERE\ login = '\ legalUser'; exec(char(0x73687574646f776e)) --\ AND\ ...$

In this case it executes as $exec(shutdown)$.

2.5.2 Injection mechanisms

- GET/POST inputs
- Cookies (can be used by the server to build SQL commands, $SELECT\ ...\ WHERE\ cookie = '\ \%s'\ ...$)
- Header fields (in PHP). Replace the XXX in $\$\_SERVER['HTTP\_XXX']$ with $HTTP\_URL$, $HTTP\_ACCEPT\_LANGUAGE$, ...

It's also possible to use second-order injection, where the input is provided so that it is kept in the system and later used. For example, registering a new user as $admin'--$; The site correctly escapes the input and accepts it, but will act as an injection on further queries.

2.5.3 Others

There exist more, less frequent, vulnerabilities in DBMS', such as:

- Blank and default passwords
- Unprotected communication (client-server)
- Several open ports
- Default (privileged) accounts
- Code vulnerabilities
- Encryption problems

2.6 Protection in Operating Systems

Protection is emplyed to ensure that objects are not accessed by unauthorized subjects. There are two aspects: separation and mediation.

2.6.1 Separation

Common operating systems (Unix, Windows) run software basically in two modes, enforced by the CPU:

- Kernel mode: software can play with any system resource (memory, I/O devices,...)
- User mode: access to resources is controlled by the OS. Software has to call the OS kernel to make privileged operations

There are several forms of separation:

- Physical separation: different processes use different devices (e.g. printers for different levels of security)
- Temporal separation: processes with different security requirements are executed at different times
- Logical separation: processes operate under the illusion than no other processes exist
- Cryptographic separation: processes use cryptography to conceal their data and/ or computations in a way that they become unintelligible to other processes

Memory Protection

Logical separation is often used to achieve memory protection. The most common solutions are segmentation (program is split in pieces with logical unit, such as code, data, stack...) and paging (program is divided in pages of the same size).

From a protection point of view, pages are similar to segments: a process can only access a segment only if appears in its segment translation table or can only see a page if it appears on its table.

2.6.2 Access Control

Access control is concerned with validating the access rights of subjects to resources of the system. It should be implemented by a reference monitor, following 3 principles:

- Completeness: it must be impossible to bypass
- Isolation: it must be tamperproof
- Verifiability: it must be shown to be properly implemented

Some basic access control mechanisms include:

- Access control lists (ACLs): Each object is associated with a list of pairs (subject, rights)
- Capabilities: Each subject has a list of objects that it may access, i.e. pairs (object, rights). Capabilities are cryptographically protected against modification and forging

- Access control matrix: A matrix with lines per subject, columns per object, rights in the cells

There are also two basic access control policies:

- Discretionary Access Control (DAC): access policy defined by the user
- Mandatory Access Control (MAC): access policy defined by an administrator

Unix Access Control

Each user in Unix has a username and a user ID (UID). Users can also belong to one or more groups, each with a group ID (GID). Objects (such as files or directories) have an owner identified by a real UID and a real GID. Access permissions are set for the owner, group, and others (world), with read (r), write (w), and execute (x) permissions.

When processes interact with objects, their effective UID and effective GID are used to determine access permissions. The kernel compares the effective UID and GID with the permissions of the object to decide whether to grant or deny access.

The setuid and setgid bits are additional permission bits that can be set on executable files. When the setuid bit is set on an executable file, the program is executed with the effective UID of the file's owner. Similarly, when the setgid bit is set, the program is executed with the effective GID of the file's group owner. Programs with setuid and owner UID 0 (root) are potential targets for privilege escalation attacks because they may execute with elevated privileges, providing an opportunity for attackers to exploit vulnerabilities.

2.7 Data Validation

There are 3 facets to validation: type, length and syntax. Validation is also tied to integrity checking and business rule enforcing.

The first problem that arises is where to perform validation:

- 1^{st} principle: data validation has to be made whenever data crosses a trust boundary
- 2^{nd} principle: there has to be a small set of well defined chokepoints where validations are done

2.7.1 White and Black Listing

White listing refers to accepting known good and black listing to rejecting known bad. Of the two, only white listing should be used, since black listing violates principle of fail-safe defaults.

2.7.2 Sanitization

Sanitization consists of eliminating or encoding characters to make input safe.

Canonical Representation

Metacharacter evasion has to be solved by doing canonicalization before validation. For example, %64*elete* in canonical form becomes *delete*.

Web canonicalization issues are important due to many ways to encode the same character (ascii, UTF, URL encoding)

The principle is to do before validation the same decodings the application + interpreter might do after validation (in the same order). For example, if the input comes from URL and is used in UTF-8 in a web page:

- Decode URL encoding (i.e., hexadecimal escapes, e.g., %20) to UTF-8
- Canonicalize UTF-8 char
- Validate

2.7.3 Encoding

Encoding is an alternative/complement to validation. It consists of encoding characters for protection, with the objective of neutralizing dangerous characters (typically metacharacters).

Microsoft .NET Anti-XSS 1.5

Encoding Method	Should Be Used If ...	Example/Pattern
HtmlEncode	Untrusted input is used in HTML output	`<a href="http://www.example.com">` `Click Here [Untrusted input]</a>`
HtmlAttributeEncode	Untrusted input is used as an HTML attribute	`<hr noshade size=[Untrusted input]>`
JavaScriptEncode	Untrusted input is used within a JavaScript context	`<script type="text/javascript">` `...[Untrusted input]...</script>`
UrlEncode	Untrusted input is used in a URL	`<a href="http://example.com/results.aspx?q=[Untrusted-input]">Click Here!</a>`
VisualBasicScriptEncode	Untrusted input is used within a Visual Basic Script context	`<script type="text/vbscript" language="vbscript">` `…[Untrusted input]...</script>`
XmlEncode	Untrusted input is used in XML output	`<xml_tag>[Untrusted input]</xml_tag>`
XmlAttributeEncode	Untrusted input is used as an XML attribute	`<xml_tag attribute=[Untrusted input]>` `Some Text</xml_tag>`

2.8 Input Validation

Trust can be misplaced when there is interaction (i.e. input).

One example of input are environment variables. The libraries used by the program may not do enough sanity checking of the environment variables, such as PATH and IFS.

2.8.1 Metadata and Metacharacters

Data is often stored with metadata, which can be represented:

- In-band: as part of the data, e.g., strings in C (a special character is used to indicate the termination)
- Out-of-band: separated from data, e.g., strings in Java (the number of characters is metadata stored separately from the characters)

In-band metacharacters are a source of many vulnerabilities. They occur because the program trusts input to contain only characters, not metacharacters. The most typical attacks using metacharacters are:

- Embedded delimiters (\n)
- NUL character injection (\0)
- Separator injection (;)

2.8.2 Format String Vulnerabilities

Format strings are used in C, in functions of the families $printf$, err and $syslog$. They are vulnerable because the parameters passed are put in the stack before the function is called. If the format string is controlled by an attacker it might be used to crash the program, print arbitrary memory addresses or write arbitrary values in memory.

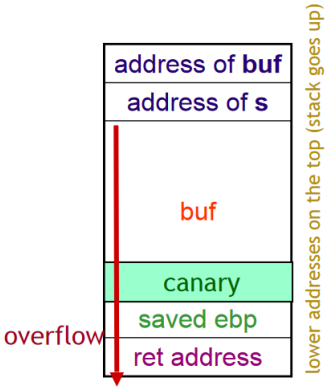
The solution is simple: always write the format string in the program.

2.9 Dynamic Protection

The idea is to block attacks that may exploit existing vulnerabilities, mostly the case of memory corruption attacks.

2.9.1 Canaries

Canaries may be used to detect some stack smashing attacks. They do not detect BO attacks that modify local variables, as those are above the canary, but detect (altough possibly too late) BO attacks againts the function arguments (these are below the ret address).



In order to allow canaries to protect local vars from overflows, a simple solution consists of reordering the stack layout, placing the local vars between the buffer and canary.

To allow the protection of function arguments, some compilers put the args in CPU registers, preventing this attack. An alternate solution consists of copying the args to the top of the stack in the beginning of a function.

2.9.2 Non-executable Stack and Heap

Many buffer overflow attacks involve injecting shell code in the stack/heap. A simple protection is to mark these memory pages as non-executable.

2.9.3 Randomization and Obfuscation

Address Space Layout Randomization (ASLR)

ASLR is efective against most BO attacks, but not those against local variables. The idea is to randomize the addresses where code and data are placed in runtime. What is randomized are not the physical addresses by shuffling pages around the RAM (which always happens anyway), but the logic/linear addresses, i.e., the organization of the virtual memory of a process. Altough this does not prevent exploitation, it makes it unreliable and harder. Only some elements can be randomized:

- Code: addresses where apps and dynamic libraries are loaded
- Stack: starting address of the stack of each thread
- Heap: base address of the heap

Instruction Set Randomization

Code injection would be almost impossible if each computer had its own random instruction set. With this approach, legitimate code is scrambled and malicious code is not, hopefully making it possible to execute.

A practical case would be SQL instruction randomization to avoid SQL injection.

Function Pointer Obfuscation

Long-lived function pointers are often the target of memory corruption exploits, as they provide a direct method for seizing control of program execution.

Pointer obfuscation mitigates this problem. The idea is to XOR the pointer with a random secret cookie, keeping it protected when not needed.

2.9.4 Integrity Verification

Windows Structured Exception Handling (SEH)

This feature is present in Windows operating systems. When an exception is generated, Windows examines a linked list of $EXCEPTION\_REGISTRATION$ structures in the stack. Such structures include pointers to the handlers, which can be overrun and an exception raised to force a jump to that address.

Array Bound Checking

BOs are caused by lack of array bound checking, so doing the check solves the problem. It is already done in languages like Java and supported in C compilers, such as *gcc*, with some restrictions.

gcc gives warning	gcc doesn't give warning

`char str[10];`
`int i;`
`for (i=0; i<10; i++) {`
`  str[i] = 'a';`
`}`
`str[10]='\0';`
	`char str[10];`
	`int i;`
	`for (i=0; i<10; i++) {`
	`  str[i] = 'a';`
	`}`
	`str[i]='\0';`

Detection Through Interception

$libsafe$ is a library with wrappers for problematic libc functions. When a function is called, the wrapper first checks if the buffer is not being overrun, using EBP as an upper bound.

Control-flow Integrity

Stack Shield has a global ret stack:

- Whenever a function is called, the return address is stored in a $GlobalRetStack$ (besides the normal stack)
- Before the function returns, check if the return address in the normal stack and the $GlobalRetStack$ match

2.9.5 Filtering

Web Application Firewalls

WAFs are application-level firewalls for webapps:

The interposition can be obtained in four ways:

- Bridge: WAF installed as a transparent bridge (switch)
- Router: network reconfigured to direct traffic through the WAF
- Reverse proxy: represents the web server for the clients
- Embedded: WAF is installed as a web server plug-in

The reaction to the detection of an attack can range from blocking the HTTP request, to ending the session and even blocking the user.