Outline

Overview

Language Based Security

Taint tracking

Information flow security by type-checking

Summary
Recap

We have looked at:

- examples of vulnerabilities and exploits
- particular programming failure patterns
- security engineering
- tools: static analysis for code review

In this lecture we examine some:

- language-based security principles

for (ensuring) secure programs.
Outline

Overview

Language Based Security

Taint tracking

Information flow security by type-checking

Summary
Security Properties

Remember the “CIA triple” of traditional properties for secure systems:

- **Confidentiality**
- **Integrity**
- **Availability**

(These are not the only security-relevant properties)

Confidentiality can be particularly tricky compared to I and A, to establish. *(Q. Why?)*

**Confidentiality**

Information is *confidential* if it cannot be learned by unauthorised principals.
Information leakage through the web
Single origin restriction

- Browser: **Single Origin Policy** (SOP): web page elements must come from same domain, or else block/warn user
- Too restrictive in practice: no mashups
- Doesn’t prevent *intentional/accidental* release
Generalised origin-based restrictions

- Web page loads script content from many places
- **Information from user/browser may leak**
- Motivates refinements of SOP in modern browsers (2014-):
  - CSP: Content Security Policy
  - CORS: Cross-origin Resource Sharing
- This gives a form of **Discretionary Access Control**

**Question.** Sometimes DAC is not good enough, why?
A different solution: separate confidential from non-confidential data

A programming approach might ensure confidentiality or integrity, by making sure that scripts do not mix data from different domains.
Language-based security approach

Idea: prevent application-level attacks inside the application.

Benefits:

- **Defence at application level** where meaningful app-level notions of user, APIs, services, etc are defined and connected to lower-level.

- **Semantics-based** security specification possible: rigorous and precise definition of what is required, based on definitions and data used inside program.

- **Static enforcement sometimes possible** if we admit a white box technique, we can examine the code, use programmer annotations and/or special type systems, drive run-time monitoring if needed.
Outline

Overview

Language Based Security

Taint tracking

Information flow security by type-checking

Summary
Dynamic taint tracking

Idea: add security labels to data inputs (sources) and data outputs (sinks). Propagate labels during computation (cf dynamic typing).

Labels are:

**Tainted**
- Data from *taint sources* (e.g., user input)
- Data arising from or influenced by tainted data

**Untainted**
- Data that is safe to output or use in sensitive ways
Stopping tainted data being stored

Legend:
- Untainted
- Tainted

Source \(\rightarrow\) \$db->query \(\rightarrow\) Sink

Sanitization

Security violation!
Preventing jumps to tainted addresses

<table>
<thead>
<tr>
<th>Line #</th>
<th>Statement</th>
<th>$\Delta$</th>
<th>$\tau_\Delta$</th>
<th>Rule</th>
<th>$pc$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>start</td>
<td>${}$</td>
<td>${}$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$x := 2*\text{get_input}()$</td>
<td>${x \rightarrow 40}$</td>
<td>${x \rightarrow \text{T}}$</td>
<td>T-ASSIGN</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$y := 5 + x$</td>
<td>${x \rightarrow 40, y \rightarrow 45}$</td>
<td>${x \rightarrow \text{T}, y \rightarrow \text{T}}$</td>
<td>T-ASSIGN</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>$\text{goto } y$</td>
<td>${x \rightarrow 40, y \rightarrow 45}$</td>
<td>${x \rightarrow \text{T}, y \rightarrow \text{T}}$</td>
<td>T-GOTO</td>
<td>error</td>
</tr>
</tbody>
</table>

See Schwartz, Avgerinos, Brumley, *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)*, IEEE Security and Privacy, 2010. This paper explains tainting with a simple operational semantics.
Taintdroid: notifying dynamic leaks on Android

Taintdroid uses a modification of the Android framework to track data flows at runtime. See the demo video.
Drawbacks of the dynamic method

Preventing code injection exploits using dynamic taint tracking is like letting a thief in your house and checking his bag for stolen goods at the very moment he tries to leave. It might work, but only if you never lose track of the gangster and if you really know your house. However, I would prefer a solution that does not let thieves in my house in the first place.

Analogy by Martin Johns used to explain dynamic taint tracking, 2007
Another drawback: implicit flows

- Simple dynamic tracking only captures *direct* flows
- To spot *implicit* flows, need to monitor every path
- Not only the ones actually taken by the program!
- Quickly impractical without severely pruning
  - special techniques like *forward symbolic execution*
- Partial solution: *type checking for information flow*
  - or *hybrid dynamic-static* methods
Outline

Overview

Language Based Security

Taint tracking

Information flow security by type-checking

Summary
Type-checking information flow

Idea: define a type system which tracks security levels of variables in the program, and adding levels to sources and sinks. Security levels may be:

**High**

- Sensitive information, e.g., personal details
- Any other data that
  - is computed directly from **high** data
  - occurs in a **high** context (high test in if)

**Low**

- Public information, e.g, obtained from user input

More generally, security labels may be taken from a multi-level security lattice. (Security lattices are a basic topic in access control, you may like to consult a textbook if you haven’t seen them before.)
The type system is designed to detect insecure information flows.

If a program can be type-checked, it will be secure on any execution, without the need to monitor dynamically.

Compare this with the idea of ordinary typing for data, to distinguish strings and numbers, etc. That provides the guarantee of memory safety: a well-typed program does not need to check types dynamically.

Theorem: Typability implies no insecure flows

If an output expression has type low, then it cannot be affected by any input of type high. Hence there can be no insecure information flows in the program.
Absence of flows

Intended security: low-level observations reveal nothing about high-level input:
Goguen and Meseguer expressed the property of non-interference for sequential programs.

For any two executions of the program which differ only in high inputs, the result of low outputs does not change.

More generally, we may use a notion of behavioural equivalence to relate values computed by the program. This allows for precise values to change, e.g., generating randomly different crypto keys on each run, and to express the restricted capability of an attacker to decrypt values.
Non-interference can be formalised using programming language semantics, as a definition like this:

\[ \forall m_1, m_2. \ m_1 =_L m_2 \Rightarrow [C]m_1 \approx_L [C]m_2 \]

**Semantic indistinguishability**

- **C is secure iff**
- Low-memory equality: \((h,l) =_L (h',l') \) iff \(l = l'\)
- C’s behavior: semantics \([C]\)
- Low view \(\approx_L\): indistinguishability by attacker
Type-checking information flow: examples

\([\text{low}] \vdash h := l + 4; \ l := l - 5\)

\([\text{pc}] \vdash \text{if } h \text{ then } h := h + 7 \text{ else skip}\)

\([\text{low}] \vdash \text{while } l < 34 \text{ do } l := l + 1\)

\([\text{pc}] \vdash \text{while } h < 4 \text{ do } l := l + 1\)
Type-checking: basic rules

Expressions:

\[
\text{exp} : \text{high} \quad \Rightarrow \quad \text{h} \notin \text{Vars(exp)} \quad \Rightarrow \quad \text{exp} : \text{low}
\]

Atomic commands (pc represents context):

\[
\boxed{[\text{pc}] \vdash \text{skip}}
\]

\[
\boxed{[\text{pc}] \vdash \text{h} := \text{exp}}
\]

\[
\boxed{[\text{low}] \vdash \text{l} := \text{exp}}
\]

**context**
Type-checking: compound rules

\[ \text{[high]} \vdash C \]
\[ \text{[low]} \vdash C \]
\[ \text{[pc]} \vdash C_1 \quad \text{[pc]} \vdash C_2 \]
\[ \text{[pc]} \vdash C_1 ; C_2 \]

**exp:** \text{pc} \quad \text{[pc]} \vdash C_1 \quad \text{[pc]} \vdash C_2

\[ \text{[pc]} \vdash \text{if exp then } C_1 \ \text{else } C_2 \]

**exp:** \text{pc} \quad \text{[pc]} \vdash C

\[ \text{[pc]} \vdash \text{while exp do } C \]

Implicit flows: branches of a high if must be typable in a high context.
Type-checking: example

\[\begin{align*}
5 : \text{low} & \quad 3 : \text{low} \\
\text{[high]} & \vdash h := h + 1 \\
\text{[low]} & \vdash l := 5, \quad \text{[low]} \vdash l := 3, \quad l = 0 : \text{low} \\
\text{[low]} & \vdash h := h + 1 \\
\text{[low]} & \vdash \text{if } l = 0 \text{ then } l := 5 \text{ else } l := 3 \\
\text{[low]} & \vdash h := h + 1; \text{ if } l = 0 \text{ then } l := 5 \text{ else } l := 3
\end{align*}\]
## Limits of simple type checking

<table>
<thead>
<tr>
<th>l:=h</th>
<th>insecure (direct)</th>
<th>untypable</th>
</tr>
</thead>
<tbody>
<tr>
<td>l:=h; l:=0</td>
<td>secure</td>
<td>untypable</td>
</tr>
<tr>
<td>h:=l; l:=h</td>
<td>secure</td>
<td>untypable</td>
</tr>
<tr>
<td>if h=0 then l:=0 else l:=1</td>
<td>insecure (indirect)</td>
<td>untypable</td>
</tr>
<tr>
<td>while h=0 do skip</td>
<td>secure (up to termination)</td>
<td>typable</td>
</tr>
<tr>
<td>if h=0 then sleep (1000)</td>
<td>secure (up to timing)</td>
<td>typable</td>
</tr>
</tbody>
</table>
Another limitation is the need to expose information carefully sometimes.

```java
if (!password.equals(inputString)) {
    System.out.println("Password wrong, please try again.");
}
```

A password check with a retry inevitably leaks 1-bit of information.

Solution: add special **declassification** points where the programmer realises that they must expose some (part of) confidential data, or output some information in a high context.
**Jif: Information Flow Checking for Java**

**Jif** extends Java by adding labels that express restrictions on how information may be used. We can give a security policy to a variable `x` with:

```java
int {Alice->Bob} x;
```

which says that information in `x` is controlled by Alice, and Alice permits the information to be seen by Bob.

The Jif compiler analyses information flows and checks whether confidentiality and integrity are ensured.

```java
int {Alice->Bob, Chuck} y;
```

```java
x = y; // OK: policy on x is stronger
```

```java
y = x; // BAD: policy on y is not as strong as x
```

Jif translates into plain Java, doing static type checking, but also allows dynamic enforcement for runtime labels.

For JavaScript, **JSFlow** is an approach using a security-enhanced interpreter with dynamic typing.
FlowDroid does *static* taint tracking for Android applications.

It includes sophisticated data flow tracking that understands pointer aliasing, as well as class and field references.

See FlowDroid web page for more information.
Outline

Overview

Language Based Security

Taint tracking

Information flow security by type-checking

Summary
Some of this lecture has been adapted from

- Information Flow lectures given by Andrei Sabelfeld at Chalmers University of Technology, Sweden.

Recommended reading: