Secure Programming Lecture 15
Information Leakage

David Aspinall

Informatics @ Edinburgh

Outline

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

Security Properties

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

- **C**onfidentiality
- **I**ntegrity
- **A**vailability

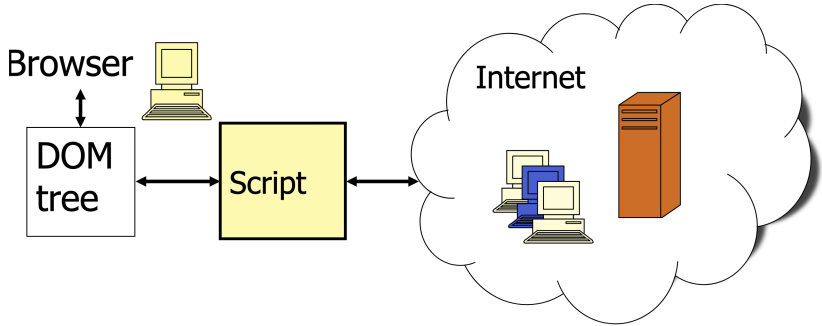
(these are not the only security-relevant properties)

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

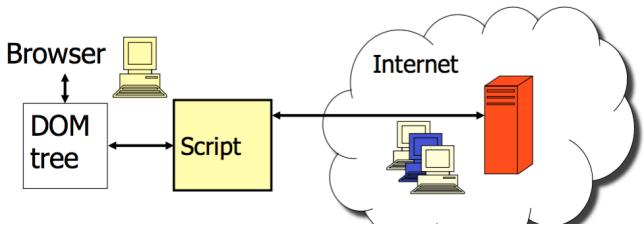
Confidentiality

Information is *confidential* if it cannot be learned by unauthorised principals.

Information leakage through the web

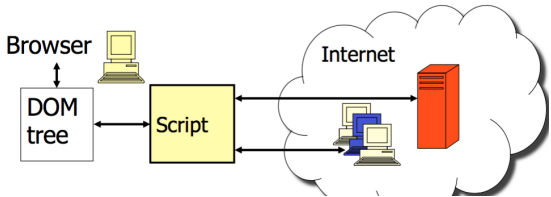


Single origin restriction



- 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

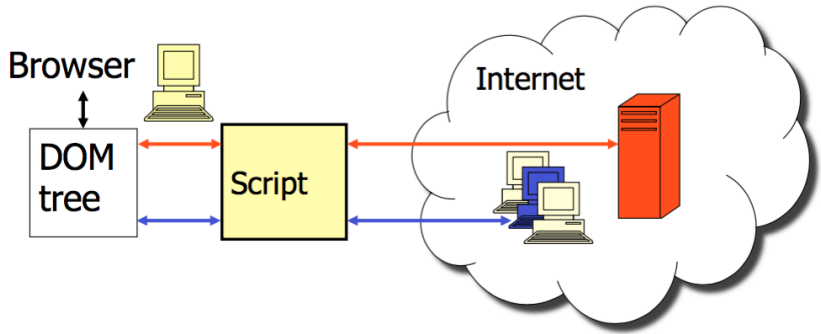
Generalised origin-based restrictions



- Web page loads script content from many places
- **Information from user/browser may leak**
- Motivates access control refinements in modern browsers (2014-):
  - CSP: Content Security Policy (restrict domains for loading)
  - CORS: Cross-origin Resource Sharing (restrict domains for access)
  - SameSite attribute cookies (restrict cookie 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

Dynamic taint tracking

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

Labels are:

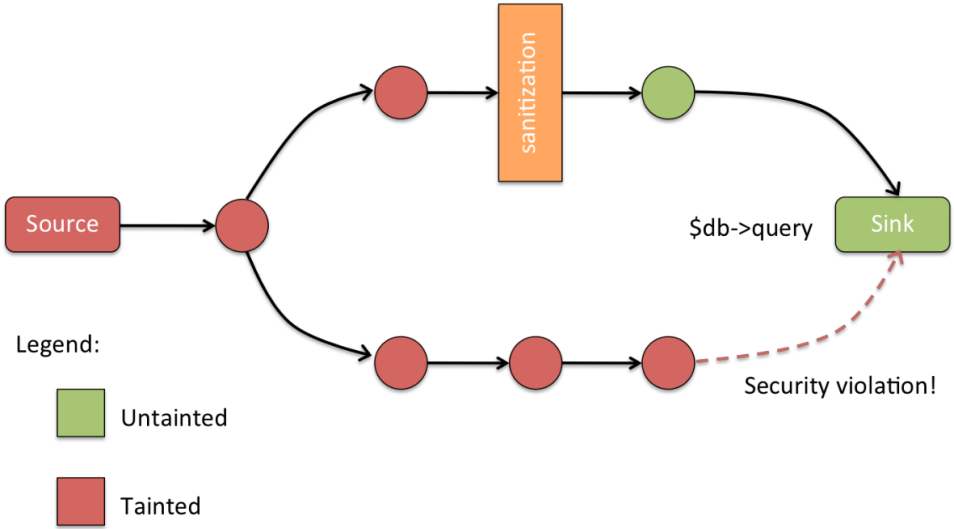
Tainted

- 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

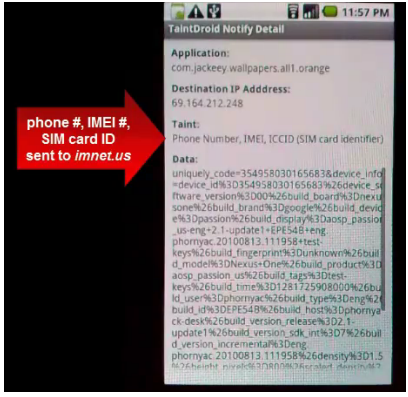


Preventing jumps to tainted addresses

Line #	Statement	Δ	τ_Δ	Rule	pc
	start	$\{\}$	$\{\}$		1
1	x := 2*get_input(·)	$\{x \to 40\}$	$\{x \to \mathbf{T}\}$	T-ASSIGN	2
2	y := 5 + x	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-ASSIGN	3
3	goto y	$\{x \to 40, y \to 45\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-GOTO	*error*

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



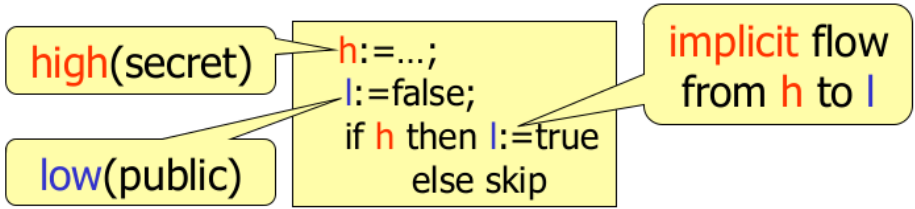
Taintdroid uses a modification of the Android framework to track data flows at runtime.
See the demo video.

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

Type checking rules

Recall that type checking rules are implemented by compilers and static checkers to explain how smaller pieces of the program and their types are combined to make larger programs and types.

They are written like logical inference rules.

Expressions:

$$\frac{\vdash exp_1 \ : \ \texttt{Int} \qquad \vdash exp_2 \ : \ \texttt{Int}}{\vdash exp_1 + exp_2 \ : \ \texttt{Int}}$$

$$\frac{\vdash exp \ : \ \texttt{Short}}{\vdash exp \ : \ \texttt{Int}}$$

Commands:

$$\frac{\vdash \texttt{x} \ : \ A* \qquad \vdash exp \ : \ A}{\vdash \texttt{x} := exp}$$

$$\frac{\vdash exp \ : \ \texttt{Bool} \qquad \vdash C}{\vdash \texttt{if}\ exp\ \texttt{then}\ C}$$

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.)

Static guarantee for security type system

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.

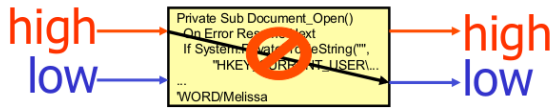
Compare this with the idea of ordinary typing for data, to distinguish strings and numbers, etc. That provides the guarantee of *memory* safety: a well-typed program does not need to check types dynamically.

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

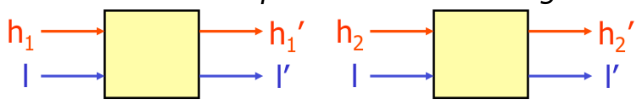
Intended security: low-level observations reveal nothing about high-level input:



Semantic property: non-interference

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.*

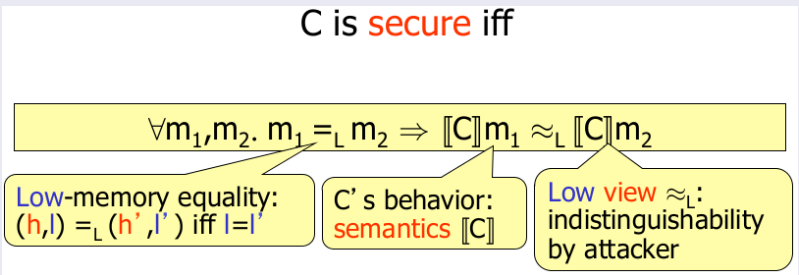


More generally, we may use a notion of *behavioural equivalence* to relate values computed by the program. This allows for precise values to change, e.g., generating randomly different crypto keys on each run, and to express the restricted capability of an attacker to decrypt values.

Formalisation of non-interference

Non-interference can be formalised using programming language semantics, as a definition like this:

Semantic indistinguishability



Type-checking information flow: examples

$[\mathsf{low}] \vdash \mathsf{h} := \mathsf{l}+4;\ \mathsf{l} := \mathsf{l}-5$

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

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

$[\mathsf{pc}] \not\vdash \text{while } \mathsf{h} < 4 \text{ do } \mathsf{l} := \mathsf{l}+1$

Type-checking: basic rules

Expressions:

$$\text{exp} : \text{high}$$

$$\frac{h \notin \text{Vars}(\text{exp})}{\text{exp} : \text{low}}$$

Atomic commands (pc represents context):

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

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

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

context

19

Type-checking: compound rules

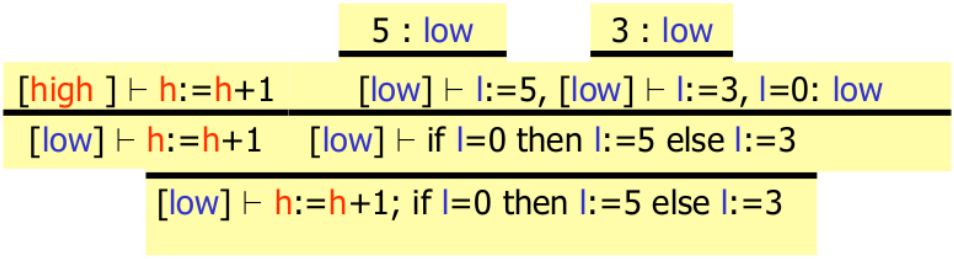
$$\frac{[\text{high}] \vdash C}{[\text{low}] \vdash C} \qquad \frac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1; C_2}$$

implicit flows: branches of a high if must be typable in a high context

$$\frac{exp{:}pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \text{if exp then } C_1 \text{ else } C_2}$$

$$\frac{exp{:}pc \quad [pc] \vdash C}{[pc] \vdash \text{while exp do } C}$$

Type-checking: example



Limits of simple type checking

l:=h	insecure (direct)	untypable
l:=h; l:=0	secure	untypable
h:=l; l:=h	secure	untypable
if h=0 then l:=0 else l:=1	insecure (indirect)	untypable
while h=0 do skip	secure (up to termination)	typable
if h=0 then sleep (1000)	secure (up to timing)	typable

Inevitable leaks: Declassification

An obvious limitation is the need to expose information sometimes.

```
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 (2002-16) extends Java with labels to express restrictions on information usage. A security policy for a variable x is:

```
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.

```
int {Alice->Bob, Chuck} y;
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is not as strong as x
```

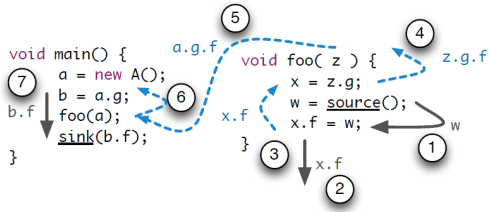
Jif translates into plain Java, doing static type checking, but also allows dynamic enforcement for runtime labels.

For JavaScript, JSFlow (2012-21) uses a security-enhanced interpreter with dynamic typing.

FlowDroid: static taint tracking on Android

FlowDroid demonstrated *static* taint tracking for Android applications.

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



See FlowDroid web page for more information.

Outline

Review Questions

Taint tracking

Give two drawbacks of relying on dynamic taint tracking to prevent information leakage.

Information flow typing

Explain the use of type-checking rules to to do static information flow checking, where types are **H** and **L** (high and low security).
What property does a correctly type-checked program have?

References and credits

Some of this lecture has been adapted from

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

Recommended reading:

- Sabelfeld and Myers, *Language-Based Information-Flow Security*, IEEE Journal on Selected Areas In Communications, **21**(1), 2003.

Further reading

Taint checking

- *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)*, IEEE S&P, 2010. A simple operational semantics for tainting.
- Watch the Taintdroid demo video

Information Flow

- Read about the Jif extension to Java, and the FlowDroid web page.
- Sabelfeld and Myers, *Language-Based Information-Flow Security*, IEEE Journal on Selected Areas In Communications, **21**(1), 2003. See the homepage of Andrei Sabelfeld for more papers on this topic.

Further reading: cross-site attacks

CSP and CORS allow servers and browsers to restrict *where* (what HTTP domain) scripts/content can be included from and *who* (what HTTP domain) can access which server resources. Both use security policies in headers. SameSite attributes are a simpler mechanism for cookies. See:

- The Mozilla Developer pages for CSP and CORS.
- OWASP: CSP Cheat Sheet, CSRF Cheat Sheet, Samesite info

Issues:

- Inaccuracies in policies, e.g. this 2016 study from Google
- Domain-level may be too coarse: Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web, USENIX 2021.
- Side-channel related leakage: OWASP XS Leaks Cheet Sheet