Secure Programming Lecture 15: Information Leakage

David Aspinall

11th March 2016

Recap

We have looked at:

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

In the last two lectures we examine some:

- **language-based security** principles

for (ensuring) secure programs.

Security Properties

Remember the "CIA" triple of traditional properties for secure systems:

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

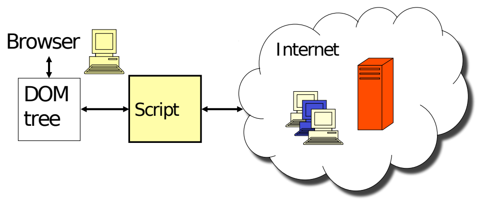
(these are not the only security-relevant properties)

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

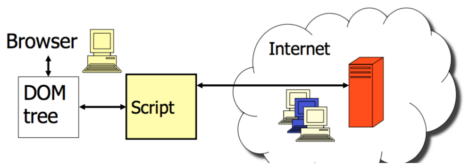
Confidentiality

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

Information leakage through the web

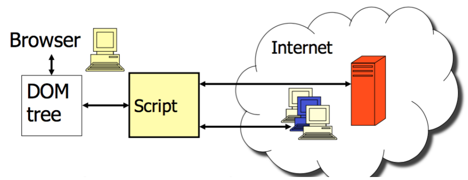


Origin-based restrictions



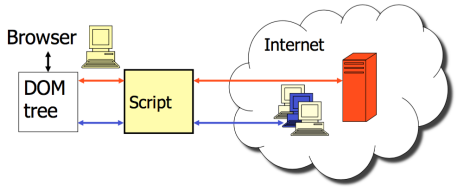
- Browser: **Single Origin Policy** (SOP): web page elements must come from same domain, or else block/warn user
- Restrictive in practice
- Doesn't prevent *intentional/accidental* release

Relaxing origin-based restrictions



- Web page loads script content from many places
- **Information from user/browser may leak** to other places
- Reason for SOP

Solution: separate confidential from non-confidential data



End-to-end security

General need:

- **end-to-end** confidentiality, integrity

which requires

- **protection at all levels**

To provide protection of *application level* concepts.

Problems of standard mechanisms (reminder)

Security in higher level applications *requires* lower-level mechanisms, but these aren't *sufficient*.

OS-level access control

- isolates users, files, processes
- but: what if one part of a process should be protected from parts of the same process?

Firewalls:

- stop some bad things entering programs
- but: massive leakage via port 80; web app firewalls are a fragile, losing game (**Q.** Why?)

Encryption

- secures a communication channel
- but not the endpoints, where data enters or leaves

Problems of standard mechanisms, continued

Antivirus scanning:

- Good with known malware, recognize by signature
- Little use on zero-day exploits

Code signing

- Digital signatures identify code producer/packager
- but don't actually guarantee code is secure

Sandboxing and OS-based monitoring

- Can block low-level accesses
- But not information transfer within applications
- Pure sandboxes too strict (witness rise of "sharing" in mobile applications)

Language-based security

Idea: prevent application-level attacks inside the application.

Benefits:

- **Semantics-based** security specification: 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.

Dynamic taint tracking

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

Labels are:

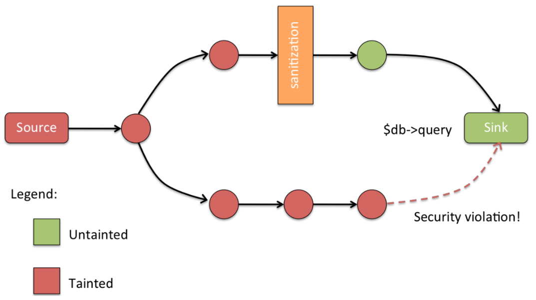
Tainted

- 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

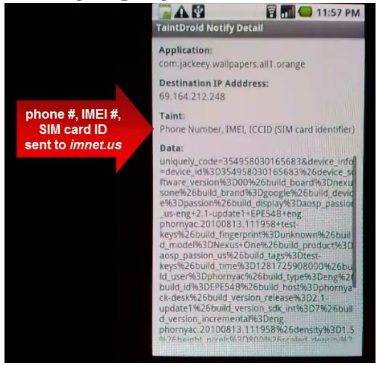


Preventing jumps to tainted addresses

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



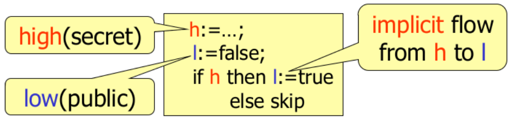
Taintdroid uses a modification of the Android framework to track data flows at runtime. See the demo video.

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*

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* as described in 3rd year *Computer Security*.

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.

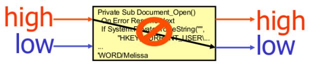
Compare this with the idea of ordinary typing for data, to distinguish strings and numbers, etc. That provides the guarantee of *memory* safety: a well-typed program does not need to check types dynamically.

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

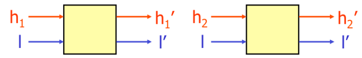
Intended security: low-level observations reveal nothing about high-level input:



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

C is secure iff

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

- 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

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

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

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

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

Type-checking: basic rules

Expressions: exp : high $\qquad \dfrac{h \notin Vars(exp)}{exp : low}$

Atomic commands (pc represents context):

$[pc] \vdash$ skip

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

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

context (pointing to [low])

Type-checking: compound rules

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

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

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

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

Type-checking: example

$$\frac{\dfrac{[high] \vdash h{:=}h{+}1}{[low] \vdash h{:=}h{+}1} \qquad \dfrac{[low] \vdash l{:=}5, \; [low] \vdash l{:=}3, \; l{=}0{:}\ low}{[low] \vdash \text{if } l{=}0 \text{ then } l{:=}5 \text{ else } l{:=}3}}{[low] \vdash h{:=}h{+}1; \text{ if } l{=}0 \text{ then } l{:=}5 \text{ else } l{:=}3}$$

where $\dfrac{}{5 : low}$ and $\dfrac{}{3 : low}$

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

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:

```
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.

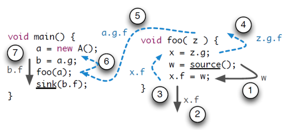
```
int {Alice->Bob, Chuck} y;
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is not as strong as x
```

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

FlowDroid: static taint tracking on Android

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.

References and credits

Most 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.
- Amusing academic publicity video made by Sabelfeld's group at Chalmers.