Introduction to Information Flow

Ana Matos
Software Security 2025/26, IST

November 11, 2025

Goals

- To become acquainted with basic concepts of **taint flow**, and understand it as a special case of information flow.
- To gain insight on capabilities and limitations of taint analysis tools, by considering the example of **Perl's Taint mode**.
- To become familiar with basic concepts for defining **information flow policies**, to understand the limitations of access control.
- To gain intuitions on how programs **encode information flows**.
- To develop **components of a tool** for detecting code injection vulnerabilities that **configure information flow policies**.

1 Taint Flow

1. Consider the following script, discussed in the theoretical class. Does it guarantee integrity of the query to the database? Why is this a problem?

```
$a = $_GET['user'];
$b = $_POST['pass'];
$c = "SELECT * FROM users WHERE u = '".mysql_real_escapes_string($a)."'";
$b = "wap";
$d = "SELECT * FROM users WHERE u = '".$b."'";
$r = mysql_query($c);
$r = mysql_query($d);
$b = $_POST['pass'];
$query = "SELECT * FROM users WHERE u = '".$a."' AND p = '".$b."'";
$r = mysql_query($query);
```

2. **(Analysis of) Perl's Taint Analysis**

   First time with Perl? This class is *not* about the Perl language, and you do not need to know the language in order to follow these exercises. We will focus on analysing its built-in taint mode tool, perhaps the most widely used information-flow analysis mechanism.

   > [Perl's] tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought. (Wall, Christiansen and Schwartz, 1996)

   Intro to Perl: `https://perldoc.perl.org/perlintro.html`

   More on Perl Security: `https://perldoc.perl.org/perlsec.html`

   More on Perl Regular Expressions: `https://perldoc.perl.org/perlrequick.html`

(a) Write a Hello world script in Perl and run it. Turn on the taint mode by using the `-T` flag in the hashbang line, and run it again.

What programs produce executions that are never altered by activating taint mode?

(b) Run the following script, and explain what happens:

```
#!/usr/bin/perl -T
print "Give name to new file:\n";   # prints message to screen
$filename=<STDIN>;                  # assigns input to $filename
open(FOO,"> $filename");            # opens file for output, handled by FOO
```

*Can you write a program that can **both** produce executions which **are** altered by taint mode, and others which are **not**?*

(c) What kinds of information flows does Perl's taint mode catch? Which ones does it not catch? Modify the above script so that it executes in taint mode, without using Perl's endorsement mechanism (regular expressions).

(d) Add the following regular expression check, to make sure that the inputted value contains nothing but "word" characters (alphabetics, numerics, and underscores), a hyphen, an at sign, or a dot before opening the file.

```
...                                     # $filneame tainted
if ($filename =~ /^([-\@\w.]+)$/) {     # match regular expression
  $filename = $1;                       # $filename here untainted
  ...
} else {
  print "Boo!";                         # $filename tainted
}
...                                     # $filename tainted
```

Run it again (making sure that you choose a fresh name for the file!), and explain what happens.

(e) *What is Perl's endorsement mechanism (regular expressions) intended to catch? How can it be misused?* Modify the above script so that it executes in taint mode, by using Perl's endorsement mechanism to bypass the purpose of taint mode.

2 Information flow policies

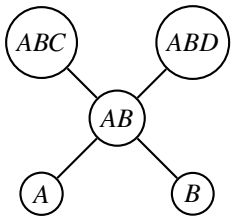
1. Define the information flow policy that is implicit in Perl's Taint mode.

2. The *principal-based integrity* policy results from considering security classes as sets of principals with write-access permissions.

   (a) Represent this information flow policy for a system with 2 principals using a Hasse diagram.

   (b) Define the above information flow policy by giving its set of classes *SC*, the can-flow relation $\rightarrow$, and the join operator $\oplus$.

3. Consider the following Hasse diagram:



(a) Define an information flow policy that is represented by the above Hasse diagram by giving its set of classes *SC*, the can-flow relation $\rightarrow\!\blacktriangleright$, and a join operator $\oplus$.

(b) The above diagram does not define $\oplus$ for all pairs of security classes. Using a Hasse diagram, represent an information flow policy that contains the above policy, and for which $\oplus$ is always defined.

4. Consider the following description of an access control policy:

- Alice allows Eve to read and write to her objects;
- Eve allows Bob to read and write to her objects;
- Owners of an object have all rights over it.

(a) Represent the access control matrix for a system that includes users Alice, Bob and Eve, and manages the rights 'r' (read) and 'w' (write) over files named Alice.txt, Bob.txt, Eve.txt (respectively owned by Alice, Bob and Eve).

(b) Consider a system that enforces the above access control policy. Are the following statements true? Why?

i. "Only Alice and Eve will ever know the contents of file Alice.txt."

ii. "Information contained in Alice.txt can only have originated from Alice or Eve"

(c) Suppose that a principal-based mandatory access control policy for confidentiality that prevents illegal information flows is imposed over the above policy. Define such an information flow policy regarding reading rights ('r') using sets of users as security classes, by giving:

- The set of security classes SC
- The "can-flow" relation $\rightarrow\!\blacktriangleright$.
- The binary class-combining operator $\oplus$.
- The Hasse diagram of the policy.

Note: It is enough to include the security classes that are used in the above matrix, as well as those needed for $\oplus$ to be well defined for all levels in *SC*.

(d) Consider an extension of the information flow policy defined as above in question 4c, where security classes are pairs of security classes for rights 'r' and 'w', i.e., for confidentiality and integrity. What should be the label of an object that contains information originating from Alice.txt and Eve.txt?

5. A lattice $(L, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ is a partially ordered structure for which there is a greatest lower bound operation $\sqcap$ and a least upper bound $\sqcup$ that are defined for all elements of L, a top element $\top$ and a bottom element $\bot$.

(a) Show that any lattice $(L, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ can be seen to represent an information flow policy.

(b) Given two lattices $(L_1, \sqsubseteq_1, \sqcap_1, \sqcup_1, \top_1, \bot_1)$ and $(L_2, \sqsubseteq_2, \sqcap_2, \sqcup_2, \top_2, \bot_2)$, then a new *product* lattice $(L_1 \times L_2, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ can be obtained by defining $\sqsubseteq$ as:

$$(l_1, l_2) \sqsubseteq (l'_1, l'_2) \textit{ iff } l_1 \sqsubseteq_1 l'_1 \textit{ and } l_2 \sqsubseteq_2 l'_2$$

As a result, two lattice-based information flow policies can be combined by defining their product. Define the product of the High-Low information flow policies for integrity and for confidentiality using a Hasse diagram.

3 Encoding Information Flow

1. Consider a *High (H) - Low (L)* security policy. For each of the following programs written in a standard imperative language, say whether they preserve confidentiality with respect to an attacker of level L. For each program, justify your answers by arguing why the program does not leak information, or give two inputs (initial values for variables) that would reveal the existance of an information leak.

   (a) $y_H := x_L; x_L := y_H + 1$

   (b) $x_L := 42; x_L := y_H$

   (c) $x_L := y_H; x_L := 42$

   (d) if $y_H = 0$ then $x_L := 42$ else $x_L := 42$

   (e) if $x_L = 0$ then $x_L := 42$ else $x_L := 4242$

   (f) $x_L := y_H$; if $x_L = 0$ then $x_L := 42$ else $x_L := 4242$

   (g) $x_L := 0$; while $y_H = 0$ do $x_L := 1$

   (h) $x_L := y_H$; while $x_L \neq 0$ do $x_L := x_L - 1$

   (i) $x_L := 0$; while $x_L \neq y_H$ do $x_L := x_L + 1$

4 Tool development - Patterns and Flow Labels

As we have seen, and you will study further in future classes, programs can encode security vulnerabilities in the form of potentially dangerous information flows, by enabling untrusted information inserted by an attacker via input functions (known as *sources*) to reach the arguments of sensitive functions or variables (so called *sinks*) and induce the program to perform security violations.

Suppose you wish to develop a tool for detecting such vulnerabilities, by tracking illegal flows from sources to sinks, and analysing whether they are processed by *sanitizers*, which are special functions that can neutralize the potential vulnerability, or validate that there is none. (This will be an exercise running through the following exrecise sheets.)

Such a tool will need to receive as input information about *vulnerability patterns* that should be detected. It will also need to work with an *information flow policy* that determines which *security labels* should be assigned to resources in a program, and how these labels can be combined and updated. To this end, implement the following components for the tool.

1. Develop a class Pattern, that represents a vulnerability pattern, including all its components. It should include at least the following basic operations:

   (a) Constructor of a Pattern object, receiving as input a vulnerability name, lists of possible source, sanitizer and sink names.

   (b) Selectors for each of its compomnents.

   (c) Tests for checking whether a given name is a source, sanitizer or sink for the pattern.

2. Develop a class Label, that represents the integrity of information that is carried by a resource. It should capture the sources that might have influenced a certain piece of information, and which sanitizers might have intercepted the information since its flow from each source. It should include at least the following basic operations:

   (a) Design a structure for the labels. Have in mind that a resource might be influenced by a same source in different ways – are your labels refined enough to distinguish the different cases?

   (b) Constructors of a Label object, and operations for adding sources to the label, and sanitizers that are intercepting the flows.

   (c) Selectors for each of its components.

(d) Combinor for returning a new label that represents the integrity of information that results from combining two pieces of information.

Note: Labels must be mutable, which means that the new labels should be independent from the original ones.

3. Develop a class MultiLabel that enables your tool to track multiple vulnerabilities at the same time. It should generalize the Label class in order to be able to represent distinct labels corresponding to different vulnerability patterns. Include the corresponding constructors, selectors and combinor. Have in mind that sources and sanitizers should only be added to labels corresponding to patterns for which that name is a source/sanitizer.