DD2460 Software Safety and Security: Part III
Exercises session 1: Taint + Type

Gurvan Le Guernic

DD2460 (III, E1)
February 21st, 2012

1 Python's taint library

Download:

- the Python's taint library, either from the course outline webpage or the library's webpage (`http://www.cse.chalmers.se/~russo/juanjo.htm`);
- the provided bash script (`testSecretLeakingOverManyNodes.sh`).

In the following, unless specified, you should not use the functions of the taint library to write your programs.

1.1 A first module

Write a module `lib` containing the following two functions:

- `get_secret()`: returns an integer between 0 and 9 included. If the environment variable "SECRET" exists, it fetches its value (using the `get` method of the dictionary `os.environ`) and converts it into an integer (using function `int`). Otherwise, it returns a random integer (between 0 and 9);
- `send_value(v)`: prints the value v prefixed with the string "`<sent>`" and suffixed with the string "`</sent>`" (`send_value(1)` prints the string "`<sent>1</sent>`")

1.2 Guessing a secret

1. In a module `app`, write and call a function `send_randomValue()` that sends a random integer between 0 and 9 using the function (`send_value(v)`) from the `lib` module you previously coded.
2. Test your `app` module using the provided bash script (`testSecretLeakingOverManyNodes.sh app.py`).
3. Comment the call to your function, but keep its definition.

1.3 Sending the secret

1. In your module `app`, write and call a function `send_secret()` that reads the secret value (`get_secret()`) and sends it (`send_value(v)`).
2. Test your `app` module using the provided bash script (`testSecretLeakingOverManyNodes.sh`).
3. Comment the call to your function, but keep its definition.

1.4 Protecting the secret

1. Secure your module `lib` by adding the necessary statements (can be done in 4 lines) such that:
   - the taint aware `INT` replaces default `int`,
   - any value returned by the `get` method of `environ` is tainted,
   - `send_value(v)` fails if v is tainted.

   What is the goal of those modifications? Why are each one needed? What do you expect to happen when performing the next step?
2. Retest the two previous functions (`send_randomValue()` and `send_secret()`) with the script (`testSecretLeakingOverManyNodes.sh`).

What happened? Why?

1.5 Evaluating the taint analysis

1. In your module `app`, using the functions of the taint library if needed, write and call a function `test_flows()` that tests which types of flows are handled by the taint analysis (i.e. the analysis is able to track tainted information flowing through those flows). For each type of flow, your function should return a string similar to "Handles ... flows." or "Does not handle ... flows."
2. Test your `app` module on the command line. Do not forget to set the environment variable SECRET for your test (for bash, `env SECRET=0 python app.py`). What happens if you do not set the variable SECRET (use `unset` if you already set it globally)? Why?
3. Comment the call to your function, but keep its definition.

1.6 Breaking the taint analysis

1. Using the information learned previously, in your module `app`, write and call a function `break_taint()` that successfully sends (`send_value()`) the secret data (`get_secret()`).
2. Test your `app` module using the provided bash script (`testSecretLeakingOverManyNodes.sh`).
3. Comment the call to your function, but keep its definition.

1.7 Handling explicit indirect flows

1. Design (do not code) a program transformation $\mathbb{T}(S)$ (on the Python's equivalent of the simple while language, i.e. handling assignment, sequence, if and while) such that explicit indirect flows in the original program are handled by the taint analysis in the transformed program.
   Remember that booleans are not handled by this library. However, assume the existence of a function $\mathcal{V}^{op}(e)$, where *op* is a commutative infix operator, that returns the application of *op* to all the variables in e. For example, $\mathcal{V}^{+}(x/y+z) \rightarrow x+y+z$.
   A transformation subtracting the sum of all the variables in the RHS of any assignment would be:

$$\mathbb{T}(x := e) \rightarrow x := e - (\mathcal{V}^{+}(e))$$
$$\mathbb{T}(S_1;\ S_2) \rightarrow \mathbb{T}(S_1);\ \mathbb{T}(S_2)$$
$$\mathbb{T}(\texttt{if}(e)\ S_1\ \texttt{else}\ S_2) \rightarrow \texttt{if}(e)\ \mathbb{T}(S_1)\ \texttt{else}\ \mathbb{T}(S_2)$$
$$\mathbb{T}(\texttt{while}(e)\ S) \rightarrow \texttt{while}(e)\ \mathbb{T}(S)$$

2. Apply manually that transformation into copies of functions `test_flows()` and `break_taint()` (`test_flows_withEIF()` and `break_taint_withEIF()`); and retest them.

1.8 Breaking the taint analysis with explicit indirect flows

1. Using the information learned previously, write and call a function `break_taint2_withEIF()`, on which you apply your transformation for explicit indirect flows, that still successfully sends (`send_value()`) the secret data (`get_secret()`).
2. Test your `app` module using the provided bash script (`testSecretLeakingOverManyNodes.sh`).
3. Test your `app` module using the kill correction mechanism of the provided bash script:
   `testSecretLeakingOverManyNodes.sh app k`
   What happens? Why?
4. Comment the call to your function, but keep its definition.

1.9 The correction pitfall and 1 bit legend

In the examples above, the "kill" correction mechanism is more efficient than the "drop bad outputs" mechanism.

1. Write a function (`guess_secret_withEIF()`), on which you apply the transformation to handle explicit indirect flows, equivalent to the program I talked about during the first lecture concerning the difference between an average leak of 1 bit and the real amount of beats liked for a given run. The program guesses the secret and then use the "kill" correction mechanism to validate the guess to the party listening on the other side of the network (i.e. the party receiving sent messages).
2. Test your function using the 2 correction modes.
   What happens? Why?
3. Comment the call to your function, but keep its definition.

1.10 Handling implicit indirect flows

1. Design a program transformation $\mathbb{T}(S)$ such that implicit indirect flows in the original program are handled by the taint analysis in the transformed program. AsXSsume the existence of a function $\mathcal{A}^{S_2}(S_1)$ which, for every variable y assigned into S_1 returns the statements S_2 where variable `x` is replaced by variable y. For example, $\mathcal{A}^{x:=0}(x := z; y := x * 3) \rightarrow x := 0; y := 0$.

2. Apply manually that transformation into copies of functions `test_flows()` and `break_taint2()` (`test_flows_withIIF()` and `break_taint2_withIIF()`); and retest them.

With this transformation, the noninterference analysis seems unbreakable. However, the taint analysis of the library does not handle cases where you call a sensitive function from a branch of a secret test (**if** secret == 1: send_value(1)). This weakness obviously allows to leak the secret. Many dynamic information flow analyses handle this case by killing processes trying to generate an observable event (send_value(1)) while inside a branch of a secret test. Would it then be impossible to leak the full secret at least sometimes like in the previous exercise? If yes, write the function that would sometimes leak the full secret.

1.11 A final word

The aim of the Pyhton's taint analysis is not to track all flows. Its aim is to help developers write safe programs by tracking direct flows. The library is efficient for that. We have seen that it is possible to combine a taint tracking mechanism with other mechanisms to track more flows. However, the resulting security mechanisms are never bullet proof even if more secure. It is important to realize what is the real level of security provided by a security mechanism (some security guarantees is good, and even better when we exactly know what they are). Many of the resulting security mechanism shown in this exercises session can be proved to be sound with regard to noninterference by carefully stating the soundness statement (mainly discarding executions that terminated abnormally). Do not be lured by soundness proofs. A soundness proof is good and better than nothing. However do not assume it means that the security mechanism protects against all possible types of attacks.

2 Noninterference type systems challenge

Connect to `http://www.vinosv.dk/ifc/` and find the solution to all the challenges.
Advice: backup the (login, password) pairs.

2.1 Challenge 1

No type system.

2.2 Challenge 2

Expressions: $e := 0 \mid 1 \mid x$ (1 is True for conditions).
Type system:

$$\frac{x \in \{h1, h2, h3, h4, h5\}}{\Gamma \vdash x := e \ : \ \texttt{cmd}} \qquad \frac{\Gamma \vdash e \ : \ \texttt{L}}{\Gamma \vdash x := e \ : \ \texttt{cmd}} \qquad \Gamma \vdash \mathsf{skip} \ : \ \texttt{cmd}$$

$$\frac{\Gamma \vdash S_1 \ : \ \texttt{cmd} \qquad \Gamma \vdash S_2 \ : \ \texttt{cmd}}{\Gamma \vdash S_1; S_2 \ : \ \texttt{cmd}} \qquad \frac{\Gamma \vdash S_1 \ : \ \texttt{cmd} \qquad \Gamma \vdash S_2 \ : \ \texttt{cmd}}{\Gamma \vdash \textbf{if} \ (e) \ \{ S_1 \} \ \textbf{else} \ \{ S_2 \} \ : \ \texttt{cmd}}$$

$$\frac{\Gamma \vdash S \ : \ \texttt{cmd}}{\Gamma \vdash \textbf{while} \ (e) \ \{ S \} \ : \ \texttt{cmd}}$$

2.3 Challenge 3

Expressions: $e := 0 \mid 1 \mid x$ (1 is True for conditions).
Type system:

$$\frac{x \in \{hi \mid i \in [1,5]\}}{\Gamma \vdash x := e \ : \ l} \qquad \frac{\Gamma \vdash e \ : \ \texttt{L}}{\Gamma \vdash x := e \ : \ \texttt{L}} \qquad \Gamma \vdash \mathsf{skip} \ : \ l \qquad \frac{\Gamma \vdash S_1 \ : \ l \qquad \Gamma \vdash S_2 \ : \ l}{\Gamma \vdash S_1; S_2 \ : \ l}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{L} \qquad \Gamma \vdash S_1 \ : \ l \qquad \Gamma \vdash S_2 \ : \ l}{\Gamma \vdash \textbf{if} \ (e) \ \{ S_1 \} \ \textbf{else} \ \{ S_2 \} \ : \ l} \qquad \frac{\Gamma \vdash S_1 \ : \ \texttt{H} \qquad \Gamma \vdash S_2 \ : \ \texttt{H}}{\Gamma \vdash \textbf{if} \ (e) \ \{ S_1 \} \ \textbf{else} \ \{ S_2 \} \ : \ l}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{L} \qquad \Gamma \vdash S \ : \ l}{\Gamma \vdash \textbf{while} \ (e) \ \{ S \} \ : \ l} \qquad \frac{\Gamma \vdash S \ : \ \texttt{H}}{\Gamma \vdash \textbf{while} \ (e) \ \{ S \} \ : \ l}$$

2.4 Challenge 4

Expressions: $e := 0 \mid 1 \mid x$ (1 is True for conditions).
Type system:

$$\frac{x \in \{hi \mid i \in [1,5]\}}{\Gamma \vdash x := e \ : \ l} \qquad \frac{\Gamma \vdash e \ : \ \texttt{L}}{\Gamma \vdash x := e \ : \ \texttt{L}} \qquad \Gamma \vdash \mathsf{skip} \ : \ l \qquad \frac{\Gamma \vdash S_1 \ : \ l \qquad \Gamma \vdash S_2 \ : \ l}{\Gamma \vdash S_1; S_2 \ : \ l}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{L} \qquad \Gamma \vdash S_1 \ : \ l \qquad \Gamma \vdash S_2 \ : \ l}{\Gamma \vdash \textbf{if} \ (e) \ \{ S_1 \} \ \textbf{else} \ \{ S_2 \} \ : \ l} \qquad \frac{\Gamma \vdash S_1 \ : \ \texttt{H} \qquad \Gamma \vdash S_2 \ : \ \texttt{H}}{\Gamma \vdash \textbf{if} \ (e) \ \{ S_1 \} \ \textbf{else} \ \{ S_2 \} \ : \ l}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{L} \qquad \Gamma \vdash S \ : \ l}{\Gamma \vdash \textbf{while} \ (e) \ \{ S \} \ : \ \texttt{L}}$$