Dynamic taint analysis exercises

In this report, the following "fake" functions will be used in order to simulate DTA:

- makeTainted(var) -> makes variable tainted
- makeUntainted(var) -> makes variable untainted
- makeCondTainted(var, conditions) -> makes var untainted if and only if one of the vars in condition are tainted

All remaining commands are taken for clarity from the C language (if, loops, and exit calls).

We assume a variable is not tainted unless we explicitly run the command makeTainted (so, for example, initialized variables with just integers are considered untainted).

Additionally, all code is to be found within the material of the homework: pasting it would have rendered it messy and unreadable.

Exercise 1 – Integer Overflow

In this exercise, we add makeTainted calls for item_choice (line 14) and item_quantity (line 19).

As with the static analysis, the if branch at line 21 could apparently untaint item_quantity by terminating instances with negative values, but this is not the case as also high values can be considered tainted in this case, as they can overflow later variables.

Subsequently, at line 33 we set price tainted conditionally on item quantity, as it can cause overflows.

At lines 38 and 68 we terminate the program for tainted instances of price. It must noted that, at line 60, we can make item_quantity theoretically untainted, as the previous if conditions make it fall necessarily between 1 and 3. This, however, can be implemented in a conservative way, and so it was commented in the code.

Finally, regarding the flow code, it must be noted that in line 41 we still assume that price is tainted since the previous condition only checked for equality and not less or equal, making it still tainted even if the previous branch is not taken. Either way, the sink was placed just before the if statement, which renders it useless.

For the execution, several simulations were made for the two user-inserted variables:

	Sim. 1	Sim. 2	Sim. 3	Sim. 4
Item_choice	2	3	2	3
Item_quantity	1	1	214748364	214748364

Sim. 1 and Sim. 2 are straight-forward and the code safely reaches the print statements without harming the execution flows. Similarly, Sim. 3 gets interrupted by the item_quantity > 3 statement in the else and therefore the code is interrupted. Sim 4 is where it gets interesting. The sink in line 36 gets triggered, due to price reaching 0 (with 214748364 being the integer making price overflow to 0).

Exercise 2 – SQL injection

In this exercise, due to the absence of if conditions, there are not many differences from the static analysis. As usual, retrieve_users strings are immediately tainted from user_id and password variables, and checks

are made to verify if the actual retrieve_user string is tainted. Additional XSS checks are made in the for loop, in order to assess whether tainted data was already present in the database.

Several simulations were made, with various combinations of user_id, password, and extracted entries from the database (which are always considered as tainted). Each time, the execution was interrupted at the very first sink (in the tracking code, be it the sink at line 24 for the SQL injection or the sink at line 45 for the XSS check in the information retrieval) – of course, as long as the items actually contained malevolent data.

Final considerations

The difference between the DTA made in this exercise and the STA in the previous analysis, from a practical standpoint, is negligible as the vulnerabilities are very clear and can be spotted easily just by reading the code. Nevertheless, adopting different policies can sometimes yield different results. Moreover, usually the dynamic analysis would sometimes generate false negatives and letting some code run, while the static analysis would straight out prevent its compilation.