VINCI NICOLO' 220229

DYNAMIC TAINT ANALYSIS

Exercise 1: SQL Injection

```
import sys
import os
import sqlite3
conn = None
try:
   conn = sqlite3.connect('users.db')
except Exception:
   print("Can't connect to the database")
   sys.exit(-1)
print("Welcome to this vulnerable database reader")
print("You have to login first")
print("Insert your user-id")
user_id = input() # Source1 (1)
# makeTainted("user_id")
print("Insert your password")
password = input() # Source2 (2)
# makeTainted("password")
retrieve_user = "SELECT * FROM credentials WHERE user_id = '" + user_id + "' and password = '" + password
+ "';" (3)
# makeCondTainted("retrieve_user", array("user_id", "password"))
# if (isTainted("retrieve_user") && isSQLInjectionAttack(retrieve_user)):
#      sys.exit("Security violation")
cursor = conn.execute(retrieve_user) # Sink1 (4)
# makeCondTainted("cursor", array("retrieve_user"))
entries = cursor.fetchall() (5)
# makeCondTainted("entries", array("cursor"))
if len(entries) > 0:  (6)
   print("\n===Logged-in=====")
   retrieve_user = "SELECT * FROM accounts WHERE user_id = '" + user_id + "';" (7)
```

```
    # makeCondTainted("retrieve_user", array("user_id", “len(entries)”))
    # if (isTainted("retrieve_user") && isSQLInjectionAttack(retrieve_user)):
    #       sys.exit(“Security violation”)
    cursor = conn.execute(retrieve_user) # Sink2 (8)
    # makeCondTainted("cursor", array("retrieve_user", “len(entries)”))
    entries = cursor.fetchall() (9)
    # makeCondTainted("entries", array("cursor", “len(entries)”))
    for entry in entries: (10)
        # makeCondTainted("entry", array("entries", “len(entries)”))
        user_id, first_name, last_name, phone = entry (11)
        # makeCondTainted("user_id", array("entry", “len(entries)”))
        # makeCondTainted("first_name", array("entry", “len(entries)”))
        # makeCondTainted("last_name”, array("entry", “len(entries)”))
        # makeCondTainted("phone", array("entry", “len(entries)”))
        print()
        # if (isTainted("user_id") && isXSSAttack(user_id)):
        #       sys.exit(“Security violation”)
        print("Here is {} data:".format(user_id)) # Sink3 (12)
        # if (isTainted("user_id") && isXSSAttack(user_id)):
        #       sys.exit(“Security violation”)
        print("user-id=", user_id) # Sink4 (13)
        # if (isTainted("first_name") && isXSSAttack(first_name)):
        #       sys.exit(“Security violation”)
        print("first_name=", first_name) # Sink5 (14)
        # if (isTainted("last_name") && isXSSAttack(last_name)):
        #       sys.exit(“Security violation”)
        print("last_name=", last_name) # Sink6 (15)
        # if (isTainted("phone") && isXSSAttack(phone)):
        #       sys.exit(“Security violation”)
        print("phone", phone) # Sink7 (16)
else:
    # makeUntainted("entries")
```

```
print("Wrong credentials")
```

Input trials:

user_id = 1, password = passwd :

1) user_id (Source1) = 1 → T

2) password(Source2) = passwd → T

3) retrieve_user → user_id, password → T

4) cursor (Sink1) → retrieve_user → T → LEAK

5) entries → cursor → T

6) len(entries)>0 → True

7) retrieve_user → user_id, len(entries) → T

8) cursor (Sink2) → retrieve_user, len(entries), → T → LEAK

9) entries → cursor, len(entries), → T

10) entry → entries, len(entries), → T

11) user_id, first_name, last_name, phone → entry, len(entries), → T

12) user_id (Sink3) → T → LEAK

13) user_id (Sink4) → T → LEAK

14) first_name(Sink5) → T → LEAK

15) last_name (Sink6) →T → LEAK

16) phone (Sink7) → T → LEAK #end of program execution

Assumption: the variables depend on the value “len(entries)” implicitly inside the branches generated by the if statement at line 6.

There are not exit at run time, so the inputs are not tainted. However, all variable are tainted due to lack of sanitization, so you may execute a SQL injection attack on queries at line 4 and 8 or a XSS attack at line 12 and 13. It does not need to consider the else statement, because there is not variable that depends on “len(entries)”, but only a print statement.

user_id = 1, password = passwd’ AND ‘1’=’1’;-- :

1) user_id (Source1) = admin → T

2) password(Source2) = passwd’ AND ‘1’=’1’;-- → T

3) retrieve_user → [user_id, password] → T #end of program execution

After that, the dynamic taint analysis will end the program, because the “if (isTainted("retrieve_user") && isSQLInjectionAttack(retrieve_user))” will be true. This means that it has been found an attempted SQL injection attack on the Sink 1.

Exercise 2: integer overflow

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        printf("Hello, which product do you want to buy?\n");
        printf("1) IPhone 12\n");
        printf("2) IPhone 12 Pro\n");
        printf("3) IPhone 12 Pro Max Max\n");
        int item_choice;
        scanf("%d", &item_choice); // Source1 (1)
        // makeTainted("item_choice");
        printf("Great device, how many?\n");
        int item_quantity;
        scanf("%d", &item_quantity); // Source2  (2)
        // makeTainted("item_quantity");
        if (item_quantity <= 0) { (3)
                (3T)
                printf("You should buy at least one Iphone!\n");
                return -1;
        }
        (3F)
        int insurance = 1200;
        if (item_choice == 3) (4)
        {
                (4T)
                // makeUntainted("item_choice");
                // if(isTainted("item_quantity") && isIntegerOverflowAttack(item_quantity) {
                //                  return -1;
                // }
                int price = 1500*item_quantity + insurance; // Sink1 (5)
                // makeCondTainted(“price”, array(“item_quantity”, “item_choice”));
```

```
        if (price == 0) { (6)
                (6T)
                printf("You solved the problem\n");
                printf("The Iphone Max Max is yours\n");
                return 1;
        }
        (6F)
        printf("You have to pay €%d\n", price); (7)
}
else
{
        (4F)
        if (item_quantity > 3) { (8)
                (8T)
                printf("You can buy maximum 3\n");
                return -1;
        }
        (8F)
        // makeUntainted("item_quantity");
        // if(isTainted("item_quantity") && isIntegerOverflowAttack(item_quantity) {
        //                  return -1;
        // }
        int price = 1000*item_quantity; (9)
        // makeCondTainted(“price”, array(“item_quantity”, “item_choice”));
        printf("You have to pay €%d\n", price); (10)
}
return 0;
}
```

Input trials:

item_choice = 3, item_quantity = 2 :

1) item_choice (Source1) → 3 → T

2) item_quantity (Source2) → 2 → T

3) item_quantity < 3 → False

3T)

3F)

4) item_choice == 3 → True

4T) item_choice → Untainted

5) price (Sink1)→ item_quantity, item_choice → T → LEAK

6) price == 0 → False

6T)

6F)

7) price → T // end of program execution

4F) item_choice → V → T // considering item_choice != 3

8) item_quantity > 3 → False

8T)

8F) item_quantity → Untainted

9) price (Sink2) →item_quantity , item_choice → T → LEAK

10) price → T

Assumption: “price” depends on the value of “item_choice” implicitly.

Considering that input values, the dynamic taint analysis ends at line 7. Observing the if statement at line 3 and 6 , there are not other branch to analyze, because there are only printf statements in the branches not taken.

On the other side, there is a flow that is not analyzed considering the if statement at line 4. It is an implicit flow that depends on “item_choice”. So, in the other flow there is a sanitization of “item_quantity” due to the checking at line 8 , but “price” is still tainted due to “item_choice” that is tainted. Hence, the DTA will report a false positive, because “price” is computed only with “item_quantity” that is untainted.

Anyway, at line 5 the code executes an integers computation that may be exploited by an integer overflow attack (Sink 1).

item_choice = 3, item_quantity = 2000000 :

1) item_choice (Source1) → 3 → T

2) item_quantity (Source2) → 2000000 → T

3) item_quantity < 3 → False

3T)

3F)

4) item_choice == 3 → True

4T) item_choice → Untainted

After that, the if statement “if(isTainted("item_quantity") && isIntegerOverflowAttack(item_quantity)” will be true due to “item_quantity” that it is too large, hence the program ends. This means that it has been found an attempted integer overflow attack on the Sink 1.

item_choice = 3, item_quantity = -100 :

1) item_choice (Source1) → 3 → T

2) item_quantity (Source2) → -100 → T

3) item_quantity < 3 → True

3T) item_quantity → T → LEAK //end of program execution

3F) item_quantity → V → T

4) item_choice == 3 → True

4T) item_choice → Untainted

5) price (Sink1) → item_quantity, item_choice, → T → LEAK

6) price == 0 → False

6T)

6F)

7) price → T

4F) item_choice → V → T //considering item_choice != 3

8) item_quantity > 3 → False

8T)

8F) item_quantity → Untainted

9) price (Sink2) → item_quantity, item_choice → T → LEAK

10) price → T

Assumption: “price” depends on the value of “item_choice” implicitly.

Even here, the DTA will report a false positive at line 9 due to the implicit dependency on the tainted variable “item_choice”.

Considerations:

The first difference between dynamic and taint analysis on those programs is that the static analysis was valid for every input values, instead the dynamic analysis considers only the actual input at run time. So, if I put some tainted values as input, the DTA will recognize it and stop the execution of the program, without analyzing the rest. Hence, the DTA should be tested for all possible input values.

Furthermore, the DTA can not analyze all possible paths, because it will follow the flow at run time and some branches may not be executed. However, it is possible to mitigate this issue with Branch-Not-Taken analysis. On the other side, the STA regards all paths of the code.

Eventually, I noticed that the DTA may have some false positives due to implicit flows.