Dynamic Taint Analysis

Security Testing (145322)

Francisco Manuel Colmenar Lamas
219757

Trento, December 2020

Contents

List of Figures

1. DYNAMIC TAINT ANALYSIS

1.1. Integer Overflow

In this first section the Integer Overflow file is going to be the one assessed.

The first example in which the Dynamic Taint Analysis is performed is the one in which the IoF vulnerability is exploited and the price is equal to 0.

```
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");

    // Get item
    int item_choice;
    scanf("%d", &item_choice);                                    item_choice > 3 > T
    makeTainted("item_choice");

    printf("Great device, how many?\n");
    int item_quantity;
    scanf("%d", &item_quantity);                                  item_quantity > 1288490188 > T
    makeTainted("item_quantity");

    if (item_quantity <= 0) {
        printf("You should buy at least one Iphone!\n");
        return -1;
    }

    int price = 1000 * item_quantity;
    makeCondTainted("item_quantity", array("item_quantity"));     price > 1288490188 * 1000 > T

    int insurance = 1200;
```

Figure 1.1: Example 1 belonging to the IoF code (1).

```
int insurance = 1200;
if (item_choice == 3) {
    int price = 1500 * item_quantity + insurance;
    makeCondTainted("price", array("item_quantity"));

    if(isTainted("price") && isIoF("price")){
        printf("IoF violation\n");
        exit(-1);
    }
    if (price == 0) {
        printf("You solved the problem\n");
        printf("The Iphone Max Max is yours\n");
        return 1;
    }

    if(isTainted("price")){
        printf("Printing tainted variable\n");
        exit(-1);
    }
    printf("You have to pay €%d\n", price);
} else {
    if (item_quantity > 3) {
        printf("You can buy maximum 3\n");
        return -1;
    }

    if(isTainted("price")){
        printf("Printing tainted variable\n");
        exit(-1);
    }
    printf("You have to pay €%d\n", price);
}
return 0;
```

price > 1500 * 1288490188 + 1200 = 0 > T

price > T && IoF > IoF Leak

Figure 1.2: Example 1 belonging to the IoF code (2).

The second example in which the Dynamic Taint Analysis is performed is the one in which the the item choice is 3, so the same branch as the IoF vulnerability is taken but the price is not equal to 0. Therefore, the leak occurs in this case when the variable price is printed.

```c
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");

    // Get item
    int item_choice;
    scanf("%d", &item_choice);                                    item_choice > 3 > T
    makeTainted("item_choice");

    printf("Great device, how many?\n");
    int item_quantity;
    scanf("%d", &item_quantity);                                  item_quantity > 10 > T
    makeTainted("item_quantity");

    if (item_quantity <= 0) {
        printf("You should buy at least one Iphone!\n");
        return -1;
    }

    int price = 1000 * item_quantity;
    makeCondTainted("item_quantity", array("item_quantity"));     price > 10 * 1000 = 10000 > T

    int insurance = 1200;
```

Figure 1.3: Example 2 belonging to the IoF code (1).

```c
    int insurance = 1200;
    if (item_choice == 3) {
        int price = 1500 * item_quantity + insurance;             price > 10 * 1500  + 1500 = 16500 > T
        makeCondTainted("price", array("item_quantity"));

        if(isTainted("price") && isIoF("price")){
            printf("IoF violation\n");
            exit(-1);
        }
        if (price == 0) {
            printf("You solved the problem\n");
            printf("The Iphone Max Max is yours\n");
            return 1;
        }

        if(isTainted("price")){                                   price >  T  > Leak
            printf("Printing tainted variable\n");
            exit(-1);
        }
        printf("You have to pay €%d\n", price);
    } else {
        if (item_quantity > 3) {
            printf("You can buy maximum 3\n");
            return -1;
        }

        if(isTainted("price")){
            printf("Printing tainted variable\n");
            exit(-1);
        }
        printf("You have to pay €%d\n", price);
    }
    return 0;
```

Figure 1.4: Example 2 belonging to the IoF code (2).

1.2. SQLi

Regarding the SQLi code, the inputs which are used for the first example are "Fran" for the username, which is just a normal value, and for the password ';DROP * FROM credentials;–, which it is an SQLi exploit input.

```
# Connect to database
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()                                                    user_id > Fran > T
makeTainted(user_id)


print("Insert your password")
password = input()
makeTainted(password)                                    password > ';DROP * FROM credentials;- - > T


retrieve_user = "SELECT * FROM credentials WHERE user_id = '" + user_id + "' and password = '" + password + "';"
makeCondTainted(retrieve_user, user_id, password)        retrieve_user > SELECT ... DROP * FROM credentials;- - > T

if isTainted(retrieve_user) and is_sqli(retrieve_user):   retrieve_user > T && retrieve_user > SQLi > Leak and Exploit SQLi
    print("Sql Injection")
    exit(-1)

cursor = conn.execute(retrieve_user)
makeCondTainted(cursor, retrieve_user)


entries = cursor.fetchall()
makeCondTainted(entries, cursor)
```

Figure 1.5: Example 1 belonging to the SQLi code (1).

```
if len(entries) > 0:
    print("\n===Logged-in=====")
    retrieve_user = "SELECT * FROM accounts WHERE user_id = '" + user_id + "';"
    makeCondTainted(retrieve_user, user_id, password)

    if isTainted(retrieve_user) and is_sqli(retrieve_user):
        print("Sql Injection")
        exit(-1)

    cursor = conn.execute(retrieve_user)
    makeCondTainted(cursor, retrieve_user)

    entries = cursor.fetchall()
    makeCondTainted(entries, cursor)

    for entry in entries:
        makeCondTainted(entry, entries)

        user_id, first_name, last_name, phone = entry
        makeCondTainted(user_id, entry)
        makeCondTainted(first_name, entry)
        makeCondTainted(last_name, entry)
        makeCondTainted(phone, entry)

        if isTainted(user_id) or isTainted(first_name) or isTainted(last_name) or isTainted(phone):
            print("Printing Tainted Values")
            exit(-1)

        print()
        print("Here is {} data:".format(user_id))
        print("user-id=", user_id)
        print("first_name=", first_name)
        print("last_name=", last_name)
        print("phone", phone)
else:
    print("Wrong credentials")
```

Figure 1.6: Example 1 belonging to the SQLi code (2).

For the second example from the SQLi code, the inputs which are used are normal values which do not exploit the SQLi. Therefore, the leak occurs in this case when the values are printed at the end of the program.

```
# Connect to database
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()
makeTainted(user_id)


print("Insert your password")
password = input()
makeTainted(password)


retrieve_user = "SELECT * FROM credentials WHERE user_id = '" + user_id + "' and password = '" + password + "';"
makeCondTainted(retrieve_user, user_id, password)

if isTainted(retrieve_user) and is_sqli(retrieve_user):
    print("Sql Injection")
    exit(-1)

cursor = conn.execute(retrieve_user)
makeCondTainted(cursor, retrieve_user)


entries = cursor.fetchall()
makeCondTainted(entries, cursor)
```

user_id > Fran > T

password > 123456 > T

retrieve_user > Fran + 123456 > T

cursor > Fran + 123456 > T

entries > Fran + 123456 > T

Figure 1.7: Example 2 belonging to the SQLi code (1).

```
if len(entries) > 0:
    print("\n===Logged-in=====")
    retrieve_user = "SELECT * FROM accounts WHERE user_id = '" + user_id + "';"
    makeCondTainted(retrieve_user, user_id, password)

    if isTainted(retrieve_user) and is_sqli(retrieve_user):
        print("Sql Injection")
        exit(-1)

    cursor = conn.execute(retrieve_user)
    makeCondTainted(cursor, retrieve_user)

    entries = cursor.fetchall()
    makeCondTainted(entries, cursor)

    for entry in entries:
        makeCondTainted(entry, entries)

        user_id, first_name, last_name, phone = entry
        makeCondTainted(user_id, entry)
        makeCondTainted(first_name, entry)
        makeCondTainted(last_name, entry)
        makeCondTainted(phone, entry)

        if isTainted(user_id) or isTainted(first_name) or isTainted(last_name) or isTainted(phone):
            print("Printing Tainted Values")
            exit(-1)

        print()
        print("Here is {} data:".format(user_id))
        print("user-id=", user_id)
        print("first_name=", first_name)
        print("last_name=", last_name)
        print("phone", phone)
else:
    print("Wrong credentials")
```

retrieve_user > Fran + 123456 > T

cursor > Fran + 123456 > T

entries > Fran + 123456 > T

entry > Fran + 123456 > T

user_id > Fran + 123456 > T
first_name > Fran + 123456 > T
last_name > Fran + 123456 > T
phone > Fran + 123456 > T

user_id > Fran + 123456 > T && ... > Leak by printing tainted variables

Figure 1.8: Example 2 belonging to the SQLi code (2).

1.3. Static vs Dynamic Tainting

The main differences between the static and dynamic taint analysis is the fact that the static analysis is more conservative, obtaining no false negatives but on the other hand it could be over conservative producing false positives.

Regarding the dynamic tainting, as opposed to the static tainting it could produce false negatives as well as false positives. However, in this case precise alarms are generated while when static analysis is used more general alarms are produced.

Furthermore, due to the introducing of new calls at the dynamic tainting at run time, it suffers a greater overhead compared to the static tainting analysis.