LANGUAGED-BASED TECHNOLOGY FOR SECURITY
Homework Assignment 1

Introduction

In this homework, you will be asked to design a simple functional language with primitive abstractions to protect the execution of programs from untrusted code, i.e. code that has not been evaluated or examined for correctness and may attempt to circumvent the security requirements.

Untrusted code adds new challenges for making software secure. Note that web applications have made untrusted code a part of everyday life: visiting a web page typically loads JavaScript code from multiple providers into browsers. Cloud services allow third parties to provide applications that are dynamically combined with the core cloud functionalities. Finally, even traditional desktop applications dynamically download plugins from external providers.

Overall, the benefits of freely sharing code come at a cost: dynamically combining code from multiple sources might yield an insecure application.

The assignment

In this homework you will be asked to implement in OCAML the trusted interpreter of a functional language with linguistic primitives for managing the secure execution of untrusted code.

The programming language and its trusted execution engine support sharing of untrusted code while enforcing the security requirements. Specifically, the language provides a programming model that offers primitive abstractions to declare trusted code. Intuitively, programmers can exploit this abstraction to declare and group trusted code and data. Moreover, the language provides a primitive astraction to declare how trusted code can be invoked. Finally, a primitive astraction to execute plugins from external providers is also available.

Overall, the execution engine manages the interplay between trusted/untrusted functions. Note that the attacker controls the plug-ins in terms of both code and data.

We illustrate the main feature of the language by the simple example below.

```
let trust mycode =

trust{

     let secret string password = “abcd”;

     let checkPassword (guess:string): bool =

             password = guess;

     handle checkPassword;

}
```

The **trust** abstraction supports the declaration of both trusted functions and trusted data isolated inside a suitable block. Trust blocks allows programmers to group trusted functions and trusted data together so that they can be treated as a single trusted element. In our example, the trusted block allows programmer to declare a piece a code to manage password authentication. The trust block can be invoked only by calling the *checkPassword function*. In other words, the **handle** annotation acts as the interface between the trust block and the external environment. Note the secret annotation is used to identify secret data to ensure that their values should not be leaked outside the trust block.

In our example, the *checkPassword* function is the object of the *handle* annotation. The *checkPassword* function accepts a string from the esternal environment and compares it with the (secret) password, the result of the comparison is returned to the external environment as a boolean value. The return value of the handle function must not leak the secret information.

In addition to trust blocks, the language is equipped with abstractions to **include** and **execute** plugins, namely unstrusted code provided by external suppliers.

The code

```
Let myFilter = include {

     filter : ('a -> bool) -> 'a list -> 'a list = <fun>;

}
```

In our example we assume to exploit in our programm the plug-in providing the code of a filter function which removes from a list all the elements that do not satisfy a property (given by the predicate represented by the first argument of the filter function). Note that the code of the plugin cannot be inspected.

The code

```
let even n = n mod 2 = 0 in

 execute(myFilter, even [1; 2; 3; 4]
```

invokes the the plug-in and produces as result the list `[2; 4]`

We emphasize that the attacker controls the plug-in code. Let us take the code

```
let lst = execute(myFilter,

                  Mycode.checkPassword,

                  ["abcd", "efgh"])
```

the attacker can control the release of values and therefore can obtain the value of the secret password.

To enforce *integrity* of the interplay between trusted and untrusted code the execution engine of the language will exploit dynamic taint analysis. Dynamic taint analysis supports the control of the flow of tainted data inside the program. The implementation of the execution engine must avoid data leakages.

The assignment consists of two mandatory contributions:

1. Discuss the language design to support trust programming and plug-in inclusion.
2. Describe the implementation of its execution environment (interpreter, dynamic taint tracking and run-time support) and evaluate its usage by presenting some simple case studies.

Note. You can design the programming language and its trusted execution engine in total freedom. For example, you can freely equip the language with further suitable linguistic abstractions. but you have to justify all the choices made.

The homework also includes an *optional* part briefly discussed below. An assertion is a predicate (a Boolean-valued function over the state space, usually expressed as a logical proposition on the variables of a program) connected to a point in the program, that always should evaluate to true at that point in code execution. Assertions can help a programmer to detect the bugs or defects.

Typically, the checking of assertions is actually done when the program run. Indeed, if the assertion is not evaluated to true – an assertion failure – the program

engines considers the program to be broken and throws an assertion failure exception.

The optional part of the homework consists in design and developing the assertion based testing of taintness for the proposed execution engine.

Learning objectives

The goals of the homework are to appreciate and understand the trade-offs in the design and implementation of an experimental secure-aware programming language. Considering the issues of trusted code and data integrity in the setting provided by the assignment has the advantage of addressing an effective instance of the problems.

Constraints

To participate in the oral examination, students must compulsorily submit two homeworks. This is the first homework.

Submission Format

The submitted homework assignment (zipped folder format) must include:

1. The definition of the programming language along with some examples.
2. The source code of the execution environment of the language.
3. A short description of the design choices.

Submissions Guidelines

Both assignments must be submitted no later than midnight of

- June 2, 2024 for the first exam call set for June 7
- June 23, 2024 for the second exam call set for June 28
- July 14, 2024 for the third exam call set for July 19

Submission will be by email to the address: gian-luigi.ferrari@unipi.it
The structure of the email subject is mandatory: [LBT23-24 HW1-2]: <name>.
Please in the email indicate: Name and email address of all the group members.
Please include all files in a zipped folder.