Language Based Technology for Security

Homework 1

Federico Ramacciotti
f.ramacciotti4@studenti.unipi.it

25/04/2023

Contents

1 Introduction

The language is similar to `stack_inspection`, the one presented in class: in order to write programs we have to write their Abstract Syntax Tree.

The language is a simple functional language, with support to integers, booleans, strings, print statements, functions and a way to sandbox processes and secure the system from untrusted code.

2 Implementation

The language is implemented in pure OCaml, defining libraries in the `lib/` directory. The interpreter is composed of two files: `env.ml` and `interpreter.ml`. In the former file there is the specification of the environment, while the latter defines the function to evaluate a program.

2.1 General interpreter definition

2.1.1 Env.ml

The environment is a map, from strings to values, to speed up search and insertion. The keys of the map are `type ide = string`, identifiers of the variables, and the values are of type value[1]. Then I defined the functions to manage the map: `lookup` searches for an object in the map and fails if it does not find it, `has_key` returns whether the elements is in the map and `insert` adds an element to the map. Other than these three simple functions, I also defined the functions to print integers, booleans and strings from the environment.

```
type ide = string
module StringMap = Map.Make (String)
let create_env : 'v StringMap.t = StringMap.empty

let lookup env k =
  try StringMap.find k env
  with Not_found -> failwith (k ^ " not found in the environment")

let has_key env k = StringMap.mem k env

let insert env k v = StringMap.add k v env
```

2.1.2 Interpreter.ml

The main function of the interpreter is the `run` function defined at the bottom of the file. It creates an empty environment and runs the real evaluation function `eval`, that takes as input the current expression and the current environment and returns the value associated to the evaluated expression.

```
let run code =
  let env = create_env in
  eval code env
```

There are defined two types, as mentioned above, expressions and values:

- `expr`: an expression is a part of the input program, that corresponds to a statement; it can be a variable, a function, a primitive, an if, a call, an enclave, an untrusted piece of code or an execution of untrusted code. When some expression has type `* expr` at the end, it means that it expects another expression to be evaluated after that.

```
    type expr =
      | CstSkip
      | CstI of int
```

[1]Explained in section 2.1.2

```
    | CstB of bool
    | CstS of string
    | Var of ide
    | Prim of ide * expr * expr
    | If of expr * expr * expr
    | Let of ide * expr * expr
    | Fun of ide * expr
    | Call of expr * expr
    | Enclave of expr * expr
    | Secret of ide * expr * expr
    | Gateway of ide * expr
    | Print of expr
    | End
    | IncUntrusted of expr
    | Execute of ide * expr
```

- `value`: a value is a token that corresponds to the evaluated expression; it can be an integer, a boolean, a string or a closure for a function, an enclave or an untrusted library. Apart from the basic values, the only complex one is the closure: it represents a function and it's composed of its identifier, its body, its internal environment and the security, that will be explained below.

```
  type value =
    | Skip
    | Int of int
    | Bool of bool
    | String of string
    | Closure of ide * expr * value StringMap.t * security
    | UntrustedClosure of expr
```

Other than these two types, there is also a third type called `security`, associated to function closures, that specifies if a function is `Free`, `Enclaved` or `Untrusted`. In the first case the function can be read and executed by anyone, in the second case the function can only be executed by a trusted process and in the third case in can only be executed in an `Execute` directive, in order to sandbox the process.

The definition of an expression and a value of type `Skip` is meant mainly for testing purposes: normally a program should complete doing something useful (like a print), but in the tests it's useful to have a skip directive, in order to have something like `let x = 5 in skip` and only test the variable declaration and nothing else.

2.2 Enclaves

An enclave is, as how I interpreted it, like an unnamed private class of Java. Everything inside it must be declared as a `Secret`, so that no one can see it outside the enclave. When a function has to be seen outside the enclave, it must be mentioned in a `Gateway`. In the language, enclaves are separated from the other part of the program and behave in an imperative way, like a library. That's why to end an enclave it is mandatory to mention the `End` directive.

When evaluating an enclave a new environment is created, dedicated only to the specified enclave, that inherits the global environment and is discarded when `End` is read. In this way only the closures of the functions are pushed to the global environment and we are sure that there cannot be leaks of the secret variables, since they don't exist anymore. Enclaves are also unnamed, since they're evaluated at the time of reading them and later "discarded" for security purposes.

Furthermore, in order to keep the system as secure as possible, gateway functions are marked with the `Enclaved` type: in this way untrusted code cannot execute them.

2.3 Untrusted code and execution

Untrusted code behaves differently from enclaves, as it is not seen as a library, and it behaves like normal functions: it has to be associated to a variable with a `Let` directive, and it doesn't need `End`. When reading an `IncUntrusted`, the code is simply saved in the environment and not executed until it is not needed.

In order to execute an untrusted piece of code, an `Execute` command is needed, together with the name of the variable that stored the untrusted code. Now we are evaluating it with a sort of sandbox: the code inherits the global environment but does not add anything to it. In this way, when the process ends, possibly malicious code doesn't leave traces in the environment. Moreover inside untrusted code it is not possible to call enclaved functions, so that it cannot access sensitive information.

Therefore, the `Execute` command sandboxes a process and secures the system against web pages and cloud applications by completely isolating their code from the operating system.

3 Language examples

3.1 Declare an enclave that checks if a password is correct

```
Enclave(
    Secret("pass", CstS "rightpw",
        Secret("checkpw", Fun("arg", Prim("=", Var "arg", Var "pass")),
            Gateway("checkpw", (* make checkpw available outside the enclave *)
                End
            )
        )
    ),
    (* create a variable and check it with checkpw *)
    Let("pw", CstS "rightpw",
        Let("b", Call(Var "check", Var "pw"),
            If(Var "b", Print(CstS "pw ok"), Print(CstS "pw wrong"))
        )
    )
)
```

3.2 Executed untrusted code

```
Let("untr",
    IncUntrusted(
        (* untrusted code *)
    ),
    Execute(
        "untr", (* this line reads the untrusted code and executes it *)
        CstSkip
    )
)
```

4 Testing

The tests of the language are placed in the `bin/main.ml` file, in which I tested the various functionalities and edge cases or errors. In this file I defined two functions `execWithFailure` and `execWithoutFailure` that execute some code, capture possible failures and assert if the code should fail or not. Then there is a list of tests, that gets executed by the `execute_tests` function at the end of the file. If everything goes as planned, the file should execute without any issues and should print "Tests passed".

In order to add tests to the file it is possible to concatenate more `execWithFailure` and `execWithoutFailure` functions to the `tests` list, that will be executed by the intepreter on a new empty environment. Then, to execute the tests, build the project with `dune build` and run the main with `dune exec hw1`.

4.1 Tests

In the `bin/main.ml` file I provided 11 different tests:

1. `[ ok ]` the code executes a gateway function of an enclave;
2. `[fail]` the code accesses secret variables of an enclave;
3. `[fail]` an enclave declares something that is not a secret or a gateway;
4. `[ ok ]` untrusted code executes a free function declared outside (untrusted code is sandboxed but can still use standard free and harmless functions of the code);
5. `[fail]` untrusted code executes an enclaved gateway function (the untrusted code cannot use functions that can manage sensitive information);
6. `[fail]` untrusted code accesses secret variables of an enclave;
7. `[fail]` execute a function that is not untrusted code (execute should be used only to sandbox untrusted code);
8. `[fail]` execute something else like variables, primitives, ...;
9. `[fail]` the code uses some variable of the untrusted code (we don't want to be able to access untrusted, and possibly malicious, code from the outside);
10. `[ ok ]` execute untrusted code;
11. `[fail]` an enclave declares a variable that is already declared outside (shadowing is not permitted as it leads to undefined behavior);