University of Pisa

Department of Information Engineering

Language-Based Technology for Security

Nicolò Mariano Fragale
March 2025

Contents

Information

These notes are intended for educational purposes only and cover essential concepts in the field of data systems and security. The aim is to provide a comprehensive understanding of topics such as system vulnerabilities, protection techniques, and defense strategies in cybersecurity.

This document includes topics related to access control, authentication mechanisms, database security, cryptographic methods, and advanced persistent threats, with a particular focus on practical applications in real-world scenarios.

1 Web Assembly

1.1 Introduction WASM is not a programming language, but a binary format generated from other language like C, C++ or Rust. WASM permit this hig level language to run efficiently and properly. It is executed in safe place like browser or other runtime enviroment.

It is safe because it runs in isolated sandbox.

It is used to increase performance in web application as:

1. 3D games in Browser;
2. Figma etc...
3. Editing image/video software online...
4. Ai, ML, blockchain, criptoghraphy...
5. Allow to execute C, C++, Rust online;
6. It can be used on server.

1.2 Key characteristics

1. Stack-Based (push and pop) <-> Does not use registers; -> Operations;
2. Executabel in web broser -> using WebAssembly JavaScript API. -> API is the only way to communicate from sandbox to outside;
3. Secure -> Sandbox and permission denied to access system resources;
4. Platform-independent -> runs on any device that has WASM runtime.

Listing 1: Code Example

```
(func $calcola (param $x i32) (result i32)
    local.get $x
    local.get $x
    i32.mul
    i32.const 2
    i32.mul
    i32.const 1
    i32.add
)
```

Analyze the example:

1. *func $calcola (param $x i32) (result i32)* :
   (a) *func* it is the key word declaring the function;
   (b) *$calcola* function's name;
   (c) *param* it is the key word declaring the parameter;

   (d) *$x* parameter's name;

   (e) *i32* indicates the data-type (32 bit integer);

   (f) *result i32* indicates the result will be a i32 data type.

   (g) **if $ is omitted the code will still work.**

2. *local.get $x* push X in stack with index 0 (Func Starts wih stack empty);
3. *local.get $x* push X in stack with index 1;
4. *i32.mul* pop 0 and 1 mul, then mul them (both x) and push temporary result in index 0;
5. *i32.const 2* push in stack the value 2 as type i32 and index 1;
6. *i32.mul* pop 0 and 1, then mul them and push as temporary result as index 0;
7. *i32.const 1* add 1 as i32 in index 1 ;
8. *i32.add* pop 0 and 1, add index 0 and 1, result is pushed in index 0.

1.3 Data-Type

1. **i32** integer with or without sign in 32 bit -> (from 0 to 4.294.967.295) or (from -2.147.483.648 to 2.147.483.647);
2. **i64** integer with or without sign in 64 bit;
3. **f32** floating poin in 32 bit;
4. **f64** floating point in 64 bit.

1.4 Storing Values

1. Stack -> push and pop (for operations) of the parameter and costant;
2. Function context -> variabale passed as parameter or declared inside the function -> Example: *local $temp i32*; ->

Listing 2: Code Example

```
1         (func $quadrato (param $x i32) (result i32)
2         (local $temp i32)
3         local.get $x
4         local.get $x
5         i32.mul
6         local.set $temp
7         local.get $temp
8     )
```

3. Single global memory -> Linear memory to handle complex data structure -> Used by many functions to store data in long term or to share data beetween more functions -> Example:

Listing 3: Code Example

```
(global $contatore (mut i32) (i32.const 0))

(func $incrementa (result i32)
    global.get $contatore
    i32.const 1
    i32.add
    global.set $contatore
    global.get $contatore
)
```

(mut i32) mutable variabale type i32 (otherwise immutable during the execution), **(i32.const 0)** initialized at 0.

1.5 Operations

1. *local.get $x:* Push x onto the stack;
2. *local.set $x:* Assign the value in top stack to x;

1.5.1 Memory

There is a single linear memory built as a contigous array of byte where u can read and write data. The Address Memory is a number -> offset.

1.5.2 (data-type(i32,i64...)).load

Using ().load, u can interact with the memory -> U read 4 byte (if i32) in the memory starting from the offset -> Convert in number (i32 in this case) and push onto the stack. It is used when u want read data from the memory.

Example:

Listing 4: Code Example

```
(memory (export "mem") 1)
(func $leggi_memoria (param $ptr i32) (result i32)
local.get $ptr
i32.load
)
```

The function will read 4 byte from the value stored in ptr (this value represent the offset). If ptr = 100 (offset), then from 100 the function will read 4 bytes (100-101-102-103).

(memory (export "mem") 1): Before use load and store u must declare the memory from whom u are going to read data with the commands **memory**.

N.B: 1 : 64 Kb (1 page is 64 kb -> 2 pages are 128 kb and so on.)

N.B: If u try to ask for a offset+value ¿ 56 536 byte -> *RuntimeError: invalid memory access out of bounds* -> WASM permit dynamic memory allocation.

1.5.3 Store

Store allow to write in memory.

Listing 5: Code Example

```
1         (memory (export "mem") 1)
2         (func $leggi_memoria (param $ptr i32) (result i32)
3         local.get $ptr
4         local.get $val
5         i32.store
6         )
```

Work the same way as before, while *local.get $val* specify the value to write in memory.

1.6 Control Flow

1.6.1 Program Counter

It is a register that keep track about the next istructions to execute, increment each time by 1. It acts like a pointer.

1.6.2 Loop/br -> Break/br_if

Loop create a label (code of block) which runs infinitely if **br** is at the end of the label.

N.B: You cant use **br** outside the scope of the label.

Listing 6: Code Example

```
1         (func $loop_example (param $x i32)
2         i32.const 0
3         local.set $x
4         (loop $loop
5             local.get $x
6             i32.const 1
7             i32.add
8             local.set $x
9             br $loop
10        )
11      )
```

To block the infinite loop we need a condition -> br_if and a label to call -> Break:

Listing 7: Code Example

```
1     (func $loop_example (param $x i32)
2         i32.const 0
3         local.set $x
4         block $out(
```

```
        (loop $loop
            local.get $x
            i32.const 1
            i32.add
            local.set $x

            local.get $x
            i32.const 10
            i32.eq
            br_if $out

            br $loop
        )
    )
)
```

U can indent block, the outest gets label 0, the second enclosing block gets 1 and so on...

Listing 8: Code Example

```
(block $outer_block ;; label 0

    (block $inner_block ;; label 1
    ))
```

N.B: We can call br 0 in the inner_block and in this case it will jump directly outside label 0.

N.B: U can omit the name of the label. -> br ¡label_name¿ is replace with br 0.

N.B: U can omit the name of the variable. Instead use indeces for references (not reccomended).

1.6.3 Call & Call_indirect

1. **Call** a function:

Listing 9: Code Example

```
            (func $add (param $a i32) (param $b i32) (result i32)
            local.get $a
            local.get $b
            i32.add
            )

            (func $main
            i32.const 5
            i32.const 10
            call $add ;; Chiamata della funzione $add
            )
```

2. **Call_indirect** a function using its index stored in a table containing functions: *(call_indirect (type ¡type¿) ¡index¿)*

1.6.4 If & Else

Listing 10: Code Example

```
    (func $check_even (param $x i32) (result i32)
  local.get $x
  i32.const 2
  i32.rem_u ;; rem_u : % (module)
  i32.eqz
  if
    i32.const 1  ;; If $x is even, return 1
  else
    i32.const 0  ;; else 0
  end
)
```

1.6.5 Return

A function's return value is **implicitly** the value at the top of the stack. -> U dont need to write it explixitely at the end of the function.