Analisi della sicurezza binaria di WebAssembly

Alessandro Arata

Universitá di Genova

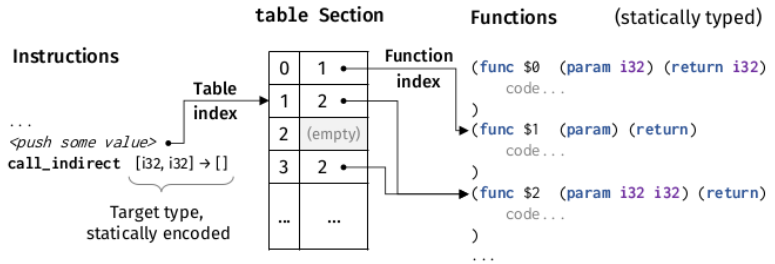
June 23, 2021

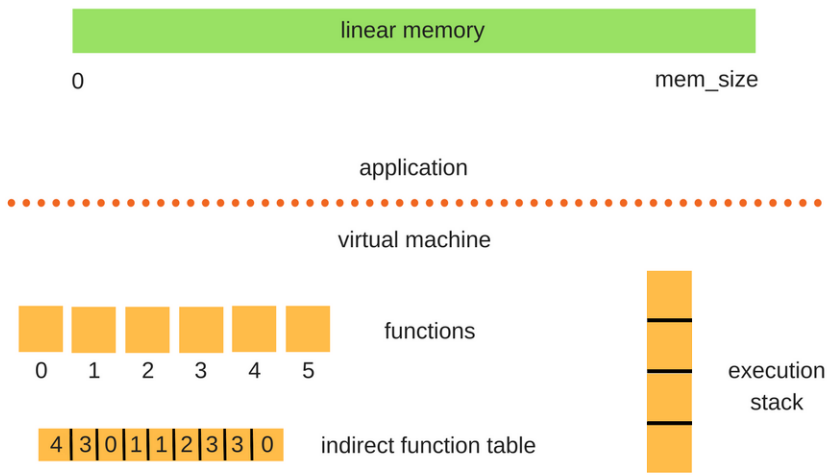
WebAssembly

- è un linguaggio bytecode che offre tempi di esecuzione rapidi e un formato portabile e compatto
- è stato ideato come un **target di compilazione**
  - esistono compilatori da C(++), Rust ed altri linguaggi
- è pensato per eseguire codice in maniera efficiente sui **browser** ed è implementato dalla maggior parte di questi
  - può essere eseguito anche in ambienti di backend come Node.js

WebAssembly - organizzazione di un programma

- gli elementi del programma (per esempio funzioni o variabili) sono identificati da **indici** interi
- staticamente tipato con quattro tipi primitivi:
  - **i32** e **i64** per interi a 32 e 64 bit
  - **f32** e **f64** per floating point a singola e doppia precisione
- tipi complessi (stringhe, indirizzi, classi, struct...) vengono salvati in un'area di memoria chiamata **memoria lineare**
- lo stack delle chiamate e le variabili (globali o locali) di tipo primitivo sono gestiti dalla macchina virtuale
- la memoria lineare è gestita dal programma





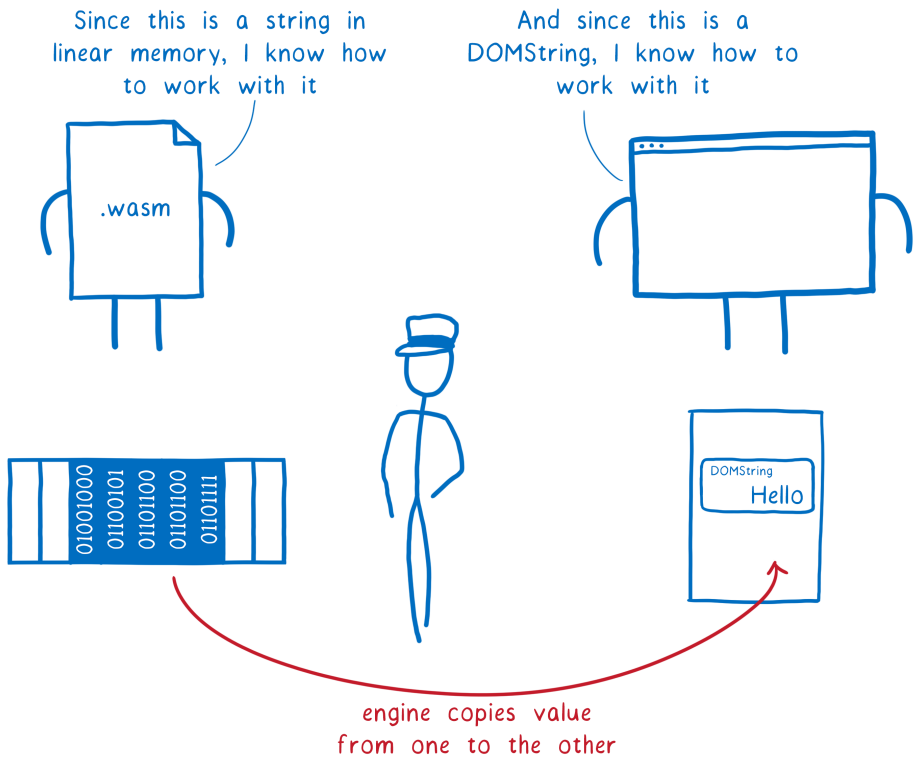
helloWorld.wat

```
puts("Hello, world!");
```

```
(module
  ;; Imports from JavaScript namespace and log function
  (import  "console"  "log" (func  $log (param  i32  i32)))
  ;; Import 1 page of memory
  (import  "js"  "mem" (memory  1))
  ;; Data section of our module
  (data (i32.const 0) "Hello, world!")
  ;; Function declaration: Exported as helloWorld(), no arguments
  (func (export  "helloWorld")
    ;; pass offset 0 to log
    i32.const 0
    ;; pass length 13 to log (strlen of sample text)
    i32.const 13
    call  $log
  )
)
```

Memoria lineare

- array **globale** di byte indirizzato da puntatori
  - per indirizzare la memoria si utilizzano indici di tipo i32
- è divisa in **regioni** per heap, stack e dati statici
- contiene tipi non primitivi
- è sia leggibile che scrivibile ma non eseguibile
- è totalmente allocata: ogni puntatore compreso tra [0, mem_max] è valido

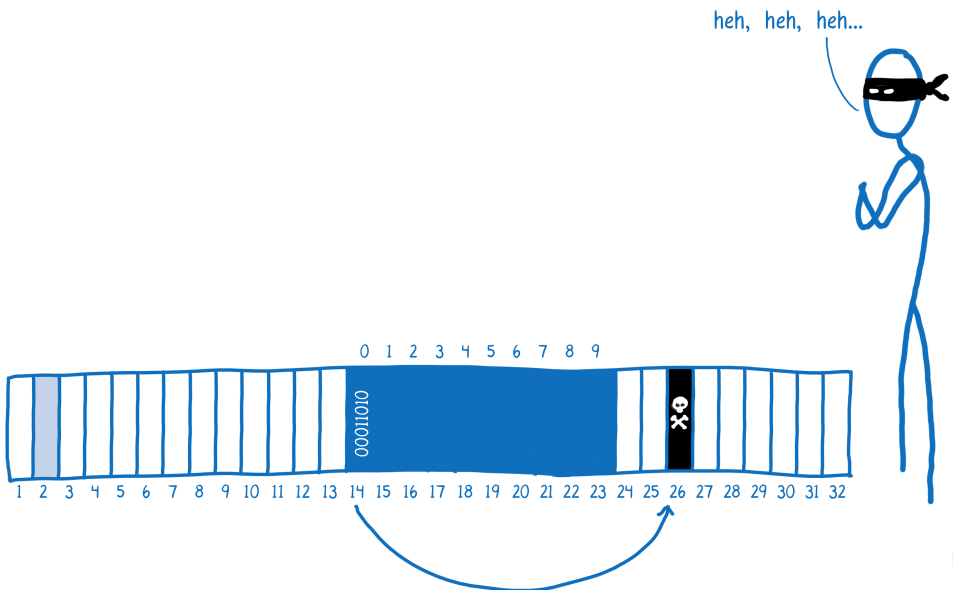


Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Layout Space Randomization** (ASLR)
  - **Guard page**
  - **Stack canary**
- esistono sezioni dedicate per i dati costanti e statici in modo che non possano essere sovrascritti
- la memoria del processo è divisa in pagine e possono non trovarsi tutte in memoria in un preciso momento dell'esecuzione
  - un attaccante deve evitare di accedere a pagine non allocate (viene sollevato un fault)

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento è "predicibile" (dal compilatore utilizzato e dal programma)
- nessuna **guard page** tra le regioni: un overflow in una regione può sovrascrivere dati in altre regioni.
- nessuna **stack canary**: il binario non controlla da sè se si scriva oltre lo spazio allocato per il buffer
- ogni area è scrivibile: dati apparentemente costanti possono essere sovrascritti
- non esistono pagine non allocate: per l'attaccante ogni puntatore è valido (non esistono page fault)



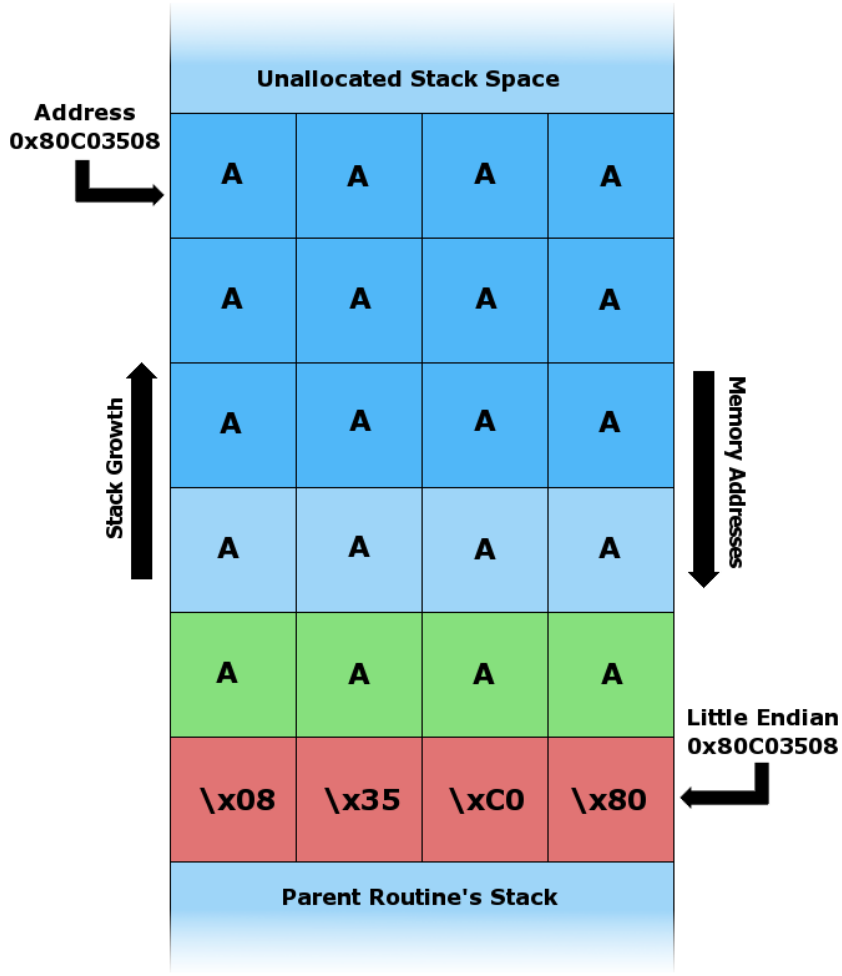
Rispetto alle mitigazioni include nei binari nativi, la memoria lineare di WebAssembly non offre molto dal punto di vista della sicurezza binaria.

Cosa succede se si compila un programma **vulnerabile** in WebAssembly?

```
void vuln() {
  char buffer[16];
  gets(buffer); // !!!
}
```

Primitiva di attacco: buffer overflow

- se la lunghezza dell'input dell'utente non viene controllata, è possibile scrivere al di fuori di un buffer
  - in particolare è possibile sovrascrivere variabili locali all'interno dello stack
- l'assenza di mitigazioni nella memoria lineare permette di sovrascrivere dati in altre regioni contigue
- funzioni come **gets()** e **strcpy()** (senza ulteriori controlli) permettono questo tipo di attacco



Sovrascrivere dati "costanti"

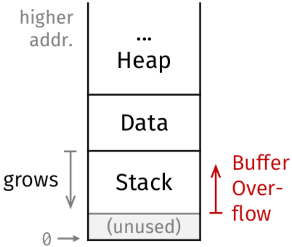
```
char *other_data = "AAAA";
static char *safe_script =
  "console.log('this should be safe, shouldn\\'t it?')";

int main() {
  emscripten_run_script(safe_script);
}

void vuln(const char* input) {
  strcpy(other_data, input);
}
```

Sovrascrivere dati "costanti"

- le variabili statiche sono salvate nella regione **data**
  - dato che non esistono aree non scrivibili, non esiste una sezione dedicata alle variabili costanti
- **strcpy()** non effettua controlli sulla dimensioni dell'input dell'utente: è possibile sovrascrivere dati sullo stack
  - in particolare (con il layout della memoria in figura) un overflow nello stack riesce a scrivere nella regione **data**
- sovrascrivendo la stringa puntata dalla variabile safe_script si possono eseguire comandi JavaScript
  - **XSS** nel browser
  - **RCE** con Node.js



Chiamando **vuln()** con la stringa ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;alert('XSS')"

Sovrascrivere dati sullo heap

- similmente è possibile sovrascrivere stringhe sullo heap
- la libreria **libpng** contiene un buffer overflow che è possibile sfruttrare convertendo un immagine da pnm a png

```
int main() {
  std::string img_tag =
    "<img src='";
  // CVE-2018-14550
  pnm2png("input.pnm", "output.png");
  img_tag += file_to_base64("output.png") + "'>";
  emcc::global("document").call("write", img_tag);
}
```

- se l'input dell'utente non viene controllato/sanitizzato, è possibile sovrascrivere la string **img_tag** situata nello heap
- questo causa un attacco di tipo XSS nel browser

Sovrascrivere dati sullo heap

Select input PNM file to convert and show: Choose File No file chosen

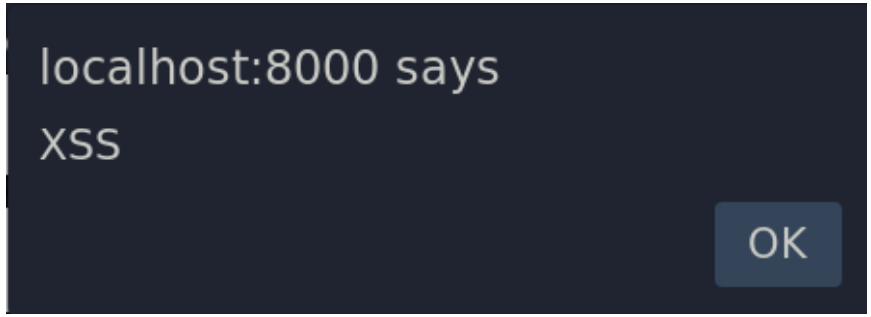
Il sito chiede all'utente un'immagine in input: non vengono fatti controlli sul tipo di file

Se l'utente immette un'immagine, questa viene convertita e mostrata nel browser

Utilizzando come payload

AAAAAAAAAAAAAAAAAAAAAAAA...
<script>alert('XSS')</script><!- -

è possibile sovrascrivere i dati nello stack (con una sequenza specifica di A) e sostituire al tag **<img>** il tag **<script>** nello heap: questo provoca un attacco di tipo XSS



Conclusioni

- WebAssembly ha reintrodotto vulnerabilitá precendentemente mitigate e introdotto nuovi tipi di attacchi, come sovrascivere dati costanti
-

Bibliografia

1. Daniel Lehmann, Johannes Kinder, Michael Pradel:
   "Everything Old is New Again: Binary Security of WebAssembly"
2. Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, Justin Engler:
   "Security Chasms of WASM"