Analisi della sicurezza binaria di WebAssembly

Alessandro Arata

Universitá di Genova

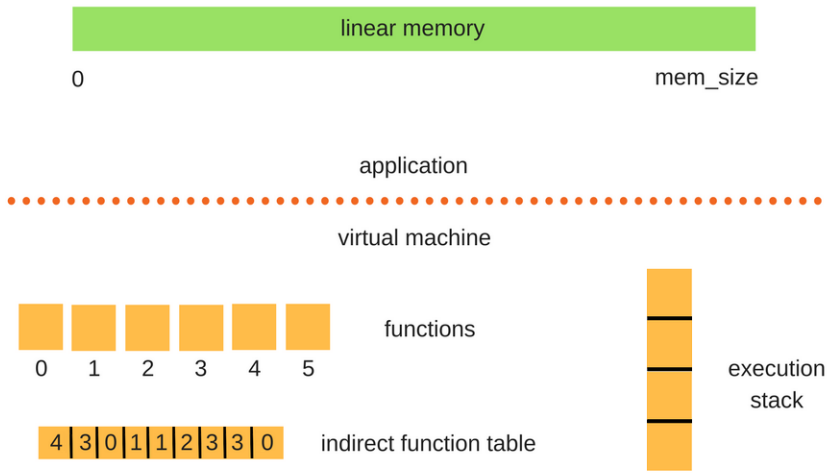
June 25, 2021

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**
  - può essere eseguito anche in ambienti di backend come Node.js

WebAssembly - gestione della memoria

- staticamente tipato con quattro tipi primitivi: **i32/64**, **f32/64**
- lo stack delle chiamate e le variabili primitive sono gestiti dalla macchina virtuale
- i tipi non primitivi (stringhe, indirizzi, classi, struct...) vengono salvati nella **memoria lineare**
- la memoria lineare è gestita dal programma



hello-world.wat

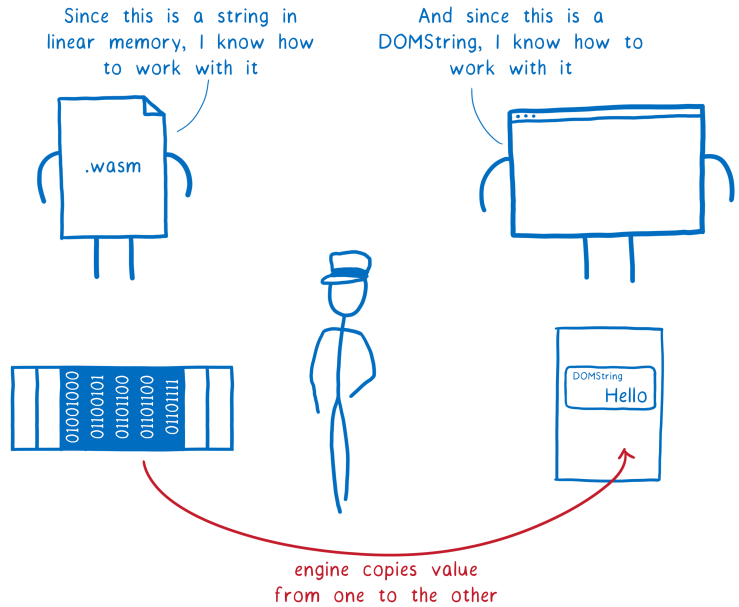
```
puts("Hello, world!");
```

corrisponde nel formato assembly di WebAssembly (**.wat**) a

```
(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 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
- tutta la memoria risulta allocata al programmatore: ogni puntatore compreso tra [0, mem_max] è valido

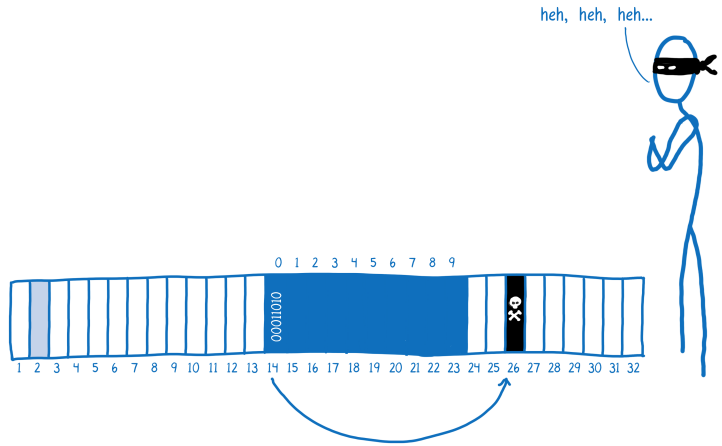


Principali mitigazioni nei binari nativi

- 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 non scrivibili per i dati costanti in modo che non possano essere sovrascritti
- la memoria del processo è divisa in pagine non necessariamente sempre mappate
  - un attaccante deve evitare di accedere a pagine non allocate (verrebbe sollevato un fault)

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento è "predicibile"
- nessuna **guard page** tra le regioni: è possibile sovrascrivere dati di regioni contigue
- 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
- per l'attaccante ogni puntatore è valido dato che non esistono page fault



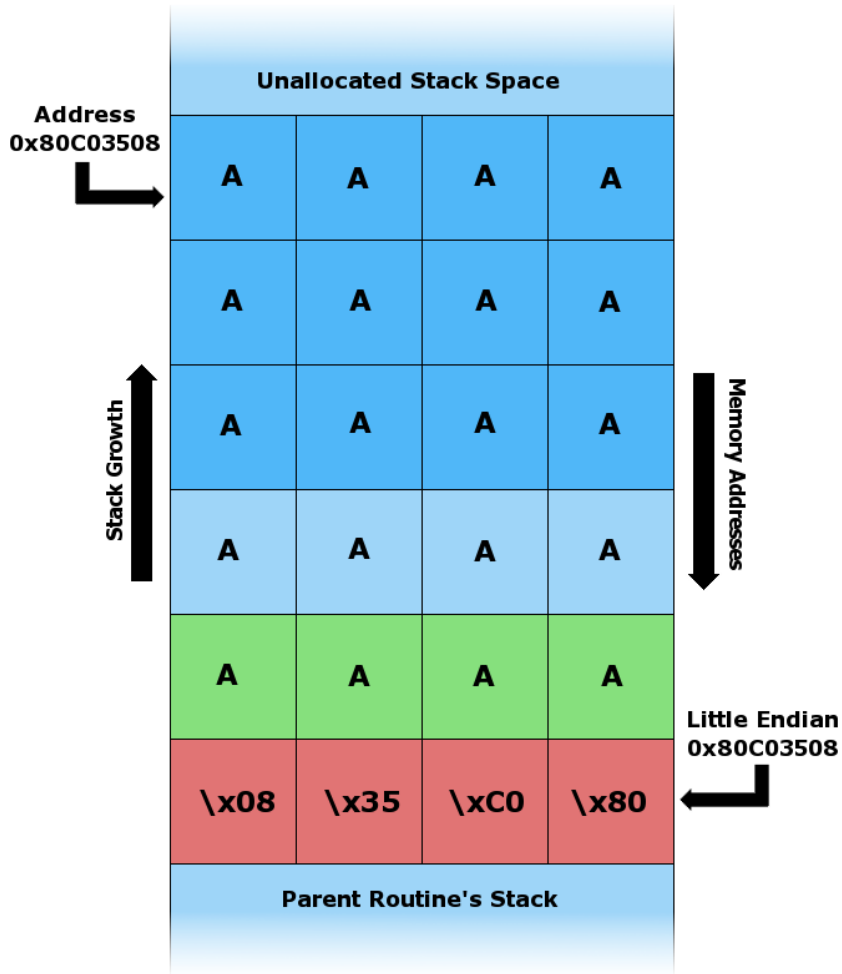
Rispetto alle mitigazioni include nei binari nativi, la memoria lineare di WebAssembly non offre molto dal punto di vista della sicurezza binaria.

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
  - si possono sovrascrivere variabili locali o indirizzi di ritorno
- funzioni come **gets()** e **strcpy()** permettono questo tipo di attacco



Sovrascrivere dati "costanti"

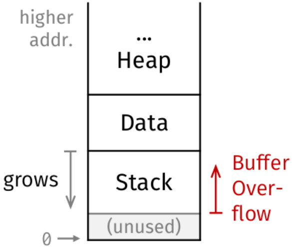
```
char *other_data = "AAAA";
static char *safe_script =
  "console.log('this should be safe, shouldn\\'t it?')";

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** (scrivibile)
- **strcpy()** non effettua controlli sulla dimensioni dell'input: è possibile sovrascrivere dati sullo stack
  - un overflow nello stack riesce a scrivere nella regione **data**
- sovrascrivendo la stringa contenuta nella variabile **safe_script** si possono eseguire comandi JavaScript arbitrari
  - **XSS** nel browser
  - **RCE** con Node.js



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

Sovrascrivere dati sullo heap

- similmente è possibile sovrascrivere dati 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><!- -

si effettua un overflow nello stack che permette di 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"