Analisi della sicurezza binaria di WebAssembly

Alessandro Arata

Universitá di Genova

10/07/2021

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** (da C, C++, Rust...)
- è pensato per eseguire codice in maniera efficiente sui **browser** o in backend (Node.js)

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** (da C, C++, Rust...)
- è pensato per eseguire codice in maniera efficiente sui **browser** o in backend (Node.js)

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** (da C, C++, Rust...)
- è pensato per eseguire codice in maniera efficiente sui **browser** o in backend (Node.js)

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
  )
)
```

hello-world.wat

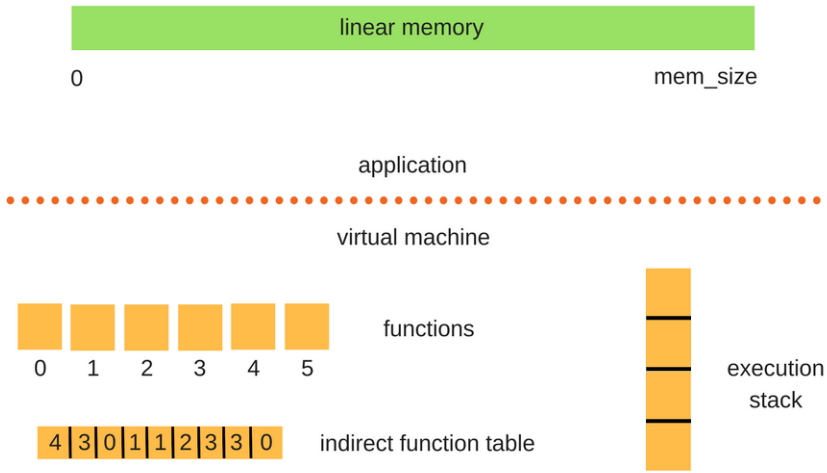
```
puts("Hello, world!");
```

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
  )
)
```

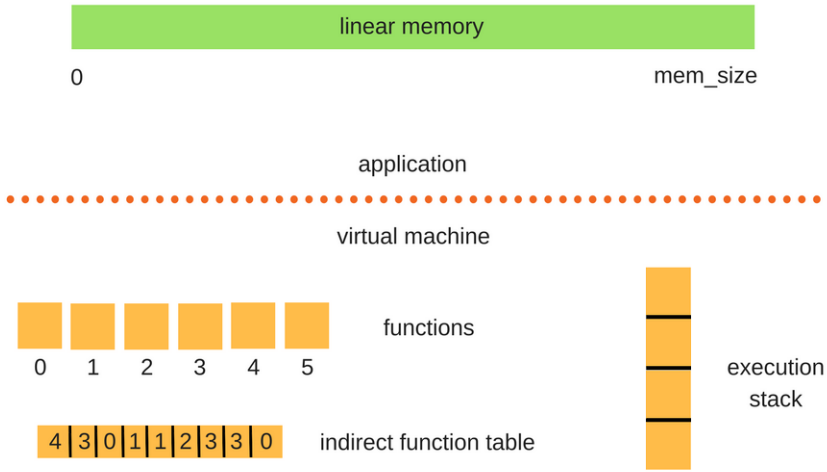
WebAssembly - tipi e gestione della memoria

- 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



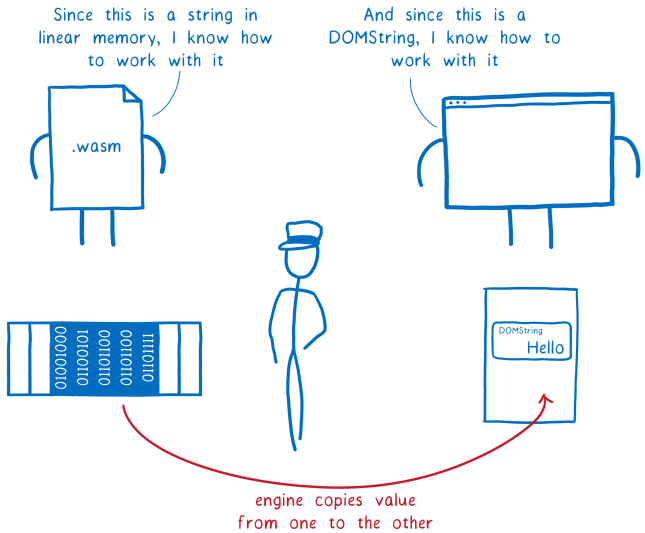
WebAssembly - tipi e gestione della memoria

- 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



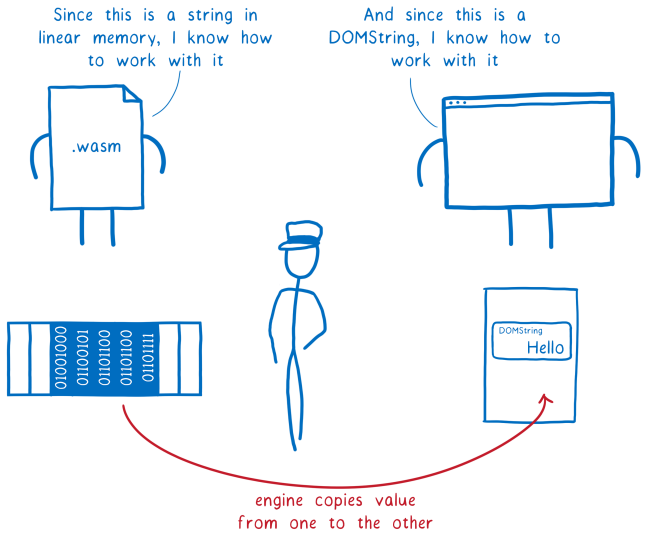
Memoria lineare

- array **globale** di byte indirizzato da puntatori di tipo i32
- è divisa in **regioni** per heap, stack e dati statici
- è sia leggibile che scrivibile ma non eseguibile
- **tutta la memoria** risulta allocata al programmatore: ogni puntatore compreso tra [0, mem_max] è valido



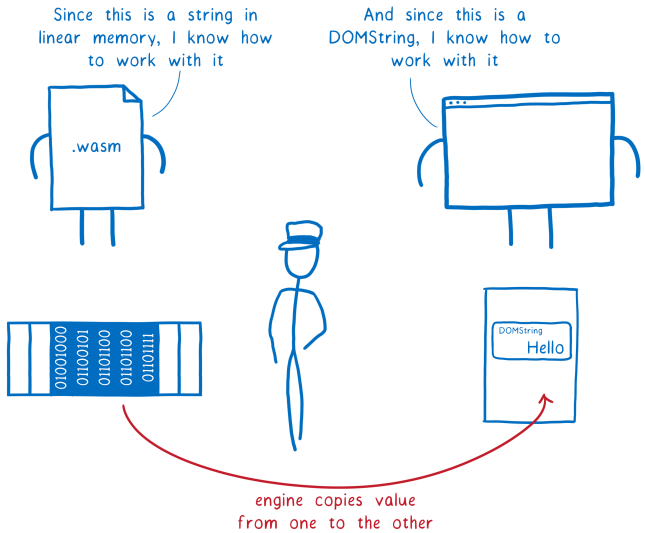
Memoria lineare

- array **globale** di byte indirizzato da puntatori di tipo i32
- è divisa in **regioni** per heap, stack e dati statici
- è sia leggibile che scrivibile ma non eseguibile
- **tutta la memoria** risulta allocata al programmatore: ogni puntatore compreso tra [0, mem_max] è valido



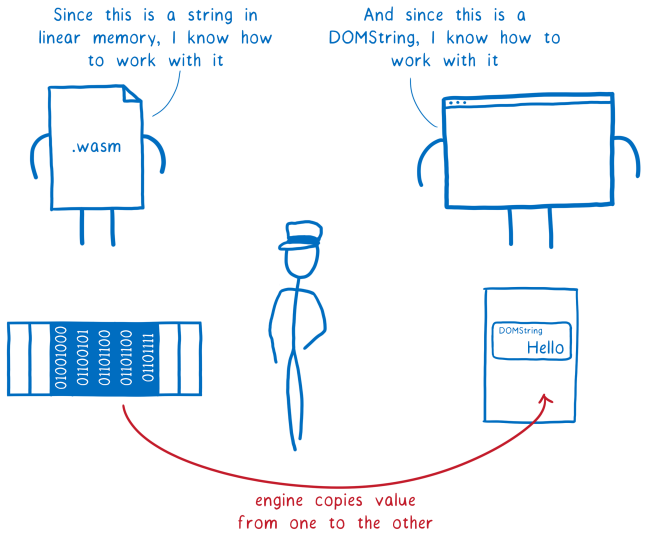
Memoria lineare

- array **globale** di byte indirizzato da puntatori di tipo i32
- è divisa in **regioni** per heap, stack e dati statici
- è sia leggibile che scrivibile ma non eseguibile
- **tutta la memoria** risulta allocata al programmatore: ogni puntatore compreso tra [0, mem_max] è valido



Memoria lineare

- array **globale** di byte indirizzato da puntatori di tipo i32
- è divisa in **regioni** per heap, stack e dati statici
- 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 Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Principali mitigazioni nei binari nativi

- i binari nativi offrono diverse mitigazioni in maniera da rendere difficile l'utilizzo di exploit della memoria
  - **Address Space Layout 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 in memoria
  - un attaccante deve evitare di accedere a pagine non allocate

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento rimane costante attraverso tutte le esecuzioni del programma
- 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

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento rimane costante attraverso tutte le esecuzioni del programma
- 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

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento rimane costante attraverso tutte le esecuzioni del programma
- 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

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento rimane costante attraverso tutte le esecuzioni del programma
- 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

Memoria lineare - sicurezza binaria

- no **ASLR**: la posizione di una elemento rimane costante attraverso tutte le esecuzioni del programma
- 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

Sulla sicurezza di WebAssembly

[...] the presence of control-flow integrity and protected call stacks prevents direct code injection attacks. Thus, common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs.

`https://webassembly.org/docs/security`

Cosa succede se un programma **vulnerabile** viene compilato in WebAssembly?

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

Sulla sicurezza di WebAssembly

[...] the presence of control-flow integrity and protected call stacks prevents direct code injection attacks. Thus, common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs.

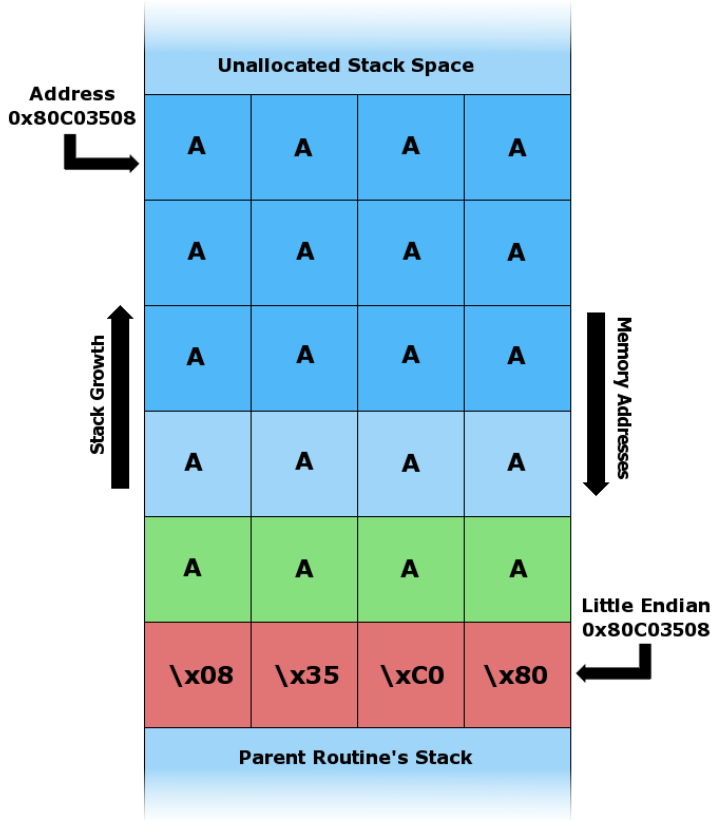
`https://webassembly.org/docs/security`

Cosa succede se un programma **vulnerabile** viene compilato in WebAssembly?

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

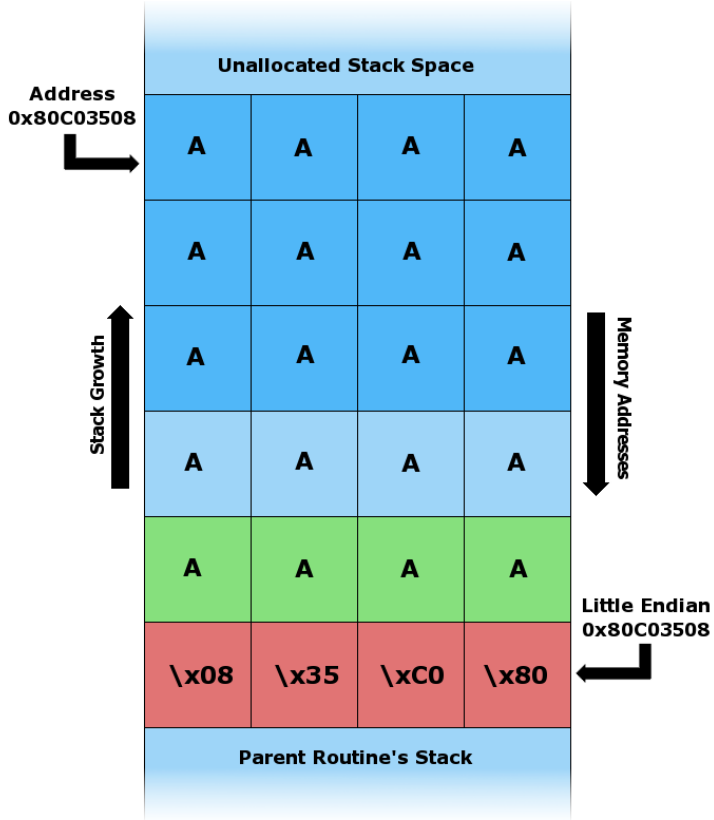
Primitiva di attacco: buffer overflow

- 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



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

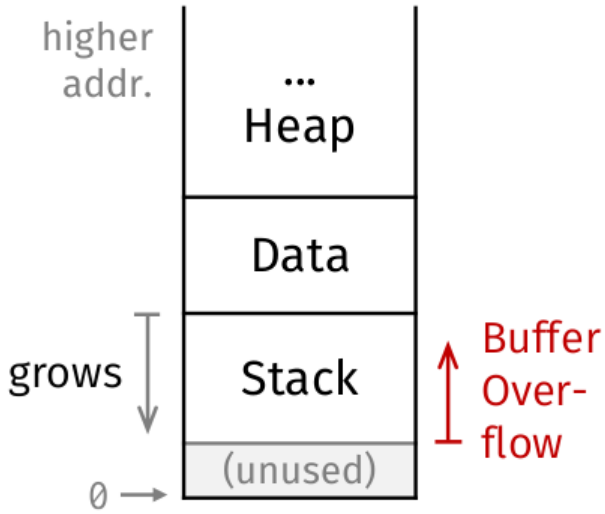


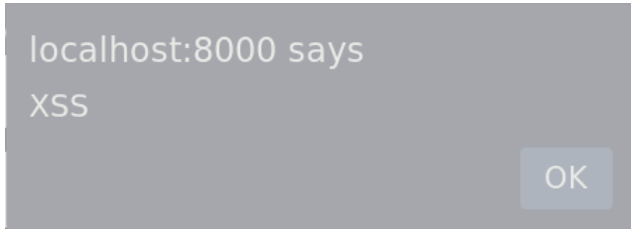
Sovrascrivere dati "costanti"

```
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 **safe_script** si possono eseguire comandi JavaScript arbitrari
  - **XSS** nel browser
  - **RCE** con Node.js

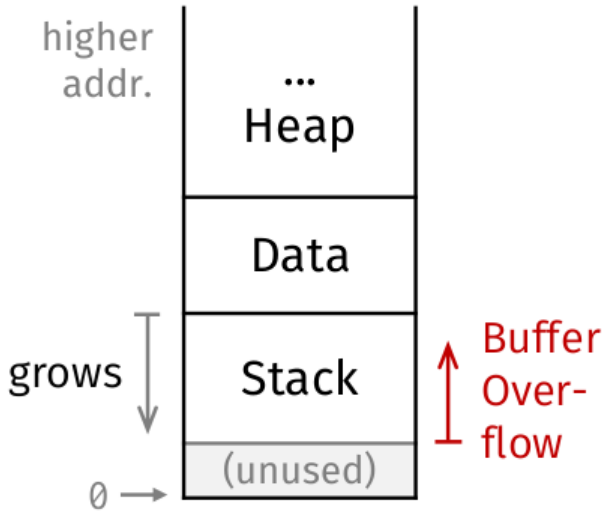


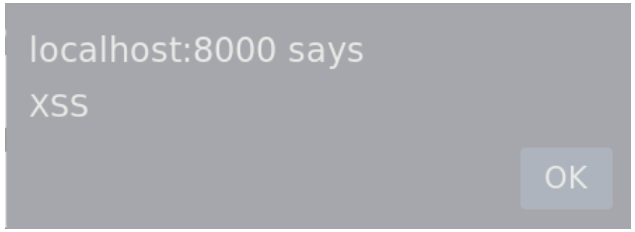


Chiamando **vuln()** con la stringa "..... alert('XSS')"

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 **safe_script** si possono eseguire comandi JavaScript arbitrari
  - **XSS** nel browser
  - **RCE** con Node.js

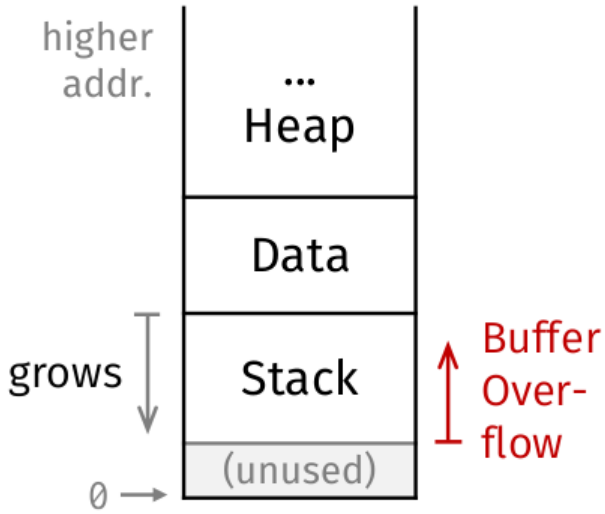


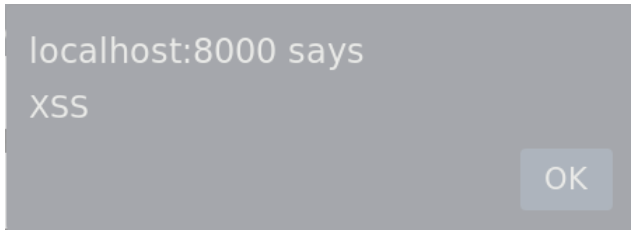


Chiamando **vuln()** con la stringa "...alert('XSS')"

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 **safe_script** si possono eseguire comandi JavaScript arbitrari
  - **XSS** nel browser
  - **RCE** con Node.js

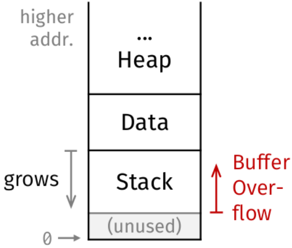




Chiamando **vuln()** con la stringa "...alert('XSS')"

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 **safe_script** si possono eseguire comandi JavaScript arbitrari
  - **XSS** nel browser
  - **RCE** con Node.js



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

Heap overflow (a partire dallo stack)

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

```
std::string img_tag =
  "<img src='";
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 stringa **img_tag** situata nello heap utilizzando un buffer overflow
  - questo causa un attacco di tipo XSS nel browser

Heap overflow (a partire dallo stack)

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

```
std::string img_tag =
  "<img src='";
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 stringa **img_tag** situata nello heap utilizzando un buffer overflow
  - questo causa un attacco di tipo XSS nel browser

Heap overflow (a partire dallo stack)

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

```
std::string img_tag =
  "<img src='";
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 stringa **img_tag** situata nello heap utilizzando un buffer overflow
  - questo causa un attacco di tipo XSS nel browser

Heap overflow (a partire dallo stack)

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

```
std::string img_tag =
  "<img src='";
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 stringa **img_tag** situata nello heap utilizzando un buffer overflow
  - questo causa un attacco di tipo XSS nel browser

Heap overflow (a partire dallo stack)

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

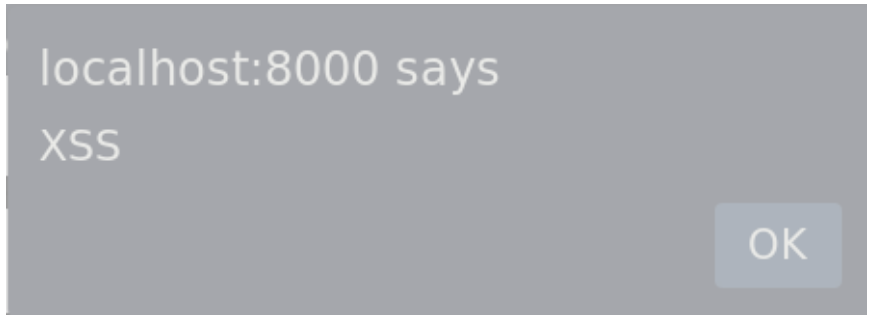
Il sito chiede all'utente un'immagine in input: non vengono fatti controlli sul tipo di file.

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

Utilizzando come payload un file contenente

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.



Heap overflow (a partire dallo stack)

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

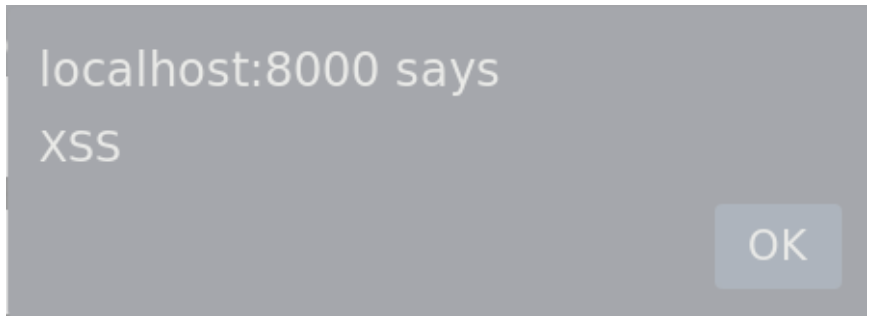
Il sito chiede all'utente un'immagine in input: non vengono fatti controlli sul tipo di file.

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

Utilizzando come payload un file contenente

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.



Heap overflow (a partire dallo stack)

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

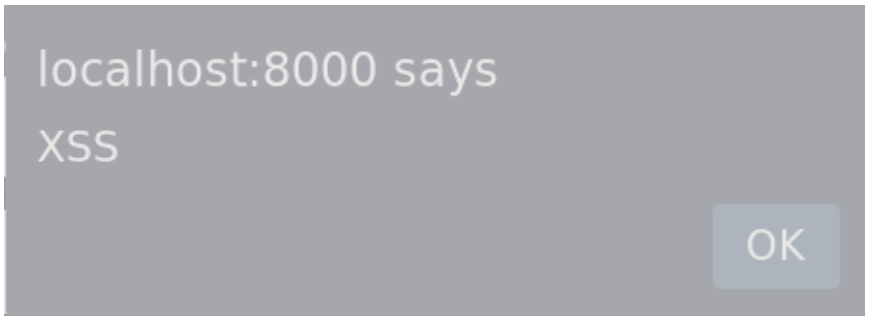
Il sito chiede all'utente un'immagine in input: non vengono fatti controlli sul tipo di file.

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

Utilizzando come payload un file contenente

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.



Heap overflow (a partire dallo stack)

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 un file contenente

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 vulnerabilità precedentemente mitigate
- Ha inoltre introdotto nel mondo web vulnerabilità legate al mondo dei binari
  - buffer/heap overflow
  - format string
  - ...
- Ha infine reso possibili nuovi tipi di attacco
  - sovrascrivere dati "costanti"
  - sovrascrivere dati di una regione a partire da un'altra

Conclusioni

- WebAssembly ha reintrodotto vulnerabilità precedentemente mitigate
- Ha inoltre introdotto nel mondo web vulnerabilità legate al mondo dei binari
  - buffer/heap overflow
  - format string
  - ...
- Ha infine reso possibili nuovi tipi di attacco
  - sovrascrivere dati "costanti"
  - sovrascrivere dati di una regione a partire da un'altra

Conclusioni

- WebAssembly ha reintrodotto vulnerabilità precedentemente mitigate
- Ha inoltre introdotto nel mondo web vulnerabilità legate al mondo dei binari
  - buffer/heap overflow
  - format string
  - ...
- Ha infine reso possibili nuovi tipi di attacco
  - sovrascrivere dati "costanti"
  - sovrascrivere dati di una regione a partire da un'altra

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"*