Robust Stack Smashing Protection for WebAssembly

Quentin Michaud (presenter) Yohan Pipereau Olivier Levillain Dhouha Ayed

2024-10-14

Section 1

Background

WebAssembly

- Overview

WebAssembly

- Overview
- Execution model

WebAssembly

- Overview
- Execution model
- Memory model

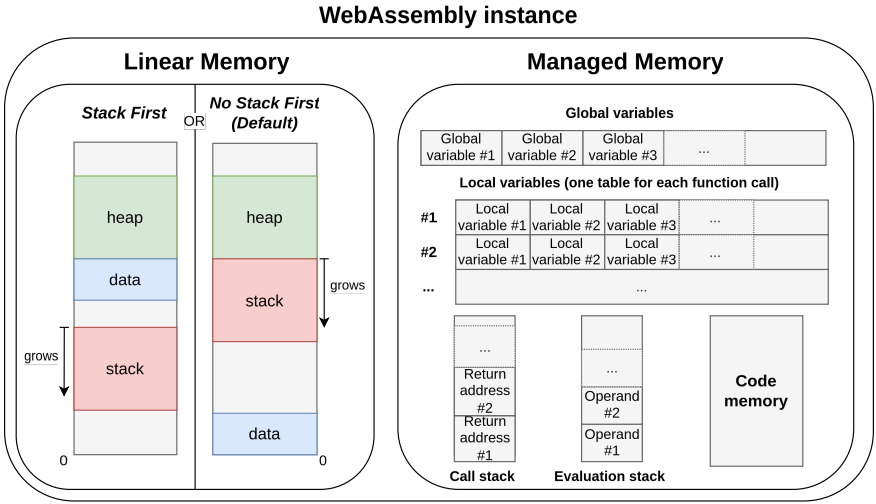


Figure 1: Memory model of WebAssembly

WASI

- Overview

WASI

- Overview
- Memory layout

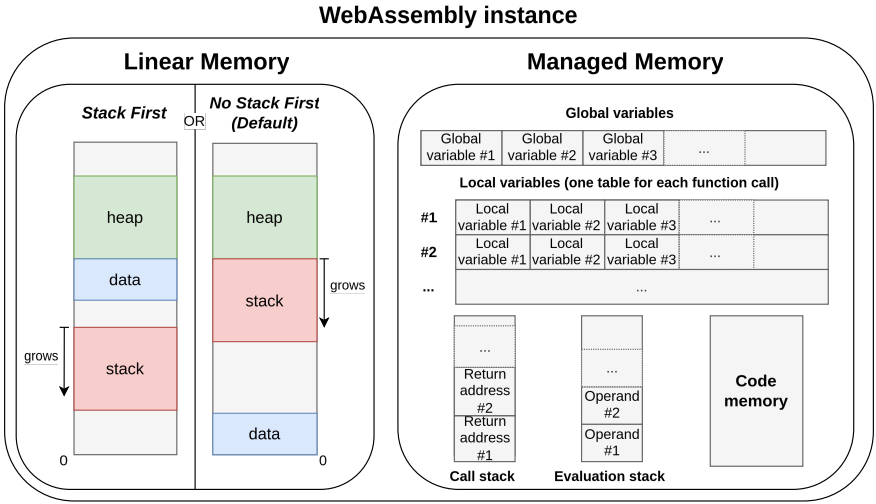


Figure 2: Memory layout of WebAssembly

Section 2

Motivation

Need of Stack Smashing Protection in WebAssembly

- Lehmann et al. show that memory attacks are possible in WebAssembly.

Need of Stack Smashing Protection in WebAssembly

- Lehmann et al. show that memory attacks are possible in WebAssembly.
- Hilbig et al. find that 4.2% of WebAssembly binaries are written in C or C++.

Need of Stack Smashing Protection in WebAssembly

- Lehmann et al. show that memory attacks are possible in WebAssembly.
- Hilbig et al. find that 4.2% of WebAssembly binaries are written in C or C++.
- Stack Smashing Protection have since been implemented in a WebAssembly / WASI toolchain

How to assess a Stack Smashing Protection implementation

Bierbaumer et al. propose a list of security properties that robust SSP implementations should satisfy:

- **P1** The canary value placed behind user-controlled buffers must be unknown to the attacker.

They also provide a framework named *CookieCrumbler* to automatically assess implementations.

How to assess a Stack Smashing Protection implementation

Bierbaumer et al. propose a list of security properties that robust SSP implementations should satisfy:

- **P1** The canary value placed behind user-controlled buffers must be unknown to the attacker.
- **P2** The reference value is placed at a location in memory that is distinct from the location of canaries and ideally mapped read-only.

They also provide a framework named *CookieCrumbler* to automatically assess implementations.

How to assess a Stack Smashing Protection implementation

Bierbaumer et al. propose a list of security properties that robust SSP implementations should satisfy:

- **P1** The canary value placed behind user-controlled buffers must be unknown to the attacker.
- **P2** The reference value is placed at a location in memory that is distinct from the location of canaries and ideally mapped read-only.
- **P3** If a canary is corrupted, the program execution terminates immediately (or as soon as possible) without accessing any attacker controlled data.

They also provide a framework named *CookieCrumbler* to automatically assess implementations.

Section 3

Security analysis

Evaluating the generation of canaries

```
void __init_ssp(void *entropy)
{
if (entropy) memcpy(&__stack_chk_guard, entropy, sizeof(uintptr_t));
else __stack_chk_guard = (uintptr_t)&__stack_chk_guard * 1103515245;
```

- **M1** We block the getrandom Linux syscall that is commonly used to acquire randomness on Linux.
- **M2** In addition to M1, we block all access to the /dev folder on Linux, which contains the other common source of randomness, the /dev/urandom block device.

Test configuration	Baseline	**M1**	**M2**
wasmtime (v19.0.1)	✓	✓	crash
wasmedge (v0.13.5)	✓	✓	✓
wasmer (v4.2.8)	✓	✓	crash
iwasm (v1.3.2)	✓	✗	✗
wasm3 (v0.5.0)	✓	✗	✗
wasmi (v0.31.2)	✓	✓	crash

Figure 3: Results of the randomness generation evaluation

Evaluating the SSP reference value location

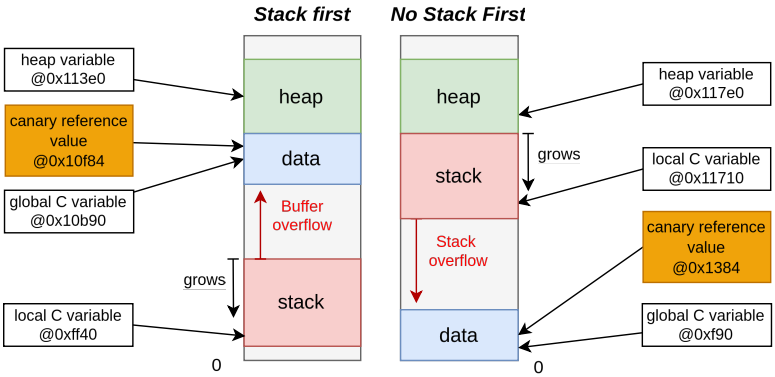


Figure 4: Results of *Cookiecrumbler*

Evaluating quick termination on canary corruption

```
(func $__stack_chk_fail (type 7)
unreachable
unreachable
)
```

Section 4

Remediation

Remediation

- Implementing protection against weak randomness: crashing on failure

Remediation

- Implementing protection against weak randomness: crashing on failure
- Implementing overwrite protection: using a global variable

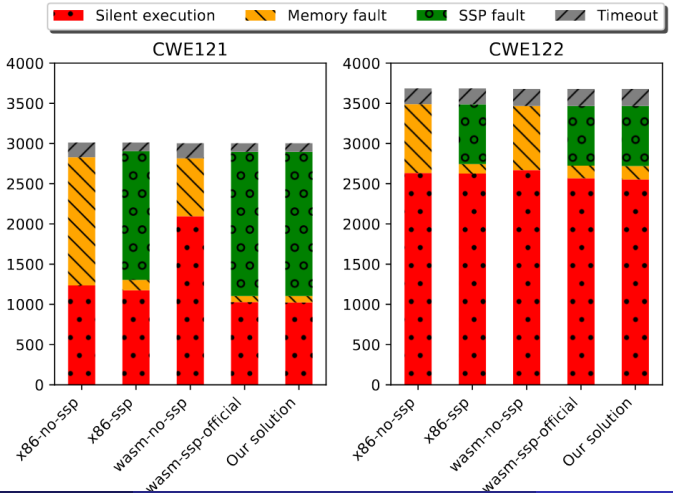
Remediation

- Implementing protection against weak randomness: crashing on failure
- Implementing overwrite protection: using a global variable
- Remediations implemented in LLVM + wasi-libc and evaluated

Section 5

Evaluation

Evaluation


Silent execution
Memory fault
SSP fault
Timeout
CWE121
CWE122
4000
3500
3000
2500
2000
1500
1000
500
0
x86-no-ssp
x86-ssp
wasm-no-ssp
wasm-ssp-official
Our solution

Section 6

Conclusion

Conclusion

- Stack Smashing Protection does have an impact on the security of WebAssembly...

Conclusion

- Stack Smashing Protection does have an impact on the security of WebAssembly...
- ... and its security in the context of WebAssembly have now been evaluated, with a robust implementation provided and evaluated.

Conclusion

- Stack Smashing Protection does have an impact on the security of WebAssembly...
- ... and its security in the context of WebAssembly have now been evaluated, with a robust implementation provided and evaluated.
- We believe that Stack Smashing Protection should become a default in all WebAssembly binaries in the future.

Thank you for your attention

Questions?