WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches

Marco Antonio Corallo

University of Pisa

Course of ICT Risk Assessment

Overview

WarpAttack exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks.

Overview

WarpAttack exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks.

- Introduce the vulnerability

Overview

WarpAttack exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks.

- Introduce the vulnerability
- Present the mechanism underlying the attack and practical PoC

Overview

WarpAttack exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks.

- Introduce the vulnerability
- Present the mechanism underlying the attack and practical PoC
- Evaluation and mitigations

Overview

Overview

Introduction

C/C++ software are prone to memory corruption bugs that often enable code execution attacks.

Introduction

C/C++ software are prone to memory corruption bugs that often enable code execution attacks.

- ASLR

Introduction

C/C++ software are prone to memory corruption bugs that often enable code execution attacks.

- ASLR
- Canaries

Introduction

C/C++ software are prone to memory corruption bugs that often enable code execution attacks.

- ASLR
- Canaries
- **CFI**

Introduction

C/C++ software are prone to memory corruption bugs that often enable code execution attacks.

- ASLR
- Canaries
- **CFI**

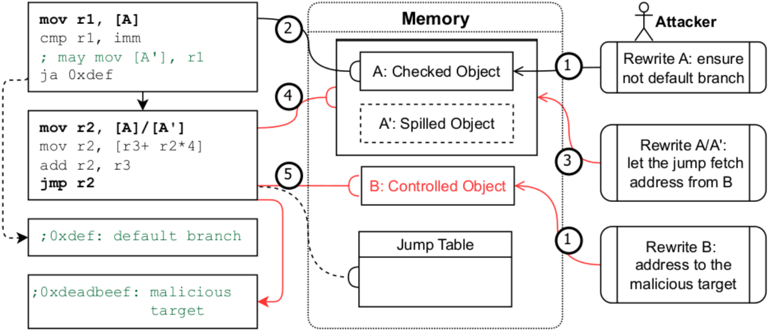
Not a total solution!

Introduction

Compiler-introduced double-fetch of a bound-checked indirect jump with a jump table

Compiler-introduced double-fetch of a bound-checked indirect jump with a jump table

30,000-foot view



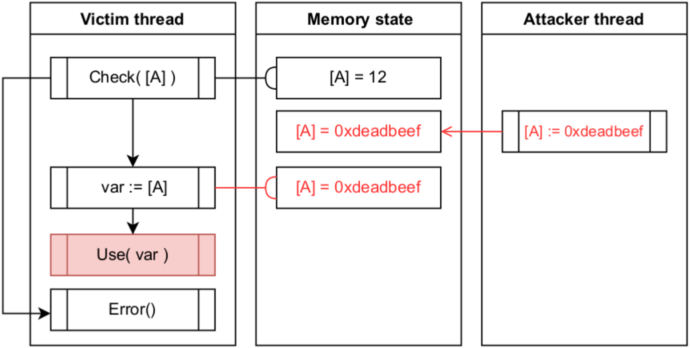
Overview

Double-Fetch

"Double-fetch bugs occur when a privilege system reads a variable multiple times, but the fetched value is inconsistent due to concurrency issues"

Double-Fetch

"Double-fetch bugs occur when a privilege system reads a variable multiple times, but the fetched value is inconsistent due to concurrency issues"



Double-Fetch

"Double-fetch bugs occur when a privilege system reads a variable multiple times, but the fetched value is inconsistent due to concurrency issues"

- CVE-2008-2252: Windows
- CVE-2005-2490: Linux kernel
- CVE-2015-1420: Linux kernel (Android)
- CVE-2022-48357: Huawei products

⋮

Bound-Checked Jump Table

The code for a jump table lookup consists of

- a bound check;
- an indirect jump (whose address is computed with the checked value).

```
;switch(obj->type) {
;   case 0:
; ...
;   default:
; ...
;}
mov     rax, rdi
mov     eax, DWORD PTR [rdi+0x30]
add     eax, 0xffffffff
cmp     eax, 0x11 ;the bound check
ja      401163  ;default branch
lea     rdi, [rax+0x30]
jmp     QWORD PTR [rax*8+0x402008]
```

Assumptions

Adversarial Capabilities

- Arbitrary read-write

Defensive Assumptions

Assumptions

Adversarial Capabilities

- Arbitrary read-write
- Thread control

Defensive Assumptions

Assumptions

Adversarial Capabilities

- Arbitrary read-write
- Thread control
- Gadgets: switch jump table with a compiler-introduced double-fetch

Defensive Assumptions

Assumptions

Adversarial Capabilities

- Arbitrary read-write
- Thread control
- Gadgets: switch jump table with a compiler-introduced double-fetch

Defensive Assumptions

- Non-Executable Memory

Assumptions

Adversarial Capabilities

- Arbitrary read-write
- Thread control
- Gadgets: switch jump table with a compiler-introduced double-fetch

Defensive Assumptions

- Non-Executable Memory
- Randomization

Assumptions

Adversarial Capabilities

- Arbitrary read-write
- Thread control
- Gadgets: switch jump table with a compiler-introduced double-fetch

Defensive Assumptions

- Non-Executable Memory
- Randomization
- Control Flow Protection

Overview

PoC Exploit

The victim: a complex and realistic target available for all common operating systems

PoC Exploit

The victim: a complex and realistic target available for all common operating systems

Version: 106.0.1
Built by: GCC 12.1.1

PoC Exploit

- Gain arbitrary read/write capability

PoC Exploit

- Gain arbitrary read/write capability: CVE-2022-26485

PoC Exploit

- Gain arbitrary read/write capability: `out-of-bound` that grants arbitrary read/write capabilities through *ArrayBuffers*.

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
    1. the address of the fetched object

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
  1. the address of the fetched object
  2. the address of the victim jump table

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
  1. the address of the fetched object
  2. the address of the victim jump table
  3. the address of one writeable memory region

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
  1. the address of the fetched object
  2. the address of the victim jump table: `.rodata` section of `libxul.so`
  3. the address of one writeable memory region

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
  1. the address of the fetched object
  2. the address of the victim jump table: `.rodata` section of `libxul.so`
  3. the address of one writeable memory region: `.bss` section of `libxul.so`

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
  1. the address of the fetched object
  2. the address of the victim jump table: `.rodata` section of `libxul.so`
  3. the address of one writeable memory region: `.bss` section of `libxul.so`

out-of-bound `Uint8Array` → leak ArrayBuffer._elements → libxul base address → __environ → stack base address

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps: *lightweight binary analysis tool*

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps
- Reaching gadget code

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps
- Reaching gadget code: libxul's `TraceJitActivation()`

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps
- Reaching gadget code: libxul's `TraceJitActivation()`
  `document.getElementById('textarea').value += x`

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps
- Reaching gadget code
- Orchestrate the thread scheduling to win the data race

PoC Exploit

- Gain arbitrary read/write capability
- Leak ASLR bases for both libxul.so and the stack
- Find double-fetch gadgets of bound-checked indirect jumps
- Reaching gadget code
- Orchestrate the thread scheduling to win the data race
- Overwrite the checked object and hijack the control flow

Overview

Gadget Code Detection

Binary Analysis tool

- designed for offensive purposes

Gadget Code Detection

Binary Analysis tool

- designed for offensive purposes
- based on Radare2

Gadget Code Detection

Binary Analysis tool

- designed for offensive purposes
- based on Radare2
- heuristics for x86/64 architectures

Gadget Code Detection

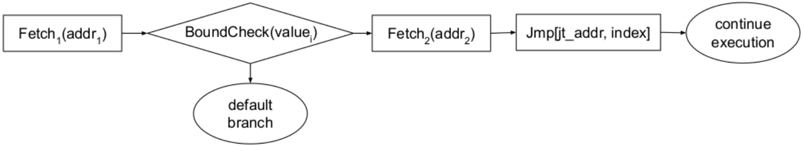
Binary Analysis tool

- designed for offensive purposes
- based on Radare2
- heuristics for x86/64 architectures
- binary pattern: combination of vulnerable elements

Gadget Code Detection

Binary Analysis tool

- designed for offensive purposes
- based on Radare2
- heuristics for x86/64 architectures
- binary pattern: combination of vulnerable elements



Overview

Benchmarks

Intel(R) Core(TM) i7-10700 CPU (8 cores) @ 2.90GHz with 32GB of memory and Fedora 36

Benchmarks

Intel(R) Core(TM) i7-10700 CPU (8 cores) @ 2.90GHz with 32GB of memory and Fedora 36

Algorithm 1 Measuring our PoC's success rate.

```
function EXPERIMENT
    while 2000 times do
        repeat                          ▷ one attempt
            attack()        ▷ run the race to overwrite
        until 20s have passed
    end while
end function
```

Benchmarks

Intel(R) Core(TM) i7-10700 CPU (8 cores) @ 2.90GHz with 32GB of memory and Fedora 36

TABLE 1. DIFFERENT SUCCESS RATES BY TUNING NUMBER OF CORES AND NUMBER OF ATTACKER THREADS (IN 2000 RUNS).

#Core	#Attacker Threads		
	1	3	7
1	0	0	0
4	0.05%	0.25%	0.2%
8	0.15%	0.15%	0.45%

Real-World Impact

- Do compiler-introduced double-fetch gadgets exist in real programs?

Real-World Impact

- Do compiler-introduced double-fetch gadgets exist in real programs?
- Which compiler is affected by such situation?

Real-World Impact

- Do compiler-introduced double-fetch gadgets exist in real programs?
- Which compiler is affected by such situation?
- Which CFI implementation is vulnerable to WarpAttack?

Real-World Impact

- Do compiler-introduced double-fetch gadgets exist in real programs?
- Which compiler is affected by such situation?
- Which CFI implementation is vulnerable to WarpAttack?
- What architectures are affected by WarpAttack?

TABLE 2. STATISTICS OF DOUBLE-FETCH GADGETS IN THE WILD. WE EXCLUDED APACHE FOR MAC OS BECAUSE WE FAIL TO FIND THE CORRECT PRE-COMPILED VERSION FOR INTEL MAC OS.

Program	Fedora	Debian	Ubuntu	Windows	Mac OS
Chrome	1024	16	23	24	16
Firefox	616	659	31	0	29
Apache	15	17	16	0	-
JVM	0	0	0	0	1
7-zip	24	24	24	0	0
Texstudio	8	9	9	230	20
Total	1687	725	103	254	66

A Study of Compilers

TABLE 3. COMPILERS THAT CAN INTRODUCE EXPLOITABLE DOUBLE-FETCH PAIRED TO THEIR COMPILATION OPTIONS. THE SYMBOL "*" INDICATES CASES OBSERVED FROM REAL WORLD PROGRAMS.

Compiler	Option	double-fetch Type	version
GCC	`O1,O2,O3,Ofast`	Var. 1 (fetch-fetch)	12.1
*G++	`O1,O2,O3`	Var. 2 (fetch-spill-fetch)	12.1
Clang	`O0`	Var. 2 (fetch-spill-fetch)	14.0.*
*Clang	`O1,O2,O3`	Var. 2 (fetch-spill-fetch)	14.0.*
*Clang	`O3`	Var. 1 (fetch-fetch)	14.0.*
*Clang++	`O3`	Var. 1 (fetch-fetch)	14.0.*
MSVC	`Od`	Var. 1 (fetch-fetch)	19.32.*

Vulnerable CFI implementations

TABLE 4. CFI IMPLEMENTATIONS VULNERABLE TO OUR ATTACK.

CFI Type	Compiler	Vulnerable CFI
Compiler-based CFI	GCC	VTV [13]
	Clang	LLVM-CFI [12]
	MSVC	CFG [43]
Binary only CFI	–	Lockdown [20]

Vulnerable Architectures

TABLE 5. CONFIRMED VULNERABLE ARCHITECTURES AND INVOLVED VARIANTS AND COMPILERS.

	Variant 1 (fetch-fetch)	Variant 2 (fetch-spill-fetch)
X86/-64	GCC O1/O2/O3	Clang O0; MSVC Od
ARM 32/64	-	Clang O0; MSVC Od
RISCV 32/64	-	Clang O0
MIPS 32/64	GCC O1/O2/O3	-

Overview

Mitigations

- Avoiding Gadget code generation

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables`

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables`
  Clang, MSVC optimization level $>$ O0

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables` Clang, MSVC optimization level > O0
- Protecting Indirect Jump

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables` Clang, MSVC optimization level > O0
- Protecting Indirect Jump: dynamic checks

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables`
  Clang, MSVC optimization level $>$ O0
- Protecting Indirect Jump: dynamic checks
- Monitoring for Attack Behavior

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables` Clang, MSVC optimization level > O0
- Protecting Indirect Jump: dynamic checks
- Monitoring for Attack Behavior: spawning several threads, constantly writing a certain memory site; crashes; ...

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables` Clang, MSVC optimization level $>$ O0
- Protecting Indirect Jump: dynamic checks
- Monitoring for Attack Behavior: spawning several threads, constantly writing a certain memory site; crashes; ...
- Making compilers aware of sensitive code

Mitigations

- Avoiding Gadget code generation: GCC's `-fno-switch-tables` Clang, MSVC optimization level $>$ O0
- Protecting Indirect Jump: dynamic checks
- Monitoring for Attack Behavior: spawning several threads, constantly writing a certain memory site; crashes; ...
- Making compilers aware of sensitive code: annotating security-related code

Based on the work of

J. Xu, L. Di Bartolomeo, F. Toffalini, B. Mao, M. Payer

Thank You.