What is “Retpoline?”
What is retpoline?
Objective

The objective is to control how a CPU guesses which instructions it should execute next.
Overview

• Computer Architectures 101
• The Problem
• What is “Retpoline”
• In-Depth Description of two different “Retpoline Sequences”
• Disadvantages
• Questions
A CPU executes a series of instructions, one at a time.
101: The “jmp” Instruction

• Jumps to a provided “address” and continues execution.

Instruction Stream

Instruction

jmp address

...
101: The “call” Instruction

- Pushes the “Return Address” to the stack
- Jumps to a provided “address” and continues execution.
101: The “ret” Instruction

- Pops the “Return Address” from the stack
- Jumps to the popped “Return Address” and continues execution.

What is "Retpoline"
101: Direct vs Indirect

• Direct: Calls a “hardcoded” address that is known at compile-time.
  1. jmp address ; jump directly to address
  2. call address ; call address directly

• Indirect: Jumps to an address stored in a register or memory and is only known at run-time.
  1. jmp rax ; jump to the address stored in RAX
  2. call [rax] ; call the address stored in memory pointed to by RAX
  3. ret

• The key difference is with direct, the CPU knows immediately what address to jump to. With indirect, the CPU must determine which address to jump to, which takes additional time.
101: The Instruction Pipeline

- When an instruction needs to execute, the CPU must perform several operations per instruction.

![Instruction Pipeline Diagram]

Instruction #1
- Fetch Instruction
- Decode Instruction
- Fetch Instruction Inputs
- Execute Instruction
- Save Results

Instruction #2
- Fetch Instruction
- Decode Instruction
- Fetch Instruction Inputs
- Execute Instruction
- Save Results

What is "Retpoline"
101: The Instruction Pipeline

- The instruction pipeline executes instructions in parallel, attempting to always keep each step in the pipeline busy.

If instruction #1 is not complete, how does the CPU know if instruction #2 should be executed?
A CPU’s speculative execution engine attempts to fill the instruction pipeline by “guessing” which instruction will execute next. If the CPU’s guess is right, execution continues without interruption. If the CPU’s guess is wrong, the CPU must undo the execution of the instruction... perfectly.
101: The “call” Instruction (Updated)

- The call instruction also pushes the “Return Address” to a hidden stack called the Return Stack Buffer or RSB (RAS on AMD)
The speculative execution engine gets its “guess” from the RSB when a “ret” instruction is executed.

The RSB is a lot faster than the regular stack but might hold an incorrect “guess”.

101: The “ret” Instruction (Updated)
The Problem With Speculation

• Most speculative execution engines *do not perfectly undo their execution*. For example, some CPUs leave the results of speculative execution in the cache.

• If the CPU is *convinced* speculatively executes an instruction that operates on a “secret”, an attacker might be able to read this secret using a side-channel attack to access the side effect.

• This is the foundation for a Spectre Attack

• *We need a way to control how the CPU speculatively executes indirect “jmp” and “call” instructions.*
What is “Retpoline”

• Retpoline stands for *return and trampoline*.

• The goal of a retpoline sequence is to *control how the CPU performs speculation* when executing “jmp” and “call”.

• Does not apply to “far” jumps, or direct jumps.
Indirect Branch Retpoline

• The following code implements a retpoline sequence for the “jmp” instruction:

1. jmp [address]  →  1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3

• Let's examine this in more detail
Indirect Branch Retpoline: Step #1

- Perform a direct “call”. This performs three different actions:
  - Pushes the address of “capture_spec” to the stack
  - Pushes the address of “capture_spec” to the RSB
  - Jumps to “set_up_target” and continues execution.

1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3
Indirect Branch Retpoline: Step #2

- Change the return address on the stack from "capture_spec" to "address"
- This does **not** change the RSB.

1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3
Indirect Branch Retpoline: Step #3

- Indirectly jump to “address” by popping “address” from the stack and calling “ret.”

1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3
The speculative execution engine pops “capture_spec” from the RSB and jumps to this address.

The capture_spec code loops forever, causing the speculative execution engine to execute beginning code endlessly.

The “pause” and “lfence” instructions are used to improve performance.

1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3
Extra Credit: Why did we add “int3” as this is not seen in the existing literature.

The answer is: because CPUs not only speculatively execute the top of the RSB, but they might also speculatively execute the next instruction after “ret”.

This is called Straight-Line speculation.

“int3” is never executed by the actual code, but since “int3” is not allowed to be speculatively executed, it prevents straight-line speculation.

1. call set_up_target
2. capture_spec:
3. pause
4. lfence
5. jmp capture_spec
6. set_up_target:
7. mov [rsp], address
8. ret
9. int3
Indirect Call Retpoline

• The following code implements a retpoline sequence for the “call” instruction:

1. call [address]  
2. next instruction

1. jmp end:
2. inner:
3. call set_up_target
4. capture_spec:
5. pause
6. lfence
7. jmp capture_spec
8. set_up_target:
9. mov [rsp], address
10. ret
11. int3
12. end:
13. call inner;
14. next instruction

Seriously?
Indirect Call Retpoline: Step #1

- Perform a direct “jmp” to the end of the block of assembly.

1. `jmp end:`
2. `inner:`
3. `call set_up_target`
4. `capture_spec:`
5. `pause`
6. `lfence`
7. `jmp capture_spec`
8. `set_up_target:`
9. `mov [rsp], address`
10. `ret`
11. `int3`
12. `end:`
13. `call inner;`
14. `next instruction`
Indirect Call Retpoline: Step #2

- Call “inner”.
  - Pushes the correct return address onto the stack
  - Pushes the correct return address onto the RSB
  - Jumps to “inner” and continues execution.

```plaintext
1. jmp end:
2. inner:
3. call set_up_target
4. capture_spec:
5. pause
6. lfsence
7. jmp capture_spec
8. set_up_target:
9. mov [rsp], address
10. ret
11. int3
12. end:
13. call inner;
14. next instruction
```
Disadvantages

- Impact to the size of an application or kernel
- Impact to performance due to the CPU having to execute more instructions
- Impact to performance due to the CPU’s speculative execution engine not being allowed to optimize indirect “jmp” and “call instructions.
What is "Retpoline"

April 1, 2021

Any Questions

Rian Quinn, Ph.D.

CTO

quinnr@ainfosec.com

www.ainfosec.com