Control Flow Integrity on RISC-V

Deepak Gupta – debug@rivosinc.com

Memory safety and control flow integrity

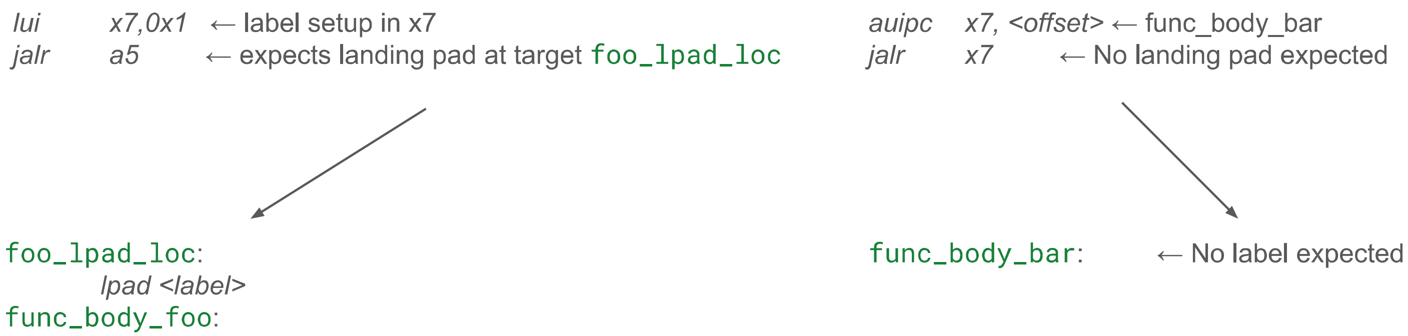
- Significant C/C++ code base in vulnerable to memory safety – [1], [2]

- Implication of memory safety issues → control flow can be subverted
  - Forward edge: Function pointers or virtual function ptr table live in RW memory
  - Return edge: Return addresses (on stack) in RW memory

[1]-https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf
[2]-https://www.chromium.org/Home/chromium-security/memory-safety/

Zicfilp - Protects forward control flow

- Zicfilp: Enforces all indirect branches must land on `lpad` (`auipc rd=x0`)
    - Except when `rs1 == (x1 | x5 | x7)`
    - Label setup in `x7` must match label encoded in `lpad` instruction on target
    - New exception (cause = 18) – software-check exception
        - *tval = 2, missing `lpad` or label didn’t match



```
lui     x7,0x1  ← label setup in x7
jalr    a5      ← expects landing pad at target foo_lpad_loc

foo_lpad_loc:
        lpad <label>
func_body_foo:
```

```
auipc   x7, <offset> ← func_body_bar
jalr    x7         ← No landing pad expected

func_body_bar:        ← No label expected
```

Zicfiss

- Zicfiss: Extends architecture with shadow stack (encoding `RWX = b010`)
  - Regular stores not allowed. Regular loads allowed.
    - Access fault on regular stores.
  - Shadow stack memory accesses strictly operate on shadow stack memory
    - SS access on RO memory → store page fault
    - SS access on RWX or XO memory or RW memory → access fault
    - `sspopchk` can raise software-check exception (*tval = 3)

```
func_main:
        lpad  <label>
        sspush x1              ← push return address on top of shadow stack
        …
        …


        ld x5, offset(sp)      ← get return address from stack
        add sp, sp, offset     ← adjust stack
        sspopchk x5            ← pop from top of shadow stack and compare with x5
        jr x5                  ← sspopchk didn't fault. Return back
```

Shadow stack & page fault

- Shadow stack is a writable memory but needs protection against stray writes.
- During fork, it becomes read-only (so that COW can be done later)
    - For mm any shadow stack access (SS load or SS store) is a COW (thus store) operation

- Following fault behavior for SS accesses
    - Read only memory – store page fault
    - Not present memory – store page fault
    - RW or RWX or X memory – access fault
        - Shadow stack instructions operating on RW* or X memory indicates fatality

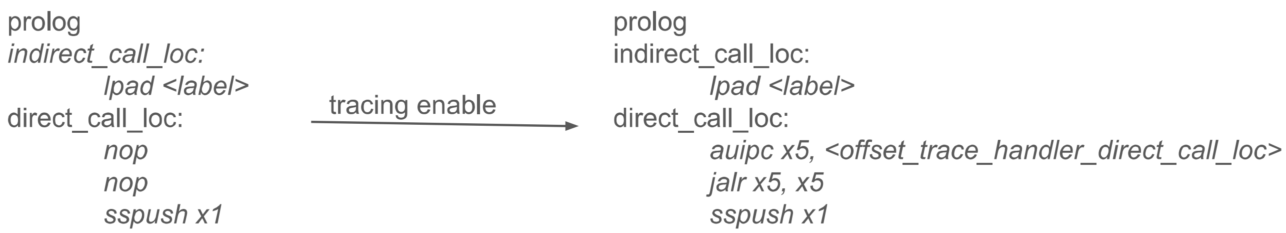
- Regular loads to shadow stack memory are allowed: useful for backtrace / debugging
- Regular stores to shadow stack memory are access fault: fatal condition and should be SIGSEGV

Runtime control-flow changes and CFI

- Text patching
    - tracing
    - breakpoints
        - probes
- eBPF
    - BPF programs JIT codegen must confirm to kernel CFI policies
    - BPF programs attach to kprobes
        - Should work as long as kprobes work

- *** Anything missed *** ?

Prolog /w CFI and tracing support



```
prolog
indirect_call_loc:
        lpad <label>
direct_call_loc:
        nop
        nop
        sspush x1
```

tracing enable

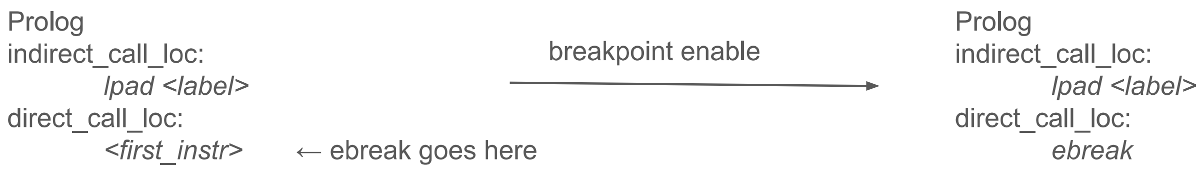
```
prolog
indirect_call_loc:
        lpad <label>
direct_call_loc:
        auipc x5, <offset_trace_handler_direct_call_loc>
        jalr x5, x5
        sspush x1
```

*** Proposal ***

- Currently tracing enable uses `jalr x5, x5` ← should work as is
  - landing pad not expected on target trampoline
  - Return saved in `x5`
  - Target trampoline uses `x5` on return path (`rs1 == x5` doesn't require landing pad)

- `lpad` can't be patched and is always executed

Breakpoints and text patch /w CFI



```
Prolog
indirect_call_loc:
        lpad <label>
direct_call_loc:
        <first_instr>      ← ebreak goes here
```

breakpoint enable

```
Prolog
indirect_call_loc:
        lpad <label>
direct_call_loc:
        ebreak
```

- Setting breakpoint can't patch `lpad`, subsequent instruction is patched
- Normal breakpoint handling is followed

kprobes and kretprobes

kprobes

- Similar to breakpoint handling

kretprobe: probes on function returns

- Installs a kprobe on function entry
- kprobe handler does `pt_regs->ra` = `arch_rethook_trampoline`
    - Saves away original `ra`
- `arch_rethook_trampoline` gets called on return and calls retprobes
    - Eventually does `jr` to original `ra`
- None of this violates Zicfilp or Zicfiss

Shadow stack: protection flags and creation

Memory (mmap) protection flags and corresponding VMAs

- PROT_READ → VM_READ
- PROT_WRITE → (VM_READ | VM_WRITE)
- PROT_SHADOWSTACK – new protection flag for memory mapping
    - PROT_SHADOWSTACK → Only (VM_WRITE)
- x86 (and aarch64 too) have introduced VM_SHADOW_STACK (stealing VM_ARCH_5 bit)
- On riscv `#define VM_SHADOW_STACK VM_WRITE`

User control on shadow stack creation

*** Proposal ***

- Shadow stack is dedicated to store return addresses. Not worth it to have protection flag exposed to user
- x86 already have `map_shadow_stack` in mainline. aarch64 following same. ← RISCV to do same

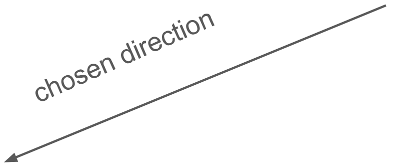
LKML discussions on topic

- https://lore.kernel.org/lkml/20230822-arm64-gcs-v5-11-9ef181dd6324@kernel.org/
- https://lore.kernel.org/lkml/20230613001108.3040476-15-rick.p.edgecombe@intel.com/
- https://www.spinics.net/lists/arm-kernel/msg1070930.html
- https://lore.kernel.org/lkml/20230613001108.3040476-35-rick.p.edgecombe@intel.com/

RISC-V user mode CFI – enabling

- Kernel can’t assume about inbuilt CFI support in all object files in address space
- Decision to enable shadow stack (SS) and landing pad (LP) is left to `ld.so` in user mode
  - Following x86 and aarch64 direction

*** Two paths here ***



ld.so starts life without SS and LP
- invoke prctls to enable CFI if all objects support CFI

ld.so starts life with SS and LP
- invoke prctls to disable CFI if any object doesn’t support CFI

LKML discussions on topic

- https://lore.kernel.org/all/20220130211838.8382-1-rick.p.edgecombe@intel.com/
- https://lkml.iu.edu/hypermail/linux/kernel/2303.1/04556.html
- https://lore.kernel.org/lkml/CAHk-=wgP5mk3poVeejw16Asbid0ghDt4okHnWaWKLBkRhQntRA@mail.gmail.com/

RISC-V user mode CFI – glibc enabling

Tunability options

- Some application may still want to disable CFI
- Some applications may want CFI but don’t know if all dependent objects have CFI support or not
  - May want to start application with CFI support but may want to disable if dlopen to non-CFI object file
  - May want to start application with CFI support and want to exit if dlopen to non-CFI object file

*** Proposal ***

- Follow x86 glibc tunables for shadow stack and indirect branch tracking (https://www.gnu.org/software/libc/manual/html_node/Hardware-Capability-Tunables.html)
- glibc.cpu.riscv_lp for landing pads and glibc.cpu.riscv_ss for shadow stack
  - On → Strict on and any dlopen to a shared library with no cfi support leads to exit
  - Off → Off irrespective of ELF bit marker or shared libraries
  - Permissive → Any incoming object in address space with no support will turn the feature off

Discussion / Q&A