CFI, SFI, and All That Jazz

Benjamin Lim
@jarsp

“I want to run **untrusted** (but not actively malicious) code...

“I want to run **untrusted** (but not actively malicious) code...

...but **VMs/containers** are **insufficient/unsupported**”

Why?

… You want to **run code in the kernel**

… You are on an **embedded system** without the appropriate support

… You are a Cyber Reasoning System and the **humans** didn’t give you any other choice

… You want it as a **defense-in-depth** measure

What we'll be covering

- Overview of techniques to protect and sandbox code, namely:
- **Software Fault Isolation (SFI)**, *or*, how to protect running code from other code
- **Control Flow Integrity (CFI)**, *or*, how to protect running code from itself

Software Fault Isolation

- Prevent untrusted code from escaping sandbox
- Prevent untrusted code from tampering with trusted components

Portable Native Client (PNaCl)

- Google's implementation of SFI (IL -> asm)
  - x86, AMD64, ARM
- WebAssembly precursor
- Somewhere between -9% (!) to `big%` overhead
- Performance hit mitigated by speculative execution
- Implemented with modified GCC for AMD64

Portable Native Client (PNaCl)

- PNaCl setup:
  - Restrict to 4GB of addressable memory (ARM32 source compatibility)
  - Reserved register (r15) to reference start of untrusted address space
  - Unmapping 10 x 4GB of memory on both sides of untrusted address space
- Relies on page fault mechanism and zeroing behavior of 32-bit arithmetic

Portable Native Client (PNaCl)

```
mov %rax, %rsp                mov %eax, %esp
                              lea %(r15,%rsp,1), %rsp


add $0x8, %rcx                add $0x8, %ecx
mov %eax, $disp(,%rcx,scale)  mov %eax, $disp(%r15,%ecx,scale)


mov $disp(%rsp), %eax         mov $disp(%rsp), %eax
```

Are we done?

no

Portable Native Client (PNaCl)

- Statically link code, must be read-only
- Bundle and align valid instructions to 32 bytes
- Indirection only to start of bundle

```
jmp %rdx
```

```
and $0xffffffe0, %edx
lea (%r15,%edx,1), %rdx
jmp *%rdx
```

Verified SFI

- Similar work (Kroll et. al. ‘14)
- Formally specify and guarantee SFI properties of generated code
  - Memory safety
  - Ideally provable functional correctness
  - Pain and Suffering
- Proof using a proof assistant (F*, Coq) to produce a certificate/extracted program

Typical Exploit A

- Application accepts user input without proper validation
- Corruption of critical structures eventually results in code execution
- Code execution requires control over control flow
- Targets of interest:
  - Indirect branches and calls
  - Saved return addresses

Control Flow Integrity

- Force control along ‘benign’ routes
- Various granularities possible, leading to differing amounts of overhead
- Static analysis to recover CFG
- Compile-time/binary instrumentation to enforce

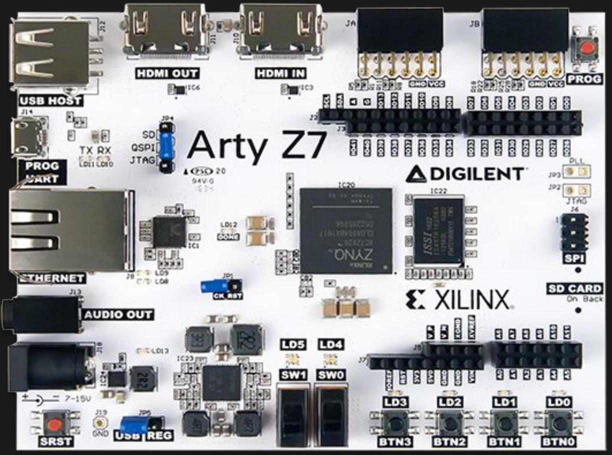
LLVM CFI

- Lightweight forward CFI
  - Virtual Calls
  - Indirect Function Calls
- Backwards CFI planned
  - Return Elision (?) for leaf functions
  - Explicit call-site checking
  - etc.
- Shadow Stack option

```
ca7fbb:       mov    (%rdi),%rcx
ca7fbe:       lea
0x7fb42c3(%rip),%rdx
ca7fc5:       mov    %rcx,%rax
ca7fc8:       sub    %rdx,%rax
ca7fcb:       rol    $0x3d,%rax
ca7fcf:       cmp    $0x17f,%rax
ca7fd5:       ja     ca8511
ca7fdb:       lea
0x6f70bc0(%rip),%rdx
ca7fe2:       testb
$0x10,(%rax,%rdx,1)
ca7fe6:       je     ca8511
ca7fec:       callq  *0x98(%rcx)
  [...]
ca8511:       ud2
```

MITRE Embedded CTF 2019

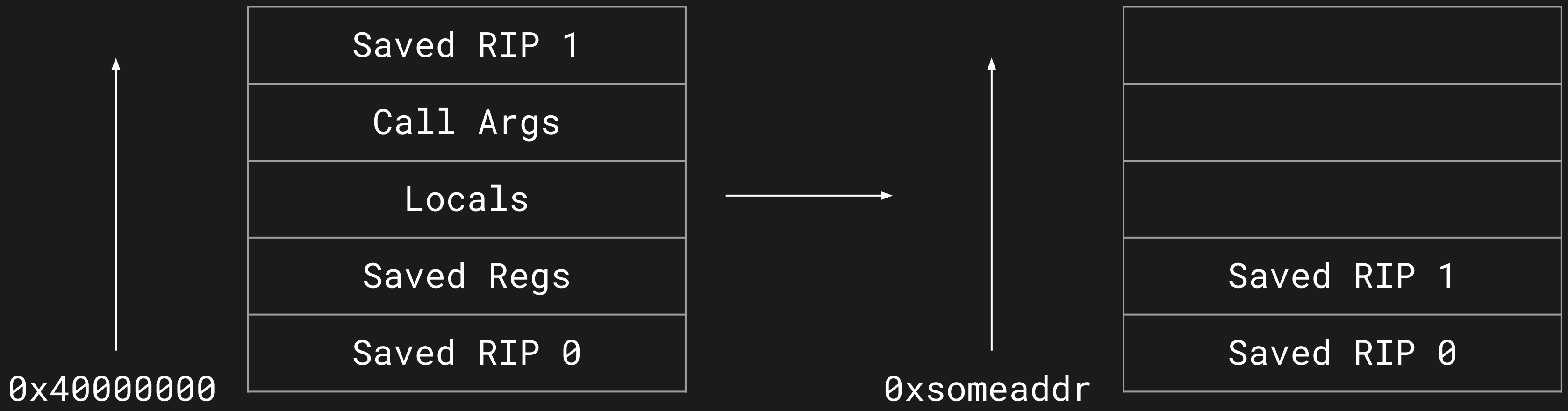
- Implement a simple ‘game console’
- (Presumably) Vulnerable C/C++ game binaries
- Assume some basic ‘sanity’ of binaries
- Prevent plaintext dump of game
- No criteria on game performance!



Shadow Stack

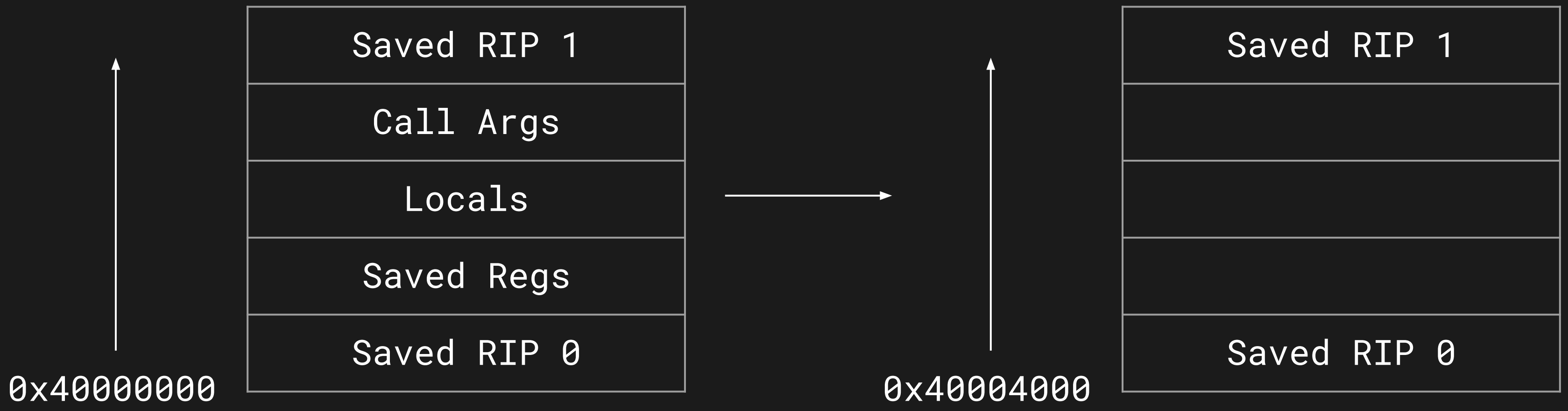
- Calls should generally be paired with returns
- Maintain a separate ‘shadow’ stack
- On return, abort if addresses do not match up
- Intel Control-Flow Enforcement Technology (CET) proposal

Shadow Stack



‘Classic’ Shadow Stack

Shadow Stack



Saved RIP 1

Call Args

Locals

Saved Regs

Saved RIP 0

0x40000000

Saved RIP 1

Saved RIP 0

0x40004000

‘Sparse’ Shadow Stack

Shadow Stack Implementation

- Dynamic instrumentation framework
  - Intel PinTools: x86, AMD64
  - DynamoRIO: x86, AMD64, ARM
- ARM-to-ARM JIT
- Hooks
  - Basic block creation
  - Signals
  - Syscalls
  - ...

Shadow Stack Implementation

- Basic idea:
  - Maintain a (thread-local) array of return addresses
  - Instrument call and return instructions to push and pop return addresses

```
if (tls->shadow_count >= MAX_SHADOW_SIZE)
{
    DERR("Call stack depth exceeded.\n");
    dr_abort();
}

res = (next_h << 16) + next_l;
tls->shadow_stack[tls->shadow_count].lr = res;
tls->shadow_count++;
```

Shadow Stack Implementation

- What is a call and what is a return?
  - ARM calls: `bl, blx`
  - ARM returns: `bx lr, (ldr pc, [sp], #4), ldmia {...,pc}`
- When to instrument?
  - Before calls (at start of function is possible but harder)
  - Before returns (to catch bad returns)

Shadow Stack Implementation

- Dynamorio/ARM specific issues
  - DynamoRIO thinking that PLT dispatch is return
    - Special case to eliminate this
  - `ldrex/strex`
    - Usually not a problem
  - `ite`
    - Insert instrumentation before `ite` block
    - Should emulate

Shadow Stack Implementation

- Dynamorio/ARM specific issues
  - ite/predicated returns
    - Instrumentation checks if return occurs

```
dr_get_mcontext(drcontext, &mc);
cpsr = mc.cpsr;
N = ((cpsr & 0x80000000) != 0);
Z = ((cpsr & 0x40000000) != 0);
C = ((cpsr & 0x20000000) != 0);
V = ((cpsr & 0x10000000) != 0);
switch (exit_pred) {
    case DR_PRED_EQ:
        cond = Z;
        break;
...
if (!cond) return;
```

Shadow Stack Implementation

- Program loading breaks
  - Extract address of main from CRT code
  - Instrument but don’t execute until main reached

```
10448:      f8df c010    ldr.w   ip, [pc, #16]    ; 1045c <_start+0x24>
1044c:      f84d cd04    str.w   ip, [sp, #-4]!
10450:      4803         ldr  r0, [pc, #12]    ; (10460 <_start+0x28>)
10452:      4b04         ldr  r3, [pc, #16]    ; (10464 <_start+0x2c>)
10454:      f01a f87a    bl   2a54c <__libc_start_main>
10458:      f01f f946    bl   2f6e8 <abort>
```

Shadow Stack Implementation

- Intentional Call/Ret mismatch
  - setjmp/longjmp
  - try/catch (C++)
- Solution:
  - Heuristically identify key functions
    - __sigsetjmp, __longjmp
    - __restore_core_regs, something else I forgot
  - Keep sp along with `lr`
  - Persist entry until function it is called from returns
  - Unroll until matching sp address

Conclusion

Cool

End.

Questions?