# Clang CFI / Shadow Stack

Good news everyone!

# Clang CFI / ShadowStack

- CFI – Control Flow Integrity
  - Also: CFG – Control Flow Guard
  - Also: CFE – Control Flow Enforcement
- Anti ROP implementation
- Forward Edge: Make sure call's are correct
- Backward Edge: Make sure return's are correct
- Available since Ubuntu 18.04

- Reference: https://github.com/dobin/clang-cfi-safestack-analysis

# Clang - Forward Edge Protection

Clang compiler options

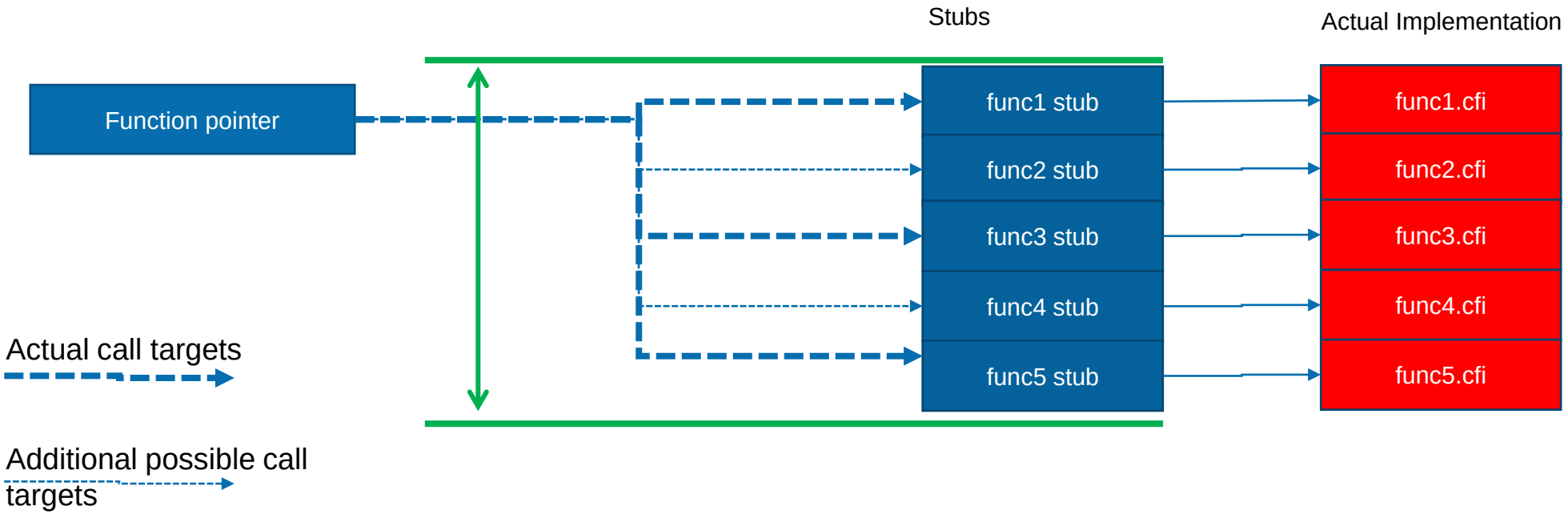
- -fsanitize=cfi –flto –fvisibility=hidden

CFI?

- Compiler identifies all starts of functions, and puts them in a whitelist
- CFI: **Add checks on every indirect branch**
  - Indirect branch: function pointer
  - Requires: Complete control flow graph (no single .o, but complete binary)
  - Function Pointers have X (1-100?) targets

https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/

# Clang - Forward Edge Protection




**pointer >= &func1_stub && pointer <= func5_stub?**

# Clang - Forward Edge Protection

Pseudocode:

```
stubSize = sizeof stubFunction = 8
distance = (functionPointer - baseStubPointer) / stubSize
if distance >= number_of_functions:
   crash()
else:
   (*functionPointer)()
```

# Clang - Forward Edge Protection

```
   0x000000000040120c <+28>:    mov    0x404048,%rax    # rax is the functionPointer we wanna call
   0x0000000000401214 <+36>:    movabs $0x401400,%rcx   # rcx is &func1_stub (base pointer)
   0x000000000040121e <+46>:    mov    %rax,%rdx        # rdx is the functionPointer we wanna call (copy from rax)
   0x0000000000401221 <+49>:    sub    %rcx,%rdx        # rdx = rdx - rcx.
                                                        # is: rdx = functionPointer - &func1_stub
                                                        # is: the distance of &func1_stub to functionPointer in bytes
   0x0000000000401224 <+52>:    mov    %rdx,%rcx        # rcx = rdx
                                                        # rcx has now the distance between the
                                                        # FunctionPointer we wanna call and the base
   0x0000000000401227 <+55>:    shr    $0x3,%rcx        # rcx >> 3: divide memory distance by 8
                                                        # each stub function is 8 bytes.
                                                        # So, the number of "stubs" between these functions in rcx
   0x000000000040122b <+59>:    shl    $0x3d,%rdx       # rdx << 0x3d. make sure the pointer is 8-byte aligned.
   0x000000000040122f <+63>:    or     %rdx,%rcx        # rcx = rcx ^ rdx
   0x0000000000401232 <+66>:    cmp    $0x4,%rcx        # check if rcx is <= 4: max addr of call target is &func5
   0x0000000000401236 <+70>:    jbe    0x40123a <bof+74>
   0x0000000000401238 <+72>:    ud2                     # rcx >= 5. Crash here
   0x000000000040123a <+74>:    callq  *%rax            # rcx <= 4. Call the functionPointer, as it is "safe"


# The function stubs look like this:
#   0x0000000000401400 <+0>:     jmpq   0x401140 <func1.cfi>
#   0x0000000000401405 <+5>:     int3
#   0x0000000000401406 <+6>:     int3
#   0x0000000000401407 <+7>:     int3
#   0x0000000000401408 <+0>:     jmpq   0x401160 <func2.cfi>
#   0x000000000040140d <+5>:     int3
#   0x000000000040140e <+6>:     int3
#   0x000000000040140f <+7>:     int3
```

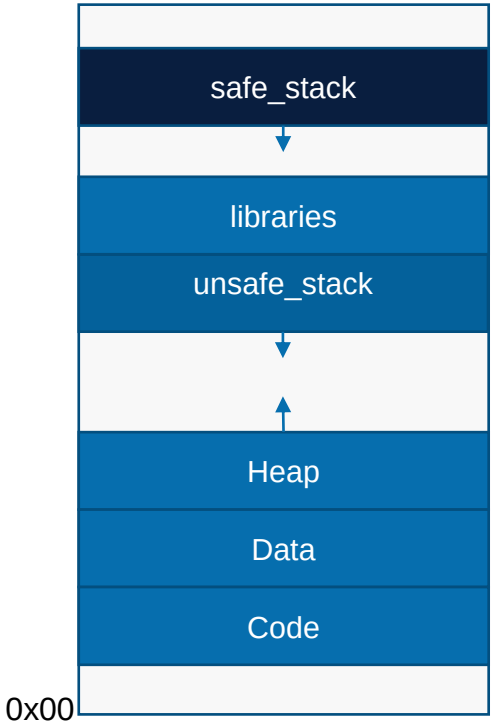
# Clang - Backward Edge Protection

SafeStack

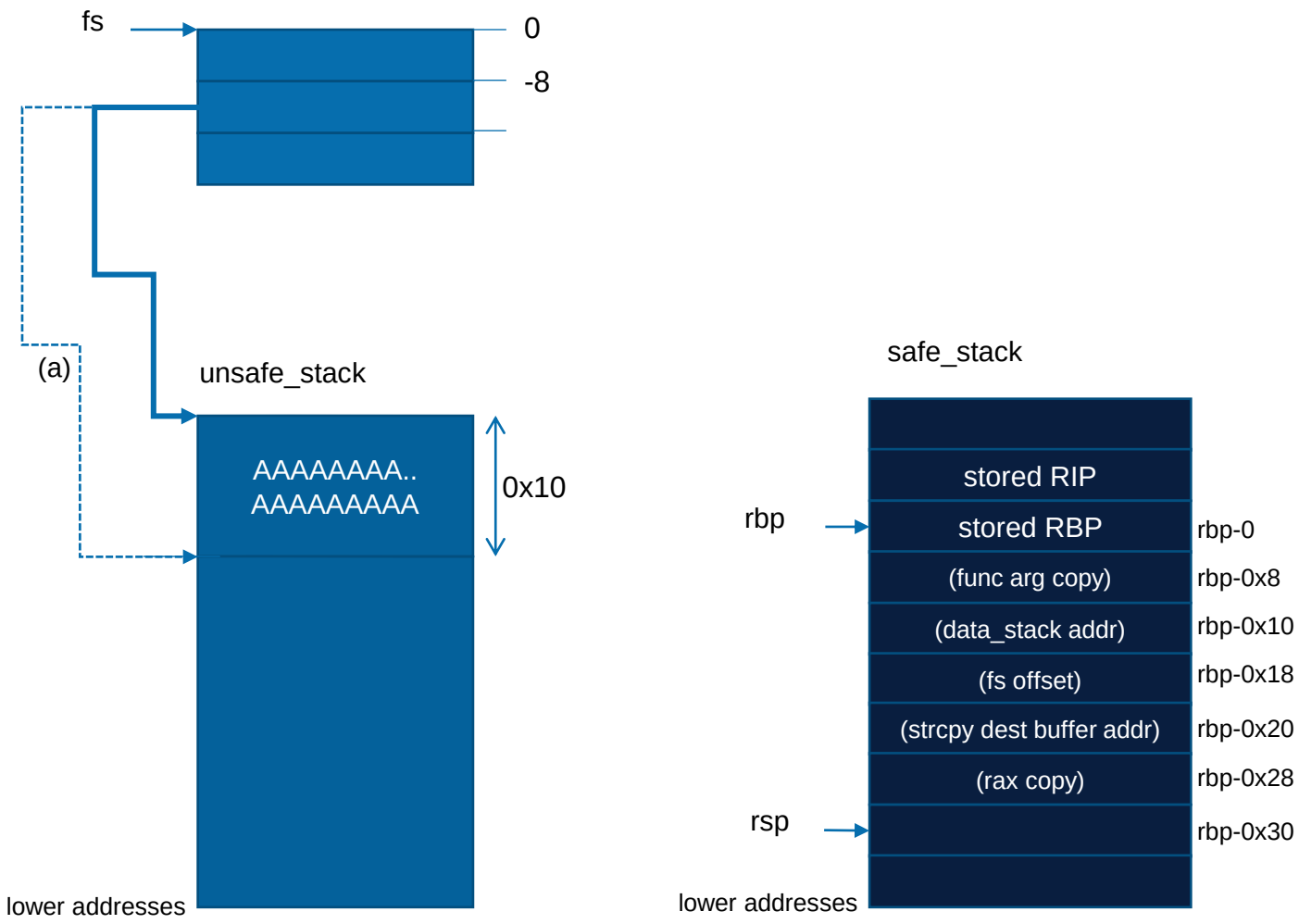
- -fsanitize=safe-stack
- Split stack into safe- (SIP etc.) and unsafe stack

- Unsafe Stack: Has local variables, buffers etc.
- Safe Stack: Has SIP, SBP

# Clang - Backward Edge Protection



# Clang - Backward Edge Protection



# Clang - Backward Edge Protection

Pseudocode:

```
# SafeStack prologue
offset = *(rip + 0x95b1)                          // -8
dataStackBaseAddress = fs[offset]                 // get base of "our" data stack
newDataStackBottom = dataStackBaseAddress - 0x10  // make some space in it
                                                  //   (expand stack to lower address)
fs[offset] = newDataStackBottom                   // store new base

# actual function
localBufferVar = dataStackBaseAddress - 8         // prepare some space in the data stack.
                                                  //   8 is size of BufferVar
strcpy(localBufferVar, ...)                       // Use data stack for local var purposes

# SafeStack epilogue
fs[offset] = dataStackBaseAddress                 // restore original offset
                                                  // (move stack up to the previous address)
```

# Clang - Backward Edge Protection

```
# safestack: prologue
# get value from fs segment, decrement by 0x10, and store it again
   0x000000000040fa28 <+8>:     mov    0x95b1(%rip),%rax           # rax = -8 = 0xfffffffffffffff8
   0x000000000040fa2f <+15>:    mov    %fs:(%rax),%rcx             # rcx = 0x7ffff7c1a000 = *fs:rax
   0x000000000040fa33 <+19>:    mov    %rcx,%rdx                   # rdx = 0x7ffff7c1a000
   0x000000000040fa36 <+22>:    add    $0xfffffffffffffff0,%rdx    # rdx = 0x7ffff7c19ff0    // rdx -= 0x10
   0x000000000040fa3a <+26>:    mov    %rdx,%fs:(%rax)             # fs:rax = 0x7ffff7c19ff0
# rcx is now base pointer to data_stack

# strcpy() part
   0x000000000040fa3e <+30>:    mov    %rdi,-0x8(%rbp)             # rdi is argument of this function, char *a
   0x000000000040fa42 <+34>:    mov    %rcx,%rdx                   # rdx = rcx               // rdx = &data_stack
   0x000000000040fa45 <+37>:    add    $0xfffffffffffffff8,%rdx    # rdx -= 8                // rdx -= 8
   0x000000000040fa49 <+41>:    mov    -0x8(%rbp),%rsi             # rsi = source            // argument argv[1]
   0x000000000040fa4d <+45>:    mov    %rdx,%rdi                   # rdi = rdx = destination // &data_stack-8
   0x000000000040fa50 <+48>:    mov    %rcx,-0x10(%rbp)
   0x000000000040fa54 <+52>:    mov    %rax,-0x18(%rbp)
   0x000000000040fa58 <+56>:    mov    %rdx,-0x20(%rbp)
   0x000000000040fa5c <+60>:    callq  0x401040 <strcpy@plt>       # rdi = destination = 0x7ffff7c19ff8
```

# Clang - Forward Edge Protection - C++

C++ Hardening:

- Mostly helps against type confusion attacks
  - -fsanitize=cfi-cast-strict: Enables strict cast checks.
  - -fsanitize=cfi-derived-cast: Base-to-derived cast to the wrong dynamic type.
  - -fsanitize=cfi-unrelated-cast: Cast from void* or another unrelated type to the wrong dynamic type.
  - -fsanitize=cfi-nvcall: Non-virtual call via an object whose vptr is of the wrong dynamic type.
  - -fsanitize=cfi-vcall: Virtual call via an object whose vptr is of the wrong dynamic type.
  - -fsanitize=cfi-icall: Indirect call of a function with wrong dynamic type.
  - -fsanitize=cfi-mfcall: Indirect call via a member function pointer with wrong dynamic type.

https://clang.llvm.org/docs/ControlFlowIntegrity.html
https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

# New iPhones: Arm64e

- Backwards edge, forward edge
- With arm 8.3 **authenticated pointers**
  - Signing pointers – use top bits to store data
- Kernel mode, and usermode
- Not fully fine-grained, but also not very coarse

- Authenticated vtable pointer in C++ (for virtual call)
  - Kills call on free'd element (UAF)

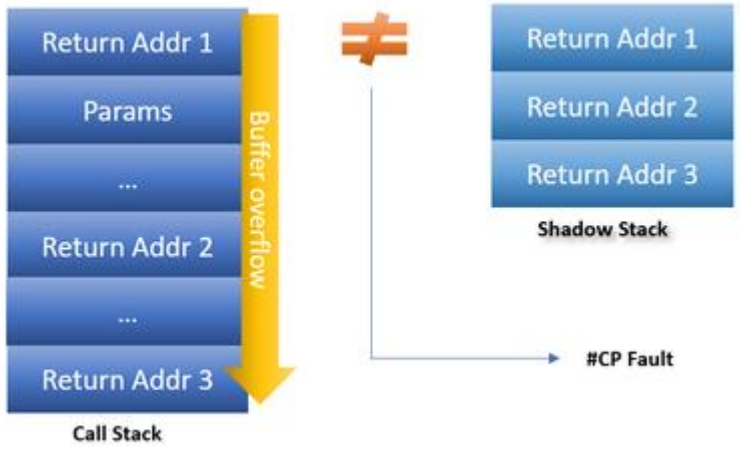
- Bypasses:
  - Pointer replacement attacks (leak signed pointers, reuse)
  - Pointer forgery attack (use CFI weakness to build signing gadgets)
  - A single valid code path with an unprotected branch is all you need (legacy apps, ASM, JIT?)

«Life as an iOS Attacker» - @qwertyoruiop, Bluehat 2019

# Windows - Shadowstack

https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815

keeping a record of all the return addresses via a Shadow Stack. On every CALL instruction, return addresses are pushed onto both the call stack and shadow stack, and on RET instructions, a comparison is made to ensure integrity is not compromised.



# CFI Conclusion

CFI:

- Forward Edge / Backward Edge
- Fine/coarse grained

Attacks:

- Infoleaks on probalistic CFI can break it
- Coarse-grained CFI can allow you to jump to groups of functions (same return type, argument)
- Data-only attacks

# Random mentioning: MS Edge

https://microsoftedge.github.io/edgevr/posts/Introducing-Enhanced-Security-for-Microsoft-Edge/

Microsoft Edge already takes advantage of advanced protections like **Code Integrity Guard (CIG)** and **Control Flow Guard (CFG).**
As of Microsoft Edge 98, **Control-flow Enforcement Technology (CET) and Arbitrary Code Guard (ACG)** will be enabled in the renderer process when a site is in enhanced security mode.

These additional mitigations prevent dynamic code generation in the renderer processes and implement **a separate shadow stack** to protect return addresses. Moreover, we are quite excited that Microsoft Edge now supports **both forwards and backwards control-flow protection**. By applying these protections, we can provide defense in depth that spans beyond JIT attacks.