EuroBSD 2018

Removing ROP Gadgets from OpenBSD

Todd Mortimer

Overview

- About Me
- Return Oriented Programming
- Polymorphic Gadget Reduction
  - Register Selection
  - Alternate Code Generation
- Aligned Gadget Reduction
  - Retguard
- Other architectures - arm64
- Remaining Work

About Me

- OpenBSD user since ~2015
- Randomly approached Theo at BSDCan 2017
  - I suggested removing ROP gadgets was possible
  - Theo expressed skepticism
- Joined project in June 2017
  - Working on ROP mitigations in clang

Return Oriented Programming

Return Oriented Programming

- W^X means attackers cannot just upload shellcode anymore
- ROP is stitching bits of existing binary together in a new way to get the same effect as shellcode
  - The bits are called Gadgets
  - The stitching is called a ROP Chain
- Attacker
  - Loads a chain in memory
  - Redirects execution to return off of the chain

ROP Gadgets

Aligned Gadget

Terminates on an intended return instruction

```
Gadget: 0xffffffff81820653 : pop rbp ; ret // 5dc3

ffffffff81820653: 5d      popq    %rbp

ffffffff81820654: c3      retq
```

Polymorphic Gadget

Terminates on an unintended return instruction

```
Gadget: 0xffffffff810f72dc : pop rbp ; ret // 5dc3

ffffffff810f72db: 8a 5d c3    movb    -61(%rbp), %bl
```

ROP Gadgets

Aligned Gadget

Terminates on an intended return instruction

Gadget: Address : Disassembly // Bytes

What the gadget does

```
Gadget: 0xffffffff81820653 : pop rbp ; ret // 5dc3
ffffffff81820653: 5d       popq    %rbp
ffffffff81820654: c3       retq
```

Polymorphic Gadget

Terminates on an unintended return instruction

What the gadget does

```
Gadget: 0xffffffff810f72dc : pop rbp ; ret // 5dc3
ffffffff810f72db: 8a 5d c3    movb    -61(%rbp), %bl
```

ROP Gadgets

Aligned Gadget

Terminates on an intended return instruction

```
Gadget: 0xffffffff81820653 : pop rbp ; ret // 5dc3
ffffffff81820653: 5d      popq    %rbp
ffffffff81820654: c3      retq
```

What the code meant to do

Polymorphic Gadget

Terminates on an unintended return instruction

```
Gadget: 0xffffffff810f72dc : pop rbp ; ret // 5dc3
ffffffff810f72db: 8a 5d c3   movb    -61(%rbp), %bl
```

What the code meant to do

ROP Chains

- Each gadget ends with ‘*ret*’
- ‘*ret*’ pops an address from the stack and jumps to it
- A ROP Chain strings many gadgets addresses together on the stack
- Gadgets are executed sequentially

```
0x00000000000905ee # pop rsi ; ret
0x00000000002cd000 # @ .data
0x000000000003b62e # pop rax ; ret
0x2f62696e2f2f7368 # "/bin//sh"
0x000000000001f532 # mov qword ptr [rsi], rax ; pop rbp ; ret
0x4141414141414141 # padding
0x00000000000905ee # pop rsi ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000000fa0 # xor rax, rax ; ret
0x000000000001f532 # mov qword ptr [rsi], rax ; pop rbp ; ret
0x4141414141414141 # padding
0x00000000000004cd # pop rdi ; pop rbp ; ret
0x00000000002cd000 # @ .data
0x4141414141414141 # padding
0x00000000000905ee # pop rsi ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000068f03 # pop rdx ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000000fa0 # xor rax, rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret

[… keep incrementing rax to 59 : SYS_execve]

0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000009c8 # syscall
```

ROP Chains

Gadget Addresses

- Each gadget ends with ‘*ret*’
- ‘*ret*’ pops an address from the stack and jumps to it
- A ROP Chain strings many gadgets addresses together on the stack
- Gadgets are executed sequentially

```
0x00000000000905ee # pop rsi ; ret
0x00000000002cd000 # @ .data
0x000000000003b62e # pop rax ; ret
0x2f62696e2f2f7368 # "/bin//sh"
0x000000000001f532 # mov qword ptr [rsi], rax ; pop rbp ; ret
0x4141414141414141 # padding
0x00000000000905ee # pop rsi ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000000fa0 # xor rax, rax ; ret
0x000000000001f532 # mov qword ptr [rsi], rax ; pop rbp ; ret
0x4141414141414141 # padding
0x00000000000004cd # pop rdi ; pop rbp ; ret
0x00000000002cd000 # @ .data
0x4141414141414141 # padding
0x00000000000905ee # pop rsi ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000068f03 # pop rdx ; ret
0x00000000002cd008 # @ .data + 8
0x0000000000000fa0 # xor rax, rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret

[… keep incrementing rax to 59 : SYS_execve]

0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000038fe # inc rax ; ret
0x00000000000009c8 # syscall
```

ROP Chain Tooling

- Building ROP Chains by hand is tedious
- Tools make this easy
  - ROPGadget.py
  - ropper
  - pwntools
  - *others…*

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.3/libc.so.92.3

Unique gadgets found: 8468

ROP chain generation

- Step 1 -- Write-what-where gadgets
    [+] Gadget found: 0x617a8 mov word ptr [rcx], dr1 ; ret
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    […]
- Step 2 -- Init syscall number gadgets
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    [+] Gadget found: 0x62a6 add al, 1 ; ret
    […]
- Step 3 -- Init syscall arguments gadgets
    [+] Gadget found: 0x4cd pop rdi ; pop rbp ; ret
    [+] Gadget found: 0x905ee pop rsi ; ret
    […]
- Step 4 -- Syscall gadget
    [+] Gadget found: 0x9c8 syscall
    […]
-  Step 5 -- Build the ROP chain
    […]
    p += pack('<Q', 0x00000000000905ee) #  pop rsi ; ret
    p += pack('<Q', 0x00000000002cd000) # @ .data
    p += pack('<Q', 0x000000000003b62e) #  pop rax ; ret
    p += '/bin//sh'
    […]
    p += pack('<Q', 0x00000000000038fe) #  inc rax ; ret
    p += pack('<Q', 0x00000000000009c8) #  syscall
```

Review - Results

Number of unique gadgets found

Identifying different types of gadgets needed

Stringing the gadgets together to get *exec("/bin/sh")*

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.3/libc.so.92.3

Unique gadgets found: 8468

ROP chain generation

- Step 1 -- Write-what-where gadgets
    [+] Gadget found: 0x617a8 mov word ptr [rcx], dr1 ; ret
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    [...]
- Step 2 -- Init syscall number gadgets
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    [+] Gadget found: 0x62a6 add al, 1 ; ret
    [...]
- Step 3 -- Init syscall arguments gadgets
    [+] Gadget found: 0x4cd pop rdi ; pop rbp ; ret
    [+] Gadget found: 0x905ee pop rsi ; ret
    [...]
- Step 4 -- Syscall gadget
    [+] Gadget found: 0x9c8 syscall
    [...]
-  Step 5 -- Build the ROP chain
    [...]
    p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
    p += pack('<Q', 0x00000000002cd000) # @ .data
    p += pack('<Q', 0x000000000003b62e) # pop rax ; ret
    p += '/bin//sh'
    [...]
    p += pack('<Q', 0x00000000000038fe) # inc rax ; ret
    p += pack('<Q', 0x00000000000009c8) # syscall
```

Removing Gadgets

- Aim: Reduce the number and variety of useful gadgets
  - Compile out unintended returns
  - Make intended returns hard to use in ROP chains
- We don't need to get to zero gadgets
  - Just remove enough to make building useful ROP chains hard / impossible
  - Use ROP tool output to measure progress

Polymorphic Gadget Reduction

Polymorphic Gadgets

- There are 4 return instructions on x86 / amd64

Byte	Instruction
C2	RET imm16 (near)
C3	RET (near)
CA	RET imm16 (far)
CB	RET (far)

Polymorphic Gadgets

- There are 4 return instructions on x86 / amd64

Byte	Instruction
C2	RET imm16 (near)
C3	RET (near)
CA	RET imm16 (far)
CB	RET (far)

Most useful form

Polymorphic Gadgets - Sources

Other Instruction Opcodes

```
Gadget: 0xffffffff8100b61e : add dword ptr [rcx], eax ; ret / / 0101c3

Instruction: ffffffff8100b61c:      83 e3 01        andl    $1, %ebx
             ffffffff8100b61f:      01 c3           addl    %eax, %ebx
```

Polymorphic Gadgets - Sources

Other Instruction Opcodes

```
Gadget: 0xffffffff8100b61e : add dword ptr [rcx], eax ; ret // 0101c3

Instruction: ffffffff8100b61c:    83 e3 01    andl   $1, %ebx
             ffffffff8100b61f:    01 c3       addl   %eax, %ebx
```

Constants

```
Gadget: 0xffffffff81050f8b : movsd dword ptr [rdi], dword ptr [rsi] ; ret // a5c3

Instruction: ffffffff81050f88:    48 c7 c7 a5 c3 84 81    movq   $-2122005595, %rdi
```

Polymorphic Gadgets - Sources

Other Instruction Opcodes

```
Gadget: 0xffffffff8100b61e : add dword ptr [rcx], eax ; ret / / 0101c3

Instruction: ffffffff8100b61c:      83 e3 01        andl    $1, %ebx
             ffffffff8100b61f:      01 c3           addl    %eax, %ebx
```

Constants

```
Gadget: 0xffffffff81050f8b : movsd dword ptr [rdi], dword ptr [rsi] ; ret / / a5c3

Instruction: ffffffff81050f88:      48 c7 c7 a5 c3 84 81    movq    $-2122005595, %rdi
```

Relocs

```
Gadget: 0xffffffff81008647 : xchg eax, ebp ; ret / / 95c3

Instruction: ffffffff81008646:      e8 95 c3 3e 00  callq   4113301 <bcmp>
```

Register Selection

- One common class of gadgets gets C3 return bytes from the *ModR/M* byte of certain instructions
  - Source register is RAX/EAX/AX/AL
  - Destination register is RBX/EBX/BX/BL
- Also operations on RBX / EBX / BX / BL
  - inc, dec, test, *etc.*

Register Selection

```
Gadget: 0xffffffff8100ca58 : dec dword ptr [rax - 0x77] ; ret // ff4889c3
Instructions:
ffffffff8100ca54:    e8 f7 f9 ff ff  callq   -1545 <uvm_pmr_insert_addr>
ffffffff8100ca59:    48 89 c3        movq    %rax, %rbx
```

```
Gadget: 0xffffffff8100ffcd : mov byte ptr [rax], 0 ; add bh, bh ; ret // c6000000ffc3
Instructions:
ffffffff8100ffcb:    0f 84 c6 00 00 00    je      198 <pckbc_attach+0x337>
ffffffff8100ffd1:    ff c3                incl    %ebx
```

```
Gadget: 0xffffffff810100f3 : or edi, edi ; ret // 09ffc3
Instructions:
ffffffff810100f2:    74 09   je      9 <pckbc_attach+0x39d>
ffffffff810100f4:    ff c3   incl    %ebx
```

Register Selection

```
Gadget: 0xffffffff8100ca58 : dec dword ptr [rax - 0x77] ; ret // ff4889c3
Instructions:
ffffffff8100ca54:      e8 f7 f9 ff ff  callq   -1545 <uvm_pmr_insert_addr>
ffffffff8100ca59:      48 89 c3        movq    %rax, %rbx
```

```
Gadget: 0xffffffff8100ffcd : mov byte ptr [rax], 0 ; add
Instructions:
ffffffff8100ffcb:      0f 84 c6 00 00 00      je      198 <pckb
ffffffff8100ffd1:      ff c3                  incl    %ebx
```

RBX / EBX / BX / BL
generate a lot of C3 bytes

```
Gadget: 0xffffffff810100f3 : or edi, edi ; ret // 09ffc3
Instructions:
ffffffff810100f2:      74 09   je      9 <pckbc_attach+0x39d>
ffffffff810100f4:      ff c3   incl    %ebx
```

Register Selection

- **Avoid using RBX/EBX/BX/BL**
- Clang allocates registers in this order:
  - RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, **RBX**, R14, R15, R12, R13, RBP
- Move RBX closer to the end of the list:
  - RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, R14, R15, R12, R13, **RBX**, RBP
- Also change order for EBX

Register Selection

- Performance cost: Zero
- Code size cost: Negligible
  - Some REX prefix bytes
- Results: Removes about 4500 unique gadgets (6%) from the kernel

Alternate Code Generation

- Sometimes you need to use RBX
- We know which instructions will have a C3 byte
- Teach the compiler to emit something else

Fixup Gadgets Pass

- Clang module that identifies instructions with possible gadgets and replaces them with safe alternatives

```
ffffffff81006bd9:    89 c3    mov    %eax,%ebx
```

Potential gadget

Fixup Gadgets Pass

- Clang module that identifies instructions with possible gadgets and replaces them with safe alternatives

Turn this …

```
ffffffff81006bd9:      89 c3    mov    %eax,%ebx
```

… into this

```
ffffffff81006bd9:      48 87 d8       xchgq   %rbx, %rax
ffffffff81006bdc:      89 d8          movl    %ebx, %eax
ffffffff81006bde:      48 87 d8       xchgq   %rbx, %rax
```

Fixup Gadgets Pass

- Performance cost: Negligible
  - xchg is cheap
- Code side cost: Small
  - 6 bytes per fixup
  - 0.15% larger kernel
- Results: Removes about 3700 unique gadgets (5%) from the kernel

Fixup Gadgets Pass

- Still more to do
  - Additional instruction cases to handle
  - Constants
  - Relocs

Aligned Gadget Reduction

Denying Gadgets

- Some RETs are impossible to avoid
  - Functions need to actually return
- Can we make them hard to use?

Retguard

- Allocate a random cookie for every function
  - Use openbsd.randomdata section to allocate random values
- On function entry
  - Compute ***cookie ^ return address***
  - Store the result in the frame
- On function return
  - Compute ***saved value ^ return address***
  - Compare to cookie
  - If comparison fails then abort

Retguard - Prologue

- On function entry
  - Compute ***cookie* ^ *return address***
  - Store the result in the frame

```
ffffffff819ff700: 4c 8b 1d 61 21 24 00     mov    2367841(%rip),%r11 # <__retguard_2759>
ffffffff819ff707: 4c 33 1c 24              xor    (%rsp),%r11
ffffffff819ff70b: 55                       push   %rbp
ffffffff819ff70c: 48 89 e5                 mov    %rsp,%rbp
ffffffff819ff70f: 41 53                    push   %r11
```

Retguard - Epilogue

- On function return
  - Compute ***saved value ^ return address***
  - Compare to cookie
  - If comparison fails then abort

```
ffffffff8115a457: 41 5b                   pop    %r11
ffffffff8115a459: 5d                      pop    %rbp
ffffffff8115a45a: 4c 33 1c 24             xor    (%rsp),%r11
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00    cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                   je     ffffffff8115a469
ffffffff8115a467: cc                      int3
ffffffff8115a468: cc                      int3
ffffffff8115a469: c3                      retq
```

Retguard - Epilogue

- The int3 instructions are important
  - They disrupt gadgets wanting to use the ret

```
ffffffff8115a457: 41 5b                 pop    %r11
ffffffff8115a459: 5d                    pop    %rbp
ffffffff8115a45a: 4c 33 1c 24           xor    (%rsp),%r11
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00  cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                 je     ffffffff8115a469
ffffffff8115a467: cc                    int3
ffffffff8115a468: cc                    int3
ffffffff8115a469: c3                    retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a459: 5d                    pop    %rbp
ffffffff8115a45a: 4c 33 1c 24           xor    (%rsp),%r11
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00  cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                 je     ffffffff8115a469
ffffffff8115a467: cc                    int3
ffffffff8115a468: cc                    int3
ffffffff8115a469: c3                    retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45a: 4c 33 1c 24             xor    (%rsp),%r11
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00    cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                   je     ffffffff8115a469
ffffffff8115a467: cc                      int3
ffffffff8115a468: cc                      int3
ffffffff8115a469: c3                      retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45a:    33 1c 24               xor    (%rsp),%ebx
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00      cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                     je     ffffffff8115a469
ffffffff8115a467: cc                        int3
ffffffff8115a468: cc                        int3
ffffffff8115a469: c3                        retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45a:       1c 24                   sbb    al, 0x24
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00          cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                         je     ffffffff8115a469
ffffffff8115a467: cc                            int3
ffffffff8115a468: cc                            int3
ffffffff8115a469: c3                            retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45a:             24              and    al, 0x4c
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00        cmp    11432963(%rip),%ebx # <__retguard_2759>
ffffffff8115a465: 74 02                       je     ffffffff8115a469
ffffffff8115a467: cc                          int3
ffffffff8115a468: cc                          int3
ffffffff8115a469: c3                          retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e: 4c 3b 1d 03 74 ae 00    cmp    11432963(%rip),%r11 # <__retguard_2759>
ffffffff8115a465: 74 02                   je     ffffffff8115a469
ffffffff8115a467: cc                      int3
ffffffff8115a468: cc                      int3
ffffffff8115a469: c3                      retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:    3b 1d 03 74 ae 00    cmp    11432963(%rip),%ebx # <__retguard_2759>
ffffffff8115a465: 74 02                   je     ffffffff8115a469
ffffffff8115a467: cc                      int3
ffffffff8115a468: cc                      int3
ffffffff8115a469: c3                      retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:          1d 03 74 ae 00       sbbl   $0xae7403, %eax
ffffffff8115a465: 74 02                         je     ffffffff8115a469
ffffffff8115a467: cc                            int3
ffffffff8115a468: cc                            int3
ffffffff8115a469: c3                            retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:             03 74 ae 00    addl   (%rsi, %rbp, 4), %esi
ffffffff8115a465: 74 02                      je     ffffffff8115a469
ffffffff8115a467: cc                         int3
ffffffff8115a468: cc                         int3
ffffffff8115a469: c3                         retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:               74 ae 00        je      -80
ffffffff8115a465: 74 02                         addb %dh, -0x34(%rdx, %rax)
ffffffff8115a467: cc
ffffffff8115a468: cc                            int3
ffffffff8115a469: c3                            retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:                    ae 00     scasb (%rdi), %al
ffffffff8115a465: 74 02                        addb %dh, -0x34(%rdx, %rax)
ffffffff8115a467: cc
ffffffff8115a468: cc                           int3
ffffffff8115a469: c3                           retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a45e:                       00
ffffffff8115a465: 74 02                         addb %dh, -0x34(%rdx, %rax)
ffffffff8115a467: cc
ffffffff8115a468: cc                            int3
ffffffff8115a469: c3                            retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a465: 74 02                  je     ffffffff8115a469
ffffffff8115a467: cc                     int3
ffffffff8115a468: cc                     int3
ffffffff8115a469: c3                     retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a465:    02               addb %ah, %cl
ffffffff8115a467: cc
ffffffff8115a468: cc                  int3
ffffffff8115a469: c3                  retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a467: cc                     int3
ffffffff8115a468: cc                     int3
ffffffff8115a469: c3                     retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a468: cc                    int3
ffffffff8115a469: c3                    retq
```

Retguard - Epilogue

- Disassemble every offset leading to the *ret*. Every gadget either
  - Must pass the comparison
  - Includes an *int3* instruction

```
ffffffff8115a469: c3                          retq
```

Retguard

- Performance Cost
  - Runtime about 2%
  - Startup cost (filling *.openbsd.randomdata*) is variable
- Code size cost
  - 31 bytes per function in binary
  - 8 bytes per function runtime for random cookies
  - + ~ 7% for the kernel

Retguard

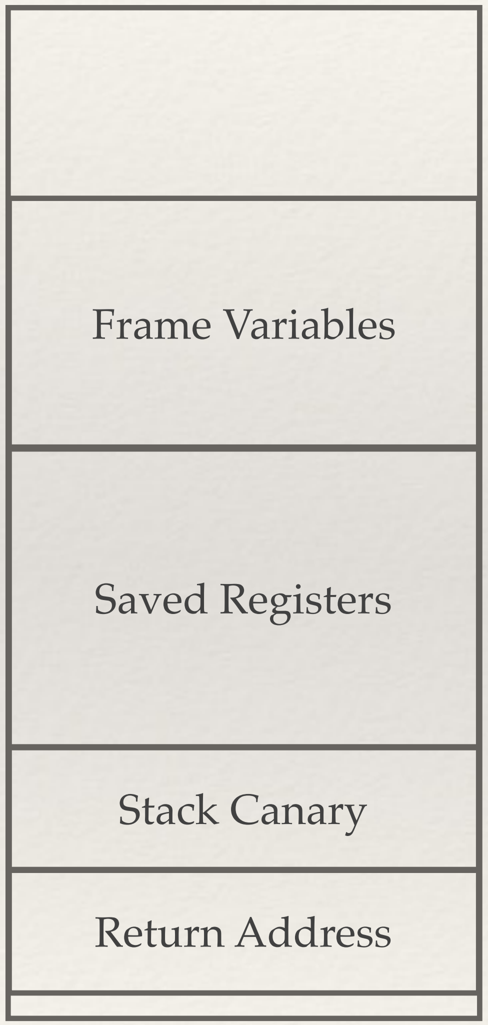
- Removes from the kernel
  - ~ 50% of total ROP gadgets
  - ~ 15 - 25% of unique ROP gadgets
- Gadget numbers are variable due to Relocs / KARL

Retguard - Bonus

- Unexpected consequence
  - Retguard verifies integrity of the return address
  - Stack protector verifies integrity of the stack cookie
  - Retguard is a better stack protector
    - Per-function cookie
    - Verifies return address directly

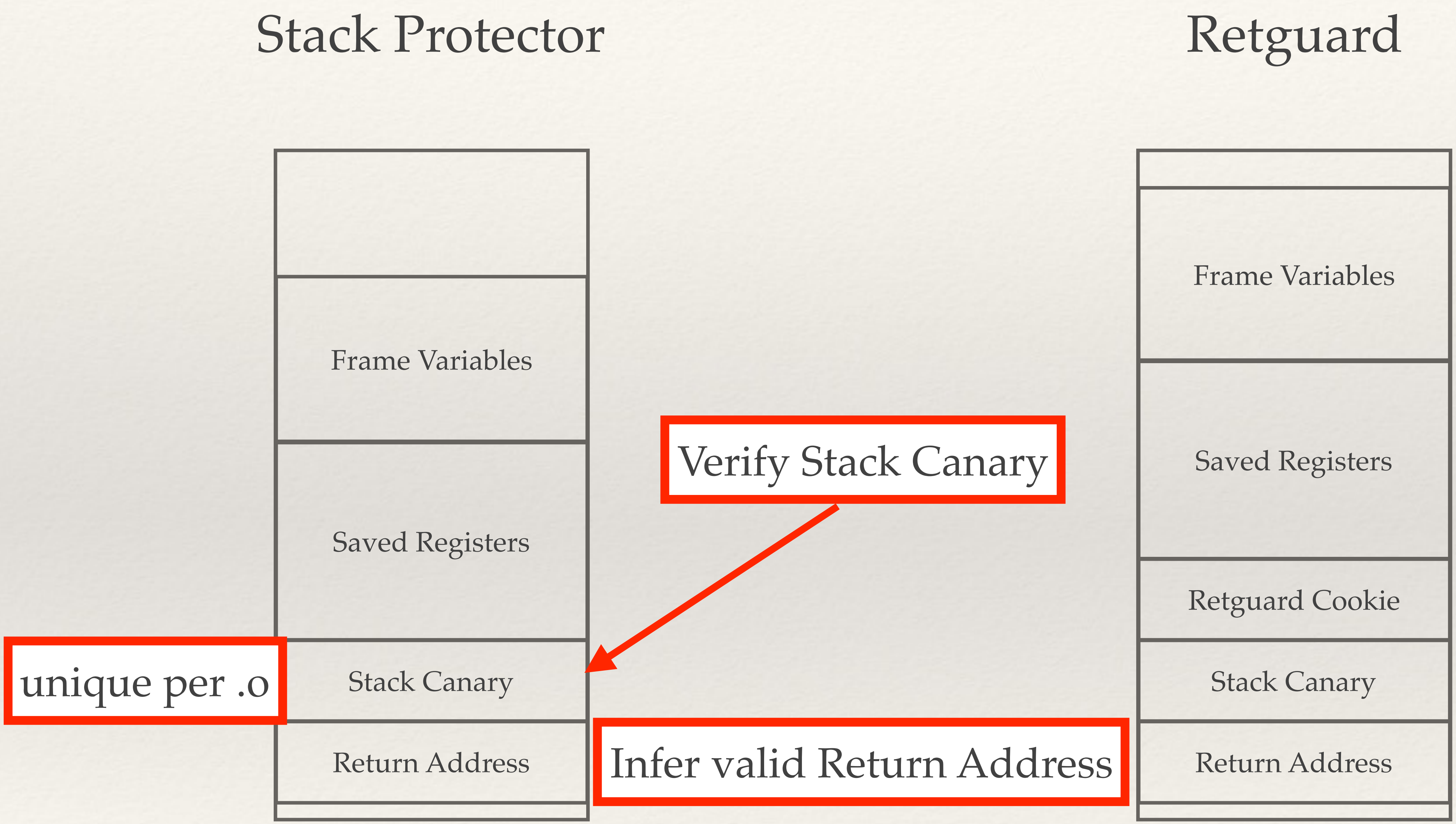
Retguard - Bonus

Stack Protector

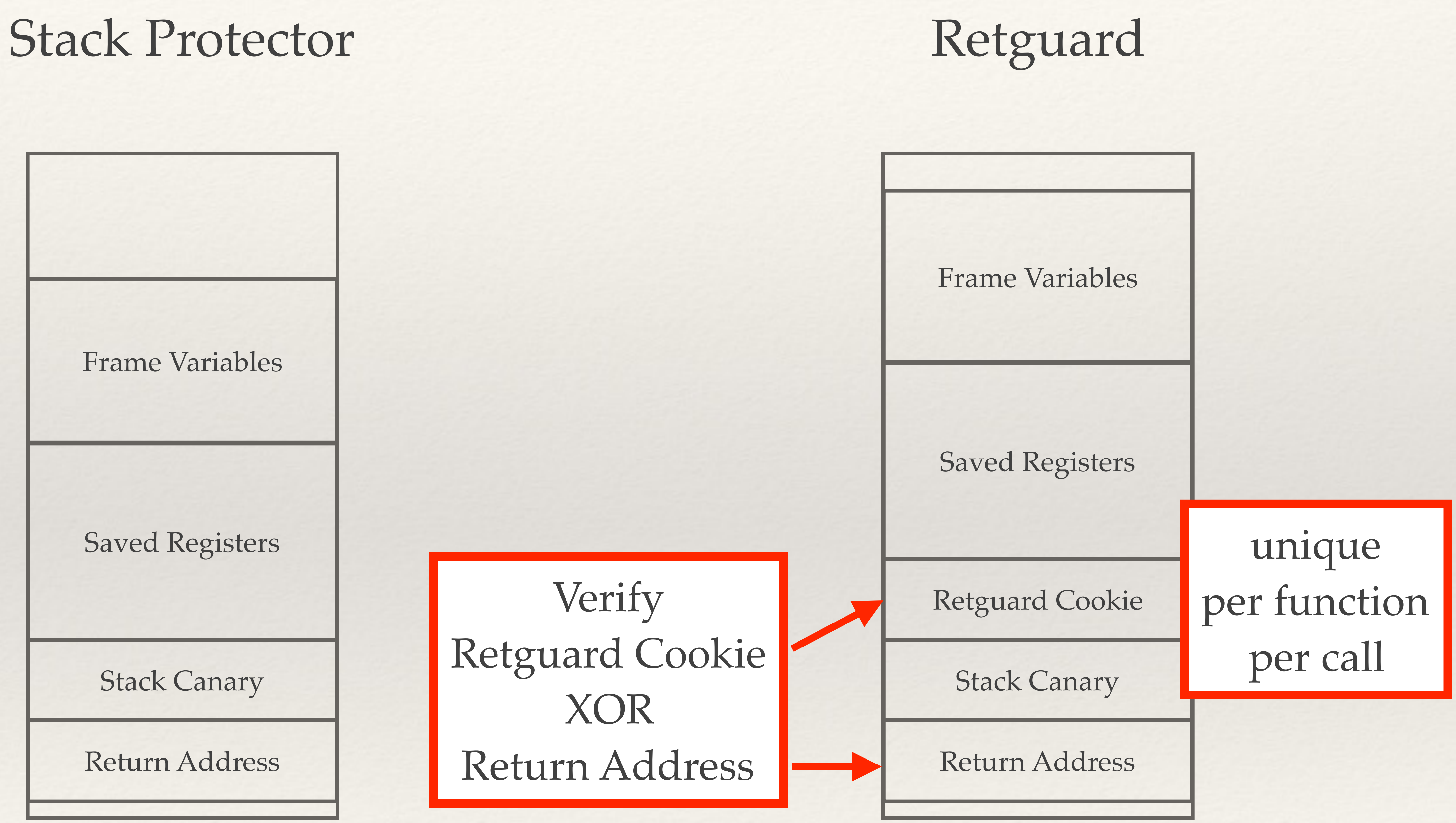


Retguard

Retguard - Bonus



Retguard - Bonus



Other Architectures - Arm64

Arm 64

- arm64 has fixed width instructions
  - No polymorphic gadgets
    - No need for register selection or alternate code changes in clang
  - Aligned gadgets
    - Retguard can instrument every return

Retguard - Arm 64

Prologue

```
ffffff8000204370:       2f 37 00 f0     adrp    x15, #7237632
ffffff8000204374:       ef 25 43 f9     ldr     x15, [x15, #1608]
ffffff8000204378:       ef 01 1e ca     eor     x15, x15, x30
ffffff800020437c:       ef 0f 1f f8     str     x15, [sp, #-16]!
```

Epilogue

```
ffffff80002043f8:       ef 07 41 f8     ldr     x15, [sp], #16
ffffff80002043fc:       29 37 00 f0     adrp    x9, #7237632
ffffff8000204400:       29 25 43 f9     ldr     x9, [x9, #1608]
ffffff8000204404:       ef 01 1e ca     eor     x15, x15, x30
ffffff8000204408:       ef 01 09 eb     subs    x15, x15, x9
ffffff800020440c:       4f 00 00 b4     cbz     x15, #8
ffffff8000204410:       20 00 20 d4     brk     #0x1
ffffff8000204414:       c0 03 5f d6     ret
```

Retguard - Arm 64

- Since there are only aligned gadgets on arm64
- and Retguard can instrument every aligned gadget
  - We can actually remove all the gadgets

Retguard - Arm 64

CVSROOT: /cvs
Module name: src
Changes by: mortimer@cvs.openbsd.org 2018/09/09 10:41:43

Modified files:
    sys/arch/arm64/arm64: locore.S

Log message:
Apply retguard to the last asm functions in the arm64 kernel. This completes retguard in the kernel and brings the number of useful ROP gadgets at runtime to zero.

ok kettenis@

Retguard - Arm 64

- Number of ROP gadgets in 6.3-release arm64 kernel
  - 69935
- Number of ROP gadgets in 6.4-beta arm64 kernel
  - 46

Retguard - Arm 64

- Remaining gadgets are assembly functions in the boot code
  - create_pagetables
  - link_l0_pagetable
  - link_l1_pagetable
  - build_l1_block_pagetable
  - build_l2_block_pagetable
- OpenBSD unlinks or smashes the boot code after boot
  - These functions are gone at runtime

Retguard - Arm 64

- Story in userland is much the same
  - Often zero ROP gadgets
  - Remaining gadgets are from assembly functions
    - crt0, ld.so, *etc.*
- Some work remains to instrument these functions

Review

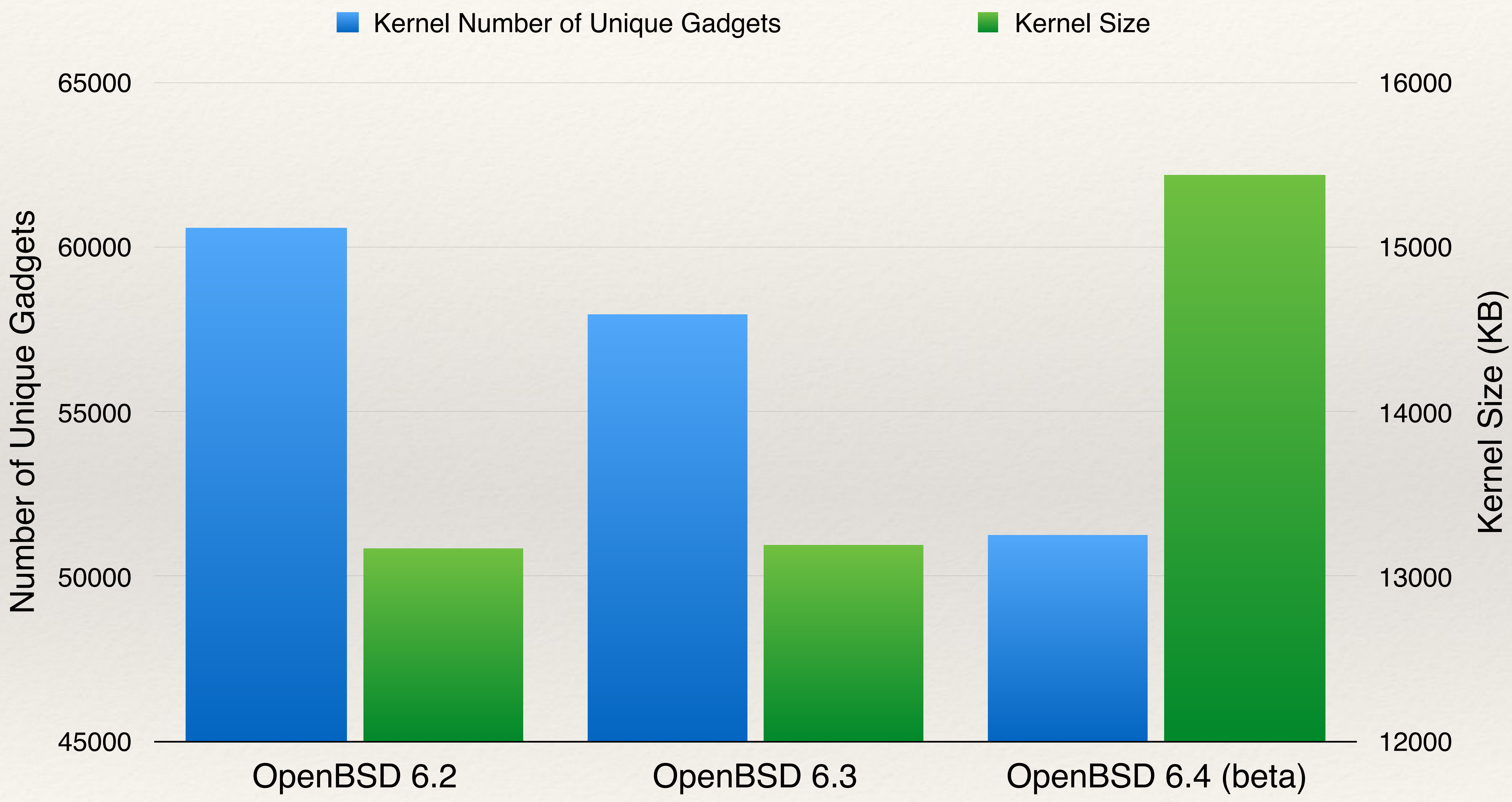
Review

- We can remove ROP gadgets
  - Alternate Register Selection
  - Alternate Code Generation
  - Retguard

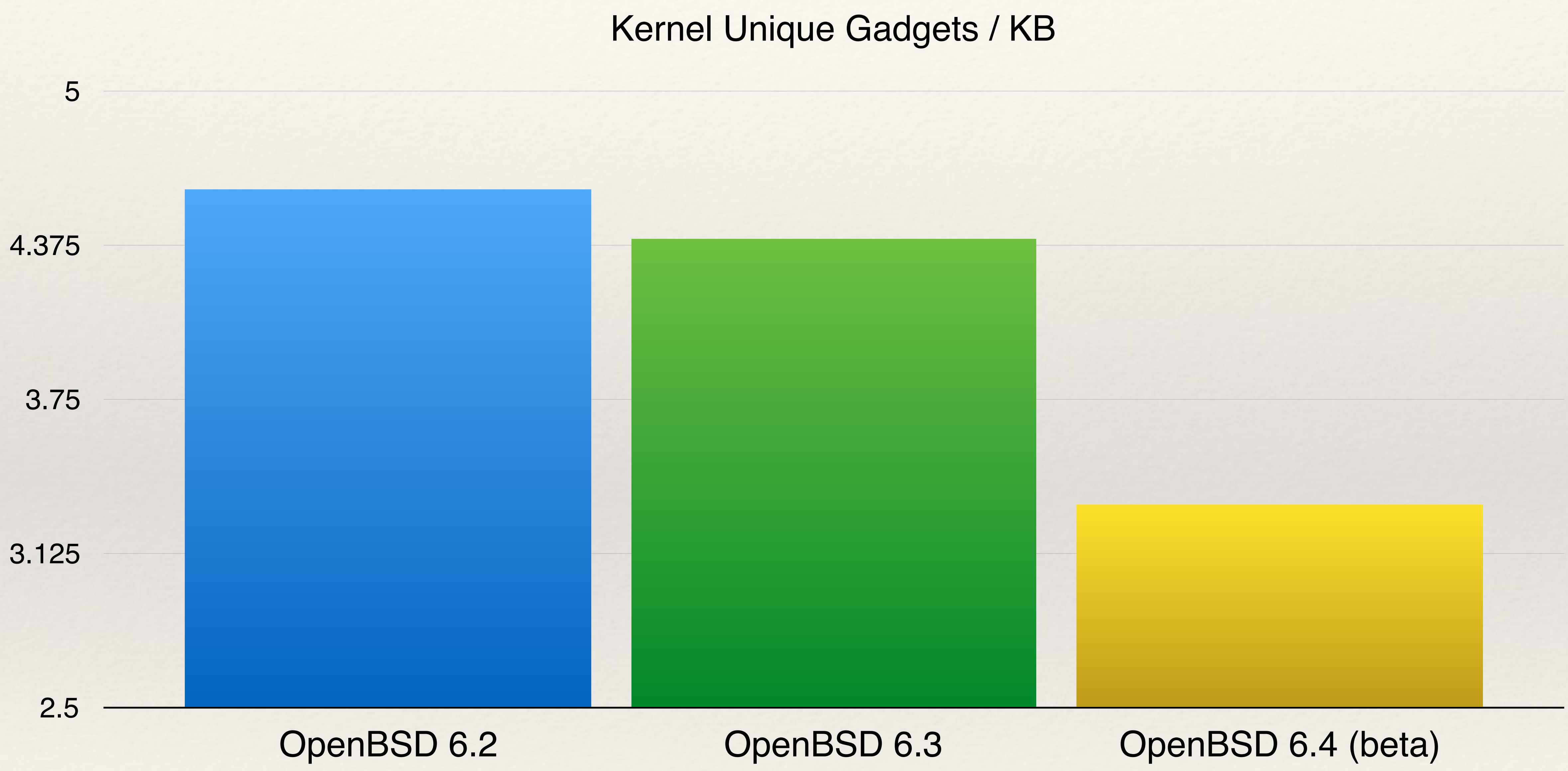
Review - Progress

- In the amd64 kernel we removed unique ROP gadgets:
  - Alternate Register Selection: ~ 6%
  - Alternate Code Generation: ~ 5%
  - Retguard: ~ 15-25%
- Similar numbers for userland

Review - Progress



Review - Progress



Does this really make a difference?

Review - Results

- Run ROPGadget.py on OpenBSD 6.3 libc
    - libc is a big juicy target
- Ask the tool for a ROP chain that pops a shell
- Tool succeeds and outputs a ROP chain

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.3/libc.so.92.3

Unique gadgets found: 8468

ROP chain generation

- Step 1 -- Write-what-where gadgets
    [+] Gadget found: 0x617a8 mov word ptr [rcx], dr1 ; ret
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    [...]
- Step 2 -- Init syscall number gadgets
    [+] Gadget found: 0xfa0 xor rax, rax ; ret
    [+] Gadget found: 0x62a6 add al, 1 ; ret
    [...]
- Step 3 -- Init syscall arguments gadgets
    [+] Gadget found: 0x4cd pop rdi ; pop rbp ; ret
    [+] Gadget found: 0x905ee pop rsi ; ret
    [...]
- Step 4 -- Syscall gadget
    [+] Gadget found: 0x9c8 syscall
    [...]
-  Step 5 -- Build the ROP chain
    [...]
    p += pack('<Q', 0x00000000000905ee) # pop rsi ; ret
    p += pack('<Q', 0x00000000002cd000) # @ .data
    p += pack('<Q', 0x000000000003b62e) # pop rax ; ret
    p += '/bin//sh'
    [...]
    p += pack('<Q', 0x00000000000038fe) # inc rax ; ret
    p += pack('<Q', 0x00000000000009c8) # syscall
```

Review - Results

- Run ROPGadget.py on OpenBSD 6.4-beta libc

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.4-beta/libc.so.92.4
```

Review - Results

- Run ROPGadget.py on OpenBSD 6.4-beta libc
- The tool fails to find a ROP chain that pops a shell
- Reduced gadget diversity foils this tool
- ROP attacks on 6.4 are harder to execute

```
$ ROPgadget.py --ropchain --binary OpenBSD-6.4-beta/libc.so.92.4

Unique gadgets found: 6007

ROP chain generation

- Step 1 -- Write-what-where gadgets

    [-] Can't find the 'mov qword ptr [r64], r64' gadget

$
```

Still many gadgets…

… but not enough diversity

Remaining Work

- There is still more to do!
- Alternate Code Generation
  - Additional instruction sequences to fix
  - Constants
  - Relocs
- What about JOP?

Questions?