Emulating Function Calls with int3

And the crappy solutions we came up with

Steven Rostedt
Open Source Engineer
rostedt@goodmis.org / srostedt@vmware.com

Twitter: @VMWopensource / Expo floor #31
The background
The background

In January 2018 Spectre and Meltdown were announced to the public. They were hardware flaws:

- Affected all operating systems
- Took advantage of flows in the way the CPU cached data
  - Speculative execution did not clear cache
  - Side channels could read any memory in the system
  - Could trick the branch predictor
2018 Kernel Recipes in Paris


Meltdown and Spectre: seeing through the magician’s tricks

by Paolo Bonzini
2018 Kernel Recipes in Paris


Mitigating Spectre and Meltdown vulnerabilities

by David Woodhouse
Mitigating Spectre and Meltdown vulnerabilities
by David Woodhouse

Spectre v2: Retpoline

- Confuse the branch predictor!
- Original code:
  
  ```
  jmp *r11
  ```

- Replaced with:
  
  ```
  call set_up_target
capture_speculation:
  pause
  jmp capture_speculation

set_up_target:
  mov %r11, %rsp
  ret
  ```

© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.
What does this mean?

Dynamic functions have added overhead!
Huh?
Normal Function

```c
int hello_world(void)
{
    printf("Hello world!\n");
    return 1;
}
```
Dynamic Function (Function Pointer)

```c
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))
```
Dynamic Function (Function Pointer)

```c
void qsort(void *base, size_t nmemb, size_t size,
          int (*compar)(const void *, const void *))
```
```c
int nums[] = { 8, 4, 3, 9, 7 };
char *strings[] = { “mutt”, “duck”, “shark”, “kernel” };
qsort(nums, 5, sizeof(int), intcmp);
qsort(strings, 4, sizeof(char *), scmp);
```
```c
int nums[] = { 8, 4, 3, 9, 7 }; char *strings[] = { "mutt", "duck", "shark", "kernel" };
qsort(nums, 5, sizeof(int), intcmp);
qsort(strings, 4, sizeof(char *), scmp);
```
```c
int nums[] = { 8, 4, 3, 9, 7 }; char *strings[] = { "mutt", "duck", "shark", "kernel" };

int intcmp(const void * _a, const void * _b)
{
    const int *a = _a, *b = _b;
    return *a < *b ? -1 : *a > *b ? 1 : 0;
}

int scmp(const void * _a, const void * _b)
{
    char * const *a = _a;
    char * const *b = _b;
    return strcmp(*a, *b);
}
```
qsort

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *))
{
    [...]  
    const void *a = base + mid * size;
    const void *b = base + lo * size;
    [...]  
    if ((compar)(a, b) < 0)
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))
{
   [..]
    const void *a = base + mid * size;
    const void *b = base + lo * size;
   [..]
    if ((*compar)(a, b) < 0)
void qsort(void *base, size_t nmemb, size_t size, intcmp)
{
    [...]
    const void *a = base + mid * size;
    const void *b = base + lo * size;
    [...]
    if (intcmp(a, b) < 0)
void qsort(void *base, size_t nmemb, size_t size, scmp)
{
    const void *a = base + mid * size;
    const void *b = base + lo * size;
    if (scmp(a, b) < 0)
Silly example of dynamic function

```c
int goldie_locks(void (*bed)(int)) { (*bed)(23); }

void minus3(int x) {
    printf("-3 is %d\n", x - 3);
}

void plus4(int x) {
    printf("+4 is %d\n", x + 4);
}

void just_right(int x) {
    printf("%d is just right!\n", x);
}

int main (int argc, char **argv) {
    goldie_locks(minus3);
    goldie_locks(plus4);
    goldie_locks(just_right);
    exit(0);
}
```
Silly example of dynamic function

$ goldie_locks
-3  is  20
+4  is  27
23  is  perfect!
WARNING!

The following slides may not be suitable for some audiences.
WARNING!

The next slide contains ASSEMBLY!
Silly example of dynamic function

```c
int goldie_locks(void (*bed)(int)) { (*bed)(23); }
```

goldie_locks:
```
sub $0x8,%rsp
mov %rdi,%rax
mov $0x17,%edi
callq *%rax
add $0x8,%rsp
retq
```

[..]
main:

```c
lea -0x88(%rip),%rdi  # 115f <minus3>
call goldie_locks
```
Silly example of dynamic function

```c
int goldie_locks(void (*bed)(int)) { (*bed)(23); }
```

goldie_locks:
```
sub    $0x8,%rsp
mov    %rdi,%rax
mov    $0x17,%edi
callq  *%rax
add    $0x8,%rsp
retq
```

direct function call

```c
main:
```
```
lea    -0x88(%rip),%rdi  # 115f <minus3>
call   goldie_locks
```
Silly example of dynamic function

```c
int goldie_locks(void (*bed)(int)) { (*bed)(23); }
```

goldie_locks:
```
sub $0x8,%rsp
mov %rdi,%rax
mov $0x17,%edi
callq *%rax
add $0x8,%rsp
retq
```

Indirect (dynamic) function call

```
lea -0x88(%rip),%rdi  # 115f <minus3>
call goldie_locks
```

whatever %rax is?
Spectre v2: Retpoline

- Confuse the branch predictor!
- Original code:
  
```assembly
  jmp *r11
```
- Replaced with:
  
```assembly
  call set_up_target
  capture_speculation:
  pause
  jmp capture_speculation
  set_up_target:
  mov %r11, %(%rsp)
  ret
```
Retpolines

goldie_locks:
  sub   $0x8,%rsp
  mov   %rdi,%rax
  mov   $0x17,%edi
  callq *%rax
  add   $0x8,%rsp
  retq
goldie_locks:
  sub   $0x8,%rsp
  mov   %rdi,%rax
  mov   $0x17,%edi
  callq retpoline_thunk_rax
  add   $0x8,%rsp
  retq

retpoline_thunk_rax:
  call   setup_target

capture_speculation:
  pause
  lfence
  jmp   capture_speculation

setup_target:
  mov    %rax,(%rsp)
  retq
Retpolines

goldie_locks:
  sub    $0x8,%rsp
  mov    %rdi,%rax
  mov    $0x17,%edi
  callq  retpolineThunk_rax
  add    $0x8,%rsp
  retq

retpolineThunk_rax:
  call   setup_target

capture_speculation:
  pause
  lfence
  jmp    capture_speculation

setup_target:
  mov    %rax,(%rsp)
  retq

CPU guesses

real path from %rax
What does this mean?

Dynamic functions have added overhead!
What does this mean?

Dynamic functions have added overhead!

Tracing uses lots of dynamic functions

```c
it_func_ptr = rcu_dereference_raw((tp)->funcs);

if (it_func_ptr) {
    do {
        it_func = (it_func_ptr)->func;
        __data = (it_func_ptr)->data;
        ((void(*)(proto))(it_func))(args);
    } while (((++it_func_ptr)->func);
} 
```
What does this mean?

Dynamic functions have added overhead!

Tracing uses lots of dynamic functions

```c
it_func_ptr = rcu_dereference_raw((tp)->funcs);

if (it_func_ptr) {
    do {
        it_func = (it_func_ptr)->func;
        __data = (it_func_ptr)->data;
        ((void(*)(proto))(it_func))(args);
    } while (((++it_func_ptr)->func);
}
```
What does this mean?

<table>
<thead>
<tr>
<th>Without Retpoline</th>
<th>With Retpoline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No events:</strong></td>
<td><strong>No events:</strong></td>
</tr>
<tr>
<td># time /work/c/hackbench 100</td>
<td># time /work/c/hackbench 100</td>
</tr>
<tr>
<td>Time: <strong>1.262</strong></td>
<td>Time: <strong>1.315</strong></td>
</tr>
<tr>
<td>real 0m1.446s</td>
<td>real 0m1.539s</td>
</tr>
<tr>
<td>user 0m1.499s</td>
<td>user 0m1.587s</td>
</tr>
<tr>
<td>sys 0m7.890s</td>
<td>sys 0m8.123s</td>
</tr>
<tr>
<td><strong>All events:</strong></td>
<td><strong>All events:</strong></td>
</tr>
<tr>
<td># time /work/c/hackbench 100</td>
<td># time /work/c/hackbench 100</td>
</tr>
<tr>
<td>Time: <strong>2.707</strong></td>
<td>Time: <strong>3.093</strong></td>
</tr>
<tr>
<td>real 0m2.985s</td>
<td>real 0m3.386s</td>
</tr>
<tr>
<td>user 0m1.393s</td>
<td>user 0m1.520s</td>
</tr>
<tr>
<td>sys 0m15.441s</td>
<td>sys 0m17.616s</td>
</tr>
</tbody>
</table>
Overhead

No retpoline vs retpoline no events:

- hackbench time: 1.262 vs 1.315 (4.19%)
- user time: 1.499 vs 1.587 (5.87%)
- system time: 7.890 vs 8.123 (2.9%)

No retpoline vs retpoline with events:

- hackbench time: 2.707 vs 3.093 (14.26%)
- user time: 1.393 vs 1.520 (9.12%)
- system time: 15.441 vs 17.616 (14.10%)
How to fix this?

Tracing indirect function pointers only change by user

- Not something that happens often
- User enables event then function pointer gets updated
How to fix this?

Tracing indirect function pointers only change by user
  • Not something that happens often
  • User enables event then function pointer gets updated

The function tracer does dynamic code updates
  • When user enables a function to be traced
  • Code gets modified to call that new function
  • Takes a bit of work to enable / disable
  • But that’s OK because it only happens sometimes
How to fix this?

Tracing indirect function pointers only change by user
  • Not something that happens often
  • User enables event then function pointer gets updated

The function tracer does dynamic code updates
  • When user enables a function to be traced
  • Code gets modified to call that new function
  • Takes a bit of work to enable / disable
  • But that’s OK because it only happens sometimes

See where I’m going with this?
Remember this?

goldie_locks:
  sub $0x8,%rsp
  mov %rdi,%rax
  mov $0x17,%edi
  callq *%rax
  add $0x8,%rsp
  retq
What if?

We make it call a stub function by default

goldie_locks:
    sub    $0x8,%rsp
    mov    $0x17,%edi
    callq  stub_func
    add    $0x8,%rsp
    retq  

We make it call a stub function by default
What if?

Register new function foo, and convert this to:

```
goldie_locks:
    sub    $0x8,%rsp
    mov    $0x17,%edi
    callq  foo
    add    $0x8,%rsp
    retq
```
What if?

Register new function foo, and convert this to:

```
goldie_locks:
    sub    $0x8,%rsp
    mov    $0x17,%edi
    callq  foo
    add    $0x8,%rsp
    retq
```

We do it for function tracing!
Function tracing

Compile code with `gcc -pg -mfentry`

```
<schedule>:
  0f 1f 44 00 00   nop
  41 55           push %r13
  41 54           push %r12
  55             push %rbp
  53             push %rbx
```
Function tracing

Compile code with `gcc -pg -mfentry`

```
<schedule>:
  <cc> 1f 44 00 00 00
  41 55
  41 54
  55
  53
  <int3>nop
  push %r13
  push %r12
  push %rbp
  push %rbx
```
Function tracing

Compile code with `gcc -pg -mfentry`

```
<schedule>:
    <cc> bb 1f 0a 00
    41 55
    41 54
    55
    53
    <int3>ftrace_caller
    push %r13
    push %r12
    push %rbp
    push %rbx
```
Function tracing

Compile code with `gcc -pg -mfentry`

```
<schedule>:
  e8 bb 1f 0a 00  callq ftrace_caller
  41 55           push  %r13
  41 54           push  %r12
  55             push  %rbp
  53             push  %rbx
```
How this works

<schedule>:

    <int3>callq foo
    push %r13
    push %r12
    push %rbp
    push %rbx
How this works

<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx

do_int3(struct pt_regs *regs) {
  regs->ip += 5;
  return
}
How this works

<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx

do_int3(struct pt_regs *regs) {
  regs->ip += 5;
  return
}
How this works

<schedule>:
  <int3>callq foo
    push %r13
    push %r12
    push %rbp
    push %rbx

  do_int3(struct pt_regs *regs) {
    regs->ip += 5;
    return
  }
Conversions

call function → nop

nop ← call function
Conversions

call function → nop

nop ← call function

A nop is in the state transition
Conversions

call function \[\rightarrow\] nop

nop \[\leftarrow\] call function

A nop is in the state transition

What about?

\[\text{call foo} \[\rightarrow\] \text{call bar}\]
This does not work

```c
<schedule>
<int3>callq foo
push  %r13
push  %r12
push  %rbp
push  %rbx

do_int3(struct pt_regs *regs) {
    regs->ip += 5;
    return
}
```
A call must be emulated by the int3!

This is not trivial

callq does two things
  • Push the return address on the stack
  • Moves the instruction pointer to the function

retq does two things
  • Pops the return address from the stack
  • Moves the instruction pointer to that return stack value
Why is this a problem for the int3 handler?

Emulating a callq must modify the process stack
The int3 exception appends the exception stack frame to the stack!
Why is this a problem for the int3 handler?

We hit the breakpoint

```
<schedule>:
  <int3>callq foo
    push  %r13
    push  %r12
    push  %rbp
    push  %rbx
```
Why is this a problem for the int3 handler?

The hardware appends the exception stack frame to the stack
Calls our int3 handler

<schedule>:

\[\text{\texttt{<int3>\texttt{callq foo}}}\]

- push \%r13
- push \%r12
- push \%rbp
- push \%rbx

SP

Stack Segment
Stack Pointer
Flags
Code Segment
Instruction Pointer

\textbf{do\textunderscore int3()}
Why is this a problem for the int3 handler?

We need the int3 handler to return to function foo()
But we also need to add the return address to the stack

<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx

Return Address
Why is this a problem for the int3 handler?

The return address needs to go where the exception stack frame is!

<schedule>:

<int3>callq foo
push %r13
push %r12
push %rbp
push %rbx

Return Address
Stack Pointer
Flags
Code Segment
Instruction Pointer
Not just a trace event issue!

It’s also a function tracing issue!

ftrace uses individual trampolines if only one callback is registered

```
<schedule>:
callq tramp1
push %r13
push %r12
push %rbp
push %rbx

<do_IRQ>:
callq tramp2
push %r13
push %r12
push %rbp
push %rbx

<tramp1>:
save_regs
callq callback1
restore_regs
retq

<tramp2>:
save_regs
callq callback2
restore_regs
retq
```
Not just a trace event issue!

What happens if we want to change from tramp1 to tramp2?

```
<schedule>:
  callq tramp2
  push %r13
  push %r12
  push %rbp
  push %rbx

<do_IRQ>:
  callq tramp2
  push %r13
  push %r12
  push %rbp
  push %rbx

<tramp1>:
  save_regs
  callq callback1
  restore_regs
  retq

<tramp2>:
  save_regs
  callq callback2
  restore_regs
  retq
```
Switching the callback of a traced function

No longer call function → nop
But call function1 → call function2
No nop in the state transition
The same problem!
Switching the callback of a traced function

No longer call function → nop
But call function1 → call function2
No nop in the state transition
The same problem!
Does it matter?
Switching the callback of a traced function

No longer call function → nop
But call function1 → call function2
No nop in the state transition
The same problem!
Does it matter?
Live Kernel Patching!
Live Kernel Patching

Uses ftrace to patch code

```plaintext
<schedule>:
callq lp_tramp
push %r13
push %r12
push %rbp
push %rbx

<schedule_fix>:
nop
push %r12
push %rbp
push %rbx

<lptramp>:
  save_regs
  callq live_patch
  restore_regs
  retq

live_patch(regs) {
  regs->ip = schedule_fix;
  return;
}
```
Live Kernel Patching

Uses ftrace to patch code

<schedule>:
  callq lp_tramp
  push %r13
  push %r12
  push %rbp
  push %rbx

<schedule_fix>:
  nop
  push %r12
  push %rbp
  push %rbx

<lptramp>:
  save_regs
  callq live_patch
  restore_regs
  retq

live_patch(regs) {
  regs->ip = schedule_fix;
  return;
}
Live Kernel Patching And Tracing!

Tracing schedule too!

```
<schedule>:
callq   gen_tramp
push   %r13
push   %r12
push   %rbp
push   %rbx

<gen_tramp>:
save_regs
callq   ftrace_loop
restore_regs
retq

live_patch(regs) {
  regs->ip = schedule_fix;
  return;
}
```

```
ftrace_loop(regs) {
  foreach op in ftrace_op {
    op->func(regs);
  }
  return;
}
```
Currently, int3 only emulates a nop!

See the issue here?

```
<schedule>:
  <int3>callq gen_tramp
  push %r13
  push %r12
  push %rbp
  push %rbx

<schedule_fix>:
  nop
  push %r12
  push %rbp
  push %rbx

do_int3(regs) {
  regs->ip += 5;
  return;
}
```
Changing a call

How to go from `callq lp_tramp` to `callq gen_tramp`

A nop state will bring back the broken function!

```
<schedule>:
  callq  lp_tramp
  push  %r13
  push  %r12
  push  %rbp
  push  %rbx
<schedule>:
  callq  gen_tramp
  push  %r13
  push  %r12
  push  %rbp
  push  %rbx
```
Solution #1 (for function tracing only)

Incorporate a trampoline

- Have the int3 handler return to it
- Hard coded to the ftrace loop function (to handle all callers)

The per task descriptor gets a [save stack] array of four (contexts)

- normal
- soft interrupt
- hard interrupt
- NMI

Also adds a depth count (to know what depth it is in)
Solution #1

<schedule>:
  <int3>callq lptramp
  push %r13
  push %r12
  push %rbp
  push %rbx

  do_int3(regs) {
    d = curr->depth++;
    curr->stack[d] = regs->ip+5;
    regs->ip = int3_tramp;
    return;
  }

<schedule_fix>:
  nop
  push %r12
  push %rbp
  push %rbx

  save_regs
  d = curr->depth-1;
  *regs->sp = curr->stack[d];
  callq ftrace_loop
  restore_regs
  retq
Solution #1

Linus Torvalds (and pretty much everyone else) hated it!
Solution #1

Linus Torvalds (and pretty much everyone else) hated it!
Solution #2

Move the stack on interrupt / exception entry
  • Make a buffer to store the ret

The int3 handler can now extend the process stack
  • Add the return address
  • Modify the stack pointer

Call the function directly!
Solution #2

We hit the breakpoint

<schedule>:

<int3>callq foo
push %r13
push %r12
push %rbp
push %rbx
Solution #2

The hardware appends the exception stack frame to the stack.

```
<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx
```

![Stack Segment Diagram]

- Stack Segment
- Stack Pointer
- Flags
- Code Segment
- Instruction Pointer

**SP**
Solution #2

The exception assembly code moves it (before calling do_int3!)

<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx

Stack Segment
Stack Pointer
Flags
Code Segment
Instruction Pointer

SP
Solution #2

do_int3() knows about the space and can add the return address

<schedule>:

<int3>callq foo

push %r13
push %r12
push %rbp
push %rbx

do_int3()
Solution #2

do_int3() knows about the space and can add the return address

<schedule>:

\begin{verbatim}
<int3>callq foo
push %r13
push %r12
push %rbp
push %rbx
\end{verbatim}

Return Address
Solution #2

The problem with x86 32 bit!
Solution #2

The exception stack frame

```
Old SP

Stack Segment
Old SP
Flags
Code Segment
Instruction Pointer

SP

64 Bit
```
Solution #2

The exception stack frame

64 Bit

Old SP

Stack Segment
Old SP
Flags
Code Segment
Instruction Pointer

32 Bit

Old SP

Flags
Code Segment
Instruction Pointer

SP
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = regs->sp;
```
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = regs->sp;
```
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = regs->sp;
```

stack_pointer == 1!
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = regs->sp;
```

regs->sp will fault!
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = regs->sp;
```

32 Bit

- Old SP
- SP

Flags
Code Segment
Instruction Pointer
Solution #2

The exception stack frame

```c
struct pt_regs {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};

unsigned long stack_pointer;
stack_pointer = (unsigned long)&regs->sp;
```
Solution #2

The hardware appends the exception stack frame to the stack

```assembly
<schedule>:
    <int3>call foo
    push %ebp
    mov %esp, %ebp
    push %ebx
```
Solution #2

The exception assembly code moves it (before calling do_int3!)

<schedule>:
  <int3>call foo
  push %ebp
  mov %esp,%ebp
  push %ebx
Solution #2

Linus Torvalds hated it!
Solution #2

Linus Torvalds hated it!

But everyone else liked it! (it seemed the best of the choices)
Solution #2

Linus Torvalds hated it!

But everyone else liked it! (it seemed the best of the choices)

I think Linus has a special place for the i386 architecture
  • He doesn’t want it to change!
Solution #3

Linus’s solution #1

• Use per cpu variables
  - Save the return address
  - For trace events: save the call
  - For ftrace: hard code the ftrace generic loop function

• Requires several trampolines
  - ftrace update (for updating trampoline)
  - ftrace caller (for updating call site)
  - ftrace update sti (keep interrupts enable)
  - ftrace caller sti
  - ftrace update nmi (for when it is called from an NMI)
  - ftrace caller nmi
Solution #3

do_int3() assigns the proper trampoline to jump to

```
<schedule>:
  <int3>callq foo
  push %r13
  push %r12
  push %rbp
  push %rbx

  do_int3(regs) {
    if (in_nmi())
      per_cpu(nmi_ret) = regs->ip+5;
      regs->ip = ftrace_nmi_tramp;
    else {
      per_cpu(norm_ret) = regs->ip+5;
      if (regs->flags & IRQS_DISABLED)
        regs->ip = ftrace_norm_tramp;
      else {
        regs->ip = ftrace_sti_tramp;
        regs->flags |= IRQS_DISABLED;
      }
    }
  }
```
do_int3() returns to trampoline that emulates the call

```
<frtrace_nmi_tramp>:
    push    per_cpu(nmi_ret)
    jmp     *ftrace_regs_caller

<frtrace_norm_tramp>:
    push    per_cpu(norm_ret)
    jmp     *ftrace_regs_caller

<frtrace_sti_tramp>:
    push    per_cpu(norm_ret)
    sti    // enables interrupts!
    jmp     *ftrace_regs_caller
```
Solution #3

Linus obviously liked this

Nobody else did due to issues

- Lockdep problems
- Interacting with kernel shadow stack security logic
- A trampoline for every call type

Linus asked for to compare it with Solution #2
Solution #3

Linus asked for to compare it with Solution #2
Solution #3

Linus asked for to compare it with Solution #2
  • He finally agreed #2 was better
Solution #4

Linus’s solution #2

- Only modify int3
- Only add a one word gap after saved regs
- Shift the entire saved regs and exception frame by one word!
- Need separate do_kernel_int3 to act different that from user space
- do_kernel_int3 returns new shifted regs!
Solution #4

Entry from the int3 (after hit, before calling do_int3)

<int3_entry>:

- mov %esp,%eax
- sub $8,%esp
- call do_kernel_int3
- mov %eax,%esp
Solution #4

Add space after saved regs

<int3_entry>:

mov %esp, %eax // regs
sub $8, %esp
call do_kernel_int3
mov %eax, %esp
Solution #4

Call the kernel int3 handler

\[
\text{<int3_entry>:}
\]
- \text{mov } \%\text{esp},\%\text{eax} \ // \text{regs}
- \text{sub } \$8,\%\text{esp}
- \text{call } \text{do_kernel_int3}
- \text{mov } \%\text{eax},\%\text{esp}
Solution #4

Non emulating call breakpoint returns the regs that was passed (in %eax)

```
<int3_entry>:
    mov    %esp,%eax // regs
    sub    $8,%esp
    call   do_kernel_int3
    mov    %eax,%esp
```
Solution #4

Setting to the passed in regs, returns the stack back

```
<int3_entry>:
  mov %esp,%eax // regs
  sub $8,%esp
  call do_kernel_int3
  mov %eax,%esp
```

![Diagram showing instruction sequence]
Solution #4

But for emulating a call

<int3_entry>:

```
    mov       %esp,%eax    // regs
    sub       $8,%esp
    call      do_kernel_int3
    mov       %eax,%esp
```

```
Flags
Code Segment
Instruction Pointer
```
Solution #4

do_kernel_int3 does

- `memmove((void *)regs - 4, regs, sizeof(regs))`

```c
<int3_entry>:
    mov    %esp,%eax  // regs
    sub    $8,%esp
    call   do_kernel_int3
    mov    %eax,%esp
```

```
SP   Flags
Code Segment
Instruction Pointer
```
Solution #4

do_kernel_int3

- Adds return address to new space
- Replaces regs->ip to the target call function

<int3_entry>:

\[
\begin{align*}
&\text{mov} & \%esp,\%eax & \text{// regs} \\
&\text{sub} & $8,\%esp \\
&\text{call} & \text{do_kernel_int3} \\
&\text{mov} & \%eax,\%esp
\end{align*}
\]
Solution #4

do_kernel_int3

- Adds return address to new space
- Replaces regs->ip to the target call function

<int3_entry>:

```
mov   %esp,%eax // regs
sub   $8,%esp
```

```
call  do_kernel_int3
```

```
mov   %eax,%esp
```

```
```
Solution #4

Linus obviously liked this

Caused too much change in x86_64

- Why break the hardware of the future for the hardware of the past
What did we do?

Solution #2

- For x86_64 only (so far)

x86 32bit

- Still not implemented
- We may still do #2?
What did we do?

Solution #2

- For x86_64 only (so far)

x86 32bit

- Still not implemented
- We may still do #2?
- Or just let it die!
Thank You