
ENTRYBLEED

A Universal KASLR Bypass against KPTI on Linux

William Liu, Joseph Ravichandran, Mengjia Yan



compute. collaborate. create.

Where does **EntryBleed** fit in?

hardware

**μarch
side-channel
attacks**

software

**memory
corruption
defenses**

hardware

**μarch
side-channel
attacks**

software

**memory
corruption
defenses**

ENTRYBLEED

hardware

**μarch
side-channel
attacks**

Contributions

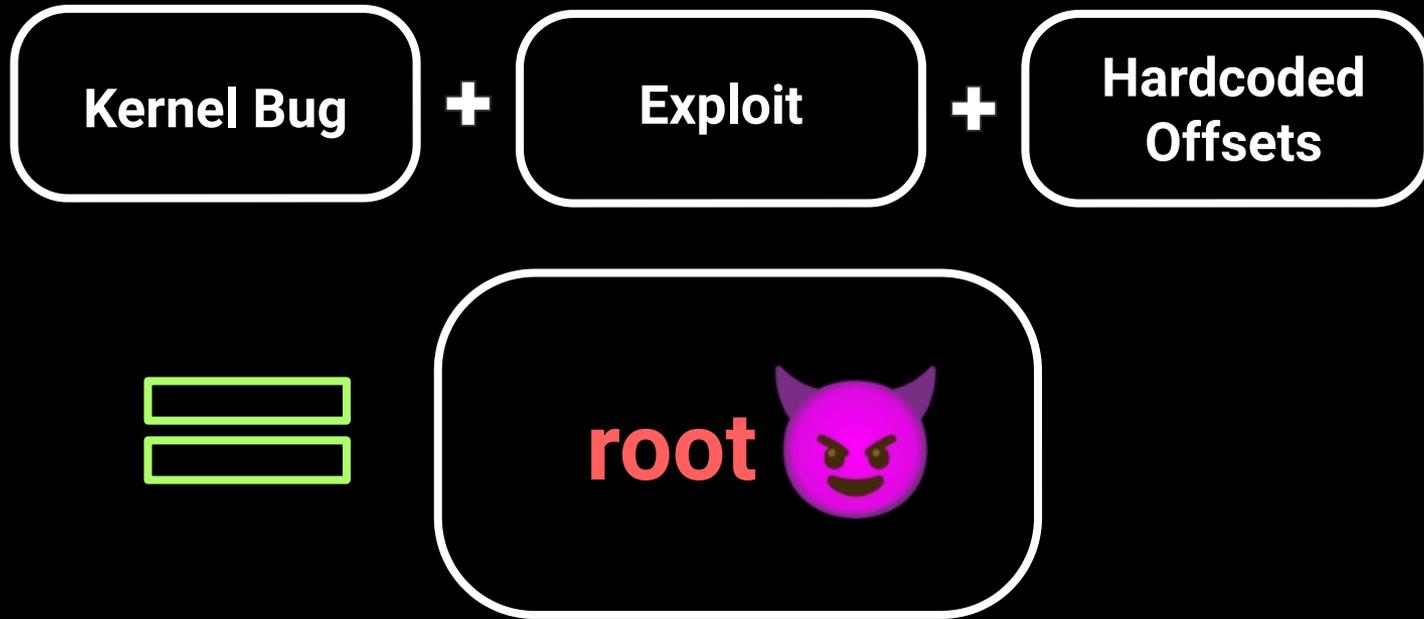
Reveal
misconception in
KPTI security

CVE-2022-4543:
KASLR bypass on
bare metal and VT-x

Provide **root cause**
analyses

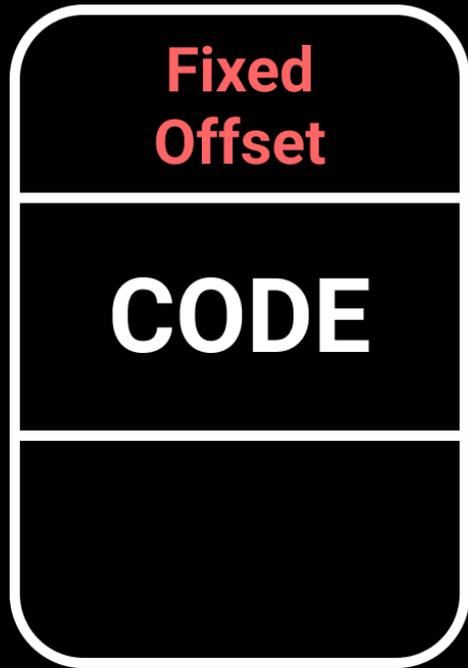
Why should we care? 🤔

Back in the old days...

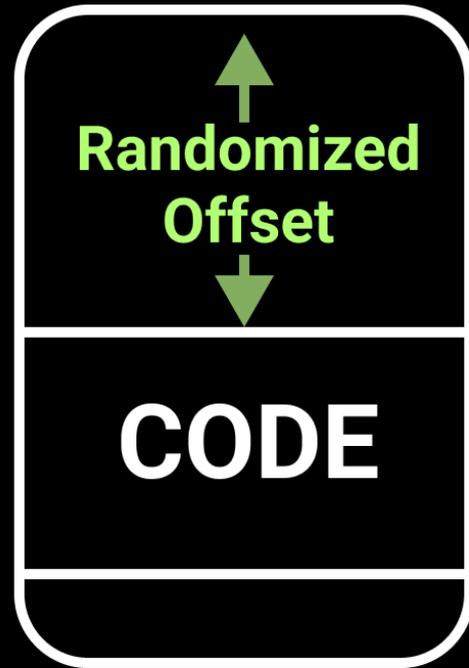


KASLR

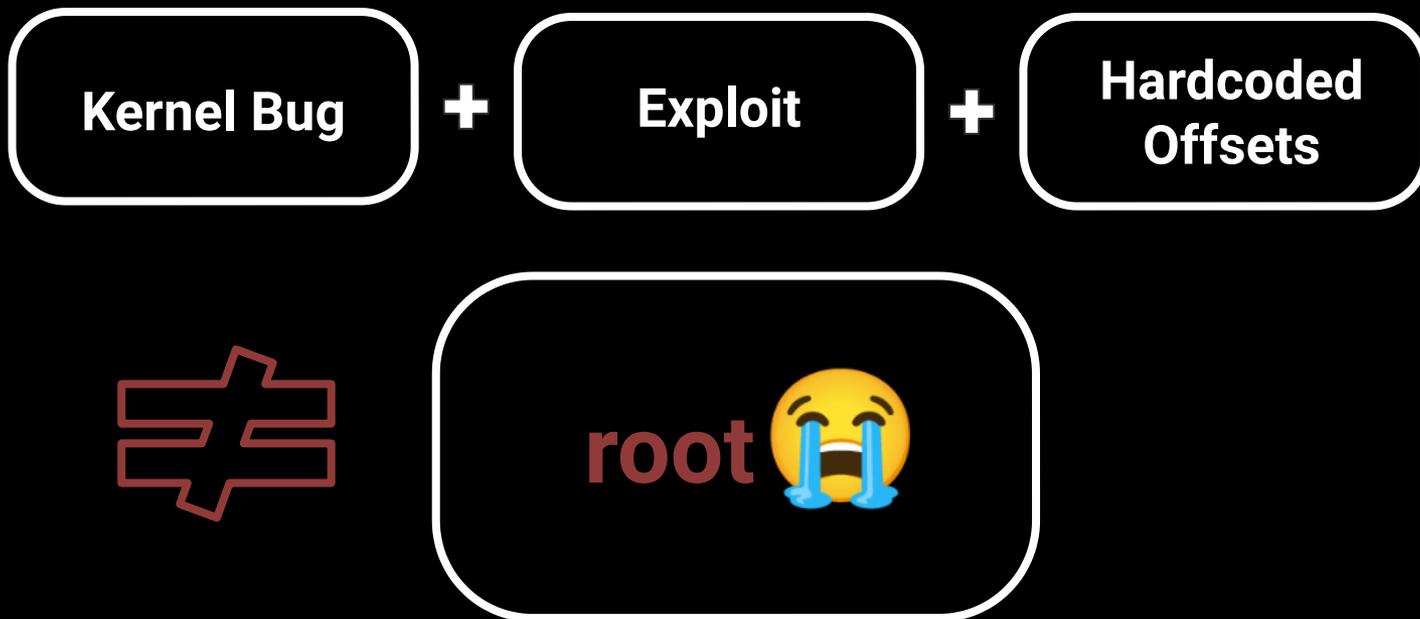
pre-ASLR



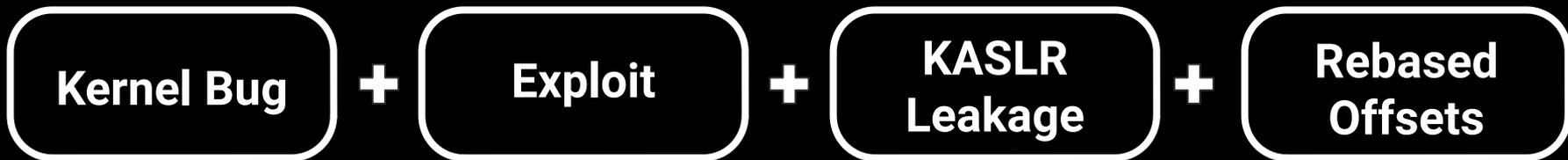
ASLR



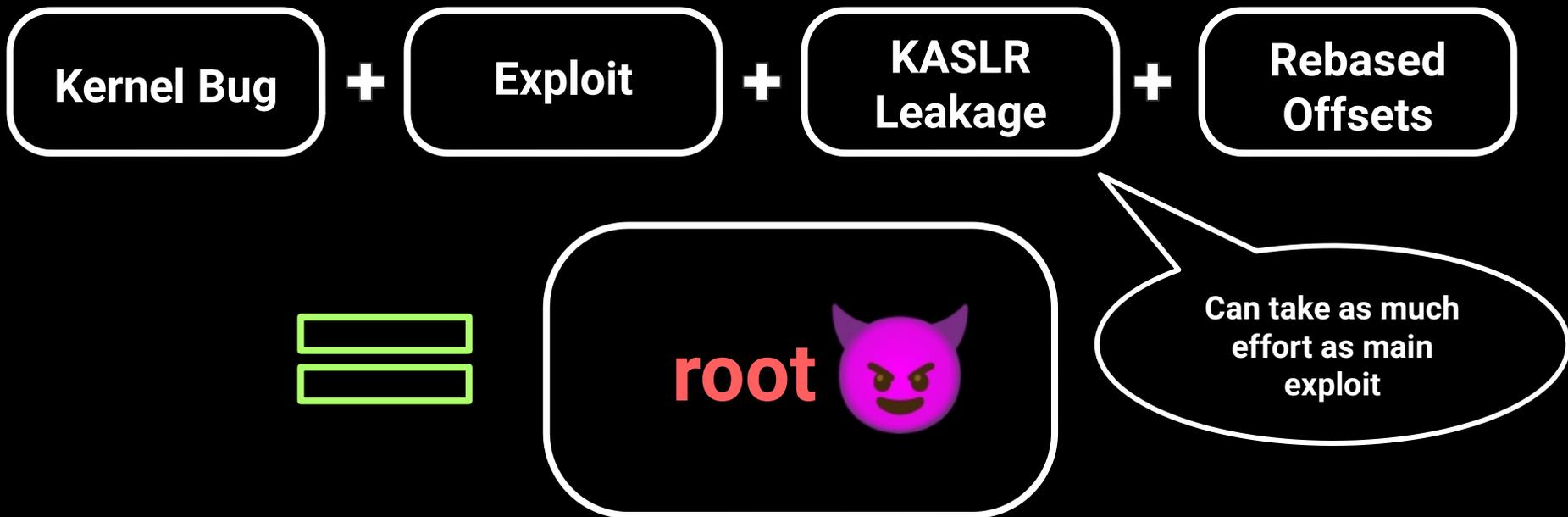
With KASLR



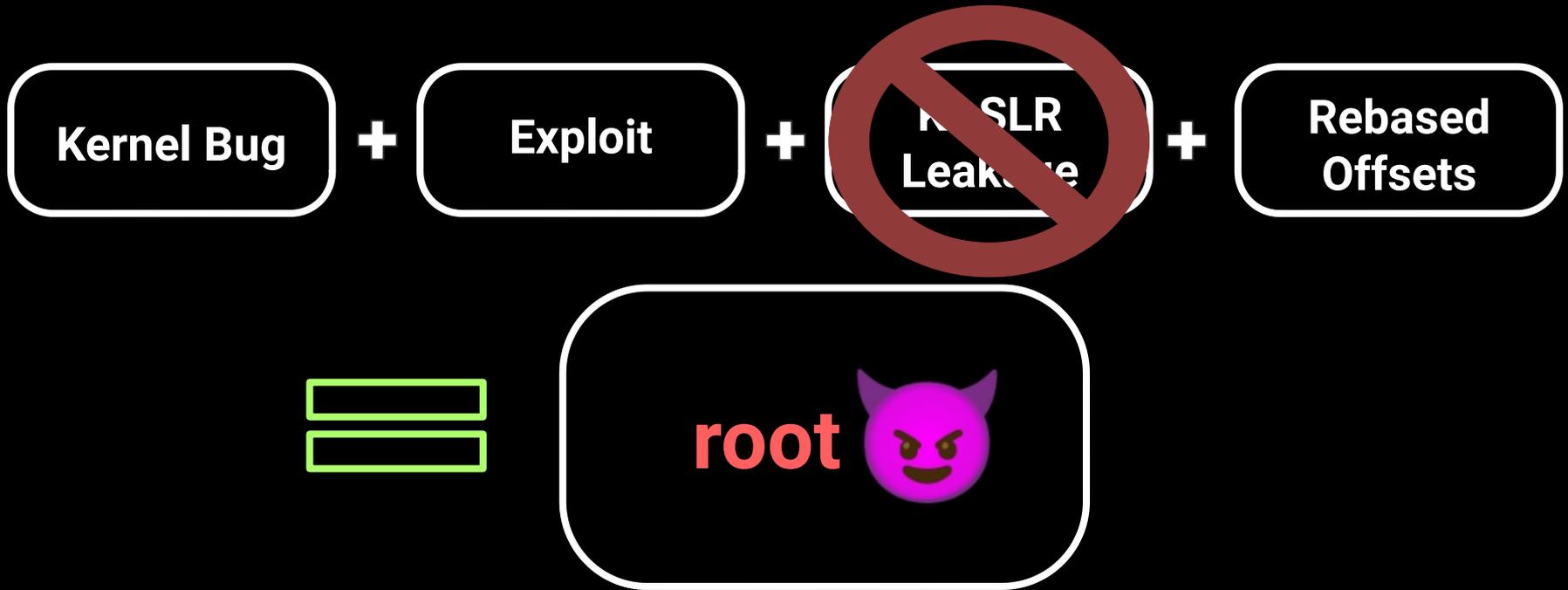
With KASLR



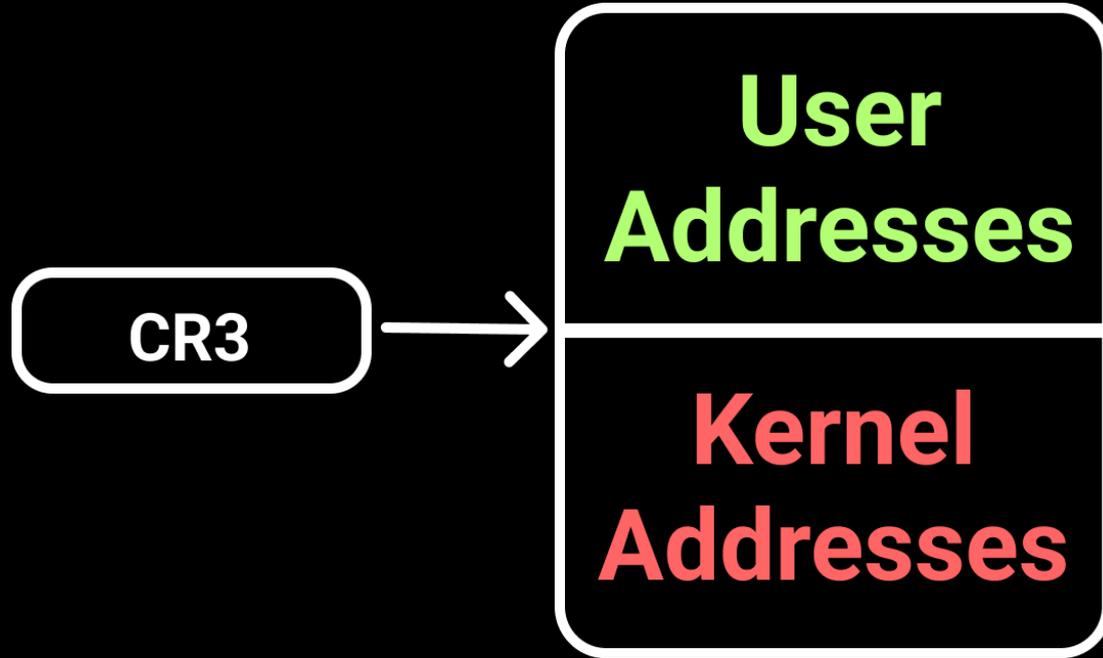
With KASLR



With EntryBleed



Pre-Meltdown



Prefetch Attack

Prefetch Attack

- If a VA is **invalid**

Prefetch Attack

- If a VA is **invalid**
 - **No ISA exceptions**

Prefetch Attack

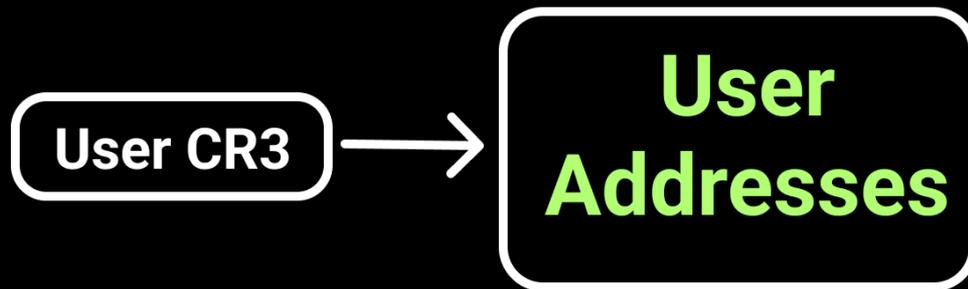
- If a VA is **invalid**
 - No ISA exceptions
 - But takes **longer**

Prefetch Attack

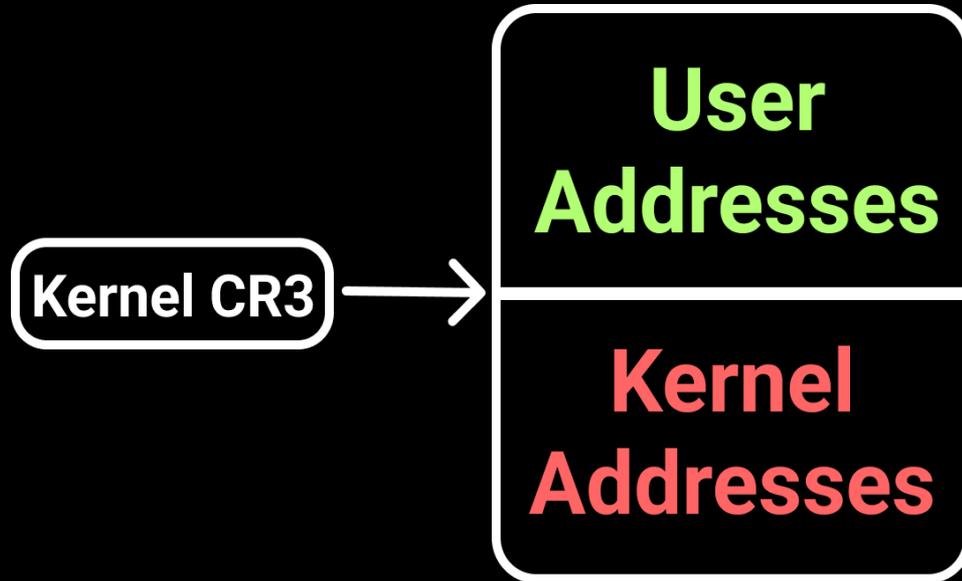
- If a VA is **invalid**
 - No ISA exceptions
 - But takes **longer**

side-channel vector 

KPTI



KPTI



Does the prefetch attack still work?

Prefetch vs. KPTI

Fetching the KASLR slide with prefetch

Upon reporting this bug to the Linux kernel security team, our suggestion was to start randomizing the location of the percpu `cpu_entry_area` (CEA), and consequently the associated exception and syscall entry stacks. This is an effective mitigation against remote attackers but is insufficient to prevent a local attacker from taking advantage. 6 years ago, Daniel Gruss et al. [discovered a new more reliable technique for exploiting the TLB timing side channel in x86 CPU's](#). Their results demonstrated that prefetch instructions executed in user mode retired at statistically significant different latencies depending on whether the requested virtual address to be prefetched was mapped vs unmapped, even if that virtual address was only mapped in kernel mode. **kPTI was helpful in mitigating this side channel** however, most modern CPUs now have innate protection for Meltdown, which kPTI was specifically designed to address, and thusly kPTI (which has significant performance implications) is disabled on modern microarchitectures. That decision means it is once again possible to take advantage of the prefetch side channel to defeat not only KASLR, but also the CPU entry area randomization mitigation, preserving the viability of the CEA stack corruption exploit technique against modern X86 CPUs.

There are surprisingly few fast and reliable examples of this prefetch KASLR bypass technique available in the open source realm, so I made the decision to write one.

Prefetch vs. KPTI

Fetching the KASLR slide with prefetch

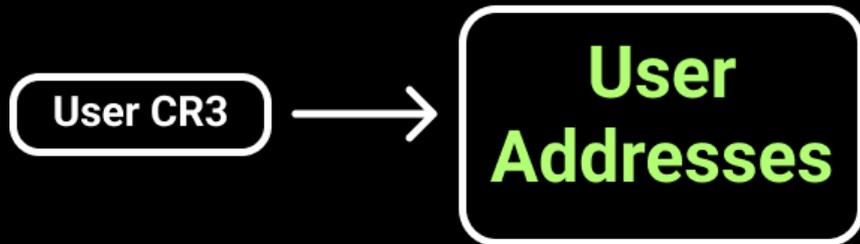
Upon reporting this bug to the Linux kernel security team, our suggestion was to start randomizing the location of the percpu `cpu_entry_area` (CEA), and consequently the associated exception and syscall entry stacks. This is an effective mitigation against remote attackers but is insufficient to prevent a local attacker from taking advantage. 6 years ago, Daniel Gruss et al. [discovered a new more reliable technique for exploiting the prefetch side channel](#) actions executed **kPTI was helpful in mitigating this side channel** requested virtual address to be prefetched was mapped vs unmapped, even if that virtual address was only mapped in kernel mode. **kPTI was helpful in mitigating this side channel** however, most modern CPUs now have innate protection for Meltdown, which kPTI was specifically designed to address, and thusly kPTI (which has significant performance implications) is disabled on modern microarchitectures. That decision means it is once again possible to take advantage of the prefetch side channel to defeat not only KASLR, but also the CPU entry area randomization mitigation, preserving the viability of the CEA stack corruption exploit technique against modern X86 CPUs.

There are surprisingly few fast and reliable examples of this prefetch KASLR bypass technique available in the open source realm, so I made the decision to write one.

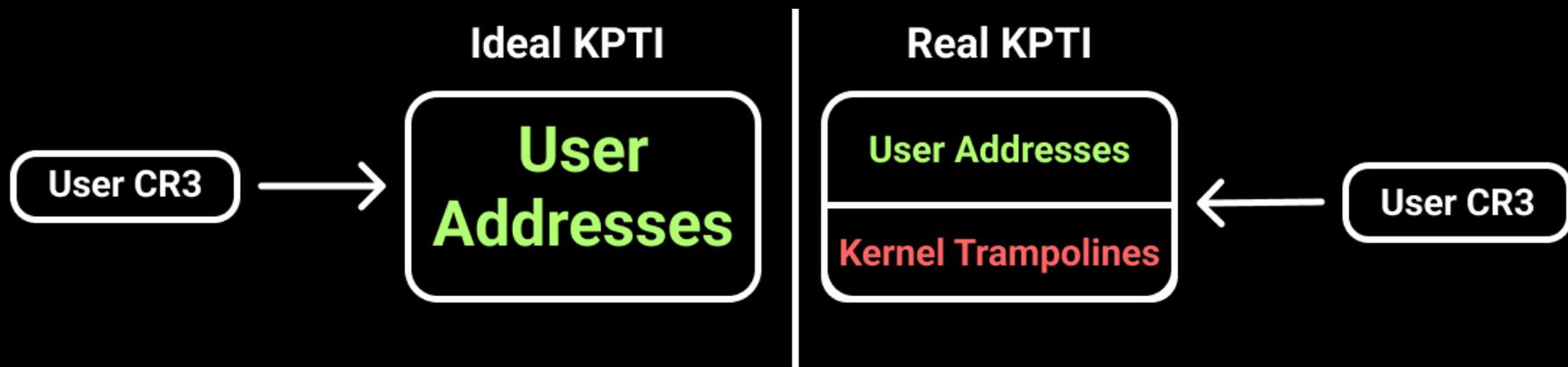
But...

Isolation Flaw in KPTI

Ideal KPTI

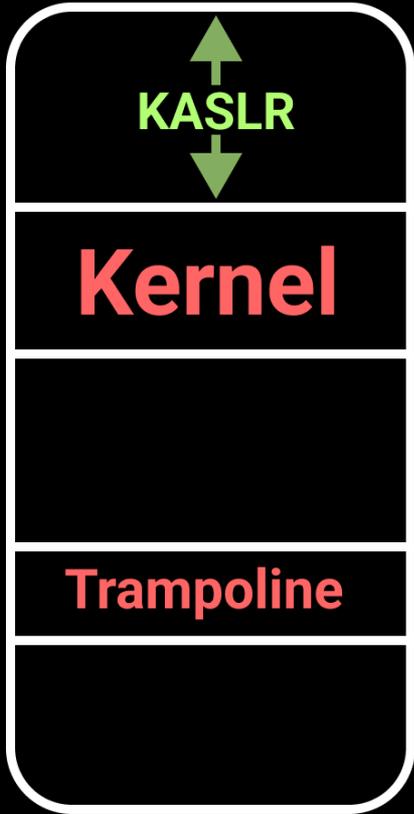


Isolation Flaw in KPTI

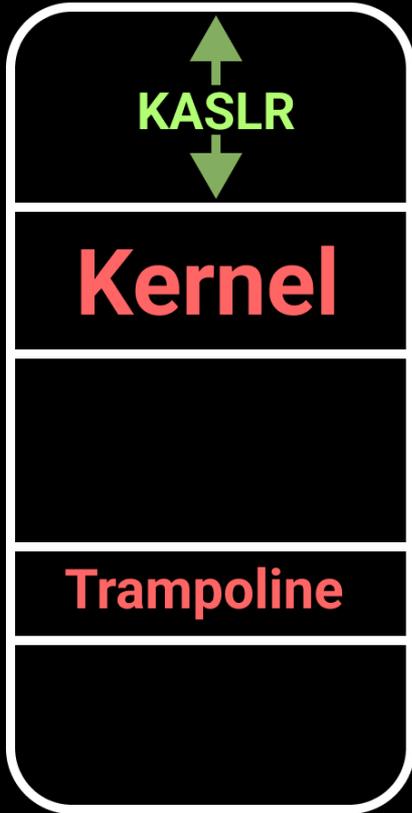


What VA is reasonable for this mapping?

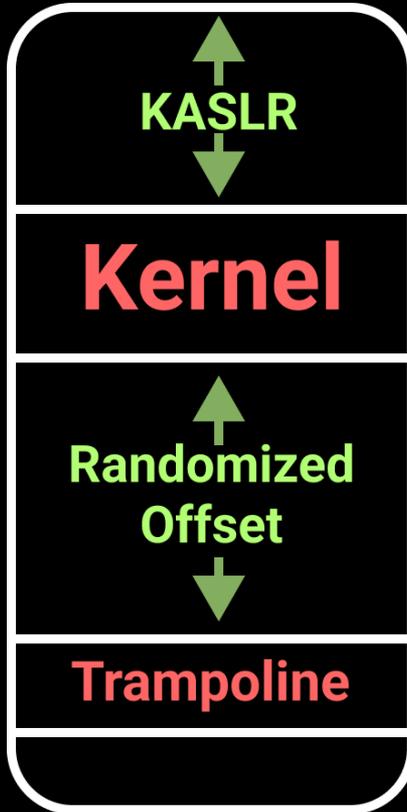
1



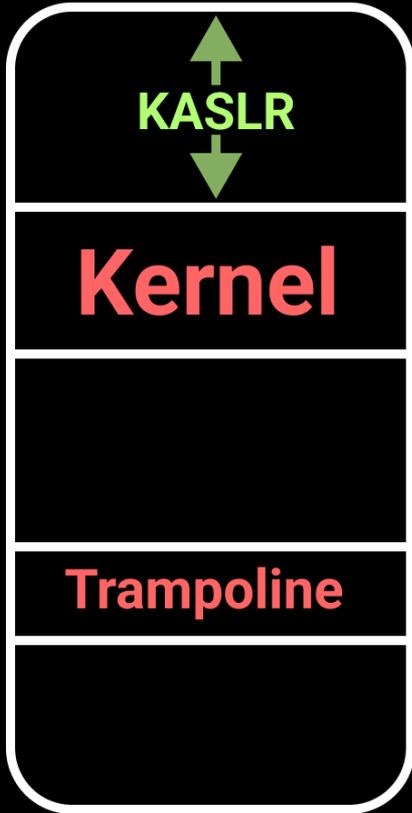
1



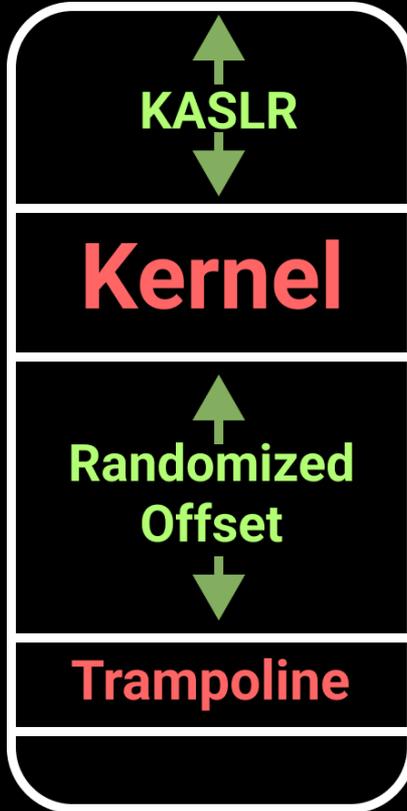
2



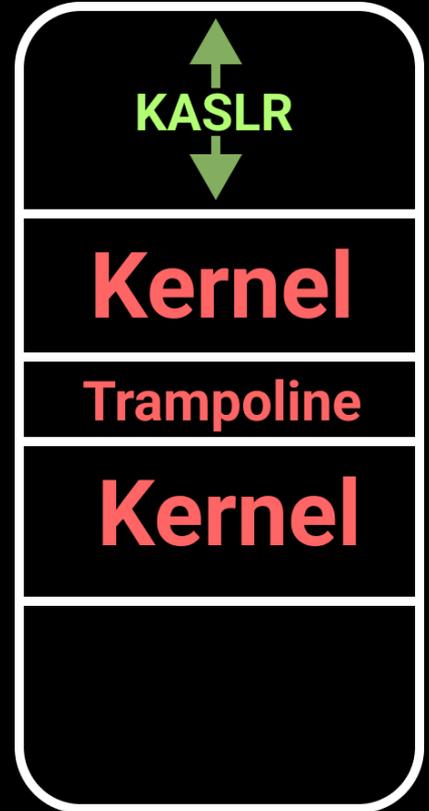
1

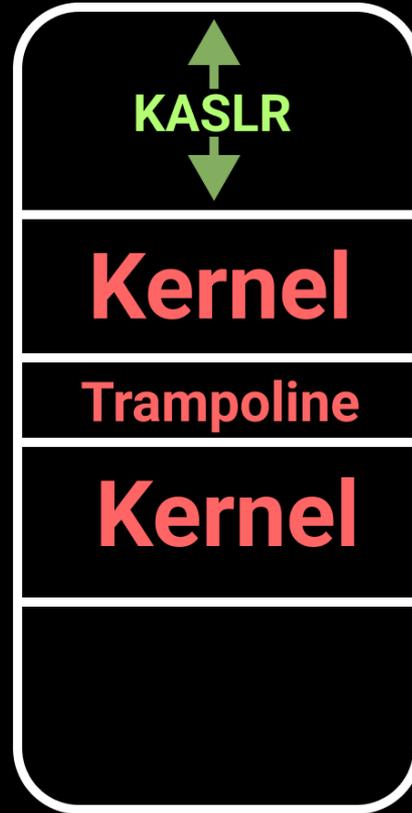


2

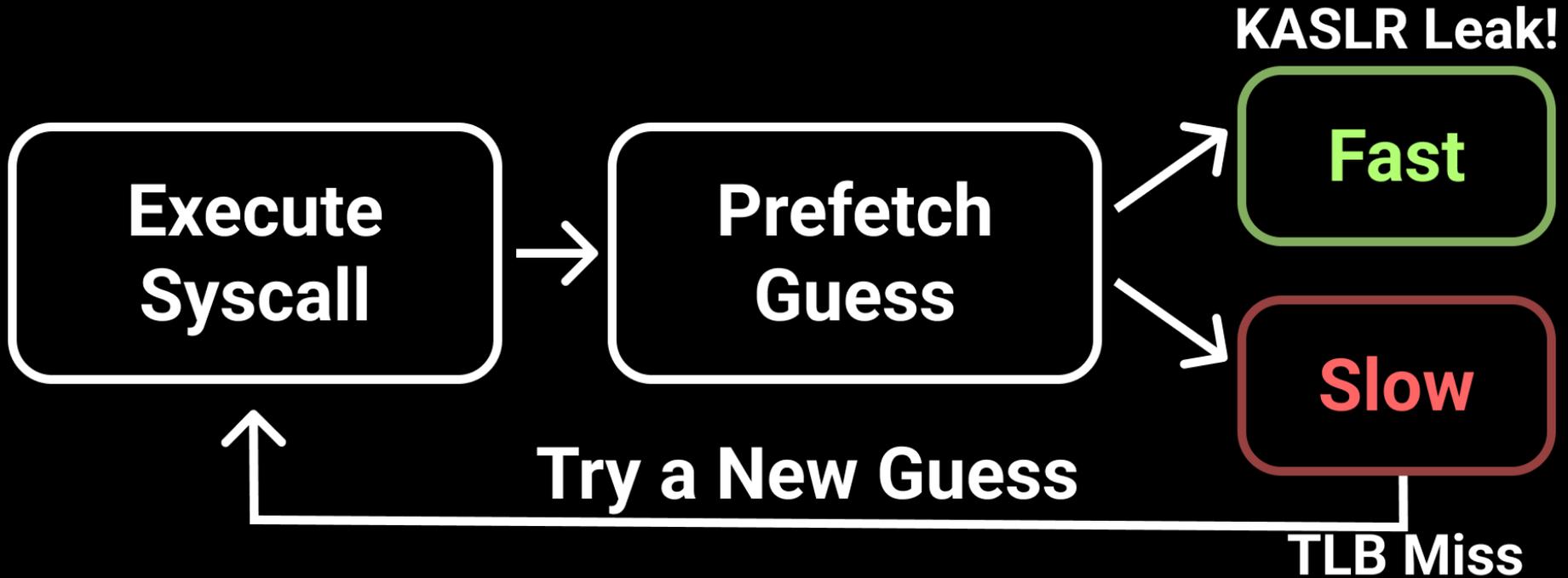


3





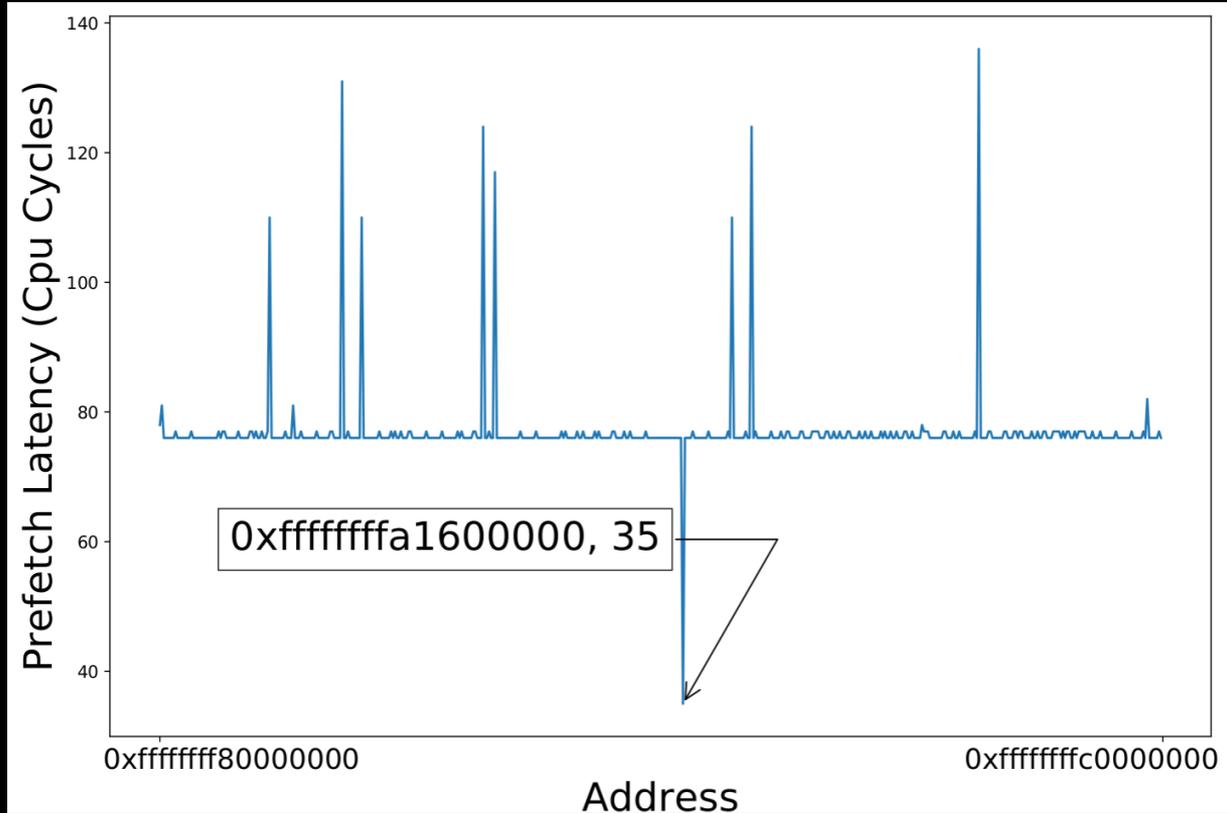
Attack Strategy



Attack Strategy

- **Bruteforce range**
 - **Start: 0xffffffff80000000**
 - **End: 0xffffffffc0000000**
- **Increment by 2MB**

Results



Results

CPU Model	Kernel Version	Average Leakage Time (s)	Accuracy Rate
Intel i5-4590	5.4.0-146	0.2236	100%
Intel i7-7950H	5.15.0-83	0.2761	99.7%
Intel i7-6700	5.15.0-67	0.1334	99.6%
Intel i7-7950H (KVM)	5.15.0-58	0.4148	99.9%

DEMO

USER

```
test@arch-sec-xss:~$
```

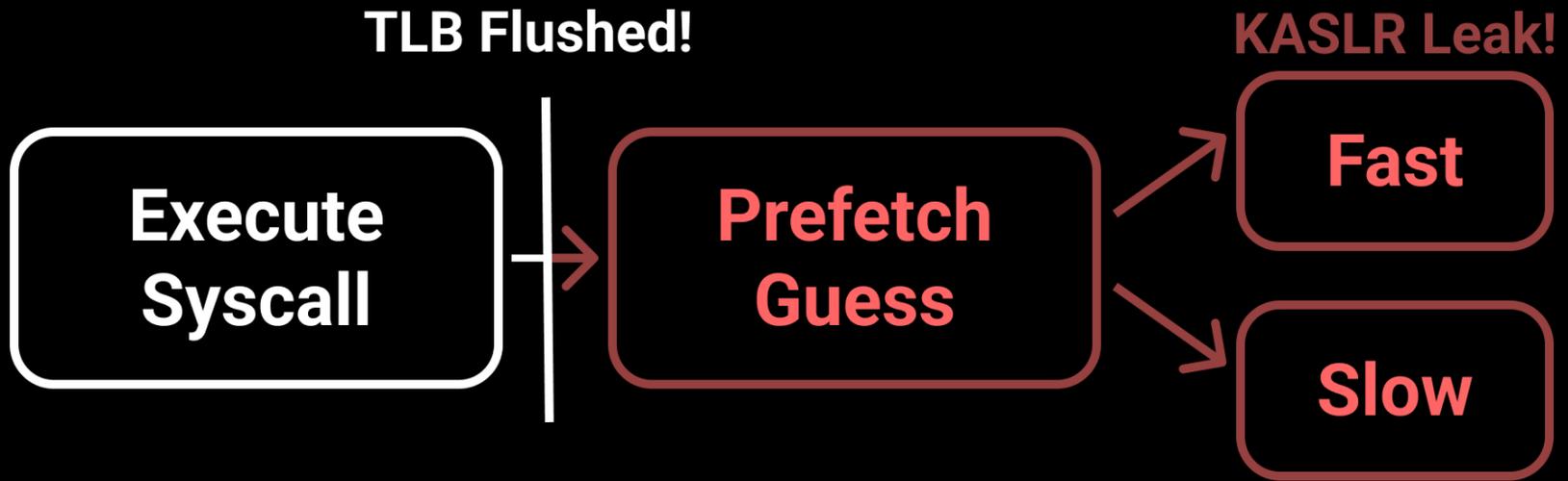
```
[0] 0:bash*
```

ROOT

```
root@arch-sec-xss:/home/test#
```

```
"arch-sec-xss.csail.mit.edu" 14:04 20-Oct-23
```

How can prefetch work after address space switch?



```

198 syscall_return_via_sysret:
199     IBRS_EXIT
200     POP_REGS pop_rdi=0
201
202     /*
203      * Now all regs are restored except RSP and RDI.
204      * Save old stack pointer and switch to trampoline stack.
205      */
206     movq    %rsp, %rdi
207     movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
208     UNWIND_HINT_END_OF_STACK
209
210     pushq   RSP-RDI(%rdi) /* RSP */
211     pushq   (%rdi)        /* RDI */
212
213     /*
214      * We are on the trampoline stack. All regs except RDI are live.
215      * We can do future final exit work right here.
216      */
217     STACKLEAK_ERASE_NOCLOBBER
218
219     SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi
220
221     popq    %rdi
222     popq    %rsp
223     SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
224     ANNOTATE_NOENDBR
225     swpgs
226     sysretq
227     SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_L_GLOBAL)
228     ANNOTATE_NOENDBR
229     int3
230     SYM_CODE_END(entry_SYSCALL_64)

```

syscall_return_via_sysret:

```
199     IBRS_EXIT
200     POP_REGS pop_rdi=0
201
202     /*
203      * Now all regs are restored except RSP and RDI.
204      * Save old stack pointer and switch to trampoline stack.
205      */
206     movq   %rsp, %rdi
207     movq   PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
208     UNWIND_HINT_END_OF_STACK
209
210     pushq  RSP-RDI(%rdi) /* RSP */
211     pushq  (%rdi)        /* RDI */
212
213     /*
214      * We are on the trampoline stack. All regs except RDI are live.
215      * We can do future final exit work right here.
216      */
217     STACKLEAK_ERASE_NOCLOBBER
218
219     SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi
220
221     popq   %rdi
222     popq   %rsp
223     SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
224     ANNOTATE_NOENDBR
225     swapgs
226     sysretq
227     SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_L_GLOBAL)
228     ANNOTATE_NOENDBR
229     int3
230     SYM_CODE_END(entry_SYSCALL_64)
```

```

198  syscall_return_via_sysret:
199      IBRS_EXIT
200      POP_REGS pop_rdi=0
201
202      /*
203       * Now all regs are restored except RSP and RDI.
204       * Save old stack pointer and switch to trampoline stack.
205       */
206      movq   %rsp, %rdi
207      movq   PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
208      UNWIND_HINT_END_OF_STACK
209
210      pushq  RSP-RDI(%rdi) /* RSP */
211      pushq  (%rdi) /* RDI */
212
213      /*
214       * We are on the trampoline stack. All regs except RDI are live.
215       * We can do future final exit work right here.
216       */
217      STACKLEAK_ERASE_NOCLOBBER

```

SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

```

221      popq   %rdi
222      popq   %rsp
223      SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
224      ANNOTATE_NOENDBR
225      swapgs
226      sysretq
227      SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_L_GLOBAL)
228      ANNOTATE_NOENDBR
229      int3
230      SYM_CODE_END(entry_SYSCALL_64)

```

```

198  syscall_return_via_sysret:
199      IBRS_EXIT
200      POP_REGS pop_rdi=0
201
202      /*
203       * Now all regs are restored except RSP and RDI.
204       * Save old stack pointer and switch to trampoline stack.
205       */
206      movq   %rsp, %rdi
207      movq   PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
208      UNWIND_HINT_END_OF_STACK
209
210      pushq  RSP-RDI(%rdi) /* RSP */
211      pushq  (%rdi) /* RDI */
212
213      /*
214       * We are on the trampoline stack. All regs except RDI are live.
215       * We can do future final exit work right here.
216       */
217      STACKLEAK_ERASE_NOCLOBBER
218
219      SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi
220
221      popq   %rdi
222      popq   %rsp
223      SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
224      ANNOTATE_NOENDBR
225      swapgs
226
227      SYM_CODE_LABEL(entry_sysretq, _end, SYM_L_GLOBAL)
228      int3
229
230      SYM_CODE_END(entry_SYSCALL_64)

```

Attacking Guest OS

- How does side-channel fare under VM exits? 🤔

Read the paper!

**Effects of VM
MMU
Optimizations**

**Fully Working
POC**

**Mitigation
Proposal**

**EntryBleed against
VM Exits**

More Data

Takeaways

Takeaways

- **KPTI** is **insufficient** against prefetch

Takeaways

- **KPTI** is **insufficient** against prefetch
- An **unpatched Linux KASLR bypass** on Intel

Takeaways

- **KPTI** is **insufficient** against prefetch
- An **unpatched Linux KASLR bypass** on Intel
- Lowers **exploitation** difficulty



willsroot.io



compute. collaborate. create.

Questions?



will@willsroot.io