ENTRYBLEED

A Universal KASLR Bypass against KPTI on Linux

William Liu, Joseph Ravichandran, Mengjia Yan
Where does EntryBleed fit in?
hardware

μarch
side-channel
attacks
software
memory
corruption
defenses

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...

Kernel Bug + Exploit + Hardcoded Offsets = root 😈
KASLR

pre-ASLR

Fixed Offset

CODE

ASLR

Randomized Offset

CODE
With KASLR

Kernel Bug + Exploit + Hardcoded Offsets

root 😭
With KASLR

Kernel Bug + Exploit + KASLR Leakage + Rebased Offsets

root 😈
With KASLR

Kernel Bug + Exploit + KASLR Leakage + Rebased Offsets

root 🧟‍♂️

Can take as much effort as main exploit
With EntryBleed

Kernel Bug + Exploit + KASLR Leakage + Rebased Offsets = root 😈
Pre-Meltdown

CR3 → User Addresses

Kernel Addresses
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 😈
Kernel CR3

User Addresses

Kernel Addresses
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 CPUs. 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 KASLR slide with prefetch. 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.
But...
Isolation Flaw in KPTI
Isolation Flaw in KPTI
What VA is reasonable for this mapping?
KASLR

Kernel

Trampoline
Attack Strategy

1. Execute Syscall
2. Prefetch Guess
   - Fast (KASLR Leak)
   - Slow (TLB Miss)

Try a New Guess
Attack Strategy

- Brute force range
  - Start: 0xffffffff80000000
  - End: 0xfffffffffc000000
- Increment by 2MB
Results
## Results

<table>
<thead>
<tr>
<th>CPU Model</th>
<th>Kernel Version</th>
<th>Average Leakage Time (s)</th>
<th>Accuracy Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel i5-4590</td>
<td>5.4.0-146</td>
<td>0.2236</td>
<td>100%</td>
</tr>
<tr>
<td>Intel i7-7950H</td>
<td>5.15.0-83</td>
<td>0.2761</td>
<td>99.7%</td>
</tr>
<tr>
<td>Intel i7-6700</td>
<td>5.15.0-67</td>
<td>0.1334</td>
<td>99.6%</td>
</tr>
<tr>
<td>Intel i7-7950H (KVM)</td>
<td>5.15.0-58</td>
<td>0.4148</td>
<td>99.9%</td>
</tr>
</tbody>
</table>
USER

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

ROOT

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

[0] 0:bash*   "arch-sec-xss.csail.mi" 14:04 20-Oct-23
How can prefetch work after address space switch?
syscall_return_via_sysret:
  IBR5_EXIT
  POP_REGS pop_rdi=0

  /*
   * Now all regs are restored except RSP and RDI.
   * Save old stack pointer and switch to trampoline stack.
   */
  movq    %rsp, %rdi
  movq    PER_CPU_VAR(cpu_tss_rw TSS_sp0), %rsp
  UNIND_HINT_END_OF_STACK

  pushq   RSP-RDI(%rdi) /* RSP */
  pushq   (%rdi)      /* RDI */

  /*
   * We are on the trampoline stack. All regs except RDI are live.
   * We can do future final exit work right here.
   */
  STACKLEASE_ERASE_NOCLOBBER

  SWITCH_TO_USER_CR3_STACK scratch_reg%rdi

  popq    %rdi
  popq    %rsp

  SVM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SVM_L_GLOBAL)
  ANNOTATE_NOENDBR
  swaps
  svmretq

  SVM_INNER_LABEL(entry_SYSRETQ_end, SVM_L_GLOBAL)
  ANNOTATE_NOENDBR
  int3

  SYM_CODE_END(entry_SYSCALL_64)
### syscall_return_via_sysret:

199 IBRS_EXIT
200 POP_REGS pop_rdl0
201 /*
202 * Now all regs are restored except RSP and RDI.
203 * Save old stack pointer and switch to trampoline stack.
204 */
205 movq %rsp, %rdi
206 movq PER_CPU_VAR(cpu_tss_rw TSS_sp0), %rsp
207 UNIND_HINT_END_OF_STACK
208
209 pushq RSP-RDI(%rdi) /* RSP */
210 pushq (%rdi) /* RDI */
211 /*
212 * We are on the trampoline stack. All regs except RDI are live.
213 * We can do future final exit work right here.
214 */
215 STACKLEAK_ERASE_NOClobber
216
217 SWITCH_TO_USER_CR3_STACK scratch_reg%rdi
218
219 popq %rdi
220 popq %rsp
221 SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
222 ANNOTATE_NOENDBR
223 swapgs
224 sysretq
225 SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_L_GLOBAL)
226 ANNOTATE_NOENDBR
227 int3
228 SYM_CODE_END(entry_SYSCALL_64)
syscall_return_via_sysret:

    IBRS_EXIT
    POP_REGS pop_rdi=0

    /*
    * Now all regs are restored except RSP and RDI.
    * Save old stack pointer and switch to trampoline stack.
    */
    movq %rsp, %rdi
    movq PER_CPU_VAR(cpu_tss_rw # TSS_sp0), %rsp
    UNIND_HINT_END_OF_STACK

    pushq RSP-RDI(%rdi) /* RSP */
    pushq (%rdi) /* RDI */

    /*
    * We are on the trampoline stack. All regs except RDI are live.
    * We can do future final exit work right here.
    */
    STACKLEAK_ERASE_NOClobber

    SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

    popq %rdi
    popq %rsp

    SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR
    swaps
    syscall
    SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_L_GLOBAL)
    ANNOTATE_NOENDBR
    int3

    SYM_CODE_END(entry_SYSCALL_64)
syscall_return_via_sysret:
  IBRS_EXIT
  POP_REGS pop_rdl=0

/*
 * Now all regs are restored except RSP and RDI.
 * Save old stack pointer and switch to trampoline stack.
 */
  movq  %rsp, %rdi
  movq  PER_CPU_VAR(cpu_tss_rw # TSS_sp0), %rsp
  UNIND_HINT_END_OF_STACK

  pushq RSP-RDI(%rdi)  /* RSP */
  pushq (%rdi)         /* RDI */

/*
 * We are on the trampoline stack. All regs except RDI are live.
 * We can do future final exit work right here.
 */
  STACKLEAK_ERASE_NOCLOBBER

  SWITCH_TO_USER_CR3_STACK scratch_reg%rdi

  popq  %rdi
  popq  %rsp

SYM_INNER_LABEL(entry_SYSRETQ_unsafe_stack, SYM_LGLOBAL)
ANNOTATE_NOENDBR
swapgs

SYM_INNER_LABEL(entry_SYSRETQ_end, SYM_LGLOBAL)
ANNOTATE_NOENDBR
int3
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
- Mitigation Proposal
- EntryBleed against VM Exits
- Fully Working POC
- More Data

46
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**