# Speculative Probing: Hacking Blind in the Spectre Era

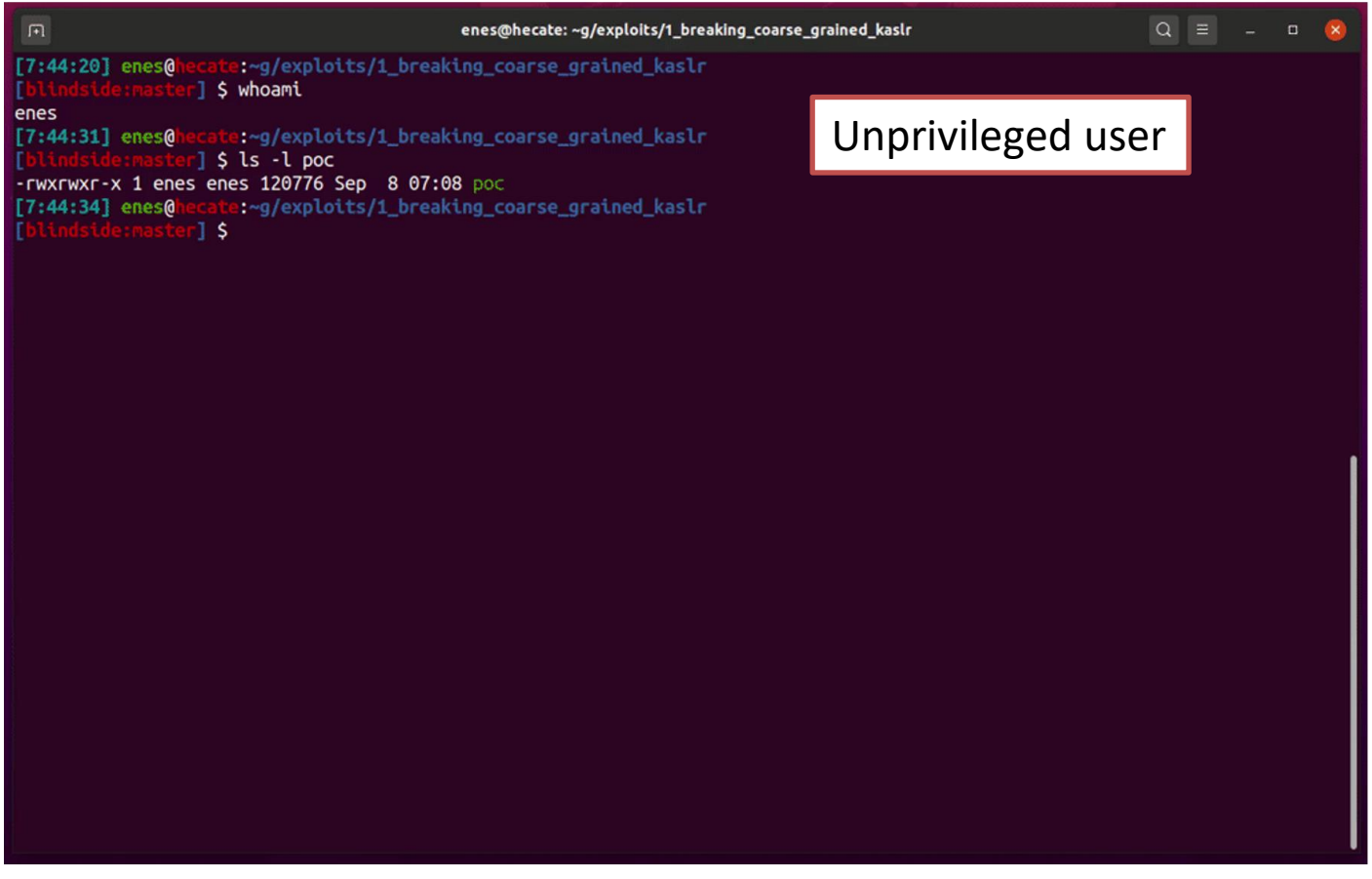
Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, Cristiano Giuffrida



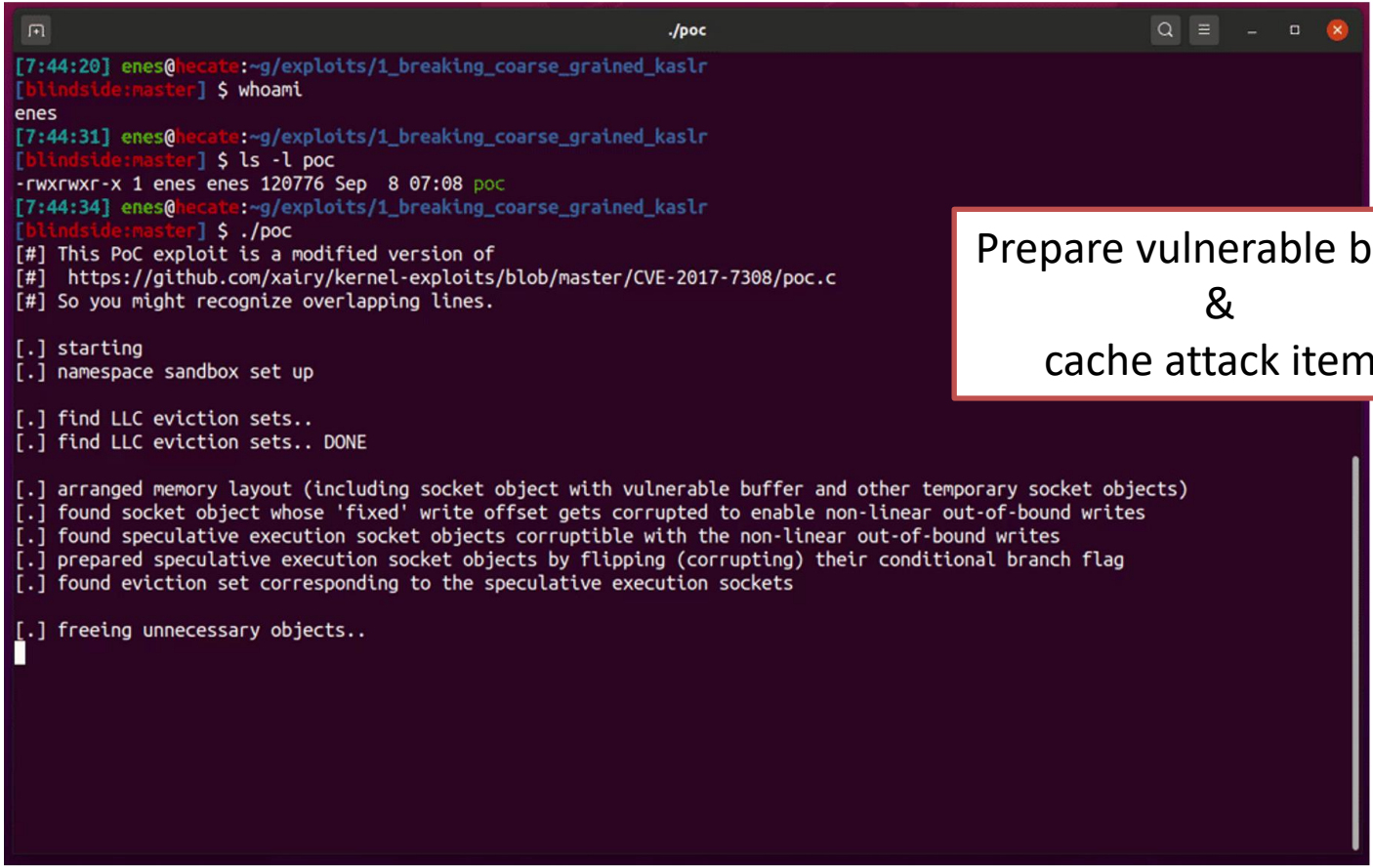
# BlindSide demo

- Target: elevate privileges through a ROP attack
  - Leak **kernel image location** for ROP gadgets
  - Leak **kernel heap location** that will contain the ROP payload
  - Leak **payload location** in the kernel heap for a reliable ROP attack
- We start with a single buffer overflow
- No direct information leakage primitive, but instead use *BlindSide* to leak
- Code-reuse mitigations in place (e.g. kernel address-space layout randomization)
- Spectre mitigations in place (e.g. eIBRS, retpoline, array index masking)

# BlindSide demo

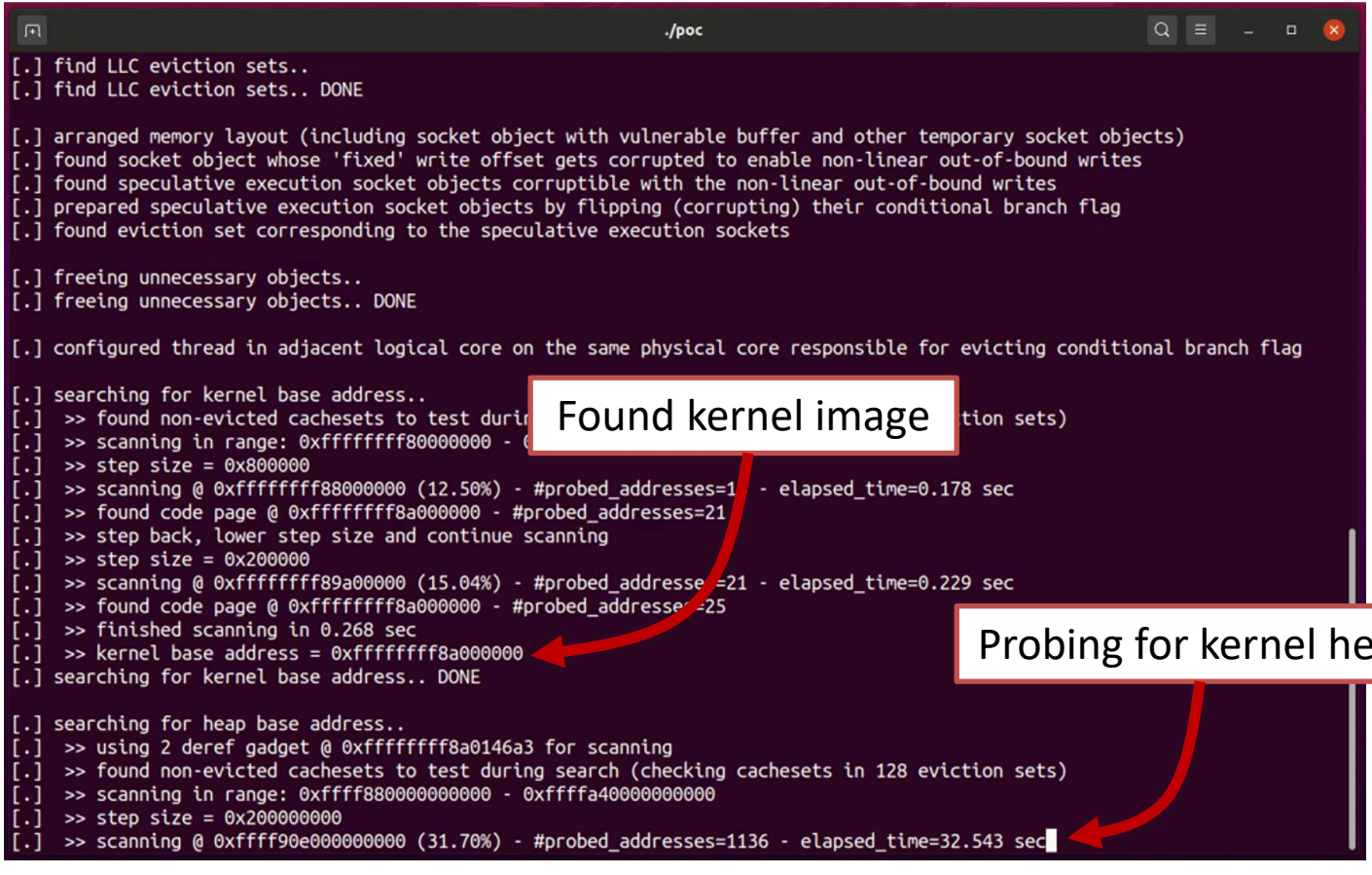


# BlindSide demo

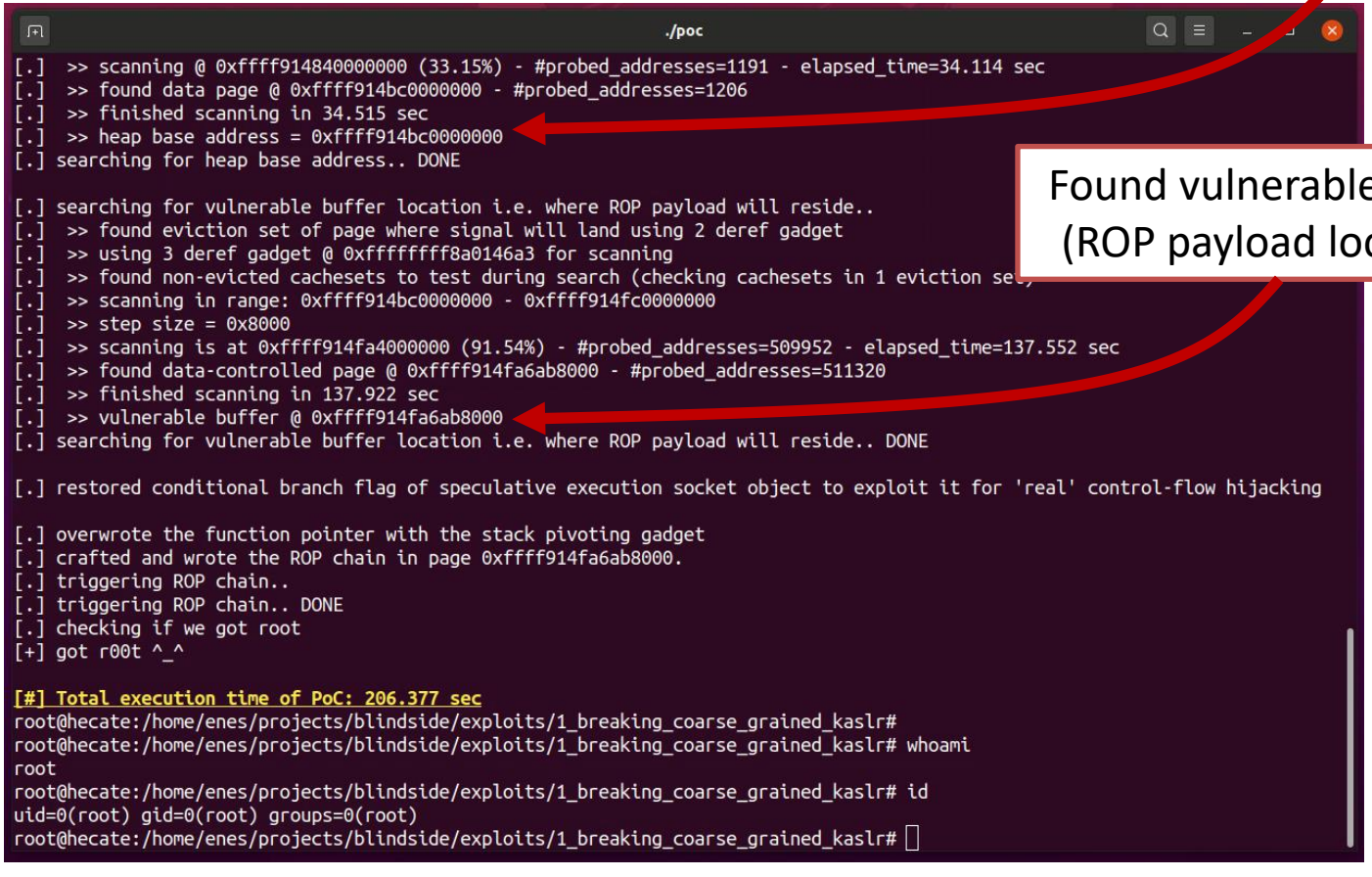



Prepare vulnerable buffer
&
cache attack items

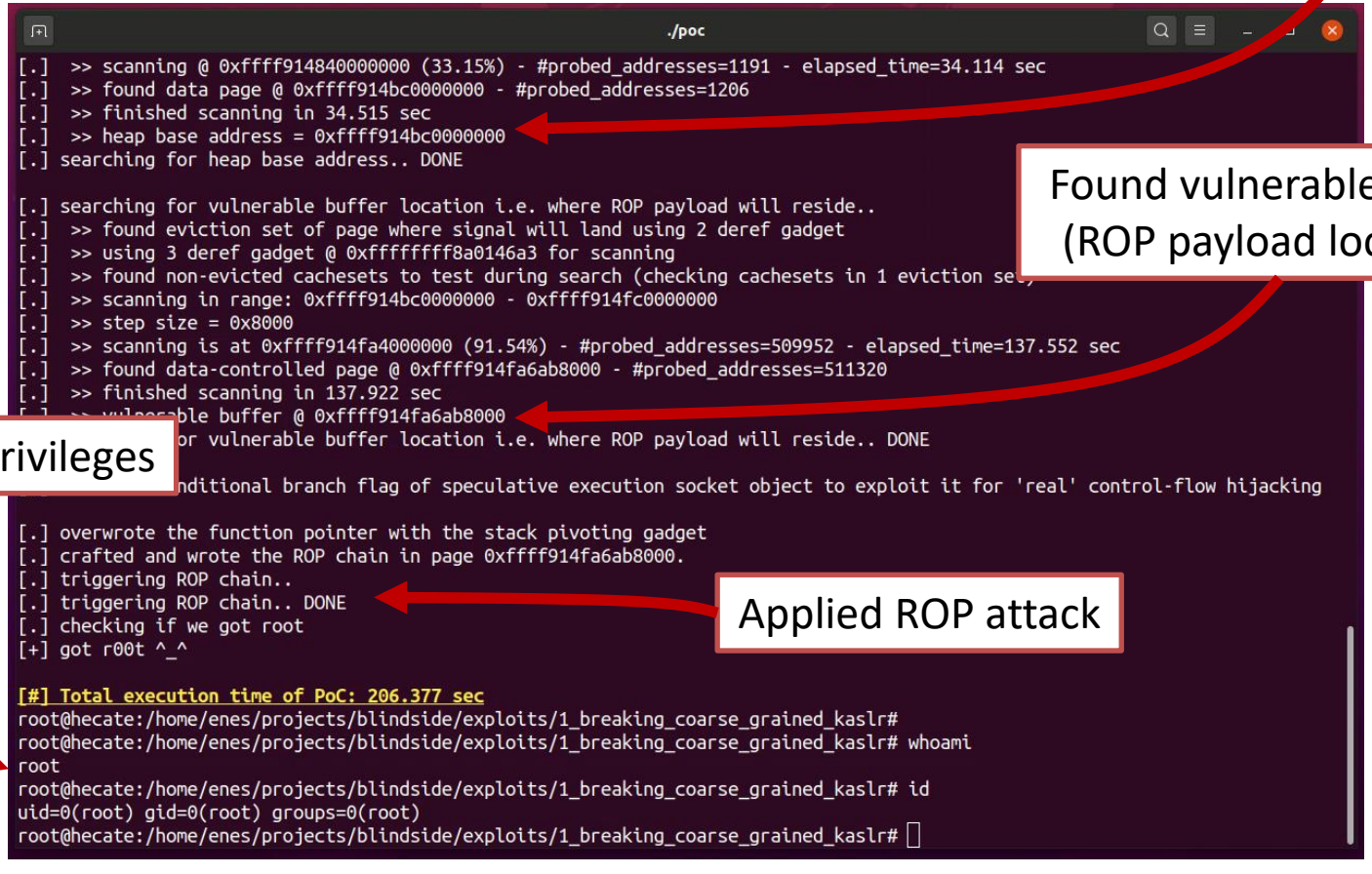
# BlindSide demo



# BlindSide demo



# BlindSide demo




Found kernel heap

Found vulnerable buffer (ROP payload location)

Elevated privileges

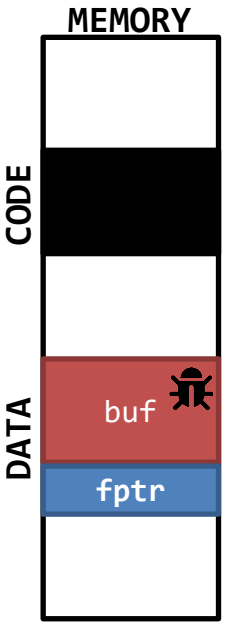
Applied ROP attack

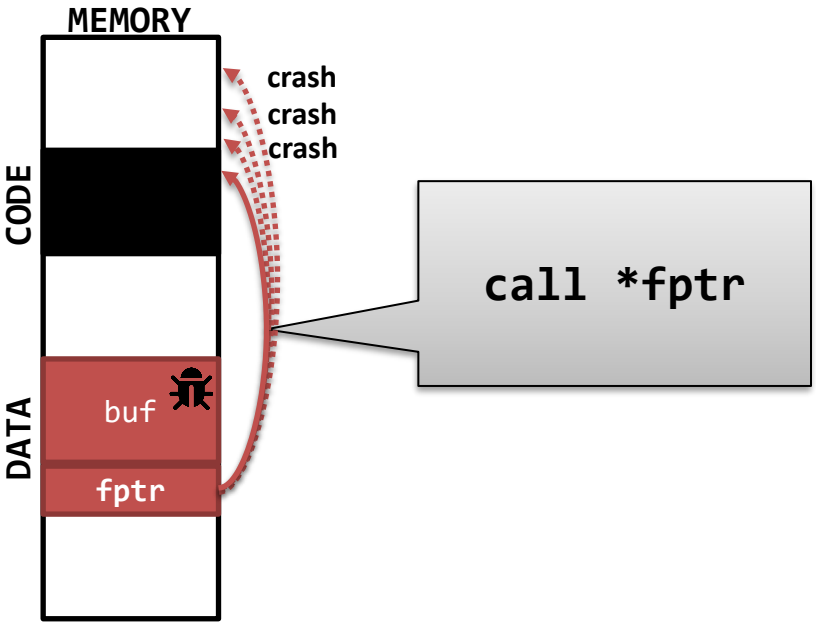
# What just happened?

- Combined buffer overflow with speculative execution, to be able to perform a **speculative** control-flow hijack

- On top of this hijacking primitive, we then **craft stronger primitives to blindly probe** for elements in memory under the speculative execution domain:
  - First, find kernel image location with *Code Page Probing* primitive
  - Then, find kernel heap location with *Data Page Probing* primitive
  - Finally, find ROP payload location with *Object Probing* primitive

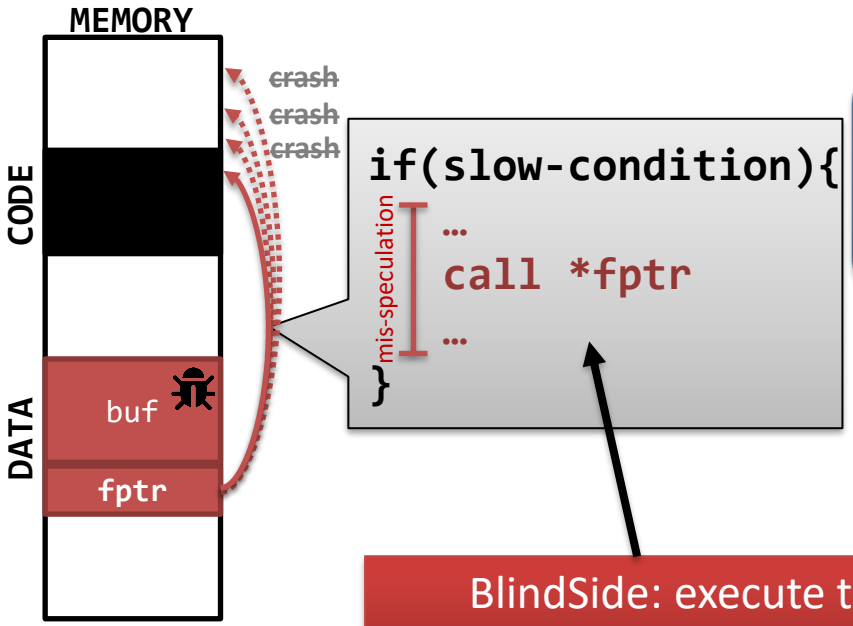
- **Finish with a ROP attack** for privilege escalation


MEMORY
CODE
DATA
buf
fptr




With code randomization and with the lack of an info leak, BROP-like attacks can probe **crash-resistant programs**

Probe results are inferred through a **fault** covert channel




With code randomization and with the lack of an info leak, BROP-like attacks can probe **crash-resistant programs**

Probe results are inferred through a **fault** covert channel

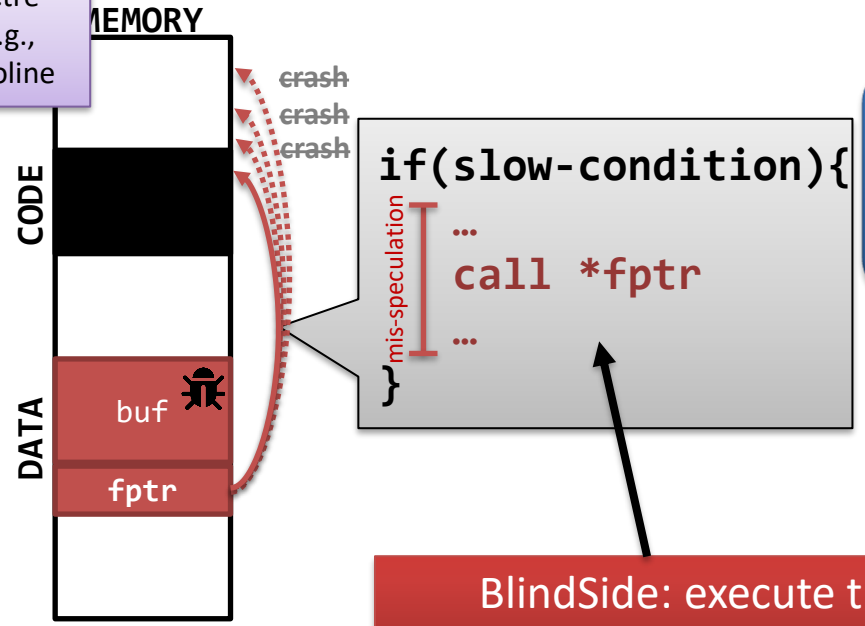
Impossible in a high value target like the **kernel** (crash-sensitive)

BlindSide: execute the indirect branch **speculatively**

Probe results are inferred through a **microarchitectural** covert channel

**BlindSide**

Bypasses Spectre mitigations, e.g., eIBRS and retpoline




With code randomization and with the lack of an info leak, BROP-like attacks can probe **crash-resistant programs**

Probe results are inferred through a **fault** covert channel

Impossible in a high value target like the **kernel** (crash-sensitive)

BlindSide: execute the indirect branch **speculatively**

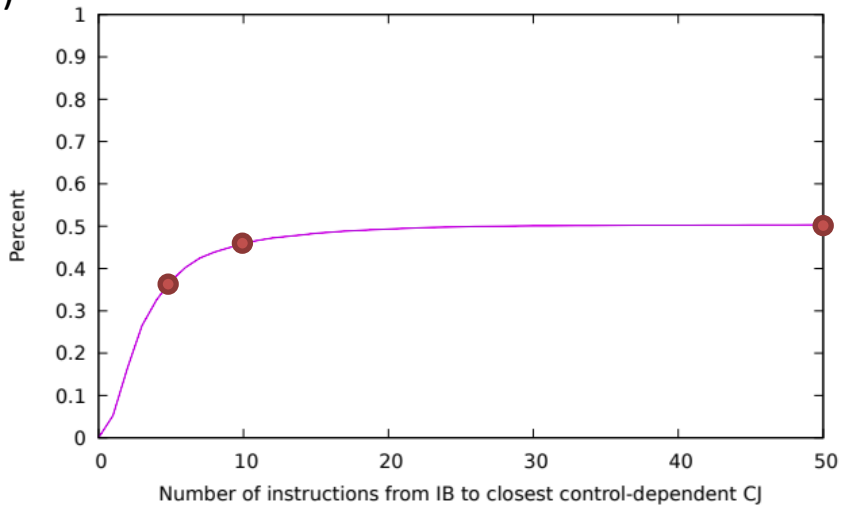
Probe results are inferred through a **microarchitectural** covert channel

# Speculative Probing Primitives


- Code page probing *(**no** gadget)*
- Gadget probing *(probe for a gadget, e.g. a Spectre gadget)*
- Data page probing *(gadget with **two** memory dereferences)*
- Object probing *(gadget with **three** memory dereferences)*
- Spectre probing *(probe with a Spectre gadget; **four** memory deref.)*

# Availability of Indirect Branches

- **~50%** (7,929) of all indirect branches (15,762) control-dependent on a conditional branch within a distance of **50** instructions
- **~45%** (7,239) of IB are within a distance of **10** instructions
- **~37%** (5,843) of IB are within a distance of **5** instructions




- Speculative Execution window can fit > 100 instructions

# Availability of Gadgets

| Source Register | # Dereferences | | |
|---|---|---|---|
| | 2 | 3 | 4 (Spectre) |
| RAX | 3086 | 540 | 1 |
| RBX | 4385 | 640 | 8 |
| RCX | 317 | 35 | 0 |
| RDX | 682 | 114 | 1 |
| RSI | 667 | 125 | 0 |
| RDI | 3842 | 844 | 15 |
| RBP | 3774 | 506 | 14 |
| RSP | 482 | 85 | 1 |

| Source Register | # Dereferences | | |
|---|---|---|---|
| | 2 | 3 | 4 (Spectre) |
| R8 | 96 | 14 | 0 |
| R9 | 75 | 11 | 0 |
| R10 | 85 | 8 | 0 |
| R11 | 36 | 5 | 0 |
| R12 | 2070 | 344 | 1 |
| R13 | 1278 | 182 | 1 |
| R14 | 1166 | 161 | 6 |
| R15 | 1114 | 149 | 0 |

# Availability of Gadgets

| Source Register | # Dereferences | | |
|---|---|---|---|
| | 2 | 3 | 4 (Spectre) |
| RAX | 3086 | 540 | 1 |
| RBX | 4385 | 640 | 8 |
| RCX | 317 | 35 | 0 |
| RDX | 682 | 114 | 1 |
| RSI | 667 | 125 | 0 |
| RDI | 3842 | 844 | 15 |
| RBP | 3774 | 506 | 14 |
| RSP | 482 | 85 | 1 |

| Source Register | # Dereferences | | |
|---|---|---|---|
| | 2 | 3 | 4 (Spectre) |
| R8 | 96 | 14 | 0 |
| R9 | 75 | 11 | 0 |
| R10 | 85 | 8 | 0 |
| R11 | 36 | 5 | 0 |
| R12 | 2070 | 344 | 1 |
| R13 | 1278 | 182 | 1 |
| R14 | 1166 | 161 | 6 |
| R15 | 1114 | 149 | 0 |

- Spectre gadgets for the majority of the registers
  - Generally a single fitting gadget is sufficient
  - Relaxing the gadget template will allow to find more
  - Chaining gadgets might also be an option

# Proof-of-Concept Exploits

- Used a heap buffer overflow (CVE-2017-7308) to showcase BlindSide in the Linux kernel (version 4.8.0)

- Three exploits using Speculative Probing primitives:
  1. Breaking kernel ASLR
  2. Leaking root password hash from heap/physmap (architectural data-only attack)
  3. Leaking kernel code (fine-grained randomization + software-XoM)

- To speed up probing: Prime+Probe → Flush+Reload
  - F+R through a user page shared in physmap

# Conclusion

- BlindSide generalizes the threat models of BROP and Spectre attacks by combining them

- With the combination being stronger than the sum of the parts
  - BlindSide enhances BROP to allow hacking blind in crash-sensitive domains
  - BlindSide enhances Spectre by making mitigations ineffective

- For exploit demos and source code, check out: https://www.vusec.net/projects/blindside/