An Analysis of Speculative Type Confusion Vulnerabilities in the Wild

Authors: Ofek Kirzner and Adam Morrison

William Mitchell

Outline

- Discussion Questions
- Problem and Motivation
- Challenges
- Proposal
- Evaluation
- Potential Impacts and Limitations
- Discussion

Discussion Questions

What is a speculative type confusion attack? How is it categorized in relation to Spectre V1?

Is speculative type confusion a reliable method for reading arbitrary memory?

How can we defend against speculative type confusion attacks?

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

foo(valid_x)

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

foo(valid_x)

Train prediction:
if taken

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

foo(&secret - array1)

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

foo(&secret - array1)

Misprediction:
out of bounds

```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space

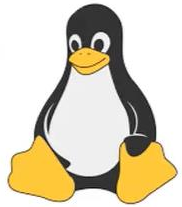
The secret is leaked


```
void foo(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space



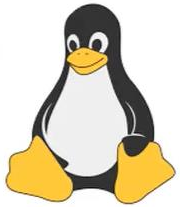
Attacker - unprivileged user

Victim - the Linux kernel

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space



Attacker - unprivileged user

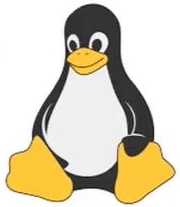
Victim - the Linux kernel

```
void function_called_from_syscall(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre Variant 1: Bounds Check Bypass

Goal: Leak data from the victim address space



Attacker - unprivileged user

Victim - the Linux kernel

foo(&secret - array1)

The secret is leaked

Read from kernel → Read any physical address

```
void function_called_from_syscall(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Mitigation in the Linux Kernel

A special API to ensure bounds checks are respected under speculation

```
void function_called_from_syscall(long x) {
  // ...
  if (x < array1_len){
    y = array1[x];
    z = array2[y * 4096];
  }
  // ...
}
```

```
void function_called_from_syscall(long x) {
  // ...
  if (x < array1_len){
    y = array_index_nospec(array1[x]);
    z = array2[y * 4096];
  }
  // ...
}
```

Problem and Motivation

Spectre V1 is more than just the bounds check bypass

From Spectre paper (Kocher et al., 2019):

> **Variant 1: Exploiting Conditional Branches.** In this variant of Spectre attacks, the attacker mistrains the CPU's branch predictor into mispredicting the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise.

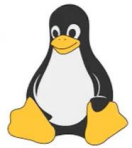
Problem and Motivation

Speculative type confusion:

- Mispeculation makes the victim execute with some variables holding values of the wrong type, and thereby enabling leakage of memory content

Problem and Motivation

Example



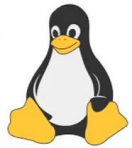
```
void syscall_helper(struct Base* obj) {
  if (obj->type == TYPE1){
    struct Type1* o = (struct Type1*) obj;
    leak(o->value);
  }
  if (obj->type == TYPE2){
    ...
  }
}
```

```
struct Common {
  enum Type type;
};

struct Type1 {
  struct Base base;
  ...
  uint32_t value;
};
```

Problem and Motivation

Example

```
void syscall_helper(struct Base* obj) {
  if (obj->type == TYPE1){
    struct Type1* o = (struct Type1*) obj;
    leak(o->value);
  }
  if (obj->type == TYPE2){
    ...
  }
}
```

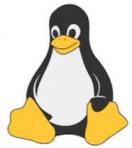


```
struct Common {
  enum Type type;
};

struct Type1 {
  struct Base base;
  ...
  uint32_t value;
};
```

Problem and Motivation

Example

```
void syscall_helper(struct Base* obj) {
  if (obj->type == TYPE1){
    struct Type1* o = (struct Type1*) obj;
    leak(o->value);
  }
  if (obj->type == TYPE2){
    ...
  }
}
```

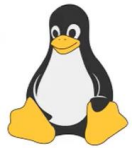


```
struct Common {
  enum Type type;
};

struct Type1 {
  struct Base base;
  ...
  uint32_t value;
};
```

Problem and Motivation

Example

```
void syscall_helper(struct Base* obj) {
  if (obj->type == TYPE1){
    struct Type1* o = (struct Type1*) obj;
    leak(o->value);
  }
  if (obj->type == TYPE2){
    ...
  }
}
```



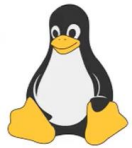
```
struct Common {
  enum Type type;
};

struct Type1 {
  struct Base base;
  ...
  uint32_t value;
};
```

Problem and Motivation

Example

Speculative type confusion

```
void syscall_helper(struct Base* obj) {
  if (obj->type == TYPE1){
    struct Type1* o = (struct Type1*) obj;
    leak(o->value);
  }
  if (obj->type == TYPE2){
    ...
  }
}
```



```
struct Common {
  enum Type type;
};

struct Type1 {
  struct Base base;
  ...
  uint32_t value;
};
```

Problem and Motivation

Observation: speculative type confusion may be much more prevalent than previously hypothesized.

Authors analyzed the Linux kernel, looking for speculative type confusion.

Authors found new types of speculative type confusion

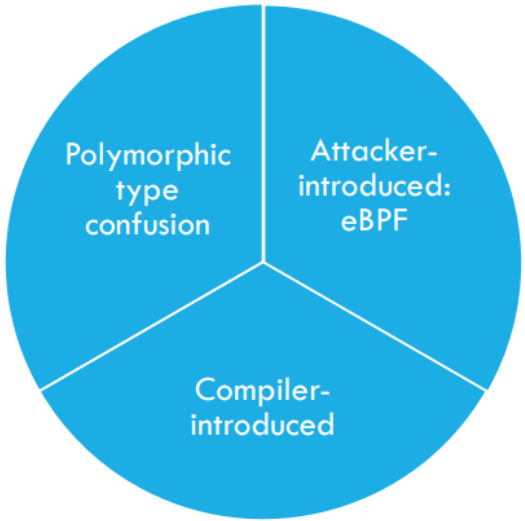
Challenges

Answering the question: Are OS kernels vulnerable to speculative type confusion attacks?

Speculative type confusion vulnerabilities are not well studied.

Proposal

Speculative type confusion in three contexts:



Attacker-Introduced: EBPF

EBPF is a Linux subsystem, enabling user-defined programs in Linux kernel space

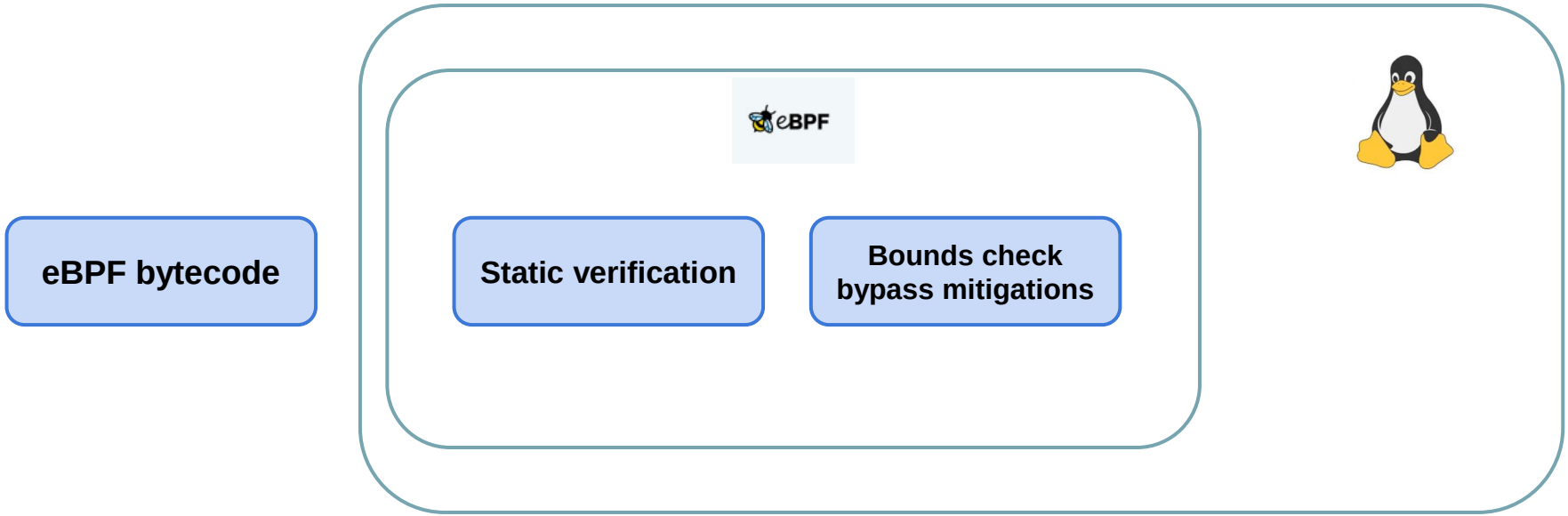
eBPF bytecode

Attacker-Introduced: EBPF

EBPF is a Linux subsystem, enabling user-defined programs in Linux kernel space

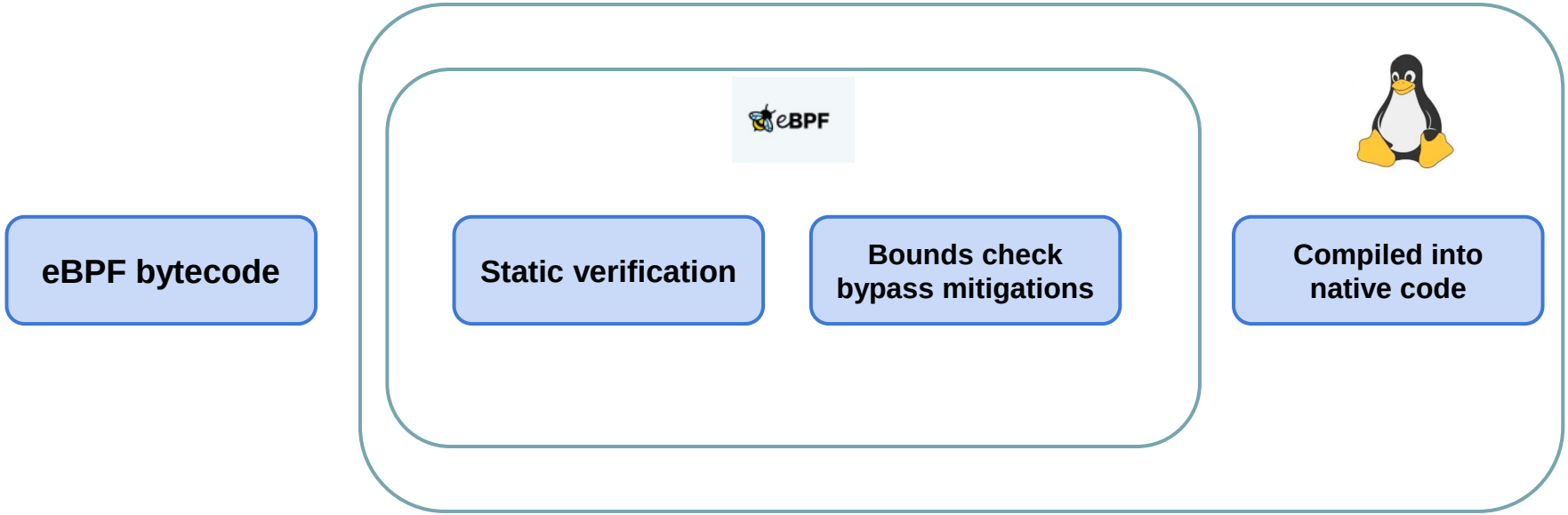
Attacker-Introduced: EBPF

EBPF is a Linux subsystem, enabling user-defined programs in Linux kernel space



Attacker-Introduced: EBPF

EBPF is a Linux subsystem, enabling user-defined programs in Linux kernel space



Attacker-Introduced: EBPF

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

```
// r0 = ptr to an array entry (verified != NULL)
```

Attacker-Introduced: EBPF

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
```

Attacker-Introduced: EBPF

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

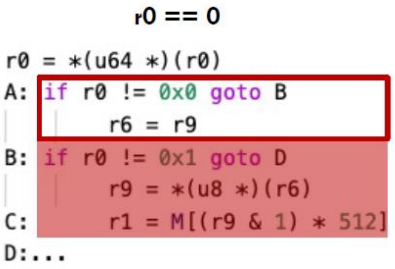
```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

Attacker-Introduced: EBPF

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

Flows considered by eBPF verifier



Attacker-Introduced: EBPF

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

Flows considered by eBPF verifier

r0 == 0

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

r0 == 1

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

otherwise

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Attacker-Introduced: EBPF

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Speculative flows are not verified!!

Attacker-Introduced: EBPF

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Predicted taken

Speculative flows are not verified!!

Attacker-Introduced: EBPF

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Predicted taken

Predicted taken

Speculative flows are not verified!!

Attacker-Introduced: EBPF

```
// r0 = ptr to an array entry (verified != NULL)
// r6 = ptr to stack slot (verified != NULL)
// r9 = scalar value controlled by attacker
```

```
r0 = *(u64 *)(r0)
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
C:     r1 = M[(r9 & 1) * 512]
D:...
```

Predicted taken

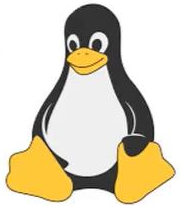
Predicted taken

```
if r0 == 0x0 and r0 == 0x1
    r6 = r9
    r9 = *(u8 *)(r6)
    r1 = M[(r9 & 1) * 512]
```

Speculative flows are not verified!!

Attacker-Introduced: EBPF

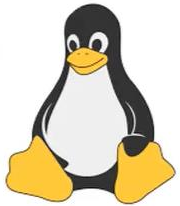
Training mutually exclusive branches



```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Attacker-Introduced: EBPF

Training mutually exclusive branches



Mutually exclusive

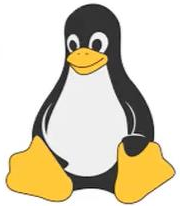
```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Attacker-Introduced: EBPF

Training mutually exclusive branches

Shadow gadget

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

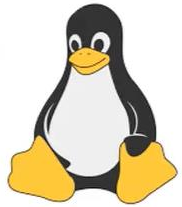


Mutually exclusive

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Attacker-Introduced: EBPF

Training mutually exclusive branches



Shadow gadget

Both branches can be taken

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Branch Prediction Unit

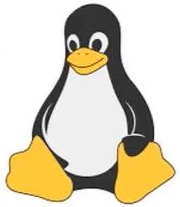
Mutually exclusive

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Attacker-Introduced: EBPF

Training mutually exclusive branches

Unprivileged process can read arbitrary memory addresses at a rate of ~6.5 KB/sec



Shadow gadget
Both branches can be taken

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Branch Prediction Unit

Mutually exclusive

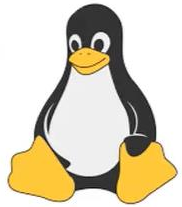
```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Attacker-Introduced: EBPF

Training mutually exclusive branches

Unprivileged process can read arbitrary memory addresses at a rate of ~6.5 KB/sec

CVE-2021-33624: fixed in mainline Linux development tree in June 2021



Shadow gadget
Both branches can be taken

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Branch Prediction Unit

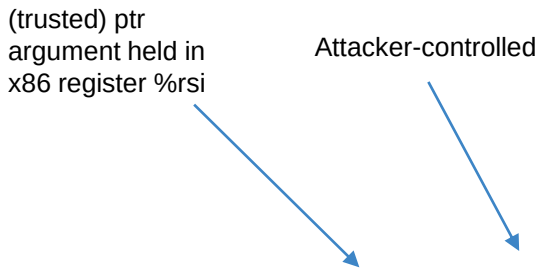
Mutually exclusive

```
A: if r0 != 0x0 goto B
       r6 = r9
B: if r0 != 0x1 goto D
       r9 = *(u8 *)(r6)
```

Compiler-Introduced

Compilers might create speculative type confusion

Innocent looking code is compiled in a way that introduces vulnerability



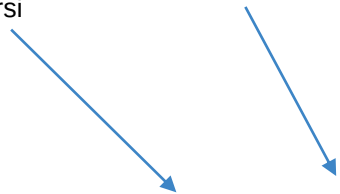

```
void syscall_helper(cmd_t* cmd, long* ptr, long x) {
  cmd_t c = *cmd;
if ( c == CMD_A)
{
  … // %rsi = x
}
if (c == CMD_B)
{
  y = *ptr; // y = *%rsi
  z = array[y * 4096];
}

  // ...
}
```

Compiler-Introduced

Compilers might create speculative type confusion

(trusted) ptr argument held in x86 register %rsi

Attacker-controlled



Innocent looking code is compiled in a way that introduces vulnerability

Compiler reasoning:
Branches are mutually exclusive

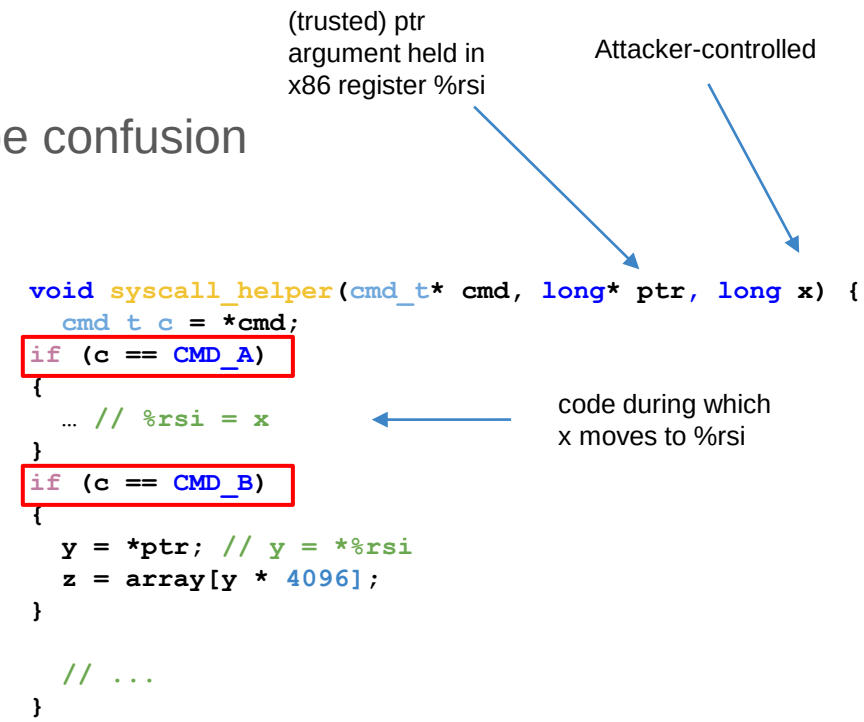
```
void syscall_helper(cmd_t* cmd, long* ptr, long x) {
  cmd_t c = *cmd;
if (c == CMD_A)
{
  … // %rsi = x
}
if (c == CMD_B)
{
  y = *ptr; // y = *%rsi
  z = array[y * 4096];
}

  // ...
}
```

Compiler-Introduced

Compilers might create speculative type confusion

Innocent looking code is compiled in a way that introduces vulnerability

Compiler reasoning:
Branches are mutually exclusive



Compiler-Introduced

Binary level analysis of Linux

Focused on system calls, which have well-defined user-controlled interface

The leakage mechanism is out of scope: aiming at finding speculative attacker-controlled memory dereference

Compiler-Introduced

Binary level analysis of Linux

Focused on system calls, which have well-defined user-controlled interface

The leakage mechanism is out of scope: aiming at finding speculative attacker-controlled memory dereference

compiler	flags	# vulnerable syscalls
GCC 9.3.0	-Os	20
GCC 9.3.0	-O3	2
GCC 5.8.2	-Os	0
GCC 5.8.2	-O3	0

Compiler-Introduced

Binary level analysis of Linux

Focused on system calls, which have well-defined user-controlled interface

The leakage mechanism is out of scope: aiming at finding speculative attacker-controlled memory dereference

processor	leak probability	
	`T=char`	`T=long`
AMD EPYC 7R32	$1/10^5$	$1/5000$
AMD Opteron 6376	$1/10^5$	$1/5000$
Intel Xeon Gold 6132 (Skylake)	$1/(5.09 \times 10^7)$	$1/(1.36 \times 10^6)$
Intel Xeon E5-4699 v4 (Broadwell)	$1/(3.64 \times 10^9)$	$1/(6.2 \times 10^9)$
Intel Xeon E7-4870 (Westmere)	$1/(1.67 \times 10^9)$	$1/(2.75 \times 10^7)$

Table 5: x86 branch squash behavior (in 30 B trials).

Polymorphic Type Confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
  /* … /*
  common->foo(common);
}
```

Polymorphic Type Confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
  /* … /*
  common->foo(common);
}
```

Polymorphic Type Confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
  /* … /*
  common->foo(common);
}
```

Polymorphic Type Confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
  /* … /*
  common->foo(common);
}
```

```
void foo_A(struct Common* common) {
  /* … /*
  char x = *((struct A*) common)->ptr;
  leak(x);
}
```

Polymorphic Type Confusion

```
struct Common { void (*foo) (void*); };
struct A { struct Common common; char* ptr; };
struct B { struct Common common; long user_controlled_scalar; };
```

```
void some_code_path(struct Common* common) {
  /* … /*
  common->foo(common);
}

void foo_A(struct Common* common) {
  /* … /*
  char x = *((struct A*) common)->ptr;
  leak(x);
}
```

B→user_controlled_scalar

Potential Impacts and Limitation

Thousands of patterns were flagged as being potentially vulnerable

Hundreds of “array indexing” instances

For all -- limited speculation window or limited control on user value

Potential Impacts and Limitation

Thousands of patterns were flagged as being potentially vulnerable

Hundreds of “array indexing” instances

For all -- limited speculation window or limited control on user value

Linux kernel security would be on shaky ground if conditional branch-based mitigation were used instead of retpolines*

*retpoline: “return and trampoline” is a software construct which allows indirect branches to be isolated from speculative execution

Discussion

What is a speculative type confusion attack? How is it categorized in relation to Spectre V1?

Is speculative type confusion a reliable method for reading arbitrary memory?

How can we defend against speculative type confusion attacks?

Discussion - Select Submitted Questions

It wasn't entirely clear how to mitigate these attacks without large performance losses. Has there been any proposed mitigation strategy that works without significantly compromising performance?

Discussion - Select Submitted Questions

When you get the gadgets from the compiler analysis, is there any generalized way to assess their exploitability or is the assumption that one would have to reason through each gadget by hand?

Discussion - Selected Questions

- Are there any hardware-leaning solutions to speculative type confusion vulnerabilities?
- Are tools that detect vulnerabilities through static or dynamic analysis evolving over time and implementing some of the insights shared by this paper?
- When you get the gadgets from the compiler analysis, is there any generalized way to assess their exploitability or is the assumption that one would have to reason through each gadget by hand?
- Is there a way to change the way a processor squashes incorrectly predicted branches in order to eliminate leakages? What sort of overhead might this entail?
- The authors found that speculative load hardening (SLH) is generally an effective mitigation against speculative type confusion vulnerabilities, but at a significant slowdown cost of up to almost 3.5x (depending on CPU). How significant is this slowdown in reality? If a CPU is fast enough, could this slowdown be imperceptible to a human?
- What would it take to make the compiler-introduced exploitation more reliable?
- Can you explain for we force branch misprediction? I found the code examples kind of confusing
- I don't understand what taint analysis is. Can you clarify?
- What other types of Speculative Type Confusion Vulnerabilities are there?
- How do other OS-s vulnerabilities compare to Linux?
- It wasn't entirely clear how to mitigate these attacks without large performance losses. Has there been any proposed mitigation strategy that works without significantly compromising performance?
- How much work has been done after this paper in defending all the novel compiler induced vulnerabilities discovered in this paper? This part seemed to have the least amount of related works available.
- How does the out of bounds misprediction attack discussed in lecture compare to the speculative type confusion vulnerabilities discussed in this paper? Which attacks would be easier to defend against or introduce mitigations for? How do the performance costs of these mitigations compare?
- Are eBPF vulnerabilites like the one shown in the paper still existing in the Linux kernel?
- Is there a formal framework or automation tools established for kernel developers so that code written don't introduce these vulnerabilites?