Reptar

by Tavis Ormandy

We have a CPU mystery! We found a way to cause some processors to enter a glitch state where the normal rules don't apply, but what does that mean…?

If you're interested what can go wrong inside modern CPUs, read on!

Introduction

If you've ever written any x86 assembly at all, you've probably used rep movsb. It's the idiomatic way of moving memory around on x86. You set the *source*, *destination*, *direction* and the *count* - then just let the processor handle all the details!

```
lea rdi, [rel dst]
lea rsi, [rel src]
std
mov rcx, 32
rep movsb
```

The actual instruction here is movsb, the rep is simply a prefix that changes how the instruction works. In this case, it indicates that you want this operation **rep**eated multiple times.

There are lots of other prefixes too, but they don't all apply to every instruction.

Prefix Decoding

An interesting feature of x86 is that the instruction decoding is generally quite relaxed. If you use a prefix that doesn't make sense or conflicts with other prefixes nothing much will happen, it will usually just be ignored.

This fact is sometimes useful; compilers can use redundant prefixes to pad a single instruction to a desirable alignment boundary.

Take a look at this snippet, this is exactly the same code as above, just a bunch of useless or redundant prefixes have been added:

```
            rep lea rdi, [rel dst]
             cs lea rsi, [rel src]
      gs gs gs std
         repnz mov rcx, 32
rep rep rep rep movsb
```

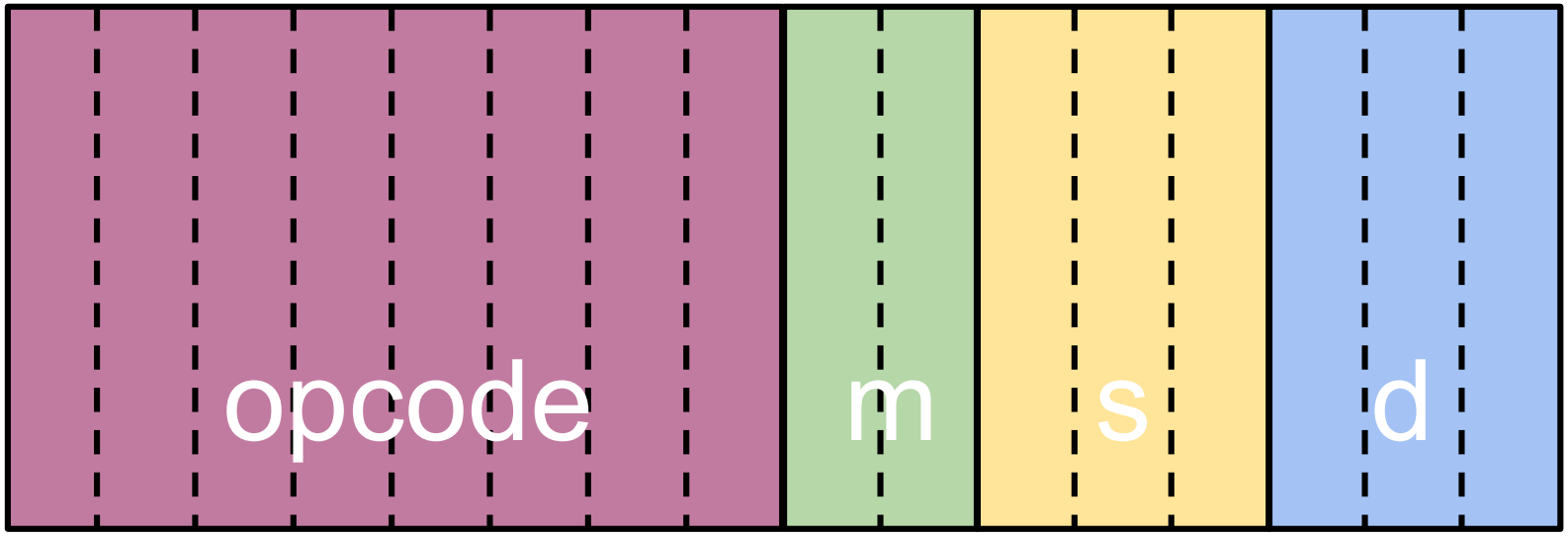
Perhaps the most interesting prefixes are rex, vex and evex, all of which change how subsequent instructions are decoded.

Let’s take a look at how they work.

The REX prefix

The i386 only had 8 general purpose registers, so you could specify which register you want to use in just 3 bits (because 2^3 is 8).

The way that instructions were encoded took advantage of this fact, and reserved *just* enough bits to specify any of those registers.



Simple 2-byte instructions that use modr/m might be encoded like this, for example mov eax, ebx.

This is an 8-bit opcode, 2 bit addressing mode (labeled m), and 3 bits each for the source (s) and destination (d).

Well, this is a problem, because x86-64 added 8 additional general purpose registers. We now have sixteen possible registers..that's 2^4, so we're going to need another bit! 😆

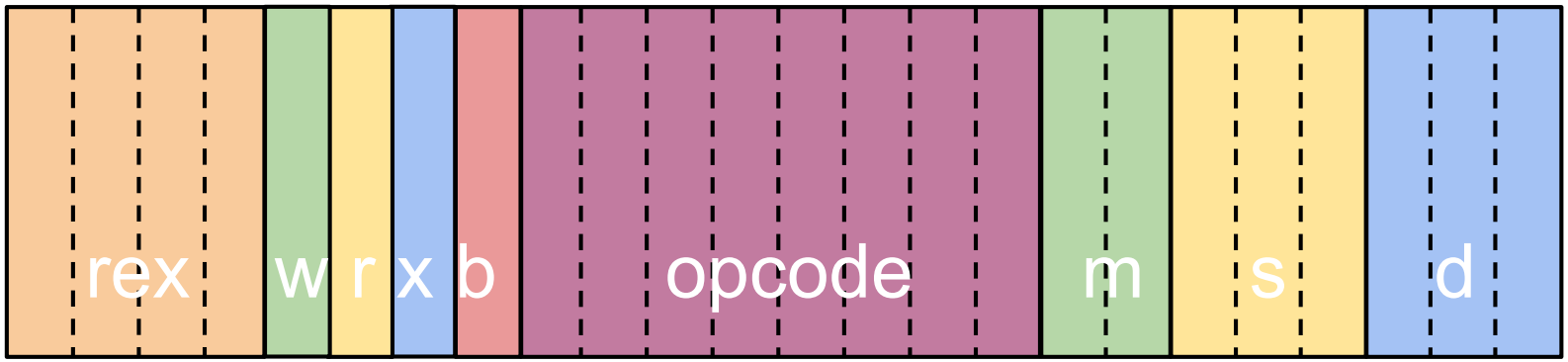
The solution to this is the rex prefix, which gives us some spare bits that the next instruction can borrow.

When we're talking about rex, we usually write it like this:

rex.rxb

rex is a single-byte prefix, the first four bits are mandatory and the remaining four bits called b, x, r and w are all optional. If you see rex.rb that means only the r and b bits are set, all the others are unset.

These optional bits give us room to encode more general purpose registers in the following instruction.



The rex prefix can lend the next instruction extra bits to use for operands, so now we can encode all 16 possible general purpose registers!

Now we're fine until someone adds another register! 😂

Encoding Rules

So now we know that rex increases the available space for encoding operands, and that useless or redundant prefixes are usually ignored on x86. So… what should this instruction do?

```
rex.rxb rep movsb
```

The movsb instruction doesn't have any operands - they're all implicit - so any rex bits are meaningless, right?

If you guessed that the processor will just silently ignore the rex prefix, you would be correct!

Well… except on machines that support a new feature called *fast short repeat move*! We discovered that a bug with redundant rex prefixes could interact with this feature in an unexpected way and introduce a serious vulnerability, oops 🙄

Fast Short Repeat Move

FSRM is a new feature introduced in Ice Lake that fixes some of the shortcomings of ERMS. Hopefully that clears up any confusion. 😆

Just kidding, let's quickly look at ERMS.

The hard part of moving strings around efficiently is getting all the buffers aligned so you can use the widest possible stores available. You *could* do this in software, but if we do it in microcode then the processor can just transparently make your existing code faster for you.

This requires some expensive setup, but once that's done you get vastly improved throughput. This feature is known as *enhanced repeat move/store*, ERMS.

If you have a processor with ERMS support, simple rep movsb operations can sometimes perform comparably with more complicated hand-tuned vector move operations.

However, there is a problem with ERMS. That initial setup is so expensive that it just isn't worth it for very short strings. This is what FSRM is designed to solve, it handles the case of only moving 128 bytes or less and makes that faster too!

I'm not aware of any documentation that explains exactly how FSRM works, but you can check if you have a processor that supports it by looking at the flags line in /proc/cpuinfo:

```
flags       : fpu vme de pse tsc msr pae mce cx8 [...] fsrm
```

Some of the processors that have this feature include:

- Ice Lake
- Rocket Lake
- Tiger Lake
- Raptor Lake
- Alder Lake
- Sapphire Rapids

> Note: This list may not be comprehensive, please see Intel advisory INTEL-SA-00950 for a complete list.

Discovery

I've written previously about a processor validation technique called *Oracle Serialization* that we've been using. The idea is to generate two forms of the same randomly generated program and verify their final state is identical.

> You can read more about Oracle Serialization in my previous writeup.

In August, our validation pipeline produced an interesting assertion. It had found a case where adding redundant rex.r prefixes to an FSRM optimized rep movs operation seemed to cause unpredictable results.

We observed some very strange behavior while testing. For example, branches to unexpected locations, unconditional branches being ignored and the processor no longer accurately recording the instruction pointer in xsave or call instructions.

Oddly, when trying to understand what was happening we would see a debugger reporting impossible states!

This already seemed like it could be indicative of a serious problem, but within a few days of experimenting we found that when multiple cores were triggering the same bug, the processor would begin to report machine check exceptions and halt.

We verified this worked even inside an unprivileged guest VM, so this already has serious security implications for cloud providers. Naturally, we reported this to Intel as soon as we confirmed this was a security issue.

Reproduce

We're publishing all of our research today to our security research repository. If you want to reproduce the vulnerability you can use our icebreak tool, I've also made a local mirror available here.

```
$ ./icebreak -h
usage: ./icebreak [OPTIONS]
    -c N,M      Run repro threads on core N and M.
    -d N        Sleep N usecs between repro attempts.
    -H N        Spawn a hammer thread on core N.
icebreak: you must at least specify a core pair with -c! (see -h for help)
```

The testcase enters what should be an infinite loop, and unaffected systems should see no output at all. On affected systems, a . is printed on each successful reproduction.

```
$ ./icebreak -c 0,4
starting repro on cores 0 and 4
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
```

In general, if the cores are SMT siblings then you may observe random branches and if they're SMP siblings from the same package then you may observe machine checks.

If you do *not* specify two different cores, then you might need to use a hammer thread to trigger a reproduction.

Analysis

We know something strange is happening, but how microcode works in modern systems is a closely guarded secret. We can only theorize about the root cause based on observations.

µops

The CPU is split in two major components, the *frontend* and the *backend*. The frontend is responsible for fetching instructions, decoding them and generating µops to send to the backend for execution.

The backend executes instructions *out of order*, and uses a unit called the ROB, *reorder buffer*, to store and organize results.

We believe this bug causes the frontend to miscalculate the size of the movsb instruction, causing subsequent entries in the ROB to be associated with incorrect addresses. When this happens, the CPU enters a confused state that causes the instruction pointer to be miscalculated.

The machine can eventually recover from this state, perhaps with incorrect intermediate results, but becoming internally consistent again. However, if we cause multiple SMT or SMP cores to enter the state simultaneously, we can cause enough microarchitectural state corruption to force a machine check.

Questions

I'm sure some readers will have questions about what is possible in this unexpected "glitch" state. Well, so do we!

We know that we can corrupt the system state badly enough to cause machine check errors, and we've also observed threads interfere with execution of processes scheduled on SMT siblings.

However, we simply don't know if we can control the corruption precisely enough to achieve privilege escalation. I suspect that it *is* possible, but we don't have any way to debug µop execution!

If you're interested in studying this, then we would love to get your input!

Credit

This bug was independently discovered by multiple research teams within Google, including the [silifuzz] team and Google [Information Security Engineering]. The bug was analyzed by Tavis Ormandy, Josh Eads, Eduardo Vela Nava, Alexandra Sandulescu and Daniel Moghimi.

Solution

Intel have published updated microcode for all affected processors. Your operating system or BIOS vendor may already have an update available!

Workaround

If you can't update for some reason, you *could* disable fast strings via the IA32_MISC_ENABLE model specific register.

This will cause a significant performance penalty, and should not be used unless absolutely necessary.

Notes

If you're interested in more CPU bugs, we publish everything we find!

Not all the bugs we discover have security consequences, but they're usually worth reading! For example, did you know that sometimes movlps just doesn't work? or that registers can sometimes roll back to previous values?