
Spectre, Meltdown, and Linux

This talk is all about the newly announced category of bugs called Spectre.

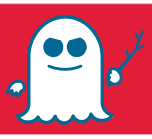
# Disclaimer

- This talk vastly over-simplifies things.
- See notes for full details and resources.

https://github.com/gregkh/presentation-spectre

In order to keep this talk within the time limit, I am vastly over simplifying things.

Please see the presentation notes at the link here for more details and a full list of resources on how to find out more information about this topic.



# Spectre

- Hardware bugs
- Valid code can be “tricked” into exposing sensitive data to attacking programs.
- Exploits the speculative execution model of modern CPUs.
- Many different variants.
- Is going to be with us for a very long time!

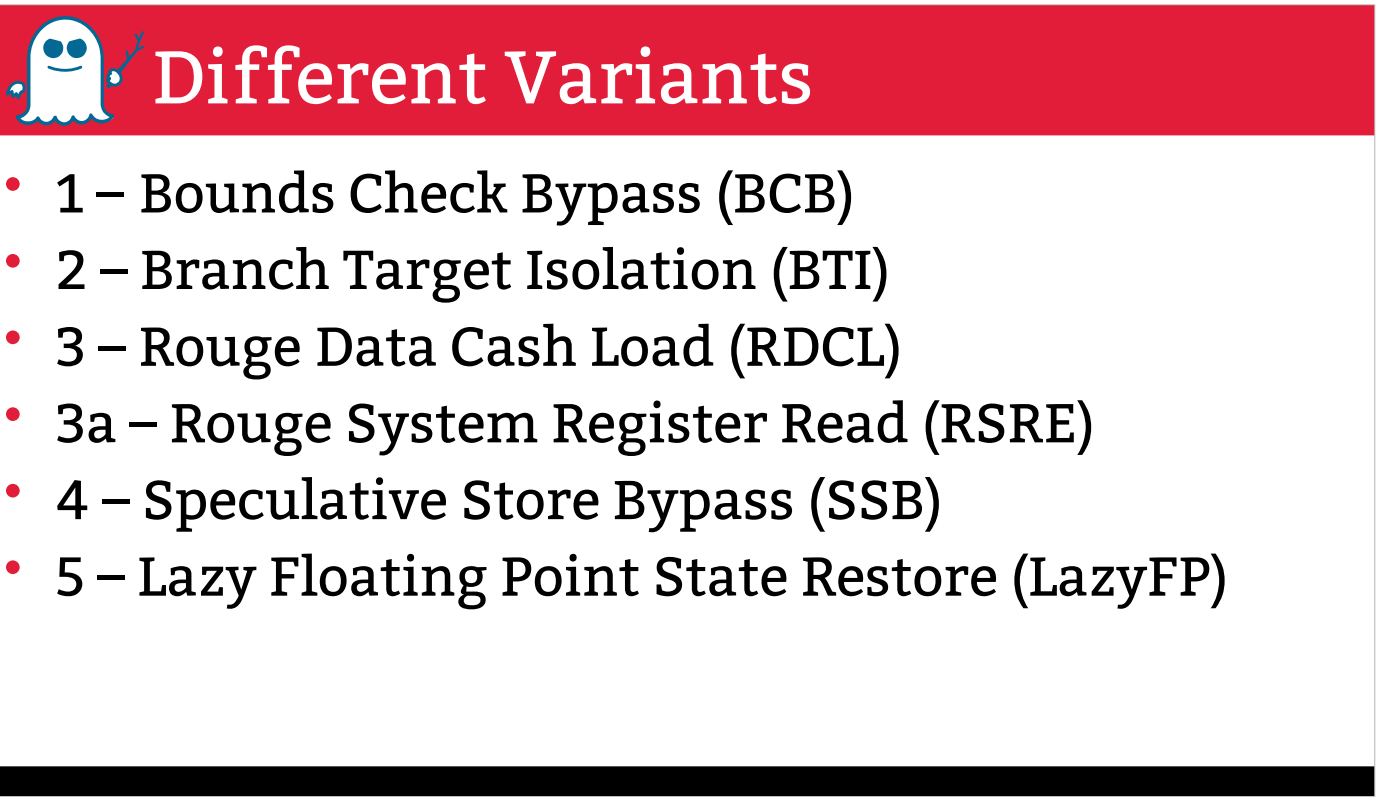
Spectre is the category of new CPU bugs that were originally discovered last year in July first by Jann Horn of Google and then independently discovered by others in the few months afterward.

These bugs are problems in the hardware itself that takes advantage of how CPUs speculatively execute code. I will describe what this means later on.

All of these bugs are ones that enable attackers to read memory of other programs or virtual machines. No memory can be modified.

There are many different variants of these bugs that have been found so far.

Due to the way that CPUs work, these problems are going to be with us for a very long time as we clean up after the mess.




Here are the different variants that have been announced so far.

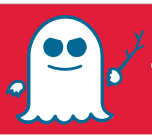
All of the variants end up doing the same thing, allowing memory to be read that should not be read, but they do it in different ways, which results in the different names.

They require different methods of protection, because they abuse different parts of the CPU in order to achieve the attack.

1, 2, and 3 were in the initial announcement, January 4, 2018

3a and 4 were made public May 21, 2018

5 “leaked” June 13, full details to be published June 27



# variant 1 – Bounds check bypass

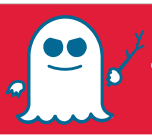
- Uses the kernel to read memory of another process or virtual machine.
- Fixed by core kernel changes.
- Lots of individual drivers still need to be fixed.

Like I said earlier, all of the variants allow memory to be read from places that should not be readable. Examples of this is reading memory of the kernel itself, or different programs running in the same machine, or even reading memory of a different virtual machine.

Because of this, these bugs can be exploited by browers running scripts that can read data from other programs or tabs, or in a virtual machine to read data from different customer's virtual machines on the same server.

The first variant uses correct code in the kernel to read the memory of another program.

It was fixed by changing some core parts of the kernel, but also requires all drivers to be audited and changed as well. That work has only just started.



# variant 1 – vulnerable code

```
int load_array(int *array, unsigned int user_value)
{
      if (user_value >= MAX_SIZE)
            return 0;

      return array[user_value];
}
```

This is an example of kernel code that is completely correct as written.

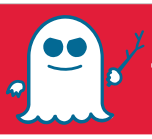
Spectre uses this correct code to take advantage of the CPU's behavior in trying to guess what the code is going to do in the future.

If you call this code a lot, the CPU learns that the check will succeed so it gets smart and preloads the value that array+user_value to be returned.

If you suddenly pass in a very large number for user_value, the CPU will preload the array+user_value memory location while it is still doing the check on the first line. When that check fails, the CPU will "stall" and unwind itself and return the error value and process continues on correctly.

But, that invalid memory location was read by the CPU and placed in the cache already. This can then be detected by a program running elsewhere in order to determine exactly what that memory value was.

Doing this is not fast, you can end up reading 2000 bytes a second, but that's all you need, you now have access to memory values you should not have access to.



# variant 1 – fixed code

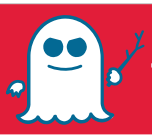
```
int load_array(int *array, unsigned int user_value)
{
      if (user_value >= MAX_SIZE)
            return 0;

      user_value = array_index_nospec(user_value, MAX_SIZE);

      return array[user_value];
}
```

To stop the CPU from speculating and reading outside of the memory range, we have to put in a “no speculation” mask in the code to force the CPU to not do this.

On Linux this call is array_index_nospec()



# variant 1 – fixed code

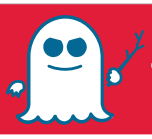
```
#define array_index_nospec(index, size)                                 \
({                                                                      \
        typeof(index) _i = (index);                                     \
        typeof(size) _s = (size);                                       \
        unsigned long _mask = array_index_mask_nospec(_i, _s);\
                                                                        \
        BUILD_BUG_ON(sizeof(_i) > sizeof(long));                        \
        BUILD_BUG_ON(sizeof(_s) > sizeof(long));                        \
                                                                        \
        (typeof(_i)) (_i & _mask);                                      \
})
```

For fun, here is what this array_index_nospec() macro does.

It verifies at build time that you are actually comparing things that should be properly compared.

And it generates a “mask” to force the values of the index to fit in the correct range that is required.

The call that does this mask calculation, is array_index_mask_nospec() and is custom to each CPU type.



# variant 1 – fixed code - x86

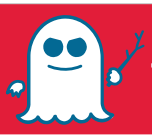
```
static inline unsigned long
      array_index_mask_nospec(unsigned long index,
                              unsigned long size)
{
      unsigned long mask;

      asm ("cmp %1,%2; sbb %0,%0;"
                     :"=r" (mask)
                     :"g"(size),"r" (index)
                     :"cc");
      return mask;
}
```

Here is the implementation of array_index_mask_nospec() for the x86 proessor.

It is just a few assembly language instructions that does the compare of the values and stops the processor from doing any speculation beyond this point.

Different processor types have different implementations of this function.



# variant 1 – fixed code

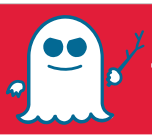
```
int load_array(int *array, unsigned int user_value)
{
	if (user_value >= MAX_SIZE)
		return 0;

	user_value = array_index_nospec(user_value, MAX_SIZE);

	return array[user_value];
}
```

Now back to our fixed example.

Because we properly tell the processor to not speculate outside the range provided, the CPU will not try to preload any values outside of the range here.



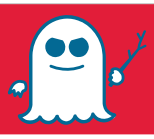
# variant 1 – Fix dates*

- x86
  - 4.14.14 17 January 2018
  - 4.9.77 17 January 2018
  - 4.4.113 23 January 2018
- ARM
  - 4.15.4 17 February 2018
  - 4.14.21 22 February 2018

Here are a listing of the different kernel versions and release dates that the variant 1 problem was fixed in.

Note that for ARM processors, no older kernels were originally fixed.

And I say “first fixed” because...

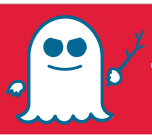


# *Fixes keep coming

- These are the “first fixed” dates.
- Later kernels get more fixes and improvements.
- Keep updating your kernel & microcode!

Fixes for these problems keep coming. We might have a “first fixed by” date, but there are almost always future fixes to make things run faster, or cover more problem areas, or just fix up more drivers where this code signature is found.

The moral of the story is to always keep updating your kernel and your processor microcode whenever new updates come out.



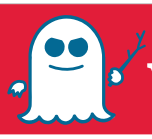
# variant 1 – Fix dates again

- x86
  - 4.16.11 22 May 2018
  - 4.14.43 22 May 2018
  - 4.9.102 22 May 2018
- ARM
  - 4.16 1 April 2018
  - 4.9.95 20 April 2018

As proof, a few months later, we got more fixes for variant 1 merged, with better coverage and better performance.

Also, for ARM chips, the fix got backported to the 4.9 stable kernel tree.

But not for 4.14, which is odd, I don’t know why that did not happen.



# variant 2 – Branch target injection

- Abuses the CPU branch predictor.
- Read data from kernel or other virtual machine.
- Fixed by compiler, kernel, & microcode updates.
- “retpoline”

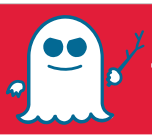
Variant 2 is much like variant 1, but instead of abusing the data lookup portion of the CPU, it abuses the ability for a CPU to predict which way it will go when a function pointer is called.

This problem can enable an attacker to read memory from the kernel, or another virtual machine.

To fix this issue it required that changes be made in the compiler, the kernel, and in the processor itself with microcode updates.

A new type of function protection was created by Matt Linn of Google called “retpoline”. This is a way for the compiler to protect function calls in a “fast” way from being abused. It is much quicker than the microcode updates that turn off branch prediction, but it is done in the compiler, so all code must be rebuilt with these changes.

For the kernel, that is not a problem to take advantage of, but for legacy code, be aware of this and turn on the microcode changes instead if you can not rebuild your problem applications.



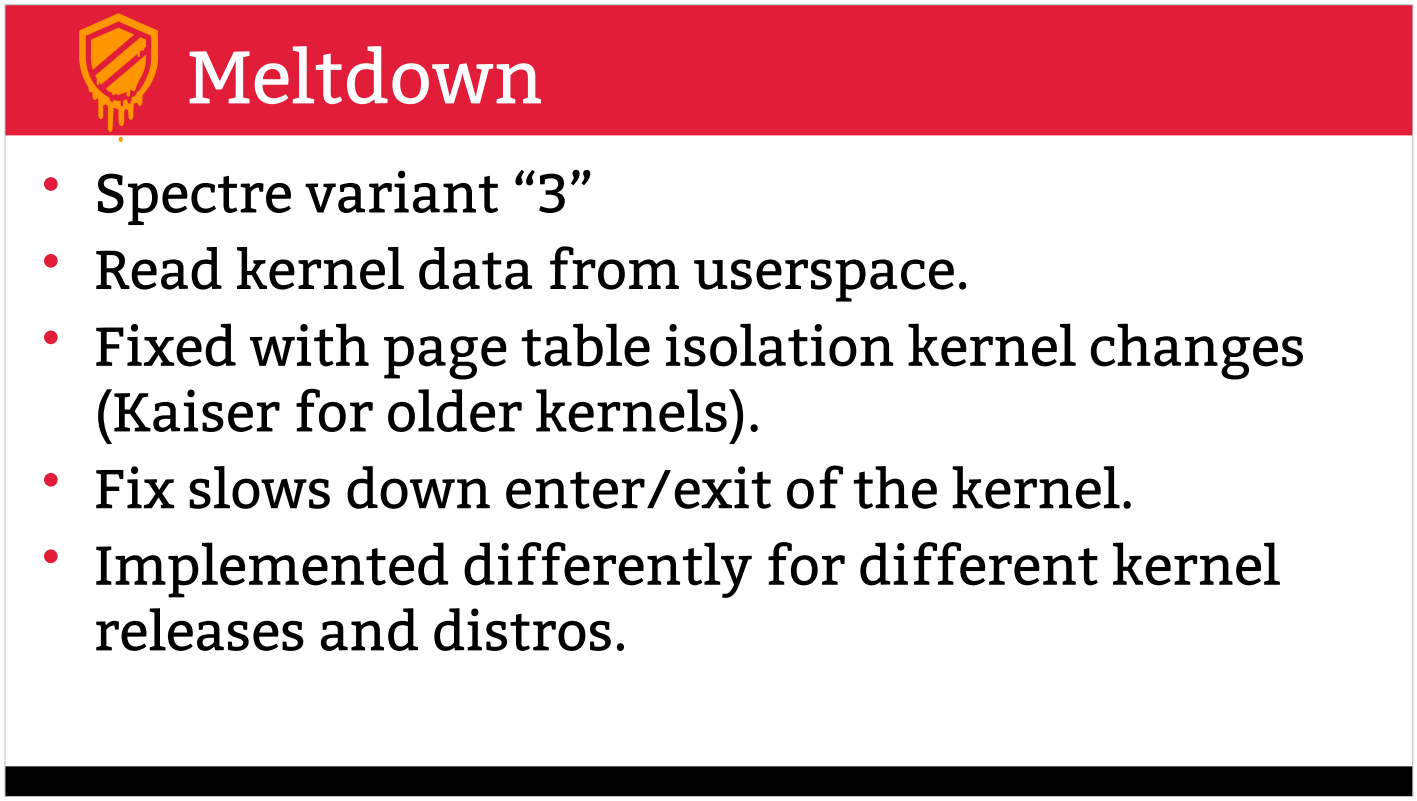
# variant 2 – Fix dates*

- x86
  - 4.15.9 11 March 2018
  - 4.14.26 11 March 2018
  - 4.9.87 11 March 2018
  - 4.4.121 11 March 2018

*More fixes and optimizations happened in later kernels

Again, here are the dates and releases that this problem was first found and fixed in. There were later releases that improved on the fixes here, so updates are still necessary.

ARM processors did not seem to be vulnerable to this class of problems.




This is variant “3” of spectre

It allows a userspace program to be able to read data from within the kernel

It is fixed with a large number of kernel patches that implement “page table isolation”. This moves all kernel memory outside of the system when entering/exiting userspace, preventing userspace from being able to see kernel memory entirely.

The “Kaiser” paper first suggested this solution for a different type of kernel vulnerability, and this is what it is sometimes called

It slows down every time you enter or exit the kernel from userspace, which means that processes that do a lot of I/O accesses are hit hard.

Different distributions backported the needed fixes here in very different ways. Running benchmarks on the distributions results in vastly different numbers. Please test yourself if you are stuck using a distribution-based kernel to see if you should just update to a newer release that works much faster.

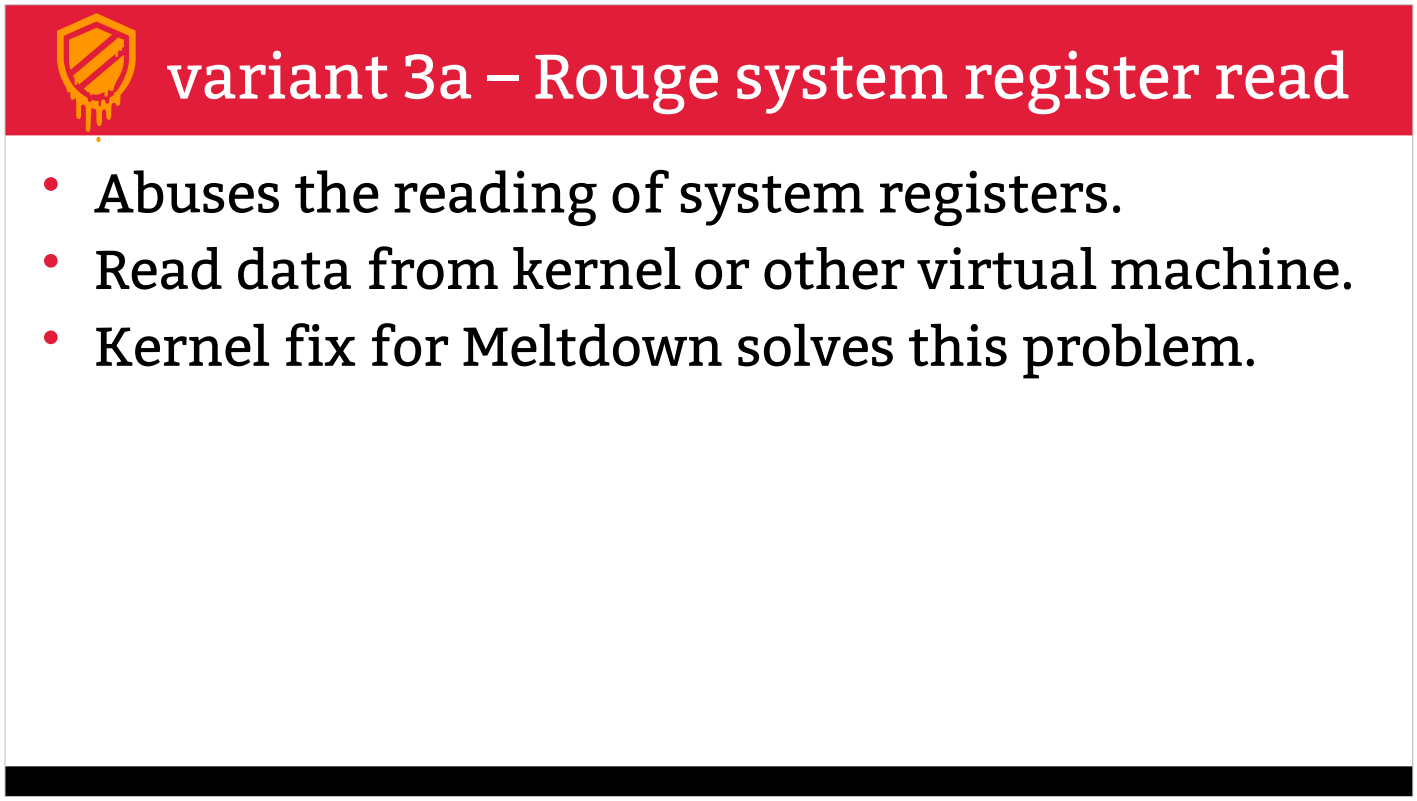
# Meltdown – fix dates

- x86
  - 4.14.11 02 January 2018
  - 4.9.75 05 January 2018
  - 4.4.110 05 January 2018
- ARM
  - 4.15.4 17 February 2018
  - 4.14.20 17 February 2018
  - 4.9.93 08 April 2018

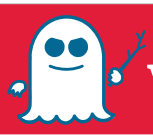
Here are the dates that this was first fixed.

Note, the backports to 4.9 and 4.4 are VERY different from what is in 4.14 and newer. There are still some know holes in the backported changes, as they are implemented in a different manner. So be careful if you need to use these old kernel versions, and test to ensure that your systems are safe.

Also note that newer kernels run faster than older ones do for this problem. So if at all possible, update to 4.14 or newer please.




This is an example of kernel code that is completely correct as written.

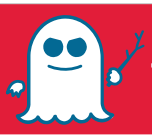


## variant 4 – Speculative Store Bypass

- Can execute and read beyond what is expected.
- Read data from kernel or other virtual machine.
- Minor kernel changes.
- Microcode update required for full protection.

This is an example of kernel code that is completely correct as written.

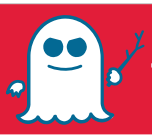

This is an example of kernel code that is completely correct as written.



# variant 5 – Lazy FP state restore

- Uses the old “lazy floating point restore” method to read memory of another process or virtual machine.
- Details to be published June 27.
- Linux kernel fixed in 2016.

This is an example of kernel code that is completely correct as written.



# variant 5 – Fix dates

- x86
  - 4.6 15 May 2016
  - 4.4.138 16 June 2018

This is an example of kernel code that is completely correct as written.

# Why this is such a big deal

- CPU bugs require software & microcode fixes.
- All operating systems are affected.
- Performance will decrease.
- Totally new class of vulnerabilities.
- We will be finding, and fixing, these for a very long time.

This class of bug shows up in all modern CPU architectures. It requires microcode fixes combined with compiler and operating system fixes, and it affects all operating systems.

These fixes will cause performance to decrease under most workloads because the advanced speculation of data has to stop, which slows the CPU down.

This is a whole new class of vulnerabilities, which does not happen very often at all.

And as can be seen by the announcements of new variants being found over the past few weeks and months, these types of bugs will continue to need to be fixed for a long time in the future.

# Linux's response

- Companies were notified, but not developers.
- Developers notified very late, resulting in delay of fixes.
- Majority of the world runs non-corporate kernels.
- Intel is now working with some developers.

When these bugs were originally announced the first week of January, the community was caught off-guard. We did not have a fix for most of the issues, and even the fixes that we had were not sufficient in some cases.

This was caused because the original vendors notified other companies, not the kernel developers responsible for these areas of the kernel.

Businesses are used to working with other businesses, but it turns out, in the big cloud systems, it is community-based kernels that are in the large majority of them. One major cloud vendor reports that less than 10% of their Linux instances run an "enterprise supported" distribution. The other 90% runs either kernel.org kernels, or community-supported distributions like Debian and Fedora.

Intel has learned from this and is now working with a small subset of developers to try to address these types of issues. So far it is getting better, but as the LazyFP announcement leak shows, it is still not ideal.

# Keeping a secure system

- Take ALL stable kernel updates.
  - Do **NOT** cherry-pick patches.
- Enable hardening features.
- Update to a newer major kernel version where ever possible.
- Update your microcode/BIOS!

These are the things that you have to do in order to keep your systems secure.

You have to take ALL of the stable kernel patches. You can not just “cherry pick” individual patches that you think you might need. You will miss fixes that you do not realize you need, as they turn out to be important later.

Always update to a new stable kernel, including major versions. Do not stay at older stable versions for no good reason. Newer versions fix problems that you do not know you have yet. Proof of this is the LazyFP bugfix that was fixed in 2016.

Update your microcode and BIOS with the latest updates from your vendor. They make these updates for a reason, and it is to fix the buggy CPU models that Spectre research is showing.


THE
LINUX
FOUNDATION