IMMUNITY
KNOWING YOU'RE SECURE

Kernel Memory disclosure & CANVAS Part 1 - Spectre: tips & tricks

IMMUNITY

October 25, 2018 By: Ricardo, Immunity Inc

Table of Contents

A lot has been written about *Spectre* (and *Meltdown*) and providing relevant new information on this topic can be a challenge. However, as far as we know, there are no publicly available real-world (reliable and functional) exploits for these bugs. Behind all real-world exploits there is a story and in this article we are going to share some implementation tips for both our Linux and Windows exploits. This is also the first of two (or more) articles dedicated to *Meltdown*-like bug exploitation.

1. Designing your Linux exploit

When we started to write the Linux exploit, it soon became clear that the amount of work required to write a reliable exploit would be formidable. Even for dedicated exploit development teams, there is a limit to the amount of time developers are allowed to spend on one task. For this reason, we had to think several steps ahead and try to answer the following:

1. Should we be using *Spectre*, *Meltdown* or even a mix of both?
2. How could we write the exploit in order to speed up the development of similar exploits in the future?
3. What are the possible strategies for dealing with an unreliable primitive?

1.1 Choosing a primitive

An interesting thing about *Spectre* and *Meltdown* is that while most people almost always understand what *Meltdown* is about, they have a hard time understanding *Spectre* even in its first variant (Spectre V1). In particular, a lot of people did not know that Spectre can leak the kernel's memory or they assumed that an *eBPF* primitive was necessary (most likely because they misread p0's paper). This is not the case. Since we had quickly found a reliable way to leak the kernel memory using *Spectre v1*, we had to choose between this primitive and *Meltdown*.

In the end, we chose *Spectre* over *Meltdown* for several reasons:

- There are two known ways to leak memory using *Meltdown*: by using the Intel's TSX extensions or by performing explicit memory accesses. The first case was not considered a viable option because one of the interesting aspects of these bugs is that they are very old[1][2] and cover a wide variety of affected CPUs. In this context, TSX is a problem since it is a recent extension (~2012 according to wikipedia). The second case requires that the exploit catches a few SIGSEGV per byte read. For a fully working exploit this would require thousands of SIGSEGV to handle, something easy to turn into a signature for an HI{D,P}S, which in our opinion is also less than ideal.
- With the *Spectre* bug we were confident that we could write a primitive that would work on both Linux and Windows with only minor changes (calling convention, potential external API). This assumption was validated when we developed the Windows version[3].
- As far as we know, there is no memory that can be leaked with *Meltdown* that can not be leaked with *Spectre*, but even if it could not, it would be somewhat irrelevant to our actual use case.

1.2 Spectre now, what about later?

While *Spectre* is interesting because it is a hardware bug, from a practical standpoint it is nothing more than a read primitive within the kernel memory (albeit a very *safe*[4] one). There were a lot of software bugs with similar impact in the past so it was safe to assume that there would be others in the future. For this reason, we decided to write a generic exploit (front end) that relies on an abstracted API whose back end primitives were implemented via *Spectre*.

The benefits are numerous:

1. When the architecture is stable, adding a new exploit means writing a simple back end to support the API abstraction. As a result an exploit writer can focus on the quality of their primitive without focusing on the use of the primitive.

2. When a bug is found and fixed within the front end, or similarly when a feature or a target is added in the front end, all the exploits are potentially improved[5].

3. In some cases this speeds up debugging because one may compare the results provided by two different back ends.

1 To exploit *Spectre*, a FLUSH+RELOAD attack is used to detect which cache line has been accessed speculatively. As explained in the original paper from Yarom et al., this targets the LLC (hence the L3 on modern processors but also the L2 on older ones such as the Core 2 Duo).

2 For the same reason *rdtscp* should *not* be used. *rdtsc* is perfectly fine as long as you understand the concepts of serialization and memory fences.

3 The primitive is as fast on Windows as it is on Linux when the memory targeted is not from the Paged Pool.

4 There is indeed 0 risk to crash the kernel.

5 In the case of targets this is only true if both exploits have common targets.

To be fair, the 2nd point is a bit optimistic. For example, when we wrote the *show_timer* exploit, we decided to focus on generic kernel targets. This meant that the exploit had to have the ability to resolve kernel symbols based on the leaked kernel memory. *show_timer* (which is fast) clearly benefits a lot from this. However the *Spectre* exploit which gained this ability in the process (sharing the same front end) proved to be too slow to be able to use it efficiently. While it works, some ground work had to be performed to reduce the amount of reads it needed to perform. This is tricky in itself because it could affect the stability of the *show_timer* exploit (less reads means less confidence). Theoretically speaking, the shared API benefit was immediate, practically speaking there were some caveats.

So far we have implemented 4 back ends and the 5th will most likely be *L1TF-VMM*:

Bug name / Reference	**Prevented by SMAP?**	**Leak speed**	**Safe?**	**Difficulty to write a *good* backend**	**Interesting targets ?**
Spectre **CVE-2017-5753** (spectre_file_leak)	**Yes**. For some reason, SMAP prevents the leak of kernel memory.	**Slow**. The more recent your CPU, the faster the primitive (i.e. i7 is faster than i5 which is faster than the Core 2 Duo). The speed also depends on whether you use virtualization or not (i.e. bare metal is faster than vmware which is faster than virtualbox).	**Yes**. The memory is never accessed directly so there are no risk of crashing the kernel.	Writing an exploit that is both reliable and fast is **quite hard** compared to other exploits.	**Yes**. No need to explain.
Meltdown **CVE-2017-5754** (meltdown_file_leak) <u>Note</u>: It has never been released and is mostly for testing purpose.	**Untested**.		**Yes**. Attempting to access to kernel memory from the userland is safe.		
CVE-2017-18344 (*show_timer_leak*)	**Yes** but only if the exploitation scenario relies on userland pages (see Part 2).	**Very fast**. The shadow is leaked within a second.	**No**. Dereferencing incorrect pointers may lead to crashes.	**Easy**.	**Yes**. A huge amount of targets.
CVE-2018-14656 dmesg_leak	**No**. The kernel transmits the leaked memory	**Slow**. This is because the kernel may suppress messages within	**Yes.** Deferencing incorrect pointers is safe (see Part 2).	**Easy.**	**No**. Unfortunately, the bug did not live very long.

	using *kmsg* therefore SMAP is useless in this case.	*kmsg*. The *Spectre* exploit running on an i7/ vmware is actually faster.			

Let's now talk about the API. We chose a very straightforward design:

int READ_init(void).

- This is what is called to initiate the back end. It may return an error code if the target is not vulnerable, if the exploitation requires a condition that is not met or if the read primitive is not working for some reason.

void READ_end(void).

- This function is called at the end of the process to clean resources (close files, mappings, free memory, etc.).

int READ_get_byte(unsigned long ptr, unsigned char *val).

- This primitive reads a byte at a specific location *ptr* and returns it in *val*. It returns an error code in case of error. This is probably one of the most important functions of this API. It is rarely used directly though and instead is wrapped within the front end's high level API.

int READ_get_kbase(unsigned long *kbase).

- This function returns the kernel base address. For example in the case of the *Spectre* exploit we use Gruss et al's prefetching technique. This is mostly because we have used this primitive for a while now and it is efficient. This specific primitive was killed by the (K)PTI patch.

int READ_reload(void).

- For some backends, the primitive might not be completely stable or the conditions required to exploit the primitive might not be met anymore. When too many errors occur, this API is called and is responsible for fixing the situation. Practically speaking and depending on back ends, it may only return 0 without doing anything.

Note: This is obviously a Linux only API as an *unsigned long* is 32 bits on Windows, thus not a valid address. The Windows API is slightly different but is not as mature.

1.3 Dealing with unreliable primitives

The initial (and incredibly annoying) problem that we had to face was the unreliability of the read primitive. In a nutshell, at any time when running the exploit, one could face the following situations:

1. *READ_get_byte()* could fail for one or more bytes at one specific address. This fail could be time dependent (fixed by process activity) or even boot dependent (fixed by a reboot). It could be frequent or it could be statistically improbable.
2. *READ_get_byte()* could return wrong results for at least two reasons: parasite measurements as well as the tricky case of the value 0.
3. *READ_get_byte()* was very slow at the beginning. When Dave tweeted one of our first achievements, the exploit would easily take 12 to 15 minutes to leak the first 128 bytes of /etc/shadow on an i7.

Since running the exploit twice in a row would most likely yield different results, the debugging was tedious. For this reason we settled on a dichotomy mechanism. Benefiting from the architecture, we would divide the problem in two. First we would write a solid front end (the shadow leak algorithm) and only then would we attempt to fix the *Spectre* back end in order to remove some of the aforementioned side effects. To do that, the solution was simply to write an additional back end that would rely on a kernel module. This way we would resume work on a deterministic primitive and since the only difference between two binaries compiled with two different back ends would be the result of the *READ_get_byte(),* the debugging was more straightforward.

While it would be natural to assume that we then mostly focused on the improvement of the read primitive, this is not how things happened. While we spent some time on the primitive, in the end we were not sure if we would be able to develop a very fast and reliable back end. If we had been working two weeks on the back end without any significant improvement, we would still not be remotely close to have any functional exploit. For this reason, we decided to take a different path and instead focus on the front end, which we rewrote based on two observations:

- Some of the data to leak is (partially) predictable. If you can predict a value then there is no need to actually read it (at least not entirely). Your prediction will be based on both the type of the variable holding the data and the expected value of that data. For example suppose you need to read an *unsigned int* whose value is smaller than 255 then in memory it will be stored in this *4 bytes array* starting at address *p*: 0 0 0 X Using *READ_get_dword()* which itself calls *READ_get_byte()* four times would be very inefficient. As such it would be wiser to create a specific API for that data that would only be reading the X value and return an *unsigned int*. Another example would be a kernel address for which the two MSB are always 0xff.
- When an error is detected, a backtracking strategy is applied. The shadow leak algorithm (which is not described in this article) is composed of several steps. At the end of each step, one could clearly identify some kernel specific patterns[6] proving that the step is completed successfully (an endpoint). Therefore instead of iterating continuously through the whole algorithm, it is smarter to go from one endpoint to another. Eventually the detection of an error

6 An obvious example of such endpoint is the ‘root:’ string which always prefixes a */etc/shadow* file. This is obviously the last endpoint of the algorithm.

would lead to the restart of the exploit at the last known endpoint which saves time and eventually reaches the page containing /etc/shadow.

Applying both strategies allowed us to significantly reduce both the *number of errors* (since we removed a lot of read operations) and *its impact on the outcome* of the algorithm.

Note: Both the predictability and the backtracking mechanisms are still implemented in the front end which means that all our *Spectre*-like exploits are using them even when their read primitives are reliable. This could be seen as a problem because our predictions could be wrong. Practically speaking, though, the exploit almost always worked no matter what the kernel version was (Ubuntu 15 to 17, Arch Linux, Fedora 24 to 27, Centos 6.9, etc.).

While it allowed us to get the first satisfying results, it was not enough for a fully working CANVAS exploit, especially considering the potential for running on slower hardware than our lab environment. At this point we knew that even minor improvements in the back end would significantly increase our performance so we focused our efforts on this part. It was a tedious task as it required a lot of trickery. We will describe a few of our optimizations here.

1.4 Improving the backend

a) Since we are dealing with non deterministic primitives, it means that we have to use statistics. There is no need to use very complex mathematics, even a simple counter is already an improvement. Empirical thresholds are working just fine but remember one golden rule in statistics: you are only as good as the size of your sample set.

b) *READ_reload()* is your friend. If for some reason one specific memory address cannot be read, then wait or trigger some system activity and read it again.

c) The specific case of the 0 value is a problem because you can not (unless we missed something) detect that specific value (this is even more problematic on Windows because of paged pools). So the problem is simple, if the FLUSH+RELOAD does not return any result, is it because the value is 0 or is it because *Spectre* is just not working on this specific byte? To avoid this situation, we used two strategies:

1. In the front end, expect the 0 as much as possible and try not to read them
2. Read several times and remember that if you can read other bytes of the same cache line then this probably indicates a 0 value.

d) Beware of parasite values. Parasite values are invalid values that appear alongside valid ones on multiple samples and these will most likely occur in even the most robust of exploitation strategies for these kinds of bugs. The good thing about them is that they should be constant for the lifetime of the process. This means the process itself may be able to predict them. This is an advantage but it does not mean that you will be able to tell the difference between a parasite and actual data holding the same value. Our only advice is to implement a back end in such a way that they would almost never occur.

The final version of the exploit leaks /etc/shadow in less than 10s on an i7 and 1m30 on a Core 2 Duo. It works on all the CPUs we tested in both bare metal and virtualized environments.

While most of our implementation has been focused on Linux so far, let's now expand our reach into Windows as well. However instead of focusing on the architecture of the exploit, this time we will be explaining algorithms.

2. Spectre VS Windows: exploitation notes

On Linux the *Spectre* exploit locates and then dumps the "/etc/shadow" (or any other file such as Kerberos tickets) from the kernel memory. While dumping the file is one thing, locating it is another and it is not always that easy.

2.1 File caches: dead end?

Just like network proxies with static content, modern kernels use a file cache. Whenever a read/write operation is performed, they need to avoid read/write disk operations as it would kill the performance immediately. In order to do that they store pieces (or all) of a file in RAM using specific structures and perform operations on these copies (including synchronization whenever required). By default, all the files may be cached (even sensitive ones). Could we use that approach on Windows?

On Windows whenever you have a file handle, you can immediately leak the kernel address of the corresponding *_FILE_OBJECT* structure using *NtQuerySystemInformation()* (see Alex Ionescu's talk for example) and from that point easily locate the file itself in memory. The trick works for all the objects and even on Windows 2016 but only if the exploit's process is not running at Low Integrity Level (afaik).

On Windows the Cache Manager is implemented using the CC-prefixed API. The Cache Manager mechanisms have not changed much across time so it is very easy to understand how they work especially using Windows' leaked/published source code.

What is important is that in the *_FILE_OBJECT* the cache manager stores:

- the *SectionObjectPointer*
- the *PrivateCacheMap*

At this point and without even using Spectre, attackers are one (or two) pointer leak(s) away from the target file's content. On Windows, several hives (stored in registry files) may be used to store (encrypted) secrets depending on the type of account and the configuration. There are at least 3 hives that attackers may want to get their hand on:

1. HKLM\SYSTEM
2. HKLM\SECURITY
3. HKLM\SAM

With the notable exception of *HKLM\SYSTEM* whose content is (at least for an attacker's needs) retrievable using the *NT/ZW* registry API, the two others require SYSTEM level privileges. They are opened (*exclusively*) by a single process which is not LSASS contrary to one may think but System.

On Windows 2016, two of *System*'s handles:

```
0364: Object: ffffe0016e83fbc0  GrantedAccess: 00020003 (Protected)
Entry: ffffc001a1c06d90
Object: ffffe0016e83fbc0  Type: (ffffe0016d52a9a0) File
   ObjectHeader: ffffe0016e83fb90 (new version)
       HandleCount: 1  PointerCount: 32745
       Directory Object: 00000000  Name:
\Windows\System32\config\SECURITY {HarddiskVolume2}


037c: Object: ffffe0016e83b550  GrantedAccess: 00020003 (Inherit) Entry:
ffffc001a1c06df0
Object: ffffe0016e83b550  Type: (ffffe0016d52a9a0) File
   ObjectHeader: ffffe0016e83b520 (new version)
       HandleCount: 1  PointerCount: 32739
       Directory Object: 00000000  Name:
\Windows\System32\config\SAM {HarddiskVolume2}
```

So the idea is to find in memory the location of the two missing hive files ("\Windows\System32\Config\SAM" and "\Windows\System32\Config\SECURITY") and to leak their content since they are opened by *System*. However this can not work because, as we would discover later, these files are always opened using the *FO_NO_INTERMEDIATE_BUFFERING* flag, shortcutting the Cache Manager. When such a flag is used, both *PrivateCacheMap* and *SectionObjectPointer* are NULL or empty.

In a nutshell, the Cache Manager exploitation is extremely simple on Windows and could be used to leak files if required but not the two sensitive hives.

2.2 Registry Keys objects

If, like the author, you come from the Unix world, sometimes you forget that not everything is a file. In fact, when dealing with the Windows kernel it is wiser to think in terms of objects. The idea of reaching the registry through the Cache Manager would have worked if the registry files were *normal* files (or manipulated in a classical way). However the registry on Windows is so important that there is an entire part of the kernel dedicated to the management of the hives, this is called the *Configuration Manager* (the corresponding API is prefixed with *Cm*).

More specifically, the Windows kernel uses an internal memory representation of the registry's content. It is obvious when you think about it that the kernel would have the whole registry mapped in memory one way or another given the amount of requests for it! This article is not the best suited to introduce registry internals especially since it has been done intensively already (and in a much better way). In particular we recommend reading Brendan Dolan-Gavitt's articles (1, 2, 3, 4), Ivan's old blog if you can read French and of course the almighty *Windows Internals* book (*Configuration Manager* chapter).

An unprivileged process can easily get a handle on a key belonging to the SAM hive. I have to thank Ivan for that tip and also both for hinting at the *Configuration Manager* and pointing out that the object corresponding to the handle was of type *_CM_KEY_BODY (*one of the kernel objects handled by the Object Manager*)*. Once you know this information and since you can retrieve the corresponding address using the *NtQuerySystemInformation()* API everything else is quite obvious if you have read the aforementioned blog posts or book chapter.

If you have not, let me sum up the key (no pun intended) points. The organization of the registry in memory is similar to that of a file system. *Values* and *Keys* are two different *Cells* represented by dedicated structures in memory. Think of a Key as the equivalent of a directory and of a Value as the equivalent of a file (it is of course slightly more complex). From the *_CM_KEY_BODY* object you can reach your first Key (*_CM_KEY_NODE* object). From any Key you can either reach (more or less directly) a *Value* (*_CM_KEY_VALUE* object) or a *Subkey* (which is also a Key). This allows you to reach any value or any Subkey below "HKLM\SAM"[7] by following a path. For example if we need to reach value *X* stored in "*HKLM\SAM\FOO*". We need to:

- Leak the "*HKLM\SAM*" object's address and find its corresponding Key object in memory.
- Enumerate all the Subkeys and find the one named "*FOO*"
- From "*HKLM\SAM\FOO*", enumerate all the values and find the one named "*X*"
- Retrieve the location of *X*'s content and leak it.

Is it that simple though? Not exactly.

- One may think that you can go from one structure to another using a simple pointer dereference but this is not how it is done. Instead, each *Cell* is located in memory using an Index (*Cell Index*) and the translation "Index to Pointer" is performed using a page-walk like mechanism.
- There is another layer of indirection between keys and subkeys and between keys and values.

2.2.1 From indexes to pointers

The resolution process is described in moyix's blog post. Instead of giving formal explanations, let's just demonstrate it through Windbg.

First let's take a registry key object and assume we leaked its address:

```
lkd> !object fffff8a0000893a0
Object: fffff8a0000893a0  Type: (fffffa80018e6650) Key
    ObjectHeader: fffff8a000089370 (new version)
    HandleCount: 1  PointerCount: 1
    Directory Object: 00000000  Name: \REGISTRY\MACHINE\SYSTEM\CONTROLSET001
```

In this case, we can see that the corresponding key is "HKLM\SYSTEM\CONTROLSET001" and the corresponding structure is *_CM_KEY_BODY:*

7 Which is only taken as a practical example.

```
lkd> dt _CM_KEY_BODY fffff8a0000893a0
nt!_CM_KEY_BODY
   +0x000 Type             : 0x6b793032
   +0x008 KeyControlBlock  : 0xfffff8a0`00023820 _CM_KEY_CONTROL_BLOCK
   +0x010 NotifyBlock      : (null)
   +0x018 ProcessID        : 0x00000000`00000004 Void
   +0x020 KeyBodyList      : _LIST_ENTRY [ 0xfffff8a0`000893c0 - 0xfffff8a0`000893c0 ]
   +0x030 Flags            : 0y0000000000000000 (0)
   +0x030 HandleTags       : 0y0000000000000000 (0)
   +0x038 KtmTrans         : (null)
   +0x040 KtmUow           : (null)
   +0x048 ContextListHead  : _LIST_ENTRY [ 0xfffff8a0`000893e8 - 0xfffff8a0`000893e8 ]
```

Clearly this structure is mostly meant for Windows management. The corresponding data is not stored within this structure but instead may be found using the information stored inside. More precisely, the interesting fields are kept within its *KeyControlBlock*:

```
lkd> dt _CM_KEY_CONTROL_BLOCK 0xfffff8a0`00023820
nt!_CM_KEY_CONTROL_BLOCK
   +0x000 RefCount         : 8
   +0x004 ExtFlags         : 0y0000000000000000 (0)
   +0x004 PrivateAlloc     : 0y1
[...]
   +0x010 KeyHash          : _CM_KEY_HASH
   +0x010 ConvKey          : 0x80f50565
   +0x018 NextHash         : (null)
   +0x020 KeyHive          : 0xfffff8a0`00024010 _HHIVE
   +0x028 KeyCell          : 0x160
   +0x030 KcbPushlock      : _EX_PUSH_LOCK
   +0x038 Owner            : (null)
   +0x038 SharedCount      : 0n0
   +0x040 SlotHint         : 0
   +0x048 ParentKcb        : 0xfffff8a0`000234a8 _CM_KEY_CONTROL_BLOCK
[...]
```

Following with the (x86) *MMU* analogy, two fields are interesting:

- The *KeyCell* (the *Cell Index)* which could be seen as the equivalent of the *virtual address* for the registry. This can be seen as an array of indexes.
- The *KeyHive* which acts like the *CR3*, providin1g a context for the resolution.

Performing the resolution step by step

The *Hive* possesses a two entry array called the *Storage*:

```
lkd> dt _HHIVE 0xfffff8a0`00024010
nt!_HHIVE
   +0x000 Signature        : 0xbee0bee0
   +0x008 GetCellRoutine   : 0xfffff800`02b07210     _CELL_DATA*  nt!HvpGetCellPaged+0
```

```
  +0x010 ReleaseCellRoutine : (null)
[...]
  +0x0a0 StorageTypeCount : 2
  +0x0a4 Version          : 5
  +0x0a8 Storage          : [2] _DUAL
```

Each of the entry is a *_DUAL* structure and the selection of the index is performed by extracting bit 31 (**0** in this case since (0x160 & 0x80000000) >> 0x1F equals **0**):

```
lkd> dt _DUAL 0xfffff8a0`00024010 +0x0a8 + 0*0x278
nt!_DUAL
  +0x000 Length          : 0xa3f000
  +0x008 Map             : 0xfffff8a0`0002a000 _HMAP_DIRECTORY
  +0x010 SmallDir        : (null)
  +0x018 Guard           : 0xffffffff
  +0x020 FreeDisplay     : [24] _FREE_DISPLAY
  +0x260 FreeSummary     : 0x7fffff
  +0x268 FreeBins        : _LIST_ENTRY [ 0xfffff8a0`02e4fe00 - 0xfffff8a0`02b57f20 ]
```

Following with more indirections, we can now extract the next index which is composed of bits 21 to 30 (hence 10 bits). The *_HMAP_DIRECTORY* is in fact an array of 2^10 = 1024 pointers. In this case, the good entry is the **first** one since (0x160 & 0x7FE00000) >> 0x15 equals 0.

```
lkd> dq 0xfffff8a00002a000 L 1
fffff8a0`0002a000  fffff8a0`0002c000
lkd> dt _HMAP_TABLE fffff8a0`0002c000
nt!_HMAP_TABLE
  +0x000 Table           : [512] _HMAP_ENTRY
```

The corresponding table is an array of 512 *_HMAP_ENTRY* structures. The whole array is stored in 4 pages since each entry uses 32 bytes. The corresponding index is composed of bits 12 to 20 (hence 9 bits). In this case, the index is **0** (again) since (0x160 & 0x1FF000) >> 0x0C equals **0**.

```
lkd> dt _HMAP_ENTRY fffff8a0`0002c000 + 0*0x20
nt!_HMAP_ENTRY
  +0x000 BlockAddress    : 0xfffff8a0`09592000
  +0x008 BinAddress      : 0xfffff8a0`09592009
  +0x010 CmView          : (null)
  +0x018 MemAlloc        : 0x1000
```

Once the correct *_HMAP_ENTRY* is found, its *BlockAddress* reveals the address of the bin as proven by its **signature**.

```
lkd> db 0xfffff8a0`09592000 L 10
fffff8a0`09592000  68 62 69 6e 00 00 00 00-00 10 00 00 00 00 00 00  hbin............
```

Finally, the 12 remaining lsb (bits 0 to 11) are used as an offset, in this case **0x160** since 0x160 & 0x0FFF equals 0x160. Since the bin has a signature of **4** bytes, they too must be skipped in order to find the corresponding structure, once again identified by its **signature**:

```
lkd> db 0xfffff8a0`09592000 + 0x160 + 4 L 60
fffff8a0`09592164  6e 6b 20 00 32 b1 3a 9e-db 12 d4 01 00 00 00 00  nk .2.:.........
fffff8a0`09592174  20 00 00 00 05 00 00 00-00 00 00 00 98 48 2c 00   ............H,.
fffff8a0`09592184  ff ff ff ff 00 00 00 00-ff ff ff ff 10 fc 00 00  ................
fffff8a0`09592194  ff ff ff ff 24 00 01 00-00 00 00 00 00 00 00 00  ....$...........
fffff8a0`095921a4  00 00 00 00 00 00 00 00-0d 00 00 00 43 6f 6e 74  ............Cont
fffff8a0`095921b4  72 6f 6c 53 65 74 30 30-31 00 00 00 f0 ff ff ff  rolSet001.......
```

The name proves that we found the correct place in memory.

Performing the resolution using !reg

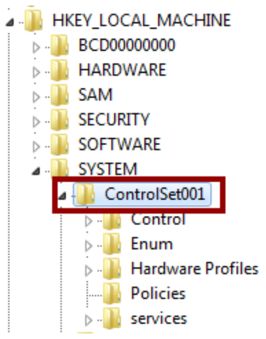
The resolution can also be performed automatically by the !reg command and one can observe that obtain the exact same data:

```
lkd> !reg cellindex 0xfffff8a0`00024010 0x160

Map = fffff8a00002a000 Type = 0 Table = 0 Block = 0 Offset = 160
MapTable     = fffff8a00002c000
MapEntry     = fffff8a00002c000
BlockAddress = fffff8a009592000

pcell:  fffff8a009592164

lkd> db fffff8a009592164 L 60
fffff8a0`09592164  6e 6b 20 00 32 b1 3a 9e-db 12 d4 01 00 00 00 00  nk .2.:.........
fffff8a0`09592174  20 00 00 00 05 00 00 00-00 00 00 00 98 48 2c 00   ............H,.
fffff8a0`09592184  ff ff ff ff 00 00 00 00-ff ff ff ff 10 fc 00 00  ................
fffff8a0`09592194  ff ff ff ff 24 00 01 00-00 00 00 00 00 00 00 00  ....$...........
fffff8a0`095921a4  00 00 00 00 00 00 00 00-0d 00 00 00 43 6f 6e 74  ............Cont
fffff8a0`095921b4  72 6f 6c 53 65 74 30 30-31 00 00 00 f0 ff ff ff  rolSet001.......
```

Clearly the resolution process requires a lot of read operations.

2.2.2 Walking through a registry path in memory

One may think that a Key would keep a reference (*Cell Index*) to its subKeys or Values. However the registry is slightly more complex than this and introduces intermediate structures (*CM_KEY_INDEX*, *CM_KEY_FAST_INDEX*, *etc.*). Let's illustrate this using our example. One can observe that *ControlSet001* has 5 subkeys:



The *_CM_KEY_NODE* objects stores the **number** of subkeys (*SubKeyCounts*) as well as the **location** of the list (*SubKeyLists*) as illustrated below:.

```
lkd> dt _CM_KEY_NODE fffff8a009592164
nt!_CM_KEY_NODE
   +0x000 Signature        : 0x6b6e
   +0x002 Flags            : 0x20
   +0x004 LastWriteTime    : _LARGE_INTEGER 0x01d412db`9e3ab132
   +0x00c Spare            : 0
   +0x010 Parent           : 0x20
   +0x014 SubKeyCounts     : [2] 5
   +0x01c SubKeyLists      : [2] 0x2c4898
   +0x024 ValueList        : _CHILD_LIST
   +0x01c ChildHiveReference : _CM_KEY_REFERENCE
   +0x02c Security         : 0xfc10
   +0x030 Class            : 0xffffffff
   +0x034 MaxNameLen       : 0y0000000000100100 (0x24)
   +0x034 UserFlags        : 0y0001
   +0x034 VirtControlFlags : 0y0000
   +0x034 Debug            : 0y00000000 (0)
   +0x038 MaxClassLen      : 0
   +0x03c MaxValueNameLen  : 0
   +0x040 MaxValueDataLen  : 0
   +0x044 WorkVar          : 0
   +0x048 NameLength       : 0xd
   +0x04a ClassLength      : 0
   +0x04c Name             : [1]  " "
```

One can thus deduce the virtual address of the corresponding structure:

```
lkd> !reg cellindex 0xfffff8a0`00024010 0x2c4898

Map = fffff8a00002a000 Type = 0 Table = 1 Block = c4 Offset = 898
MapTable     = fffff8a000030000
MapEntry     = fffff8a000031880
BlockAddress = fffff8a0092ce000

pcell:  fffff8a0092ce89c

lkd> db fffff8a0092ce89c
fffff8a0`092ce89c  6c 68 05 00 30 06 00 00-81 64 c1 55 40 48 2c 00  lh..0....d.U@H,.
```

```
fffff8a0`092ce8ac  45 02 37 00 08 2e 0a 00-ba 7b 02 84 10 db 01 00  E.7......{......
fffff8a0`092ce8bc  90 5f 5c b2 d0 01 00 00-30 f7 7a 22 00 00 00 00  ._\.....0.z"....
fffff8a0`092ce8cc  00 00 00 00 00 00 00 00-00 00 00 00 d0 ff ff ff  ................
fffff8a0`092ce8dc  76 6b 16 00 04 00 00 80-03 00 00 00 04 00 00 00  vk..............
```

In this case we clearly have a *_CM_KEY_FAST_INDEX* struct with a signature, the number of elements in the following array of 8 bytes entries. Each entry of the array is composed of both a cell index and hash. Using the first or the second cell index, one can find the corresponding *_CM_KEY_NODE* structure as proved by the signature and the name.

```
lkd> !reg cellindex 0xfffff8a0`00024010 0x630

Map = fffff8a00002a000 Type = 0 Table = 0 Block = 0 Offset = 630
MapTable     = fffff8a00002c000
MapEntry     = fffff8a00002c000
BlockAddress = fffff8a009592000

pcell:  fffff8a009592634
lkd> db fffff8a009592634
fffff8a0`09592634  6e 6b 20 00 4e 5d 6d d2-f6 6a d4 01 00 00 00 00  nk .N]m..j......
fffff8a0`09592644  60 01 00 00 4f 00 00 00-03 00 00 00 50 93 12 00  `...O.......P...
fffff8a0`09592654  58 24 01 80 08 00 00 00-d8 46 2c 00 10 fc 00 00  X$.......F,.....
fffff8a0`09592664  ff ff ff ff 2c 00 01 00-00 00 00 00 3c 00 00 00  ....,.......<...
fffff8a0`09592674  8c 00 00 00 00 00 00 00-07 00 00 00 43 6f 6e 74  ............Cont
fffff8a0`09592684  72 6f 6c 00 f0 ff ff ff-08 de 01 00 78 de 01 00  rol.........x...
[...]

lkd> !reg cellindex 0xfffff8a0`00024010 0x2c4840

Map = fffff8a00002a000 Type = 0 Table = 1 Block = c4 Offset = 840
MapTable     = fffff8a000030000
MapEntry     = fffff8a000031880
BlockAddress = fffff8a0092ce000

pcell:  fffff8a0092ce844
lkd> db fffff8a0092ce844
fffff8a0`092ce844  6e 6b 20 00 65 5c 53 b9-82 6b d4 01 00 00 00 00  nk .e\S..k......
fffff8a0`092ce854  60 01 00 00 0e 00 00 00-00 00 00 00 48 3c 37 00  `...........H<7.
fffff8a0`092ce864  ff ff ff ff 2d 00 00 00-80 d5 34 00 a0 c7 06 00  ....-.....4.....
fffff8a0`092ce874  ff ff ff ff 10 00 00 00-00 00 00 00 2e 00 00 00  ................
fffff8a0`092ce884  04 00 00 00 01 00 00 00-04 00 00 00 45 6e 75 6d  ............Enum
[...]
```

The mechanism is slightly similar for Values under a specific Key. The only difficulty is that there exist many different intermediate structures so the exploit needs to be able to handle that.

2.3 Exploiting your registry dump in a few words

Windows credentials and their extraction has been intensively discussed, researched and even documented for a long time. In the context of kernel memory disclosure we are (for now) only focused on registry extraction, more specifically on attacks targeting the SAM, the LSA and the cached credentials. They can be classified in two categories:

- Registry files based techniques. The attacker manages to get access to the raw registry hives as files and she is able to extract the secrets out of them.
- In-memory techniques. The attacker is able to run arbitrary code within the LSASS process and she is able to extract the secrets using a combination of pattern matching and system functions calls.

With *Spectre*, an attacker has the ability to access the full contents of the registry (assuming it is paged in). In a way, *Spectre* is thus *reading* (in a nonconventional way) the registry hive files and as such falls into the first category. A few years ago, *moyix* made a series of blog posts providing interesting summaries on the subject. He wrote the creddump tool as a result which is a set of three offline tools. Each tool combines the extraction of data from two different hives to provide secrets.

Tool Name	**Description**	**Required Hives**
pwdump	The hashed password of a given local user of RID=$RID is stored *obfuscated* as an LM/NTLM hash within the SAM, more precisely in "HKLM\SAM\SAM\Domains\Account\Users\$RID".	HKLM\SYSTEM HKLM\SAM
lsadump	Sensitive credentials (such as default passwords, the Machine Trust Account, etc.) are stored obfuscated within "HKLM\SECURITY\Policy\Secrets".	HKLM\SYSTEM HKLM\SECURITY
cachedump	Cached domain credentials are stored obfuscated in "HKLM\SECURITY\CACHE\NL$n". The password is hashed in the *mscash* format.	HKLM\SYSTEM HKLM\SECURITY

Note: Generally speaking, I would recommend to the users interested in Windows credentials to have a look at a series of blog posts from Bernardo Damele A. G. (1, 2, 3, 4, 5, and 6). Being slightly old, it does not cover the most recent research but, in that regard, mimikatz's source code should be able to cover part of what's missing.

CANVAS' current version of the exploit (*Early Updates* only) is still very basic and only implements the pwdump technique. Please note that on Linux extracting the hash out of the shadow is certainly dangerous but not always lethal. On Windows, even if attackers are unlikely to extract LM hashes, they should still be able to do some damage using the NTLM hashes.

2.4 Tips & Tricks

One of the things that your exploit needs is reliability. In that regard, using a backtracking algorithm such as the one mentioned in section 1.3 seems mandatory. An interesting trick is to observe that some of the aforementioned structures have a signature. This leads to *obvious endpoints*:

```
lkd> dt nt!_CM_KEY_INDEX fffff8a000a74a04
   +0x000 Signature        : 0x666c
   +0x002 Count            : 3
   +0x004 List             : [1] 0x410
lkd> db fffff8a000a74a04
fffff8a0`00a74a04  6c 66 03 00 10 04 00 00-44 6f 6d 61 a0 29 00 00  lf......Doma.)..
fffff8a0`00a74a14  4c 61 73 74 e8 02 00 00-52 58 41 43 03 00 00 00  Last....RXAC....
fffff8a0`00a74a24  00 00 14 00 98 ff ff ff-6e 6b 20 00 e2 fa 60 82  ........nk ...`.
fffff8a0`00a74a34  a1 78 d3 01 00 00 00 00-b0 09 00 00 00 00 00 00  .x..............
fffff8a0`00a74a44  00 00 00 00 ff ff ff ff-ff ff ff ff 01 00 00 00  ................
fffff8a0`00a74a54  38 22 00 00 68 02 00 00-ff ff ff ff 00 00 00 00  8"..h...........
fffff8a0`00a74a64  00 00 00 00 00 00 00 00-00 00 00 00 20 00 00 00  ............ ...
fffff8a0`00a74a74  15 00 00 00 44 69 73 74-72 69 62 75 74 65 64 20  ....Distributed
```

Practically speaking, leaking memory in that area is quite the challenge. Indeed the registry is within the paged pool area which means we need to be sure that the memory to be leaked is paged in (if it is not, the cache is empty as well). This can be solved by performing system actions (ie: by calling the right API) involving the targeted memory. For example one could try to change something within their user profile and observe the effect. There should exist numerous possibilities even without any special privilege. One important thing is that it might depend on whether the server is part (or not) of a Windows network.

Another important thing that you may want to focus on is the speed of the leak. Unfortunately, unless there are undetected shortcuts, one cannot go quickly from the original *_CM_KEY_BODY* object to a specific registry Value and this has a cost. However optimizations are still possible by making the following observations:

- An attacker does not need entire Values but rather part of them. For example in the *pwdump* case, she needs to retrieve part of the content of “*HKLM\SAM\Domains\Account\F*" which is user independent. The entire Value itself on a Windows 7 testing machine is an array of 240 bytes, a lot of them being 0 (see part 1.4) so it would take a *significant* amount of time to fetch it. However we only need 48 bytes, mostly non null, located at very specific offsets and that can be fetched orders of magnitude faster.
- A cache mechanism can be implemented to avoid unnecessary queries in memory. We initially thought of two solutions to implement the cache:
    - An object type agnostic cache. For example we could have hooked *READ_get_byte()* and store/return the memory using a per address strategy (see dmesg in part 2). This would be a big problem practically speaking for several reasons, the biggest being the error management which would be … a nightmare!
    - A per object type cache. This is our chosen solution. Whenever a cacheable object is found (thanks to its signature) its address becomes trusted and is added in cache. This

way the cache possesses entries for *Hive*, *KeyNode* and *KeyValue* objects. Additionally if a specific object is considered correctly located then it means that the "Index to Address" resolution is assumed correct and so the intermediate results should be as well. We chose to also add entries for *HmapDirectory*, *BlockAddress* and *MapTable* objects which practically speaking saves a lot of time.

Even with all these tricks you have no guarantee to be able to exploit the bug efficiently and will probably need additional work. In our case, the current Windows version still has a speed issue. This is why it has not been released officially within CANVAS yet. We expect to resume the work very soon with L1TF-VMM.

The author would like to thank Alfredo, Bas, Dave, Ivan, Lurene, Mark and Skylar.

3. References

[BER1] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes.html
[BER2] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes_16.html
[BER3] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes_20.html
[BER4] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes_21.html
[BER5] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes_28.html
[BER6] http://bernardodamele.blogspot.com/2011/12/dump-windows-password-hashes_29.html
[CACHEDUMP] http://www.securiteam.com/tools/5JP0I2KFPA.html
[CREDDUMP] https://github.com/moyix/creddump
[GGP0_SPECTRE] "Reading privileged memory with a side-channel", Jann Horn
[IONESCU] "I Got 99 Problem But a Kernel Pointer Ain't One", Alex Ionescu, Recon2013
[IVAN1] http://www.ivanlef0u.tuxfamily.org/?p=64
[KASLR_PREFETCH] https://gruss.cc/files/prefetch.pdf
[KPTI] https://en.wikipedia.org/wiki/Kernel_page-table_isolation
[L1TF] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3646
[LSADUMP] https://packetstormsecurity.com/files/10457/lsadump2.zip.html
[MOYIX1] http://moyix.blogspot.com/2008/02/enumerating-registry-hives.html
[MOYIX2] http://moyix.blogspot.com/2008/02/cell-index-translation.html
[MOYIX3] http://moyix.blogspot.com/2008/02/reading-open-keys.html
[MOYIX4] http://moyix.blogspot.com/2008/02/keys-open-by-hive.html
[MOYIX5] http://moyix.blogspot.com/2008/02/syskey-and-sam.html
[MOYIX6] http://moyix.blogspot.com/2008/02/decrypting-lsa-secrets.html
[MOYIX7] http://moyix.blogspot.com/2008/02/cached-domain-credentials.html
[MS_CM] https://docs.microsoft.com/en-us/windows/desktop/fileio/file-caching
[POOLS] https://docs.microsoft.com/en-us/windows/desktop/memory/memory-pools
[PWDUMP] https://en.wikipedia.org/wiki/Pwdump
[TSX] https://software.intel.com/en-us/blogs/2013/06/07/web-resources-about-intelr-transactional-synchronization-extensions

[WINCM] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-configuration-manager
[WINSRC] (CENSORED)

Immunity, Inc:
www.immunityinc.com
sales@immunityinc.com
Phone: +1 786.220.0600
2751 N. Miami Ave.
Suite #7
Miami, Florida 33127