15:10 Windows Kernel Race Condition Analysis While Accessing User-mode Data

by BSDaemon and NadavCh

In 2013, Google's researchers Mateusz Jurczyk (J00ru) and Gynvael Coldwind released a paper entitled "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns."[39] They discussed race conditions in the Windows kernel while accessing user-mode data and demonstrate how to find such conditions using an instrumented emulator. More importantly, they offered a very thorough explanation of how the identification of such issues is possible, specifically listing these conditions of interest:

1. At least two reads of the same virtual address;
2. Both read operations take place within a short time frame. The authors specifically recommend identifying reads in the handling of a single kernel entrance;
3. The reads must execute in kernel mode;
4. The virtual address subject to multiple reads must reside in memory writable by Ring-3 threads, in order for the user mode to be able to take advantage of the race.

Interestingly most of these races are exploitable—i.e., possible for the attacker to win—on modern machines given multiple CPU cores. The exceptions would be in memory areas that are administrator-owned, or in situations that are early boot—and thus not in a memory area that can be mapped by an attacker. Even if the user-mode area is only writable by administrator-owned tasks, it might still be a problem given that it leads to code execution in kernel mode that is prohibited to the administrator and bypasses kernel driver signing. Notably, the early boot cases are only non-exploitable if they are not part of services prohibited after boot.

We reproduced Google's research using Intel's SAE[40] and got some interesting results. This paper explains our approach in the hope of helping others understand the importance of documenting findings and processes. It also demonstrates other findings and clarifies the threat model for the Windows Kernel, thanks to our discussions with the MSRC. We share all the traces that generated double fetches for Windows 8 (pre and post booting) and Windows 10 (again, pre and post boot).[41]

We also share our implementation: it contains the parameters we used for our findings, the tracer, and the analyzer—and can be used as reference to audit other areas of the system. It also serves as a good way to understand the instrumentation capabilities of Simics and SAE, even though these are, unfortunately, not open-source tools.

For the findings per se, almost all parameters appear to be probed and copied to local buffers inside of try-except blocks. We flagged them as double-fetches because some of the pointers are probed first and then accessed to copy out actual data, like `PUNICODE_STRING->Buffer`. One of them is not inside a try-catch block and is a local DoS, but we do not consider it a security issue, since it is in administrator-owned memory. Many of them are not related to Unicode strings and are potential escalations-of-privilege (see Figure 10), but once again, for the threat model of the Windows Kernel, administrator-initiated attacks are out of scope.

Microsoft nevertheless fixed some of the reported issues. Obviously, mitigations in kernel mode might still prevent or make exploiting some of those very difficult.

Our findings concern three classes of issues:

Admin ↔ kernel cases: Microsoft did fix these, even though their threat model does not consider this a security issue. They may have considered the possibility of these cases used for a CSP bypass or a sandbox bypass—even though we did not find cases where a sandboxed process had administrator privileges.

Local DoS cases: These were also fixed, considering that a symlink can be created by anyone and this was a non-admin-only case.

Other cases: The rest of the cases do not appear to be of consequence of security. We are sharing the traces with the community, in case anyone is interested in double-checking :)

[39] Mateusz Jurczyk and Gynvael Coldwind, "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns," Google, 2013. `unzip pocorgtfo15.pdf bochspwn.pdf`

[40] Nadav Chachmon et al., "Simulation and Analysis Engine for Scale-Out Workloads," Proceedings of the 2016 International Conference on Supercomputing (ICS '16), Istanbul, Turkey; `unzip pocorgtfo15.pdf chachmon.pdf`

[41] `git clone https://github.com/rrbranco/kdf ; unzip pocorgtfo15.pdf kdf.zip`

Tool Description

We implemented a Kernel Double Fetch tool (KDF), similar to the tool described in *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*.[42] The tool has a runtime phase, in which KDF candidates are identified, and a post-runtime phase, in which these KDF candidates are analyzed based on whether the fetches are actually used by the kernel.

In the runtime phase, there is a `ztool` that looks for system-call related instructions. When such an instruction is triggered, the tool will dynamically configure itself to enable memory access notifications and instruction execution notifications. Whenever the kernel reads from the same user-space address twice or more, the tool will generate a file that describes the assembly instructions and the memory access addresses. As an optimization, the tool analyzes each system call number only the first time it is called; consecutive calls to the same system call will not be analyzed. As correctly pointed out by J00ru, though, this optimization can hinder the discovery of some potential bugs that are only reached under very specific conditions—and not during the first invocation of the affected system call. The code can be easily changed to address that concern.

After this work has completed, the KDF candidates are filtered, and only if the kernel read the memory twice or more and performed some operation based on the read, a violation will be reported.

We make the KDF `ztool` source code public. You may get it from under `<zsim-kit>/src/ztools` and open the Visual Studio solution. Make sure you build an x64 version of the tool. (Look in the Visual Studio configuration.) After that you can load the tool when you boot Win10. The tool generates candidates for KDF in separate log file in the current working directory. After completing the run of the simulation you may use the `kdf_analyzer`. The real KDF candidates will be located in the results directory.

```
cd src/ztools/kdf
python3.4 kdf_analyzer \
  -id <zsim-simics-workspace> \
  -if <kdf-violations-basename> \
  -rd <results-directory>
```

[42] `http://research.google.com/pubs/pub42189.html`

Approach

The simulation tool is dependent on SAE, and runs as a plugin to it. It works by loading the KDF tool included in this paper, booting the OS, and executing whatever test bench; the plugin will capture suspicious violations. After stopping the simulation, the KDF-analyzer scans the suspected violations recorded by the plugin and outputs the confirmed cases of double-fetches. Note that while these are real double-fetches, they are not necessarily security issues.

The algorithm of the plugin works as follows. It starts the analysis upon a SYSCALL instruction, monitoring kernel reads from user addresses. It reports a violation on two reads from the same user-space address in the same instruction window. It stops the KDF analysis after Instruction-Window is reached in the same syscall scope, or upon a ring transition.

Performance is guaranteed since each syscall is instrumented only once and the instrumentation is enabled only in the system call range, supported by the tool itself.

The analyzer—responsible for post-analysis of the potential violations—is a Python script that manages the data flow dependencies. It adds a reference upon a copy from a suspected address to a register/address. It removes the dependency reference upon a write to a previously referenced register/memory, similar to a taint analysis. It reports a violation only if two or more distinct kernel reads happen from the same user-mode address.

We looked into the system call range 0–5081. We dynamically executed 450 syscalls within that range—meaning that our test bed is far from completely covering the entire range. The number of suspected cases flagged by the plugin was 67 and the number of violations identified was 8.

Interesting Cases

Figure 10 shows some of the interesting cases. The Windows version was build number 10240, TH1 RTM candidate.

You will find traces extracted from our tests in directories `win10_after_boot/` and `win8_after_boot/`. As the names imply, they were collected after booting the respective Windows versions by just using the system: opening calc, notepad, and the recycle bin.

API	Exploitable?	Why?
`nt!CmOpenKey`	No	`UNICODE_STRING`, Read the Unicode structure and then read the actual string. Both are properly probed.
nt!CmCreateKey	No	`UNICODE_STRING`
`nt!SeCaptureObject-AttributeSecurity-DescriptorPresent` nt!SeCaptureSecurity-Qos		
`nt!ObpCaptureObject-CreateInformation`	No	Reading and then Checking if NULL. Getting length, probing, and then copying data
nt!EtwpTraceMessageVa	No	Reading, checking against NULL, probing and then copying data
`nt!NtCreateSymbolic-LinkObject`	No	`UNICODE_STRING`, May lead to Local DOS. No try-catch on user mode address reference, at least not at the top function; it may be deeper in the call stack
win32kbase!bPEB-CacheHandle	No	Working on addresses of PEB structure and not on pointers, try-catch will save in case of a malformed PEB

Figure 10. Interesting cases.

The filenames include the system call number and the address of the occurrence, to help identify the repeated cases, e.g., `kdf-syscall-4101.log.data_flow_0x7ffe0320`, `kdf-syscall-4104.log.data_flow_0x7ffe0320`, `kdf-syscall-4105.log.data_flow_0x7ffe0320`. For example, the address `0x7ffe0320` repeats in both Win10 and Win8 traces. We kept these repeated traces just to facilitate the analysis.

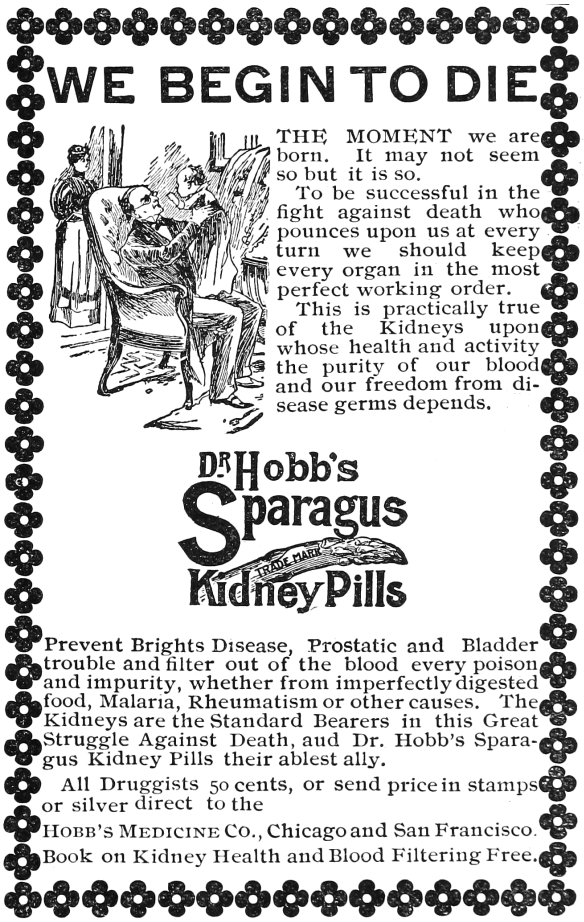
We also include the directories `results_-win10_boot/` and `result_win8_boot/`, which show the traces of interest *during* the boot process. These conditions are less likely to be exploitable, but some addresses in them repeat post-boot as well.

The format of trace files is quite straightforward, with comments inserted for events of interest:

```
--START ANALYZING KDF, ADDRESS: 0x2f7406f390
--  -> Defines the address of interest
```

Also included are the instructions performed during the analysis/trace:

```
180: 0xfffff803650acdd4
       mov rcx, qword ptr [rbx+0x10]
READ: VA = 0x2f7406f390, LA = 0x2f7406f390,
       PA1 = 0x79644390, SIZE = 0x8,
       DATA = 0x0002f746f3f8
```



The KDF detection happens on the following commentary on the trace:

```
--Data-flow dependency originated from
--line 180 is used: rcx
```

As you can see, the commentary includes the line at which the data-flow dependency was marked.

Our detection process begins when a `syscall` instruction is issued. While inside the call, we analyze kernel reads from the user address space, and report whenever two reads hit the same address; however, we remove references if a write is issued to the address. We stop the analysis once an instruction threshold is hit, or a ring transition happens.

Future Work

Leveraging our method and the toolset should make the following tasks possible.

First, it should be possible to find multiple writes to the same user-mode memory area in the scope of a single system service. This is effectively the opposite of the current concept of a violation. This may potentially find instances of accidentally disclosed sensitive data, such as uninitialized pool bytes, for a short while, before such data is replaced with the actual system call result.

Second, it should be possible to trace execution of code with `CPL=0` from user-mode virtual address space, a condition otherwise detected by the SMEP mechanism introduced in the latest Intel processors. Similarly, it should be possible to trace execution of code from non-executable memory regions that are not subject to Data-Execution-Prevention, such as non-paged pools in Windows.

Third, KDF should be studied on more operating systems.

Last but not least, other cases of cross-privilege mode double fetches should be investigated. There is far more work left to be done in tracing access to find these sorts of bugs.

Acknowledgments

We would like to thank Google researchers Mateusz Jurczyk and Gynvael Coldwind for releasing an awesome paper on the subject with enough details to reproduce their findings. (Mateusz was also kind enough to give feedback on this paper.) MSRC for helping to better define the threat model for Windows Kernel Vulnerabilities, and for their collaboration to triage the issues. We also thank Intel's Windows OS Team, specially Deepak Gupta and Volodymyr Pikhur, for their help in the analysis of the artifacts.