

A deeper UEFI dive into MoonBounce

binarily.io/posts/A_deeper_UEFI_dive_into_MoonBounce/index.html

After uncovering FinSpy several months ago, an APT threat targeting UEFI bootloaders, in the morning of January 20th 2022, Kaspersky Lab has released a new report on their latest discovery, a very interesting UEFI firmware threat dubbed MoonBounce.

Last year, the ESET researchers discovered ESpEcter another threat which also targets EFI bootloaders.

MoonBounce, FinSpy and ESpecter are examples of APT malware comprising components that target both UEFI and Legacy BIOS boot processes.

To kickstart our investigation, we leveraged VirusTotal Intelligence and discovered an archive exhibiting the detections mentioned in the Kaspersky Lab's MoonBounce report. Recently, a user uploaded the related samples, which surprised us greatly since we did not expect to find them that quickly by just querying using the detection name.

Figure1 Figure2

Let's dive into the dark waters of the MoonBounce firmware implant. In this blog we want to discuss some of the facts which weren't covered by the original report and are interesting to share.

Kaspersky Lab's original report emphasizes that the aforementioned malware consists of a number of known malware components and frameworks: Microcin, Mimikat SSP, xTalker, etc.

Binarily Research Team has analyzed the samples found in VirusTotal and discovered that the UEFI component (the first stage in the malware boot process) is quite old too. It was compiled with borrowed code from an unknown old malware project, most likely previously discovered in the wild on some Dell systems back in 2018 (according to GitHub repository information) and reconstructed in detail in the GitHub repository called "BootLoader".

Figure3

Binarily investigation focused on the firmware research of the UEFI component to provide additional technical information to the already detailed original report on MoonBounce threat.

Our deep dive uncovered similarities between the MoonBounce UEFI component and the binaries available in the BootLoader GitHub repository. Diving into the BootLoader code, visually the hooking routine `Search_OslArchTransferToKernel()` piqued our interest, as it is almost identical with the textual disassembly of the MoonBounce's `CORE_DXE` firmware dump component:

Figure4 *Code from CORE_DXE of MoonBounce firmware dump*

Figure5 *Disassembled 32-bit code of BootLoader binary*

During early runtime of the DXE phase, it is a known practice employed by malicious actors to modify or hook DXE Services to intercept a boot flow inside the firmware. From a forensics perspective, such modifications are very visible and can be detected using common integrity firmware monitoring approaches.

 Figure 6 *Disassembly code from CORE_DXE of MoonBounce firmware dump*

This code from MoonBounce component is pretty similar with code flow from [BootLoader.asm](#). These observations lead to the conclusion that MoonBounce's authors are the same or use similar code techniques or frameworks to embed their modification into the firmware and Windows kernel.


Additional details can be found in the original Kaspersky report "Technical details of MoonBounce's implementation" (p. 6, "Code that set up a hook in the ExAllocatePool function within ntoskrnl.exe").

Another technical detail we'd like to highlight here relates to the multiple infection delivery paths. The original report explains the usage of *CoreCreateEventInternal()* hook - as support for both UEFI boot and Legacy boot mode (assuming it's targeting deprecated Compatibility Support Module (CSM)).

Compatibility Support Module (CSM) - this module emulates the legacy BIOS in UEFI systems and was developed by Intel to ease the transition to UEFI world. It's pretty common on hardware released before 2020. For newer enterprise hardware is usually disabled by default.

The reason for using *CoreExitBootServices()* is to hook a call from Windows loader (Winload) and prepare the next steps for the MoonBounce boot interception process. What is the point of hooking *InternalAllocatePool()* at the beginning? The possible motivation for this could be to avoid storing the shellcode in a virtual address space along the Windows kernel, by placing it in a physical memory and then executing it directly from there. Such a technique can be used for fileless in memory code execution which would be orchestrated directly from the firmware. During incident response, this will cause complications from a forensic perspective.

Upon further analysis, we found that CORE_DXE contains the target firmware name as a string constant "E7846IMS.M30".

 Figure 7

Therefore, we became curious to determine the exact targeted platform. The first forensic artifact is the PE header timestamp for the CORE_DXE module (Fri Jul 18 03:29:55 2014).

 Figure 8

PE Tree snapshot of the CORE_DXE module

On the same day, Taiwanese company MSI released a firmware update for their hardware platform which is very similar to the firmware that has been infected by MoonBounce.

Searching on Google for the string constant "E7846IMS.M30", we found the following website:



[hxxps://www.mmnt.net/db/0/13/lupd01.jeu.msicom/ALL_BIOS_Update_Genie_update_20140831](https://www.mmnt.net/db/0/13/lupd01.jeu.msicom/ALL_BIOS_Update_Genie_update_20140831)

After obtaining the original firmware image, we were curious about the similarity between the original CORE_DXE component and the modified one in MoonBounce. According to our code analysis (binary diffing the MoonBounce UEFI component with the original firmware image from the vendor), there were no other modifications besides hooks for *InternalAllocatePool*, *CoreExitBootServices* and *CoreCreateEventInternal* services.

Figure10 BinDiff analysis with the original CORE_DXE {5AE3F37E-4EAE-41AE-8240-35465B5E81EB} driver from E7846IMS.M30 firmware

Using the Binary Cloud Platform, we found a DXE Core driver which is very similar to the MoonBounce UEFI firmware dump component. The similarity was confirmed by matching control flow graphs through further analysis with Google BinDiff tool. The code similarity search reveals exactly three modified/hooks routines that we discussed earlier.

There are other interesting questions we need to discuss, such as potential methods of delivering malware to the target system. How could such malware be written into SPI flash storage of the targeted system? It is worth noticing that **the analyzed MoonBounce UEFI component was built for a target hardware related to a MSI system from 2014**. This fact allows us to suggest two possible initial points of compromise:

- **Physical access-based implant delivery** to the target system - no Intel Boot Guard technology present or enabled thus there are no physical or hardware restrictions to get access to SPI flash storage of the system.
- **Software-based implant delivery** to the target system - keeping in mind the historical ignorance of many vendors on firmware security threats, we infer that no SPI protections were enabled on the system, hence SPI write-operations could be issued easily with no exploits required (access only to physical memory is required for working with PCH SPI controller MMIO).

Public information regarding firmware related implants is more present in the media lately, but it is just scratching the surface in terms of threat detection. Many security research papers have been published which provide examples of more advanced techniques for threat actors to persist in firmware. The supply chain complexity significantly increases the chances for the attackers to effectively reuse 1/N-day vulnerabilities ([“The Firmware Supply-Chain Security is broken: Can we fix it?”](#)).

We need to increase the industry awareness to firmware related threats and build more effective threat hunting programs with cross-industry collaboration between the vendors to mutually benefit customers and provide better detection rates.

[MoonBounce Firmware Implants Threat Intelligence Supply Chain Binary Platform](#)

[Back to overview](#)