Technical details of MoonBounce’s implementation
MoonBounce UEFI Implant

The rogue **CORE_DXE** component was patched by the attackers to incorporate an additional, malicious payload, which represents what we refer to as the MoonBounce implant.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>D94625B90DDB3F80F62BD96BD9858</td>
</tr>
<tr>
<td>SHA1</td>
<td>6FB3634F6B6C5A114121FE53279331FF821EE1E</td>
</tr>
<tr>
<td>SHA256</td>
<td>74B7581A1375B5A58A5143C02EB94D5ADCD49F284744CF2015E03DA036C2CF1A</td>
</tr>
<tr>
<td>Link time</td>
<td>Friday, 18.07.2014 03:29:55 UTC</td>
</tr>
<tr>
<td>File type</td>
<td>64-Bit EFI_BOOT_SERVICES_DRIVER</td>
</tr>
<tr>
<td>File size</td>
<td>1.698 MB</td>
</tr>
<tr>
<td>File name</td>
<td>CORE_DXE</td>
</tr>
</tbody>
</table>

This payload was appended to an unnamed section that follows the `.reloc` section and contains both shellcode and a malicious driver that are introduced in memory through a multistage infection chain during boot time. The driver, which is supposed to run in the context of the Windows kernel during its initialization phase, is in charge of deploying user-mode malware by injecting it into an `svchost.exe` process, once the operating system is up and running.

The aforementioned infection chain starts with a set of inline hooks at the beginning of several functions in the `EFI_BOOT_SERVICES` structure. This structure, which is a part of the `CORE_DXE` image itself, contains a table of pointers to routines (referred to as Boot Services) that are callable by subsequent components in the boot sequence, such as the DXE drivers, boot loader and OS loader. Hooking functions in this table facilitates the propagation of malicious code to other boot components during system startup.

The hooked functions in the underlying `EFI_BOOT_SERVICES` table, namely `AllocatePool`, `CreateEventEx` and `ExitBootServices`, have their first 5 bytes (typically referred to as the function’s prologue) replaced with a `call` instruction to a single malicious hook. The hook’s code checks the first bytes after the `call` instruction and, based on predefined byte patterns, can deduce the source function triggering its execution. Based on this trait, it can dispatch the flow to successive handlers corresponding to each of the hooked functions.

![Example of a hook installed at the beginning of the AllocatePool boot services](image_url)
The steps taken in the infection chain, as a result of deploying the above hooks, are depicted in the following diagram with accompanying explanations:

1. The first Boot Services function invoked in `CORE_DXE` after the `EFI_BOOT_SERVICES` structure initialization is `AllocatePool`, which diverts execution to its corresponding handler within the hook function.
2. **AllocatePool**’s handler:

- Restores the original prologue bytes that were previously modified by the attackers to ‘48 89 5C 24 08’ (corresponding to the instruction ‘mov [rsp+8], rbx’) and saves the state of the registers rcx, rdx, r8, r9, rsi and rdi (some of which are typically used to pass function arguments).

- Calls **AllocatePool** (which is now unhooked) with pre-configured parameters that are intended to allocate a buffer in memory and assign it with shellcode used at later stages of the infection.

- Restores the saved state of the registers and passes control back to the beginning of **AllocatePool**, which is now executed with the original arguments with which it was invoked in the first place.

```
AllocatePool_handler: ; Restore original stolen bytes
    mov    byte ptr [rax], 48H ; rax -> beginning of function
    mov    dword ptr [rax+4], 00453B08H
    pushq  rbx
    pushq  rdx
    pushq  r8
    pushq  r9
    pushq  rdi
    sub    rsp, 40h
    call   $05
    pop    rbx
    lea    r6, [rbx+0838h]; Buffer -> 10015776b
    ; replace the bytes in the marker
    ; 0x11223456567777B8 within the
    ; malicious legacy boot callback
    ; registered by CreateEventEx
    ; with the address of the allocated buffer
    mov    edx, 40000h ; Size
    mov    ecx, AllocateAnyPages ; PoolType
    call   $c
    test   rax, rax
    jnz    short func_end
    mov    rdi, [rbx+0838h]; 10015776b allocated buffer set as dest
    mov    qword ptr ds:map.driver_mapping_shellcode_to_memory, rdi ;
    ; replace marker 0x1122345656777B8
    ; with allocated address at 10015776b
    lea    rsi, [rbx+0830h]; 10015776b attacker code set as dst
    mov    ecx, 40000h ; size of attacker code to copy
    cld
    rep movsb ; copies driver mapping shellcode to allocated buffer
```

AllocatePool’s hook logic

3. The next Boot Services function invoked in **CORE_DXE** is **CreateEventEx** that diverts execution to its handler within the hook function.

4. **CreateEventEx**’s handler:

- Restores the original prologue bytes that were previously modified by the attackers to ‘48 8B C4 48 89’ (corresponding to the instructions ‘mov rax’ and ‘mov [rax+8], rbx’) and saves the state of the registers rcx, rdx, r8, r9, rsi and rdi (some of which are typically used to pass function arguments).

- Calls the now unhooked **CreateEventEx** with predefined arguments to register a callback for an event that represents a legacy boot (designated with the GUID {2A571201-4966-47F6-8B86-F31E41F32F10}, i.e. gEfiEventLegacyBootGuid). In that case, the callback is responsible for passing control to the shellcode set up in **AllocatePool**’s hook.

- Restores the saved state of the registers and passes control back to the beginning of **CreateEventEx**, which is now executed with the original arguments with which it was invoked in the first place.
The boot sequence continues, passing control to the Windows OS loader. At one point, this loader calls the hooked function ExitBootServices, which is supposed to hand control over to the OS loader and eliminate the dependency on the firmware-based Boot Services functions.

Execution is diverted to the ExitBootServices handler within the hook previously set up in CORE_DXE. The hooking of ExitBootServices was described as a technique in the Vault7 leaks.

The ExitBootServices handler conducts the following actions:

- Restores the original prologue bytes that were previously modified by the attackers to `48 89 5C 24 08` (corresponding to the instruction `mov [rsp+8],rbx`).
- Takes the previous return address from the stack (the first address after the call to ExitBootServices) and searches for the byte pattern `41 55 48 CB` (corresponding to the instructions `push r13` and `retfq`) within a region of 0x158878 bytes after it. These bytes designate the end of the function OslArchTransferToKernel in the Windows OS loader image (typically named winload.efi or osloader.exe and residing in the ESP partition on disk).
- Copies 0x229 bytes of shellcode to address 0x98000 in memory.
- Replaces the bytes starting with `48 CB ` (retfq) at the end of the OslArchTransferToKernel function to E9 <offset_to_0x98000_shellcode>, which is essentially a jump to the shellcode that was just copied to 0x9800.
- Restores the saved state of the registers and passes control back to the beginning of the now unhooked ExitBootServices, which is executed as it was originally intended in flow of the Windows OS loader.
ExitBootServices’ hook logic

8. In the further execution flow of the Windows loader, it invokes the aforementioned OslArchTransferToKernel function, which passes control from the OS loader to the Windows kernel. As mentioned in step 7, the last bytes of the function are replaced, diverting execution to a formerly allocated shellcode that effectively serves as a hook for OslArchTransferToKernel.

9. The OslArchTransferToKernel hook:
   • Locates the image base of ntoskrnl.exe in memory.
   • Resolves function addresses exported by ntoskrnl.exe, through which it uses a name-hashing algorithm with the following equivalent logic:

   ```python
   def fn_name_hash(name):
       name_hash = 0
       index = 1
       for ch in name:
           index += ord(ch)
           name_hash += index
       return (name_hash << 16) | index
   ```

   The compared function name hashes and their corresponding resolved functions are:
   • 0x42790710 – ExRegisterCallback
   • 0x2802057D – ExAllocatePool
   • 0x1C88047D – MmMapIoSpace

   • Changes the Characteristics field in each section header of ntoskrnl.exe’s image in memory:
     • The IMAGE_SCN_MEM_DISCARDABLE bit gets disabled (the section cannot be discarded);
     • The IMAGE_SCN_MEM_EXECUTE, IMAGE_SCN_MEM_WRITE and IMAGE_SCN_MEM_NOT_PAGED bits get enabled.

   • Copies 0xCC bytes of another shellcode to the virtual address of the ntoskrnl.exe’s relocation directory.

   • Sets up an inline hook at the beginning of ExAllocatePool with a call instruction to the copied shellcode by placing the bytes E8 <offset_to_shellcode> at the beginning of the function and saving the original bytes in a designated buffer.
Code that set up a hook in the ExAllocatePool function within ntoskrnl.exe

10. Control is passed to the Windows kernel, which then invokes the hooked ExAllocatePool and in turn diverts execution to its hook, which was set up in the previous stage.

11. The ExAllocatePool hook:
   • Verifies if the hook was previously executed by checking a predefined global flag. If not, the flag is set to designate that the hook was run so that any subsequent execution of ExAllocatePool will invoke the original function flow.
   • Calls MmMapIoSpace to map the driver mapping shellcode, which was set up during step 2, to the virtual address space of the Windows kernel.
   • Jumps to the address of the now mapped shellcode, passing it the following arguments on stack:
     • Pointer to a buffer with the saved ExAllocatePool prologue bytes
     • Base address of ntoskrnl.exe
     • Pointer to ExAllocatePool

ExAllocatePool hook logic
At this point, the main shellcode set up in the early stages of the infection chain and mapped to the virtual memory address space of the kernel in the previous step gets executed. The purpose of this shellcode is to map a raw PE image of a malicious driver (that is, appended at the end of the shellcode bytes) in memory and pass control to its entry point. To achieve this goal, the shellcode:

- Checks if the buffer with the saved prologue bytes of `ExAllocatePool` passed to it in the first argument is equal to `0x6F4EB841` (the original bytes in `ExAllocatePool` that were modified when it was hooked), in which case it resets the WP bit in the CR0 register in order to be able to write to read-only pages in memory and restores these original bytes to the beginning of `ExAllocatePool` (which has its address provided as the third argument of the shellcode), effectively unhooking it. After that, the shellcode restores the previous state of CR0 before that it was modified.

- Resolves exported functions from `ntoskrnl.exe` that are essential for the subsequent PE mapping. The function address resolution code makes use of yet another name-hashing algorithm, which is outlined in the equivalent logic below:

```python
def ror13(x):
    return 0xFFFFFFFF & ((x >> 13) | (x << 32 - 13))

def fn_name_hash_ror13(f_name):
    f_hash = 0
    for i in f_name:
        f_hash = ror13(f_hash)
        f_hash += ord(i)
    f_hash = ror13(f_hash)
    return f_hash
```

The functions resolved in this phase and their corresponding name hashes are the following:

- `0x0311B83F` – `ExAllocatePool`
- `0x41EBE619` – `RtlInitAnsiString`
- `0x1C4F5B64` – `RtlAnsiStringToUnicodeString`
- `0x0ADC68C7` – `MmGetSystemRoutineAddress`

- Maps the malicious driver image to the kernel memory with the following common PE-loading steps:
  - Allocates space for the image with the now unhooked `ExAllocatePool` function
  - Copies headers and sections to their corresponding virtual addresses in memory
  - Applies relocations
  - Resolves imports by getting each name in the import table, initializing its string with `RtlAnsiString` and `RtlAnsiStringToUnicodeString`, and passing the result as an argument to `MmGetSystemRoutineAddress`, following which the argument string is freed with `RtlFreeUnicodeString`.

- Finally, control is passed to the entry point of the malicious driver.

For clarity, steps 13-16, which are taken by the malicious driver and the user-mode malware it deploys, are explained in detail in the following sections.

---

**MD5**: 2228E682B2686DBE0330835B8A6F2BF (x86)  
934D06720F4CB74069A870D382AC5045 (x64)

**SHA1**: 22A4BD6BF580C3A2025B1A9F18EF1677FECA360 (x86)  
3D2E6FC3B6FDFDB44966ADB4F13679E4091D851 (x64)

**SHA256**: 707B86B0096E5742B9C6D801C12B9803F33FC518CB6BFF53B4FA5A9E72E106 (x86)  
F17C1F644CE538D07083C6D6DEB52BFD836D0376238CC3F4B3CA2B3CA6CA7D (x64)

**Link time**: Tuesday, 18.12.2018 03:48:33 UTC (x86)  
Tuesday, 18.12.2018 03:48:24 UTC (x64)

**File type**: PE32 executable (native) Intel 80386, for MS Windows  
PE32+ executable (native) x86-64, for MS Windows

**File size**: 34.63 KB (x86)  
37 KB (x64)

**File name**: None
The purpose of the malicious driver is to inject user-mode malware into a Windows service of the network services group, thereby allowing it to have access to the internet. This is achieved by first having the driver register a callback using the `PsSetLoadImageNotifyRoutine` API, which is invoked when the Windows loader maps a PE image to memory (as outlined in step 13 of figure 1). This callback in turn verifies that the inspected image is `kernel32.dll` and the underlying owning process is executed with the command line: `‘SVCHOST.EXE -K NETSVCS’` or `‘SVCHOST.EXE -K NETSVCS -P’`.

```c
if (argFullImageName && (argImageInfo->Properties & 0x100) == 0)
{
    process_peb = 0;
    if (PsLookupProcessByProcessId(argProcessId, &argProcessId) == 0)
    {
        process_peb = PsGetProcessPeb(argProcessId);
        ObDereferenceObject(argProcessId);
    }
    RtlInitUnicodeString(&us_SYSTEM32_kernel32_dll, L"_SYSTEM32\kernel32.dll");
    if (FsRtlIsNameInExpression(&us_SYSTEM32_kernel32_dll, argFullImageName, 1, 0))
    {
        if (process_peb)
        {
            if (process_peb->ProcessParameters)
            {
                RtlInitUnicodeString(&us_svchost_commandline_netsvc_k, svchost_commandline_netsvc_k);
                RtlInitUnicodeString(&us_svchost_commandline_netsvc_p, svchost_commandline_netsvc_p);
                if (FsRtlIsNameInExpression(&us_svchost_commandline_netsvc_p, &process_peb->ProcessParameters->Commandline, 1, 0)
                    || FsRtlIsNameInExpression(&us_svchost_commandline_netsvc_k, &process_peb->ProcessParameters->Commandline, 1, 0))
            {
            }
        }
    }
}
```

**Conditions to locate the target svchost.exe process for injection by MoonBounce’s driver**

If the above conditions are met, the driver continues to inject an embedded PE image, corresponding to a user-mode malware stager, to the matching `svchost.exe` process (as outlined in step 14 of figure 1). The injection leverages the Windows APC (Asynchronous Procedure Call) mechanism through the following actions:

- The driver enqueues a kernel mode APC routine, which will run in kernel mode with `APC_LEVEL IRQL`;
- The kernel APC routine initializes the following data structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>A table with pointers to various fields in the current structure and Windows API functions that are used by the PE mapping shellcode</td>
</tr>
<tr>
<td>0x28</td>
<td>PE mapping shellcode used to load the raw user mode stager PE to memory</td>
</tr>
<tr>
<td>0x800</td>
<td>Buffer with the drop zone URL carrying the payload to be downloaded by the stager</td>
</tr>
<tr>
<td>0xA00</td>
<td>Padding</td>
</tr>
<tr>
<td>0x1000</td>
<td>Buffer with the raw image of the deployed user mode stager</td>
</tr>
</tbody>
</table>

The first field, which we will refer to as the mapping shellcode argument, shows the following layout:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Pointer to the buffer with the user mode stager</td>
</tr>
<tr>
<td>0x8</td>
<td>Pointer to the buffer with the C2 URL containing the payload to be downloaded by the stager</td>
</tr>
<tr>
<td>0x10</td>
<td>Pointer to <code>VirtualAlloc</code></td>
</tr>
<tr>
<td>0x18</td>
<td>Pointer to <code>LoadLibraryA</code></td>
</tr>
<tr>
<td>0x20</td>
<td>Pointer to <code>GetProcAddress</code></td>
</tr>
</tbody>
</table>
The kernel routine initializes a **WORK_QUEUE_ITEM** structure with a pointer to a worker routine and an argument structure with the following layout:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Pointer to PE mapping shellcode</td>
</tr>
<tr>
<td>0x8</td>
<td>Pointer to the PE mapping shellcode argument described above</td>
</tr>
<tr>
<td>0x10</td>
<td>Pointer to the KTHREAD object corresponding to the current thread executing in the context of the injected process</td>
</tr>
<tr>
<td>0x18</td>
<td>Pointer to a notification event</td>
</tr>
</tbody>
</table>

It then calls the `ExQueueWorkItem` with the above structure in order to insert the worker routine to a system wide queue.

Initialization of a WORK_QUEUE_ITEM structure used to schedule the execution of a worker routine in kernel space

- The Windows kernel has a designated system thread that picks up the previously enqueued task and executes its corresponding routine, passing it a pointer to the argument structure described above. In this case, the executed routine queues the PE-mapping shellcode with its own argument structure to the APC queue of the current thread running in the context of the injected **svchost.exe** process.

```c
user_mode_apc = ExAllocatePool(NonPagedPool, 0x30u);
c_pool_buffer = user_mode_apc;
if ( user_mode_apc )
{
    KeInitializeApc(
        user_mode_apc,
        work_item_param->p_current_kthread, 0, 
        apc_rundownRoutine, 0, 
        work_item_param->p_pe_mapping_shellcode, UserMode, 
        work_item_param->p_pe_mapping_shellcode_argument_structure; 
    KeInsertQueueApc(c_pool_buffer, 0, 0, 0);
    return KeSetEvent(&work_item_param->notification_event, 0x1, 0, 0);
}
```
The user-mode malware stager, which is injected to an `svchost.exe` process by the malicious driver, is a DLL packed with a common software tool called MPRESS. It operates in a similar fashion to UPX, whereby the original sections of the PE are compressed into a new section called `.MPRESS1` and the code for unpacking is appended into another generated section named `.MPRESS2`. It gets executed during runtime in order to decompress the data and pass control to the original entry point within.

After unpacking, the malware executes a basic staging component that reaches out to a C2 URL and obtains a PE image. The DLL receives an argument from the driver in the `lpReserved` parameter of the `DllEntryPoint`, which should contain a pointer to a C2 URL. The same argument can contain additional optional data elements that can be used in a number of ways throughout execution. These are laid out in a structure of the following form:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>C&amp;C URL (may also contain a scheduling related argument)</td>
</tr>
<tr>
<td>0x11c</td>
<td>User-Agent</td>
</tr>
<tr>
<td>0x180</td>
<td>Proxy address</td>
</tr>
<tr>
<td>0x1c0</td>
<td>Proxy username</td>
</tr>
<tr>
<td>0x1e0</td>
<td>Proxy password</td>
</tr>
</tbody>
</table>

To receive a further payload to run, the malware:

- Runs a system time-dependent scheduling algorithm that postpones execution until reaching a predefined deadline value, at which point the downloading logic is initiated. This value ought to be provided as part of the aforementioned DLL argument; however, we did not observe it being passed by the driver we analysed.
- Sets up an optional User-Agent or uses the default string “IE” instead. Once again, the driver in our case did not pass any particular argument to use in this field; therefore, it is expected to be the default value.
- Registers a callback function with the `InternetSetStatusCallback` API, which detects whether the system makes use of a proxy, in which case the malware can use the proxy configuration provided in the DLL argument to issue a request.
- Sends a GET request to the C2 URL, expecting to receive a raw PE image as a response.
- Maps the retrieved image to the current memory address space and invokes its entry point.