PRESS ROOT TO CONTINUE:
DETECTING OSX AND WINDOWS
BOOTKITS WITH RDFU
Agenda

• Our motivation
• Who are we
• Introduction to...
  • Unified extensible framework interface (UEFI)
  • Previous UEFI bootkit research
• Rootkit detection framework “RDFU”
  • Framework design
  • VMWare implementation demo
• MacOS X bootkit demo
Our motivation

- UEFI is very popular
  - Windows + Android + MacOS + ...
- Full-stack: UEFI is a mini-OS
  - Memory and file manipulation, full network stack
  - Graphics APIs, device management
  - Remote boot
- Attacker’s paradise
  - No tools for analysis, low visibility, even no AV, ...
- Some good news though
  - UEFI SecureBoot (Surface RT, Android)
Who are we

- ReversingLabs
  - Founded by Mario Vuksan and Tomislav Pericin in 2009
- Focusing on
  - Deep binary analysis of PE/ELF/Mach-O/DEX and firmware
  - System reputation and anomaly detections
- Black Hat presentations and open source projects
  - TitanEngine: PE reconstruction library (2009)
  - NyxEngine: Archive format stego detection tool (2010)
  - TitanMist: Unpacking (2010)
  - Unofficial guide to PE malformations (2011)
  - FDF: disinfection framework (2012)
Thanks

• John Heasman, Black Hat 2007
• Snare, Assurance, Black Hat 2012
• Dan Griffin, Defcon 2012
• Sebastien Kaczmarek, HITB Amsterdam 2013
• DARPA CFT
UEFI

unified extensible firmware interface
UESI?

- **UEFI**: Unified extensible firmware interface
  - Originally developed by Intel, “Intel boot initiative”
  - Community effort to modernize PC booting process
  - Currently ships as a boot option alongside legacy BIOS
  - Aims to be the only booting interface in the future
  - Used in all Intel Macs and other PC motherboards
  - Managed by Unified Extensible Firmware Interface (UEFI) Forum
Booting with EFI

PROTECTED MODE

KERNEL

USERLAND

UEFI

UEFI bootloader
\EFI\Microsoft\Boot\bootmgfw.efi

winload.efi

NTOSKRNL.EXE

HAL

SMS

WIN32

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoDI 5230.29, January 8, 2009.
UEFI Conceptual Overview

- Operating system
- EFI Operating system loader
- Other interfaces (ACPI, SMBIOS...)
- EFI Boot services
- EFI runtime services
- Platform hardware
- EFI partition

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoD 5230.29, January 8, 2009.
EFI boot sequence

EFI Driver → EFI Application → EFI Boot code → OS Loader

Platform init
- Standard firmware initialization

EFI image load
- Drivers and applications loaded

EFI OS loader
- Boot from ordered EFiOS list

Boot service terminates
- Operations handed off to OS

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoDI 5230.29, January 8, 2009.
UEFI images

• UEFI images:
  • Typically PE32/PE32+ (basic format feature subset)
  • Standard also predicts that other formats can be defined by anyone implementing the specification, e.g. TE defined by Intel and used by Apple
• **UEFI drivers:**
  - Boot service driver
    - Terminated once ExitBootServices() is called
  - Runtime service driver

• **UEFI applications:**
  - EFI application
    - Normal EFI applications must execute in pre-boot environment
  - OS loader application
    - Special UEFI application that can take control of the system by calling ExitBootServices()
• **UEFI boot services:**
  • Consists of functions that are available before ExitBootServices() is called
  • These functions can be categorized as “global”, “handle based” and dynamically created protocols
    • **Global** – System services available on all platforms
      • Event, Timer and Task Priority services
      • Memory allocation services
      • Protocol handler services
      • Image services
      • Miscellaneous services
    • **Handle based** – Specific functionally not available everywhere
• **UEFI runtime services:**
  • Consists of functions that are available before and after ExitBootServices() is called
  • These functions can be categorized as “global”, “handle based” and dynamically created protocols
    • **Global** – System services available on all platforms
      • Runtime rules and restrictions
      • Variable services
      • Time services
      • Virtual memory services
      • Miscellaneous services
    • **Handle based** – Specific functionally not available everywhere
• EFI development kit
  • TianoCore – Intel’s reference implementation
  • Enables writing EFI applications and drivers in C
    • Has its own stdlibC implementation that covers a part of the standard library
    • Has a set of packages for shell, crypto, emulation and more
    • Has a set of applications built with stdlibC implementation
      • For example: Python 2.7
  • Has a build system which uses popular compilers (VS, GCC and XCode)
  • Supported CPUs: IA64, x86-64 and ARM
/**
 * Print a welcoming message.
 * Establishes the main structure of the application.
 *
 * @retval 0       The application exited normally.
 * @retval Other   An error occurred.
 */

INTN
EFIAPI
ShellAppMain ( 
    IN UINTN Argc,
    IN CHAR16 **Argv
)
{
    Print(L"Hello there fellow Programmer.
");
    Print(L"Welcome to the world of EDK II.
");

    return(0);
}
UEFI - HelloWorld.c

/ ***
 Print a welcoming message.

 Establishes the main structure of the application.

 @retval 0 The application exited normally.
 @retval Other An error occurred.
 ***/

 INTN
 EFIAPI
 UEFIAppMain (  
 IN EFI_HANDLE ImageHandle,  
 IN EFI_SYSTEM_TABLE *SystemTable / ** Boot and Runtime services **/
 }

 Print(L"Hello there fellow Programmer.\n");

 return(0);
}

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoD 5230.29, January 8, 2009.
Bootkits

attacking unified extensible firmware interface
Previous work - '07

- Hacking extensible firmware interface
  - John Heasman, NGS Consulting
  - Presented at BlackHat 2007, USA

- Research
  - Modifying NVRAM variables
  - Code injection attacks
  - Shimming boot services
  - Abusing system management mode
Previous work - '12

- Hacking extensible firmware interface
  - Snare, Assurance
  - Presented at BlackHat 2012, USA
- Research
  - Patching MacOS X kernel
  - Evil maid attack
Previous work - '13

• Dreamboot
  • Windows 8 x64 bootkit
  • Sébastien Kaczmarek, QuarksLab
  • Presented at HackInTheBox 2013, Amsterdam

• Modus operandi
  • Bypasses kernel protections (NX and Patch guard)
  • Bypasses local authentication
  • Elevates process privileges
RDFU

rootkit detection framework for uefi
What is RDFU?

- Set of EFI applications and drivers that enable:
  - Listing all EFI drivers loaded into memory
  - Probing entire memory range, scanning for executable
  - Monitoring newly loaded drivers until operating system starts
  - Listing and scanning EFI BOOT SERVICES and EFI RUNTIME SERVICES for modified function pointers
  - Continually monitoring EFI BOOT SERVICES and EFI RUNTIME SERVICES while operating system is being loaded
  - Displaying memory map and dumping all suitable regions
  - Listing and monitoring EVENT callbacks that can be used by rootkits/malware
  - Working in a standalone mode without the EFI shell
What does RDFU support?

- Supported UEFI implementations:
  - UEFI 2.x specification for 32-bit and 64-bit Implementations
  - UEFI 1.x specification
  - MacOS UEFI implementation
  - VirtualBox
  - VMWare

- Not supported UEFI implementations:
  - UEFI ARM implementation (only on Surface RT, has secure boot enabled)
How does RDFU work?

- DXE driver loaded via UEFI shell
- DXE driver loaded from USB thumb drive
- Scanner application run from UEFI shell
- Logging and dumping is done to the mounted hard drive or the USB thumb drive
Continue
Boot Manager
Boot Maintenance Manager

Select from the available operating systems or devices.
<table>
<thead>
<tr>
<th>Boot Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Bootable Operating Systems and Devices</strong></td>
</tr>
<tr>
<td>Windows Boot Manager</td>
</tr>
<tr>
<td>EFI VMware Virtual SCSI Hard Drive (0.0)</td>
</tr>
<tr>
<td>EFI VMware Virtual IDE CDROM Drive (IDE 1:0)</td>
</tr>
<tr>
<td>EFI Network</td>
</tr>
<tr>
<td>EFI Internal Shell (Unsupported option)</td>
</tr>
<tr>
<td>EFI VMware Virtual SCSI Hard Drive (1.0)</td>
</tr>
<tr>
<td>EFI VMware Virtual SCSI Hard Drive (2.0)</td>
</tr>
<tr>
<td>↑ and ↓ to change option, ENTER to select an option, ESC to exit</td>
</tr>
</tbody>
</table>

**Device Path:**
- MemoryMapped (0x80, 0xDEF0000, 0x8F33BFF0) / Fixed (CS7 AD687-0515-4008-9D21-5516 52854E27)

**VMWARE**
Press ESC in 4 seconds to skip startup.msh, any other key to continue.

Shell> _
Menu:

[0] - List all handles
[1] - List all images
[2] - Dump all images to disk
[3] - Check BootServices/RuntimeServices/SystemTable pointers in images
[4] - Install image sniffer (SniffImage, requires residency)
[5] - List all events
[6] - Install event scanner (requires residency)
[7] - Scan memory for PE images (brute-force) and dump them to disk
[8] - Display memory map
[9] - Display and dump memory map
[10] - Display and dump memory map (skip Reserved and MemoryMappedID mem)
[12] - Install EFI services scanner (requires residency)
[13] - Display IDT
[14] - Display GDT
[15] - Display Context
[16] - Dump firmware from ROM
[17] - Install all resident scanners
[18] - Quit
DEMO

rootkit detection framework for uefi
MAC OS 10.7.x

bootkit

first MacOS X bootkit example
Bootkit goals

- Create hidden folders
- Hiding (with un-hiding) processes
- Execute shell with root privileges
- Retrieve FileVault password
Running the MacOS bootkit

Mac OS X 10.7.x - Lion
Running the MacOS bootkit

Boot the OS from an USB thumb drive
VMWare / MacOS

bootkit

• MacOS can also be run in VMWare if you don’t have a MacBook Pro handy

• Running MacOS under VMWare requires an “unofficial patch” — wink wink nudge nudge

• Once patched we need to change the VMX file
  • firmware = "efi"

• After that MacOS can be installed with EFI 1.10
Bootkit workflow

UEFI
EFI\boot\bootx64.efi
BS->CreateEvent
EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE
SystemTable->ConIn->ReadKeyStroke
BS->OpenProtocol
LoadedImage->Unload
Load Mac OS X

Register event callback
HOOK!
Fail safe
Bootkit workflow

1. Load Mac OS X
2. Enumerate drives
   \System\Library\CoreServices\boot.elf
3. User choice on multiple OS X instances found
4. BS->LoadImage
5. BS->StartImage

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoD 5230.29, January 8, 2009.
Bootkit workflow

**SIGNAL**

- SetVirtualAddressMap()

**EVENT**

- EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE

**LOCATION**

- Locate syscall table

**HOOK!**

- Hook syscalls: setuid, getdirent, getdirentattr & sysctl

Distribution Statement “A” (Approved for Public Release, Distribution Unlimited)

Disclaimer: “The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.” This is in accordance with DoDI 5230.29, January 8, 2009.
Getting ROOT

/***
   executes shell with root rights
***/
#define HIDDEN_UID 1911

int main( void )
{
    setuid(HIDDEN_UID);
    system("/bin/sh");
}
Hiding processes

/***
 * sends the pid to the rootkit that should be hidden
 ***/

int main(int argc, char *argv[])
{
    pid_t pid = atoi(argv[1]);
    printf("Adding pid %d (%08x) hide list\n", pid, pid);

    int name[] = { CTL_ADD_PID, pid, KERN_PROC_ALL, 0 };

    err = sysctl((int *)name, (sizeof(name) / sizeof(*name)) - 1, NULL,
    &length, NULL, 0);

    printf("All done, sysctl returned 0x%08x\n", err);
    return EXIT_SUCCESS;
}
MacOS X bootkit
QA

Thanks!

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)
Disclaimer: "The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government." This is in accordance with DoDI 5230.29, January 8, 2009.