Behind the Scenes of iOS and Mac Security

Ivan Krstić
Head of Security Engineering and Architecture, Apple
Mac secure boot
iOS code integrity protection
Find My
Mac secure boot

iOS code integrity protection

Find My
Gatekeeper

User Privacy Protection
## Gatekeeper

**macOS Catalina**

<table>
<thead>
<tr>
<th></th>
<th>First use, quarantined</th>
<th>First use, quarantined</th>
<th>Non-quarantined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious content scan</td>
<td>No known malicious content</td>
<td>No known malicious content</td>
<td>No known malicious content</td>
</tr>
<tr>
<td>Signature check</td>
<td>No tampering</td>
<td>No tampering</td>
<td>−</td>
</tr>
<tr>
<td>Local policy check</td>
<td>All new software requires notarization</td>
<td>All new software requires notarization</td>
<td>−</td>
</tr>
<tr>
<td>First launch prompt</td>
<td>User must approve</td>
<td>Users must approve software in bundles</td>
<td>−</td>
</tr>
</tbody>
</table>
User Data Protections

Data that requires user consent to access

Contacts

Calendars

Reminders

Photos
User Data Protections

Data that requires user consent to access

Contacts
Calendars
Reminders
Photos

"Watch Grass Grow" would like to access your photos.

Your photo library is used to view & save photos of important grass-growing milestones.

Don’t Allow  OK
User Data Protections
Data that requires user consent to access

<table>
<thead>
<tr>
<th>Contacts</th>
<th>Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calendars</td>
<td>Documents</td>
</tr>
<tr>
<td>Reminders</td>
<td>Downloads</td>
</tr>
<tr>
<td>Photos</td>
<td>iCloud Drive</td>
</tr>
<tr>
<td></td>
<td>Third-party cloud storage</td>
</tr>
<tr>
<td></td>
<td>Removable volumes</td>
</tr>
<tr>
<td></td>
<td>Network volumes</td>
</tr>
</tbody>
</table>
What about secure boot?
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Apple Requirement</th>
<th>UEFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature verification of complete boot chain</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>System Software Authorization (server-side downgrade protection)</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>Authorization “personalized” for the requesting device (not portable)</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>User authentication required to downgrade secure boot policy</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>Secure boot policy protected against physical tamper</td>
<td>✅</td>
<td>❌</td>
</tr>
<tr>
<td>System can always be restored to known-good state</td>
<td>✅</td>
<td>❌</td>
</tr>
</tbody>
</table>
Mac Secure Boot
Mac Secure Boot

T2

x86
Mac Secure Boot

T2

x86
Mac Secure Boot

T2 ROM
iBoot
bridgeOS kernel
UEFI firmware

x86
Mac Secure Boot

T2
- T2 ROM
- iBoot
- bridgeOS kernel

x86
- UEFI firmware
- macOS booter
- macOS kernel
Two Critical Challenges

Thunderbolt and PCIe Direct Memory Access (DMA)
• Accessories can read/write host memory without the involvement of the CPU

PCIe Option ROMs (OROMs)
• Device-specific drivers for the early boot environment
Refresher — OS Page Tables

- **x86 CPU**
  - 64 bit protected mode
  - "Read 4 bytes from address 0x0000030000000000"

- **MMU hardware**
  - Virtual memory enabled

- **Page table hardware**

- **x86 RAM**
  - Verified UEFI firmware
  - Data 0x1570000
Refresher — OS Page Tables

x86 CPU
64 bit protected mode

“Read 4 bytes from address 0x00000f000300000000”

MMU hardware
Virtual memory enabled

Page table hardware
Consult page tables

Verified UEFI firmware

Data
0x1570000

x86 RAM
Refresher — OS Page Tables

x86 CPU
64 bit protected mode

“Read 4 bytes from address 0x0000030000000000”

MMU hardware
Virtual memory enabled

Page table hardware
Consult page tables

x86 RAM

Verified UEFI firmware

Data 0x1570000

Page tables:
virtual address 0x0000030000000000
is actually in RAM at physical address 0x1570000
Refresher — OS Page Tables

- x86 CPU
  - 64 bit protected mode

  “Read 4 bytes from address 0x0000030000000000”

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- x86 RAM
  - Verified UEFI firmware
  - Data 0x1570000
  - Page tables:
    - virtual address 0x0000030000000000 is actually in RAM at physical address 0x1570000
Refresher — OS Page Tables

x86 CPU
64 bit protected mode

“Read 4 bytes from address 0x0000030000000000”

MMU hardware
Virtual memory enabled

Page table hardware
Fetch from 0x1570000 in RAM instead of 0x0000030000000000

Page tables: virtual address 0x0000030000000000 is actually in RAM at physical address 0x1570000

x86 RAM
Verified UEFI firmware

Data
0x1570000
Unrestricted Direct Memory Access

- x86 CPU
  - 64 bit protected mode

- Network Interface Card (NIC)
  - Network packet

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- x86 RAM
  - macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer

- Fetch instructions
Unrestricted Direct Memory Access

- x86 CPU
  - 64 bit protected mode
- MMU hardware
  - Virtual memory enabled
  - Page table hardware
- NIC kernel extension
- macOS kernel
- x86 RAM
- macOS kernel
- Kernel heap
- Packet buffer

Network Interface Card (NIC)

Network packet

Fetch instructions
Unrestricted Direct Memory Access

x86 CPU
64 bit protected mode

Fetch instructions

MMU hardware
Virtual memory enabled
Page table hardware

x86 RAM
macOS kernel
NIC kernel extension
Kernel heap
Network packet

Network Interface Card (NIC)
Unrestricted Direct Memory Access

- x86 CPU
  - 64 bit protected mode

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- x86 RAM
  - macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer

- Network Interface Card (NIC)
  - Network packet
Unrestricted Direct Memory Access

x86 CPU
64 bit protected mode

Fetch instructions

MMU hardware
Virtual memory enabled
Page table hardware

x86 RAM

macOS kernel
NIC kernel extension

Network Interface Card (NIC)

Network packet

Packet buffer
Kernel heap
Unrestricted Direct Memory Access

- x86 CPU
  - 64 bit protected mode

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- x86 RAM
  - NIC kernel extension
  - Network packet
  - Kernel heap
  - Packet buffer

- Network Interface Card (NIC)

Fetch instructions
VT-d

Intel Virtualization Technology for Directed I/O (VT-d) is a mechanism by which the host can place restrictions on DMA from peripherals.

VT-d creates an I/O Memory Management Unit (IOMMU) to manage DMA.

We’ve used VT-d to protect the kernel since OS X Mountain Lion in 2012.
Direct Memory Access with VT-d

- x86 CPU
  - 64 bit protected mode

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- x86 RAM
  - macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer
  - VT-d IOMMU hardware
    - Page table hardware

- Network Interface Card (NIC)
  - Network packet
Direct Memory Access with VT-d

- x86 CPU
  - 64 bit protected mode

- MMU hardware
  - Virtual memory enabled

- x86 RAM
  - macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer

- VT-d IOMMU hardware
  - Page table hardware
  - Consult page tables

- Network Interface Card (NIC)
  - Network packet
Direct Memory Access with VT-d

- x86 CPU
  - 64 bit protected mode
- MMU hardware
  - Virtual memory enabled
  - Page table hardware
- VT-d IOMMU hardware
  - Page table hardware
  - Consult page tables
- VT-d page tables:
  - "Packet buffer R/W, everything else unmapped"
- x86 RAM
  - macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer

Network Interface Card (NIC)
- Network packet
Direct Memory Access with VT-d

- x86 CPU
  - 64 bit protected mode

- Network Interface Card (NIC)
  - Network packet

- MMU hardware
  - Virtual memory enabled
  - Page table hardware

- VT-d IOMMU hardware
  - Consult page tables

- macOS kernel
  - NIC kernel extension
  - Kernel heap
  - Packet buffer

- x86 RAM
  - macOS kernel
  - VT-d IOMMU hardware
  - VT-d page tables: “Packet buffer R/W, everything else unmapped”
DMA Protection for Thunderbolt

- T2
- x86
- UEFI firmware
- macOS kernel
DMA Protection for Thunderbolt
DMA Protection for Thunderbolt

T2

UEFI firmware

x86

macOS kernel

VT-d setup

Thunderbolt malicious DMA
DMA Protection for Thunderbolt

T2

UEFI firmware

macOS kernel

VT-d setup

Thunderbolt malicious DMA
DMA Protection for Thunderbolt

Thunderbolt malware DMA

VT-d setup

UEFI firmware

macOS kernel

x86

T2
DMA Protection for Thunderbolt

- T2
- UEFI firmware
- macOS kernel
- x86
- VT-d setup
- Thunderbolt malicious DMA
DMA Protection for Thunderbolt

**Diagram:**
- T2
- UEFI firmware
- x86
- macOS kernel
- VT-d setup
- Thunderbolt malicious DMA
DMA Protection for Thunderbolt

Thunderbolt malicious DMA

VT-d setup

macOS kernel

UEFI firmware

x86

T2
DMA Protection for PCIe

- T2
- UEFI firmware
- x86
- macOS kernel
- VT-d setup
- Thunderbolt malicious DMA
DMA Protection for PCIe

- T2
- UEFI firmware
- x86
- macOS kernel
- PCIe malicious DMA
- Thunderbolt malicious DMA
- VT-d setup
DMA Protection for PCIe

- T2
- UEFI firmware
- x86
- macOS kernel
- PCIe malicious DMA
- Thunderbolt malicious DMA
- DMA Protection for PCIe
- VT-d setup
- VT-d
- x86
DMA Protection for PCIe

- T2
- UEFI firmware
- x86
- macOS kernel
- PCIe malicious DMA
- Thunderbolt malicious DMA
- VT-d setup
- VT-d
- Thunderbolt
- PCIe
DMA Protection for PCIe Bus 0

- T2
- macOS
- kernel
- x86
- VT-d
- setup
- PCIe malicious DMA
- Thunderbolt malicious DMA
- UEFI firmware
- macOS kernel
DMA Protection for PCIe Bus 0

Pre-RAM firmware -> Post-RAM firmware -> macOS kernel

- PCIe malicious DMA
- Thunderbolt malicious DMA

VT-d setup
DMA Protection for PCIe Bus 0

T2

Pre-RAM firmware

Post-RAM firmware

macOS kernel

PCIe Bus 0 malicious DMA

PCIe malicious DMA

Thunderbolt malicious DMA

VT-d setup
x86 CPU
32 bit protected mode

MMU hardware
Pass-through

Page table hardware

T2

Verified, read-only UEFI firmware

VT-d IOMMU hardware

Page table hardware

RAM not initialized
VT-d IOMMU hardware

Page table hardware

x86 CPU
64 bit protected mode

Fetch instructions

MMU hardware
Virtual memory enabled

Page table hardware

T2
Verified, read-only UEFI firmware

Pre-RAM
UEFI
macOS

x86 RAM

VT-d IOMMU hardware

Page table hardware
x86 CPU
64 bit protected mode

VT-d IOMMU hardware

Page table hardware

MMU hardware
Virtual memory enabled

Page table hardware

T2
Verified, read-only
UEFI firmware

x86 RAM
Verified
UEFI firmware

Pre-RAM
UEFI
macOS

RAM not initialized
x86 RAM
Verified, read-only
UEFI firmware

Fetch instructions
// This array contains the root and interrupt remapping tables. Each table is
// 4kB, and must be 4kB aligned as well. We can only guarantee the alignment by
// manually mapping our 2 4kB tables into this 12kB array. By initializing the
// array to all zeros, every bus is marked as not present, and no interrupts
// are allowed.
STATIC UINT8 mTables[TABLE_SIZE * 3] = {0};

STATIC
EFI_STATUS
EFIAPI
VTdBlockDMAForUnit(UINTN VTdBar)
{
    EFI_STATUS Status;
    VTD_ECAP_REG ExtCapabilities;
    UINT64 RootTable;
    UINT64 InterruptTable;

    CHECKED_VTD_CALL(CheckCapabilities(VTdBar));

    // ExtCap needed for IOTLB register offset

}
// This array contains the root and interrupt remapping tables. Each table is
// 4kB, and must be 4kB aligned as well. We can only guarantee the alignment by
// manually mapping our 2 4kB tables into this 12kB array. By initializing the
// array to all zeros, every bus is marked as not present, and no interrupts
// are allowed.

STATIC UINT8 mTables[TABLE_SIZE * 3] = {0};

STATIC
EFI_STATUS
EFI_API
VTdBlockDMAForUnit(UINTN VTdBar)
{
    EFI_STATUS Status;
    VTD_ECAP_REG ExtCapabilities;
    UINT64 RootTable;
    UINT64 InterruptTable;

    CHECKED_VTD_CALL(CheckCapabilities(VTdBar));

    // ExtCap needed for IOTLB register offset
    ExtCapabilities.Uint64 = MmioRead64(VTdBar + R_ECAP_REG);
UINT64 InterruptTable;

CHECKED_VTD_CALL(CheckCapabilities(VTdBar));

// ExtCap needed for IOTLB register offset
ExtCapabilities.Uint64 = MmioRead64(VTdBar + R_ECAP_REG);

RootTable = (UINT64)mTables;

// Align the root table to a 4kB boundary within the table buffer.
RootTable = (RootTable + TABLE_SIZE - 1) & ~(TABLE_SIZE - 1);

// Set deny-all root table
SetRootTable(VTdBar, RootTable);

// Put the interrupt remapping table right after the root table
InterruptTable = RootTable + TABLE_SIZE;

// Set deny-all interrupt table
SetInterruptRemapTable(VTdBar, InterruptTable);
RootTable = (UINT64)mTables;

// Align the root table to a 4kB boundary within the table buffer.
RootTable = (RootTable + TABLE_SIZE - 1) & ~(TABLE_SIZE - 1);

// Set deny-all root table
SetRootTable(VTdBar, RootTable);

// Put the interrupt remapping table right after the root table
InterruptTable = RootTable + TABLE_SIZE;

// Set deny-all interrupt table
SetInterruptRemapTable(VTdBar, InterruptTable);
SetRootTable(VTdBar, RootTable);

// Put the interrupt remapping table right after the root table
InterruptTable = RootTable + TABLE_SIZE;

SetInterruptRemapTable(VTdBar, InterruptTable);

Same for MSI VT-d interrupts
<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
<th>Type</th>
<th>Subtype</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3C7ABA6-494A-47C6-9F66-F6DE5E42A6E7</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>ApplePlatformInfoDatabaseDxe</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>C24A946F-BBC6-412B-9ACE-387E86BB6125</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>D072678D-D0C2-4768-8182-99BB4AEEF5EDC</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>3F81A55F-04EF-42D9-8FA9-891839769F8D</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>636B2604-8FA3-408B-838A-95355B4932BE</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>A1F39391-B841-4C3E-5A5B-71E3120D6C9B</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>PEI dependency section</td>
<td>Section</td>
<td>Section</td>
<td>Section</td>
<td>PEI dependency</td>
</tr>
<tr>
<td>Raw section</td>
<td>Section</td>
<td>Section</td>
<td>Section</td>
<td>Raw</td>
</tr>
<tr>
<td>TE image section</td>
<td>Section</td>
<td>Section</td>
<td>Section</td>
<td>TE image</td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>53F22BE-8FBB-49B9-81B8-578348AB8F60</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>PlatformInitPreMem</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>PiSmCommunicationPei</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>S3Resume2Pei</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>SInitPreMem</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>CpuMPei</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>SInit</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>6305F2B8-8F76-40A5-8A4F-D2500FDDC1C</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>30BFC263-7672-45C5-8BB6-6028BE94EE4</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>AD70DBB-25F4-4F63-893C-7D0CB04C52</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>D8B116EE-5353-492B-AB3C-AC89F0A458B</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>F25EC59D-56AF-D179-671E812896</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>75135A8-4D66-4528-AE09-02BF9FA6A762</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Pad-file</td>
<td>File</td>
<td>Pad</td>
<td>Pad</td>
<td></td>
</tr>
<tr>
<td>DxlplPei</td>
<td>File</td>
<td>PEI module</td>
<td>PEI module</td>
<td></td>
</tr>
<tr>
<td>Ef1BiosIdGuid</td>
<td>File</td>
<td>Freeform</td>
<td>Freeform</td>
<td></td>
</tr>
<tr>
<td>AppleRomInformation</td>
<td>File</td>
<td>Freeform</td>
<td>Freeform</td>
<td></td>
</tr>
<tr>
<td>Volume Free space</td>
<td>File</td>
<td>Free space</td>
<td>Free space</td>
<td></td>
</tr>
<tr>
<td>04ADEAD-61F4-4D31-B66A-64F8BF9015FA</td>
<td>Volume</td>
<td>FF52v</td>
<td>FF52v</td>
<td></td>
</tr>
<tr>
<td>04ADEAD-61F4-4D31-B66A-64F8BF9015FA</td>
<td>Volume</td>
<td>FF52v</td>
<td>FF52v</td>
<td></td>
</tr>
</tbody>
</table>
x86 CPU
64 bit protected mode

VT-d IOMMU hardware

Page table hardware

MMU hardware
Virtual memory enabled

Page table hardware

Pre-RAM
UEFI
macOS

T2
Verified, read-only UEFI firmware

x86 RAM
VT-d IOMMU hardware
Page table hardware
x86 CPU
64 bit protected mode

Fetch instructions

VT-d IOMMU hardware
Page table hardware
MMU hardware
Virtual memory enabled
Page table hardware

Bus0 VT-d page tables
“deny all”

T2
Verified, read-only UEFI firmware

x86 RAM
Verified UEFI firmware

Pre-RAM
UEFI
macOS
VT-d IOMMU hardware

Page table hardware

T2

x86 CPU

x86 CPU

64 bit protected mode

64 bit protected mode

x86 RAM

Bus0 VT-d page tables

“deny all”

MMU hardware

Virtual memory enabled

Page table hardware

Bus0 VT-d page tables

“deny all”

UEFI firmware

Verified, read-only

Verified, read-only

UEFI firmware

Pre-RAM

UEFI

macOS

Fetch instructions

PCle Bus 0 device

Malicious data

VT-d IOMMU hardware

Page table hardware

x86 RAM

Verified

UEFI firmware
x86 CPU
64 bit protected mode

PCIe Bus 0 device

Malicious data

鸸

x86 RAM

Verified, read-only UEFI firmware

x86 CPU
64 bit protected mode

Page table hardware

VT-d IOMMU hardware

Consult page tables

MMU hardware
Virtual memory enabled

Page table hardware

Bus0 VT-d page tables
"deny all"

Verified, read-only UEFI firmware

T2

Pre-RAM UEFI macOS
DMA Protection for PCIe Bus 0

- PCIe Bus 0 malicious DMA
- PCIe malicious DMA
- Thunderbolt malicious DMA
- VT-d setup
- Pre-RAM firmware
- Post-RAM firmware
- macOS kernel
DMA Protection for PCIe Bus 0

- Pre-RAM firmware
- VT-d setup
- Post-RAM firmware
- macOS kernel

PCle Bus 0 malicious DMA
PCIe malicious DMA
Thunderbolt malicious DMA
DMA Protection for PCIe Bus 0

- PCIe Bus 0: malicious DMA
- PCIe: malicious DMA
- Thunderbolt: malicious DMA

VT-d setup

Pre-RAM firmware → Post-RAM firmware → macOS kernel
PCle Option ROMs
**PCle Option ROMs**

Device drivers which PCle devices supply to UEFI

UEFI firmware, including OROMs, mostly all run at the same x86 privilege level: Ring 0

All code loaded after OROMs, including the booter and kernel, is vulnerable to overwrite
Other UEFI drivers (storage, network, etc)
Core UEFI firmware
PCIe card 1
PCIe card 2
PCIe card 3
x86 CPU
Virtual Memory Space
OROM 1
OROM 2
OROM 3

Ring 0
(More privileged)

Hardware
(More privileged)
The diagram illustrates the relationship between hardware components and software layers in a computer system. At the hardware level, we have PCIe cards labeled 1, 2, and 3. Moving up to software layers, we encounter Ring 3 (Less privileged) and Ring 0 (More privileged). Ring 0 includes the core UEFI firmware and other UEFI drivers (storage, network, etc). Above Ring 0, we find the x86 CPU, which is at the top of the virtual memory space. The OROM layers (1, 2, 3) are situated underneath Ring 0, indicating their position in the privilege hierarchy.

This visual representation helps to understand the hierarchical structure of the computer's operation system, from hardware to software layers, and how each component interacts with others.
Other UEFI drivers (storage, network, etc)

x86 CPU

Core UEFI firmware

OROM 1
OROM 2
OROM 3

PCIe card 1
PCIe card 2
PCIe card 3

Ring 3
(Less privileged)

Ring 0
(More privileged)

Hardware
(More privileged)
Other UEFI drivers (storage, network, etc)
Non-sandboxed UEFI drivers
OROM sandbox driver
Core UEFI firmware
Virtual Memory Space 1
OROM 1
Virtual Memory Space 2
OROM 2
Virtual Memory Space 3
OROM 3
PCIe card 1
PCIe card 2
PCIe card 3
x86 CPU
OROM 1
OROM 2
OROM 3
Ring 3
(Less privileged)
Ring 0
(More privileged)
Hardware
(More privileged)
OROM Sandbox

OROMs can only call a limited subset of expected UEFI interfaces
• Similar to system call filtering

OROMs can only install a limited subset of expected UEFI interfaces
• E.g. read and write to disk blocks, or draw to graphics
OROM sandbox driver

Non-sandboxed UEFI drivers

x86 CPU

Virtual Memory Space 1

Virtual Memory Space 2

Virtual Memory Space 3

Call "Write NVRAM" interface

OROM 1

OROM 2

OROM 3

Core UEFI firmware

Non-sandboxed UEFI drivers

PCIe card 2

PCIe card 3

PCIe card 1

Ring 3

(Less privileged)

Ring 0

(More privileged)

Hardware

(More privileged)
Ring 3
(Less privileged)

Ring 0
(More privileged)

Hardware
(More privileged)

Virtual Memory Space 1
OROM 1

Virtual Memory Space 2
OROM 2

Virtual Memory Space 3
OROM 3

OROM sandbox driver
Non-sandboxed UEFI drivers
Core UEFI firmware

x86 CPU

PCIe card 1
PCIe card 2
PCIe card 3
OROM sandbox driver

Non-sandboxed UEFI drivers

Ring 0
(More privileged)

Virtual Memory Space 1

OROM 1
“I’m a storage driver (for Card 1)”

Virtual Memory Space 2

OROM 2
“I’m a network driver (for Card 2)”

Virtual Memory Space 3

OROM 3
“I’m a SecureBoot driver”

Ring 3
(Less privileged)

OROM sandbox driver

Non-sandboxed UEFI drivers

Core UEFI firmware

Hardware
(More privileged)

x86 CPU

PCIe card 1

PCIe card 2

PCIe card 3
OROM can only talk to the assigned device in its sandbox
- This is the device it was embedded on

The VT-d policy allows a device to DMA to any memory allocated within its OROM’s sandbox
- Preserve high-throughput DMA but with strong VT-d protection
- OROM doesn’t even have to be VT-d aware!
OROM Sandbox will drive attackers to privilege escalation and sandbox escapes

We added a strong set of exploit mitigations to EFI on T2 systems

• Stack Cookies
• All EFI memory W^X with read-only page tables
• SMAP: Ring 0 can’t directly read/write Ring 3 data
• SMEP: Ring 0 can’t execute Ring 3 code
• Some Spectre/Meltdown mitigations
Mac Secure Boot

Summary

The T2 Security Chip brings key secure boot properties from iOS to the Mac, far outclassing UEFI SecureBoot-based systems.

Our DMA protection for PCIe Bus 0 provides state-of-the-art protection against DMA attacks targeting firmware.

The Mac OROM Sandbox provides unprecedented defense against malicious PCIe Option ROMs compromising the secure boot process.
Mac secure boot

iOS code integrity protection

Find My
Software Enforced Code Integrity
Before iOS 9

Kernelcache signature verified by iBoot at load time

Userland __TEXT pages code signed
- CodeDirectory checked at load time (or static)
- Pages checked at fault time

Compromised kernel could change its own __TEXT

Compromised kernel could disable codesigning altogether, or alter userland pages
Kernel Integrity Protection

Goal
• Maintain integrity of kernel code and read-only data after secure boot

Threat model
• Kernel arbitrary read/write
• Arbitrary kernel instruction pointer control
• Arbitrary read/write by DMA agents and system coprocessors

Out of scope
• Secure boot bypass
Kernel Integrity Protection v0
iOS 9

At system initialization, EL3 monitor creates array of kernel page table and text hashes in TZ1

Monitor periodically verifies hashes, panics on mismatch

Effective against long-lived patches, inherently vulnerable to races
Kernel Integrity Protection v0

CPU

EL3, EL1, EL0

Address, TZ1
(TZ1 can only be set by accesses from EL3)

Data

Memory controller

TZ1 base

TZ1 end

Monitor code

Page hashes

Kernel code

Kernel data

Kernel data

Kernel code

DRAM

TZ1
Access requires TZ1 bit
Lessons Learned

Must protect critical data in addition to code
• Page tables
• Global offset table entries
• Sandbox configuration

Integrity verification after boot is vulnerable to race conditions

Easier to adapt hardware architecture to fit security requirements
Kernel Integrity Protection v1
iPhone 7

New hardware design tailored to our goals

Our threat model had three hardware requirements

• CPU prevents modification of kernel memory
• CPU also prevents EL1 execution of non-kernel memory
• Memory controller prevents DMA writes to protected physical range
Kernel Integrity Protection v1

CPU

MMU
- Kernel base
- Kernel end

Memory controller
- ROR base
- ROR end

DRAM
- Kernel and kext code
- Init code
- Kernel page tables

VA → PA, XN, RO

Read Only Region (ROR)
X, RO
XN, RO
XN, RW

PA, XN, RO
Kernel Integrity Protection v1: Read-Only Data

We have a strong design for code, but protecting data requires additional finesse.

Neither KIP v0 nor KIP v1 prevent modification of TTBR1, which tells CPU where to find the kernel’s page tables.

By using a very careful initialization sequence, we make sure no instructions are available to modify TTBR1 after CPU finishes initializing.
Kernel Integrity Protection v1

Required significant rework of kernelcache layout

Build time checks that no TTBR1 write gadget exists

Very effective at protecting kernel code integrity

Only public bypass was off-by-one error in our protection range calculation
Kernel Integrity Protection v2

iPhone Xs

Applied lessons learned from KIP v1

Control bits prevent changes to TTBR1, MMU enable, and exception vector addresses

• Guarantees in hardware that MMU configuration cannot be modified
• Replaces init-only instructions from KIP v1

Configuration is retained when CPU goes into idle power-off

• Less complexity in power management transitions
Kernel Integrity Protection

Summary

Robust enforcement of kernel code and read-only data integrity

Hardware implementation tailored to software security requirements

Essential foundation for next-generation security features
Fast Permission Restrictions (APRR)

iPhone X

Builds upon software-only Hardened WebKit JIT Mapping in iOS 10

CPU register to quickly restrict permissions on RWX memory, per thread

Removes overhead of a syscall and walking page tables to change permissions
Pre-APRR VM Permissions

- Process code: R-X
- Heap: RW-
- Framework code: R-X
- JIT memory: RWX
APRR: JavaScriptCore Execution Threads

- Process code: R-X
- Heap: RW-
- Framework code: R-X
- JIT memory: RWX

APRR = ~W

Effective = R-X
APRR: JavaScriptCore JIT Compiler Thread

- Process code: R-X
- Heap: RW-
- Framework code: R-X
- JIT memory: RWX

APRR = ~X
Effective = RW-
What about userland?
Protecting Userland Integrity

KIP gives us strong integrity protection for kernel text

Page table overrides with KIP rely on kernel code being static

Userland code is dynamically loaded, so we would need dynamic overrides
Page Protection Layer (PPL)

iPhone Xs

Ensures userland code can’t be modified after code signature checks complete

Built upon KIP and APRR

Manages page tables, code signing validation

Small TCB

 Guarantees only code inside PPL can alter protected pages
Summary

System-wide dynamic code integrity enforcement
• Even with a compromised kernel!

Massive attack surface reduction

Low overhead
• No hypervisor traps
• No nested page tables
With code integrity protected, how do we protect control flow?
Pointer Authentication

New instructions in ARMv8.3

Uses spare bits in pointers to store a cryptographic hash

Designed to be robust in the presence of arbitrary read/write primitives
Pointer Authentication

Instructions

pacKK \( X_d, X_n \)

Extra data

Pointer

Key

Encrypt

Signed Pointer

\( X_d \)
Pointer Authentication

Sign

0000000100a41238
Pointer Authentication

Sign

00000000100a41238

Padding Address
Pointer Authentication

Sign

7b9352e100a41238

Signature Address
Pointer Authentication
Authenticate

7b9352e100a41238
Signature
Address
Pointer Authentication
Authenticate

00000000 100a41238
Padding        Address
Pointer Authentication
Auth failure

7b9352f100a41238

Signature                      Address
Pointer Authentication
Auth failure

Padding: 20000000
Address: 100a41238
5 secret 128-bit values
- IA, IB, DA, DB, and GA keys
- I keys for instructions, D keys for data
- GA key for data MAC

Randomly generated
- At boot (A keys)
- At process creation (B keys)

Can’t be read by attacker
## Pointer Authentication

**Pointers to code**

<table>
<thead>
<tr>
<th>Description</th>
<th>I</th>
<th>A</th>
<th>B</th>
<th>Storage Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Return Address</td>
<td></td>
<td></td>
<td></td>
<td>Storage Address</td>
</tr>
<tr>
<td>Function Pointers</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Block Invocation Function</td>
<td></td>
<td></td>
<td></td>
<td>Storage Address</td>
</tr>
<tr>
<td>Objective-C Method Cache</td>
<td></td>
<td></td>
<td></td>
<td>Storage Address + Class + Selector</td>
</tr>
<tr>
<td>C++ V-Table Entries</td>
<td></td>
<td></td>
<td></td>
<td>Storage Address + Hash(mangled method name)</td>
</tr>
<tr>
<td>Computed Goto Label</td>
<td></td>
<td></td>
<td></td>
<td>Hash(function name)</td>
</tr>
</tbody>
</table>
Pointer Authentication
Function return address before PAC

```assembly
_func:
  stp x29, x30, [sp, #-16]!
  ...
  ldp x29, x30, [sp], #16
  ret
```
Pointer Authentication
Function return address after PAC

_func:
pacibsp
stp x29, x30, [sp, #-16]!
...
ldp x29, x30, [sp], #16
retab
### Pointer Authentication

Pointers to data, code via data

<table>
<thead>
<tr>
<th>Kernel Thread State</th>
<th>G</th>
<th>A</th>
<th>*</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>User Thread State Registers</th>
<th>I</th>
<th>A</th>
<th>Storage Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>C++ V-Table Pointers</th>
<th>D</th>
<th>A</th>
<th>0</th>
</tr>
</thead>
</table>
Pointer Authentication

Improvements in iOS 13

Abort on all authentication failures in kernel

Adoption across all Apple kexts

Hardened jump tables
Pointer Authentication

Improvements in iOS 13

ObjC method dispatch hardening
• Sign and authenticate IMP pointers in method cache tables

Hardened exception handling
• Hash and verify sensitive register state

JavaScriptCore JIT and extra data hardening
Authenticated members of high value data structures

- Processes, tasks
- Codesigning
- Virtual Memory subsystem
- IPC structures

Pointer Authentication

Coming soon
Mac secure boot
iOS code integrity protection
Find My
Helping users find lost devices, even when offline

Any device in proximity can help, even if stranger to the owner

Offline device communicates via Bluetooth with participating strangers (finders)

Finders report their location and a timestamp

Owner uses a second device to find the lost device
Challenges

A static device identifier makes the device trackable

Even with a rotated identifier, finder can’t encrypt location end-to-end
  • Server would have access to the location information
Security and Privacy Goals

Protect owners, finders, and devices

• Location reports are not accessible to Apple servers
  – Cannot read, modify, or even add bogus reports
• Finder identities and location not revealed to Apple servers
  – No finder identifier recorded
  – Reported location is encrypted
• Information broadcasted by the lost device cannot be used to track it, except by the owner
Find My Setup

Generate EC P-224 key pair \{d, P = d \cdot G\}
Generate symmetric key SK_0
Store \{d, P, SK_0\} in iCloud Keychain

Encrypted \{d, P, SK_0\} in iCloud Keychain
Find My

Device broadcasting its location

A Find My time period, $i$, is 15 minutes long

Derive symmetric key $SK_i$

• $SK_i = KDF(SK_{i-1}, \text{“update”})$

Derive anti-tracking secret pair $(u_i, v_i)$

• $(u_i, v_i) = KDF(SK_i, \text{“diversify”})$

Unlinkably diversify public key $P$

• $P_i = u_i \cdot P + v_i \cdot G$

Broadcast $P_i$ to nearby finders
Find My

Reporting location of a broadcasting device

Finder ECIES-encrypts its location to public key $P_i$

Computes lookup index $i = \text{SHA256}(P_i)$

Uploads encrypted report with index $i$ to Apple servers
Find My

Owner locating their device

---

Retrieve $d_i$ from iCloud Keychain

Compute $P_i = d_i \cdot G$ for lookup period $i$

Compute lookup index $i = \text{Hash}(P_i)$

ECIES decrypt $(\text{pos}_{i,0}, \text{time}_{i,0}) = D(d_i, \text{rec}_0)$

---

Query DB for location reports at index $i$

DB responds with $[\text{rec}_0, \text{rec}_1, ...]$
Find My

Summary

Novel design to enable users to enlist the help of strangers to locate lost devices

Highly rigorous privacy properties to protect participating device owners and finders
Mac secure boot

iOS code integrity protection

Find My
Mac secure boot
iOS code integrity protection
Find My
Apple Security Bounty
### Introduced in 2016

<table>
<thead>
<tr>
<th>Platforms</th>
<th>iOS, iCloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories</td>
<td>5</td>
</tr>
<tr>
<td>Participation</td>
<td>Very small invited researcher audience</td>
</tr>
<tr>
<td>Maximum payout</td>
<td>$200,000</td>
</tr>
</tbody>
</table>
50 High-Value Reports
What’s next?
Apple Security Bounty will be open to all researchers
iCloud
iOS
tvOS
iPadOS
watchOS
macOS
Revised and expanded categories
<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Attack Method</th>
<th>Maximum Payout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unauthorized access to iCloud account data on Apple servers</td>
<td></td>
<td>$100,000</td>
</tr>
<tr>
<td>Attack via physical access</td>
<td>Lock screen bypass</td>
<td>$100,000</td>
</tr>
<tr>
<td></td>
<td>User data extraction</td>
<td>$250,000</td>
</tr>
<tr>
<td>Attack via user-installed app</td>
<td>Unauthorized access to high-value user data</td>
<td>$100,000</td>
</tr>
<tr>
<td></td>
<td>Kernel code execution</td>
<td>$150,000</td>
</tr>
<tr>
<td></td>
<td>CPU side channel attack on high-value user data</td>
<td>$250,000</td>
</tr>
<tr>
<td>Network attack requiring user interaction</td>
<td>One-click unauthorized access to high-value user data</td>
<td>$150,000</td>
</tr>
<tr>
<td></td>
<td>One-click kernel code execution</td>
<td>$250,000</td>
</tr>
<tr>
<td>Network attack with no user interaction</td>
<td>Zero-click radio to kernel with physical proximity</td>
<td>$250,000</td>
</tr>
<tr>
<td></td>
<td>Zero-click access to high-value user data</td>
<td>$500,000</td>
</tr>
</tbody>
</table>
Vulnerabilities in designated pre-release builds
50% bonus
What about getting started?
We want to attract exceptional researchers who have been focused on other platforms.

New researchers shouldn’t have to find a full chain to bootstrap research.

Existing iOS researchers shouldn’t have to hold back chains for research.
iOS Security Research Device Program
iOS Security Research Device program

Unprecedented, Apple-supported iOS security research platform

Comes with ssh, a root shell, and advanced debug capabilities

New research fusing, neither production nor development
iOS Security Research Device program

Unprecedented, Apple-supported iOS security research platform

Comes with ssh, a root shell, and advanced debug capabilities

New research fusing, neither production nor development

Program applications open to everyone with a track record of high-quality systems security research on any platform
Unprecedented, Apple-supported iOS security research platform

Comes with ssh, a root shell, and advanced debug capabilities

New research fusing, neither production nor development

Program applications open to everyone with a track record of high-quality systems security research on any platform

Coming next year
Apple Security Bounty

Summary

Participation open to all researchers in the Fall

Expanded and revised categories

Highest maximum payouts in the industry

iOS Security Research Device Program for exceptional researchers new to our platform
What about a zero-click iOS full chain with kernel code execution and persistence?
<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Attack Details</th>
<th>Maximum Payout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unauthorized access to iCloud account data on Apple servers</td>
<td>$100,000</td>
<td></td>
</tr>
<tr>
<td>Attack via physical access</td>
<td>Lock screen bypass</td>
<td>$100,000</td>
</tr>
<tr>
<td></td>
<td>User data extraction</td>
<td>$250,000</td>
</tr>
<tr>
<td>Attack via user-installed app</td>
<td>Unauthorized access to high-value user data</td>
<td>$100,000</td>
</tr>
<tr>
<td></td>
<td>Kernel code execution</td>
<td>$150,000</td>
</tr>
<tr>
<td></td>
<td>CPU side channel attack on high-value user data</td>
<td>$250,000</td>
</tr>
<tr>
<td>Network attack requiring user interaction</td>
<td>One-click unauthorized access to high-value user data</td>
<td>$150,000</td>
</tr>
<tr>
<td></td>
<td>One-click kernel code execution</td>
<td>$250,000</td>
</tr>
<tr>
<td>Network attack with no user interaction</td>
<td>Zero-click radio to kernel with physical proximity</td>
<td>$250,000</td>
</tr>
<tr>
<td></td>
<td>Zero-click access to high-value user data</td>
<td>$500,000</td>
</tr>
<tr>
<td></td>
<td>Zero-click kernel code execution with persistence</td>
<td>$1,000,000</td>
</tr>
</tbody>
</table>
We’re excited to work with you!