System Security fiche

Pierre Colson

January 2023

Contents

Markdown version on *github*
Compiled using *pandoc* and *`gpdf` script*

Side Channels & Tempest

- **Compromising Emanations**: Physical signals related to digital activity; break the assumption of higher-level abstractions; Root cause of many attacks
- **Tempest**: Passive leakage of plaintext information (E.g., video on screen)
  - Video Signal
    * Signal in wire/connector/etc. not well shielded
    * Current in wires generates EM waves
    * Modulated with the pixel values
- **Soft Tempest**: Active version of *Tempest*; leakage used to exfiltrate data
  - Vide Signal
    * Signal in wire/connector/etc. not well shielded
    * Current in wires generates EM waves
    * Modulated witht the pixel values
    * Use to transmit data
    * Or to add noise Tempest leakage
    * Possible with many other sources of leakage (e.g. memory access)
- **Side Channels**: Use leakage to attack cryptographic implementation (Only in proximity, with few exceptions)
- Type of *pysical leakage*: Execution time, Power, Magnetic and electromagnetic, optical, thermal, acoustic and vibrational, reflection of injected signals
- Type of *attack*: Passively recover plaintext, Actively exfiltrate data; attack cyrptographic implementation
- **Symmetric encryption** for *confidentiality*
  - *Stream cypher*
    * Process a message bit by bit (byte by byte)
    * $KeyStream = PseudoRandomBitStreamGenerator(seed)$
    * $CyphertextStream = KeyStream + PlaintextStream$
  - *Block cypher*
    * Process a message block by block (EAS, DES)
    * Plaintext might need *padding*

    * $Plaintext + Key = BlockCipher$
    * All block cipher leads to the cipher text
    * There is different way to concatenate block cipher
    * **Electronic CodeBlock** (ECB, *insecure*) $C_1 = P_1 \bigoplus K$, $C_2 = P_2 \bigoplus K$, ...
    * **Cipher Block Chaining** (CBC) $C_1 = (P_1 \bigoplus IV) \bigoplus K$, $C_2 = (P_2 \bigoplus C_1) \bigoplus K$, ...
- **Symmetric crypto for authentication**
  - **Message Authentication Code (MAC)**
  - Shared key between A and B
  - A sends to B message $M + MAC$ where $MAC = MACFunction(M, K)$
- **Asymmetric crypto for confidentiality**
  - A and B exchange a key pair via a trusted channel
  - A wants to send M to B, she sends: $C = \mathit{Encryption}(Pu_B, M)$ where Pu_B is B's public key
  - B decrypts the message as follow: $M' = \mathit{Decryption}(Pr_B, C)$ where Pr_B is B's private key
- **Asymmetric crypto for authentication**
  - A send to B message M and Signature S, where $S = \mathit{Encryption}(Pr_A, M)$ where (A's private key)
  - B verifies the signature by checking: $\mathit{Decryption}(Pu_A, S) = M$ where (Pu_A is A's public key)
- Combine the best of two: Asymmetric key exchange + Symmetric encryption
- **Security** of cryptographic algoritms
  - We *model* system and possible attackers
  - Security properties are valid under certain *assumptions*
- Side Channel concrete example
  - **Timing**: Measure execution time
    * Classic timign attack against RSA
    * Remote attack are possible
    * Modern example of remote attack onc cryptocurrencies
  - **Power** Measure some physical quantity influenced by execution
    * Simple Power Analysis (SPA)
    * Differential Power Analysis (DPA)
    * Correlation Power Analysis (CPA)
- Reminder on **RSA**
  - *Key generation*
    1. Chose numbers p, q such that p and q are prime and $p \neq q$
    2. Compute $n = pq$
    3. Compute $\Phi(n) = \Phi(p-1)\Phi(q-1)$
    4. Chose e such that e and $\Phi(n)$ are relative prime and $1 < e < \Phi(n)$
    5. Compute d as such that $de \mod \Phi(n) = 1$
    6. Public key $PU = \{e, n\}$
    7. Private key $PR = \{d, n\}$
  - *Encryption*
    * Plaintext $m < n$
    * Ciphertext $C = m^e modn$
  - *Decryption*
    * Ciphertext C
    * Plaintext $m = C^d modn$
  - *Signature*
    * Plaintext $m < n$
    * Signature $s = m^d modn$
  - The *security* of RSA is based on two hard problems
    * The RSA problem, i.e., computing the e^{th} root of m modulo n from $C = m^e \mod n$
    * FActoring large numbers into smaller primes
- Exponentiation is implemented using *Square and multiply*
  - Problem 1
    * Key dependant branching
    * Execution time depends on the key d, if bit i of d is 0 is will be faster than if bit i of d is 1

- Problem 2
    - Montgomery used for modular multiplication because it is more efficient
    - Montgomery execution time T_{mont} depends on the plaintext m; there is a reduction step done only if necessary
- **Countermeasures**
- **Constant time**
  - Relatively easy for specific cases
    - E.g., modular multiplication without conditional reduction
  - Generic protection is hard
    - Identify and elininate all dependencies of time with plaintext and key
    - Can have performance issues
- **What if we artificially add noise**
  - An attacker jus tneed more measurements to dig the signal out of the noise
- **Masking**: Can we make it impossible fot the attacker to guess
  - Mask with random number C different for each message:
    - $md \mod n \to [(m.X)d \mod n].[(X^{-1})d \mod n] \mod n$
  - Intuitively, given m and d_i the attacker cannot guess slow/fast any more
- A **logic gate**
  - Electronic component that implements a logic operator (not, and, nand, or, xor)
  - Stateless (Combinatorial)
  - Together with memory elements it is used to implement finit state machines
- *MOS transistor*: electronic switch
- Logical gate can be implemented with MOS
- **Data dependency**: There are physical phenomena that create a data dependency between logic values and their transitions and the power consumption of the circuit
- **Measure**
  - We can measure the power consumption and observe these phenomena
  - Signals are small, many measurements and statistical analysis are often needed
- **Model**: we know how it works: given some logic data manipulated by the software/hardware, we can predict the corresponding power consumption
- **Countermeasures**
  - *Problem*: There is a data dependency (of some order) between plaintext,key and the power
  - Add noise
    - Desyncronize ther traces
    - Inject random noise
    - Defeated with better signal processing and more measurements
  - Try to balance the hardware
    - Filtering shielding (Filtering is not perfect, expensive, can be tempered)
    - Make a processor where every instruction/operands consumes the same power (Not easy and expensive)
  - N^{th} order masking
    - Multiply each data with a random variable
    - This algorithmically breaks the dependecy making it impossible to guess the intermediate value
- **EM side channel**
  - Currents flowing in cables produce EM signals
  - Clock might act as a carrier
  - Emissions from localized areas, are not all overall power consumption
- **Sound side channel**
  - Currents in certain capacitors make them vibrate and produce sounds
- We don't always need a physical access
- **Tamper resistant** systems take the bank vault approach
  - Prevention of break in
- **Tamper responding** systems use the burglar alarm approach
  - Real-time detection of intrusion and prevention of access to sensitive data

- **Tamper evident** system are designed to ensure that is a break-in occurs, evidence of the break in is left behind
  - Detection of intrusion

Architectural Support for Security

- **Application security**: Requirements
  - **Launch-time integrity**: correct application was started or loaded
  - **Run-time isolation**: no interference from malicious software, peripherals
  - **Secure persistent storage**
- **OS Security**: Privilege rings
  - Ring 3: applications
  - Ring 2: Device drivers
  - Ring 1: Device drivers
  - Ring 0: Kernel
  - Currently, only *ring 0* and *ring 3* are used
  - CPU tracks the current privilege level CPL using two register bits
  - Main uses: limiting access to privileged instructions, I/O-ports
  - Legacy use: kernel memory protection
  - Privilege in Modern OS (only two levels)
    * Level 0 for the kernel
    * Level 3 for users
- Calls across Privilege Rings
  1. Before **syscall**, put argument into registers
  2. Issue **syscall** → CPU changes privilege level and calls *Hook*
  3. *Hook* performs access policy checks and validates arguments
  4. Kernel performs the requests action
  5. Return to the application
- **Discretionary Access Control** (Linux Security Model)
  - A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission to any other subject
- **Users and Groups**
  - A *user-account* (user, uid)
    * Represents someone capable of using files
    * Associated both with human and processes
  - A *group-account* (group, gid)
    * Is a list of user-accounts
    * Users have a main group
    * May also belong to other groups
- **Permissions**
  - A file has owner and group id
  - A process has owner and group id
  - Kernel verifies permissions before executing system calls
    * If owner uid = 0(root), everything in allowed
    * Otherwise the uid and gid of the process and object are compared in this order and permission for the operation is searched based on owner, group, and other rights
- A **basic transaction**, wherein a *subject* (user or process) attemps some action (read, write, execute) against some *object* (file, directory, special file)
- `setuid` bit means program run with same privileges as owner
  - No matter who executes it
- `setgid` bit means run with same privileges as a member of the group which owns it
  - Again regardless of who executes it
- Password are changed using the program `/bin/passwd`

- Real UID (RUID): UID of the user running the program
- Effective UID (EUID): UID of uer with whose privileges the program runs
- `getuid()` returns RUID
- `geteuid()` returns EUID
- `setuid(uid)` set UID
  - If EUID == root, set EUID and RUID
  - If not root, sets EUID if certain conditions are met
- `setruid(uid)` set RUID
- `seteuid(uid)` set EUID
- `fork()` functions shall create a new process. The new process (child process) shall be an extact copy of the calling process (parent process)
- Linux uses a DAC security model but **Mandotory Access Control (MAC)** imposes a global security policy on all users
  - User may not set controls weaker than policy
  - Normal admin done with accounts without authority to change the global security policy
  - but MAC systems have been hard to manage
- **SE linux**
  - In SELinux, all access must be explicitely granted
  - Allows no acces by default, regardless of the linux user/groups ids
  - There is no default superuser in SELinux unlike the `root` in standard linux
  - Each individual subject and object in SELinux is governed by a **security context** being a:
    * User - individual user (human or daemon)
    * Role - like a group, assumed by user
    * Domain (type) - a sandbox being a combination of subjects and objects that may interact with each other
    * this model is called **Type enforcement (TE)**
  - Decision making
    * **Access** decisions: when subject do things to objects that already exist, create new things in expected domain
    * **Transition** decisions: invocation of processes in different domains than the one in which the subject process is running; creation of objects in defferent types (domains) than their parents domain; transitions must be authorized by SELinux policy
  - Access in granted by specifying access from a subject type (that is, a damain) and an object type using an allow rule
  - A domain transition is allowed only when the following three conditions are tue
    * The process' new domain type has *entrypoint* access to executable file type
    * The process' current domain type has *execute* access to the entry point file type
    * The process' current domain type has *transition* access to the new domain type
- **Paging-based Security**
  - Security-relevant data in page table entries
  - *Supervisor bit*: if set, this page is accessible only in ring 0 (isolates OS from applications)
  - *RW bits*: to distinguish between read-only and writeable pages
  - *Non executable (NX) bit*: if set, the page is not executable (prevents run-time code injection)
- **Virtual address space**
  - Every process has its own virtual address space
  - Kernel address space is protected with the supervisor bit
- **Firewire DMA**
  - Key idea: Access RAM is tightly controlled by the CPU, but this can be circumvented through **DMA** (Direct Memory Access)
  - Firewire technology allow for fast communication speeds between devices, use DMA
  - The attacker uses a Firewire cable to connect to a (locked) PC and issue a DMA request to fetch the contents of RAM
  - Later on, can look in the collected data and leak keys and other passwords
- **BadUSB**

    - Key idea: change a USB device controller to mimic another device class
- **DMA remapping**
    - Setup by the OS, similar to MMU
    - Control DMA access to physical memory
- *OS trust assumptions*
    - Intel Managment Engine
    - BIOS/UEFI
    - Periferical firmware
    - etc.
- **Physical access Attacks** are harder to defend for the OS
    - Remove the hard drive from a machine left unattended
    - Boot from a USB key and copy the data out/change the password
    - Trivial (and possibly broken) solutions:
        * Prevent booting from an external source from the BIOS (Broken; we can reset the bios by removing the battery)
        * Protect the BIOS with a password (Broken; we can reset the bios by removing the battery)
        * Partial/Full disk encryption
        * Data hiding
- **Disk encryption**
    - *Simple approach*: use password only
        * The disk is encrypted with a key that is protected using a user-privided password
        * Problem: password can be brute-forced
    - *Better approach*: leverage a secure element
        * The disk key encryption is stored in a secure element
        * Example: Trusted Platform Module (TPM) chip on motherboard
- **Cold Boot Attack Process**
    - *Normal operation*
        * The user logs with a password
        * The disk encryption key is kept in RAM when the computer is locked
    - *Attack process*
        * Attacker opens the machine to expose the RAM chips
        * Remove power
        * Cool down RAM (dust-sprayer upside down (-50 degree), LN2 (-192 degree))
        * Plug RAM module to another (acquisition) platform
        * Recover key
- Cold Boot Attack: *Countermeasures*
    - Erase key from memory on every (controlled) suspend
        * User needs to type in password often
        * Does not help sudden power loss
    - Prevent booting from external media
        * Does not prevent DRAM component transfers
    - Physical protection
        * Components that respond to enclosure opening or low temperatures
        * Expensive for commodity systems
    - Avoid placing the key in memory
        * Performs encryption in disk controller
        * Requires architectural changes
- **TPM Support**
    - **Secure boot**: OS boots only if the chain of trust is valid
    - **Authenticated boot**: System records chain of trust but the OS boots even if the chain of trust in invalid
- **UEFI** instead of BIOS
- Hardware-supported OS-based - Application Security

.	Hardware support for OS-based Security
CPU	Privilege rings
	Memory Management Unit
Chipset	DMA Remapping tables
Periferals	Trusted Platform Module
	Normal HDD with OS-enforced access control

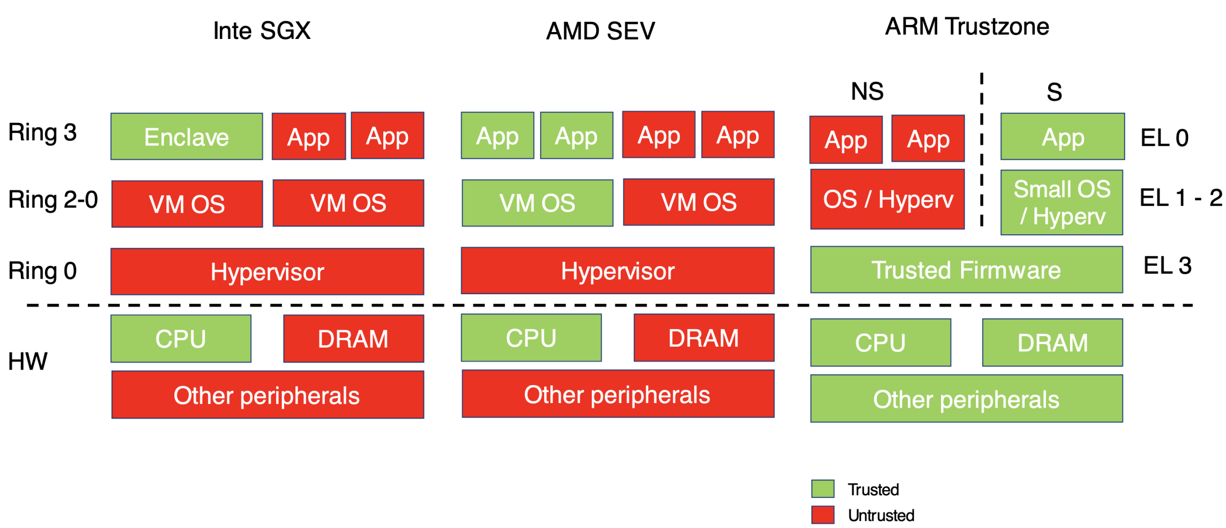
- Runtime Attacks: Typical Attacks
  - Buffer overflows
  - Format string
  - Double free
  - Use-after-free
- Runtime Attacks: Mitigation Techniques
  - **Non-executable pages (NX)**
    - Attackers used to place shell code in normal data pages/stack
    - Stack should not be executable
    - Mark all pages that do not contain code as Non-executable (NX)
    - Hardware support required
    - Mark all pages with executable code as read-only
    - Not always possible
  - **Stack canaries**
    - Add random stack canaries
    - To overwrite the return pointer, the stack canary must also be overwritten
    - Check the value before jumping to return address
    - Can prevent some attacks
    - Assumes the attacker cannot read memory
  - **Address space layout randomization (ASLR)**
    - Return to libc: overwrite return pointer with address of `exec(/bin/sh)`
    - What if we move the binary by a few MB in its virtual address space?
    - The location of `exec` will be unknown to the attacker
    - Implemented in all major operating systems
    - KASLR (linux) also defends against Meltdown
    - Broken if the attacker can read memory (separate info leak exploit)
  - **Control flow integrity**
    - Generate all legal control flow transfers
    - Verify every transfer at runtime
    - Straight forward for direct-control transfers
    - Static destination addresses
    - Forward edges: switch statements, indirect calls, etc.
    - Backward edges: returns

Trusted Execution Environment

- **Defense in depth**:
  - Small Trusted Computing Base (TCB)
  - Even if the OS/Hypervisor are comprised, code/data can be protected
- **Confidential computing**:
  - Client can verify what is running on remote system; operator cannot violate the integrity of the execution
  - Operator cannot see the code/data of the client
- Enforced in Hardware (typically via special CPU instructions or co-processor)
- We need to trust the HW manufacturers: Intel, AMD
- Vulnerable to some side channel attack

- **Computing Systems Trust Model**
  - The **Trusting Computing Base** (TCB) is the set of software and hardware components that need to be trusted for an application to execute securely
  - For instance, the TCB of a banking application includes the Operating System, the CPU, the DRAM chips, the disk ...
    * If any of these are malicious the execution integrity and confidentiality cannot be guaranteed
  - Other applications are not in the TCB
  - In principle, they can be malicious and the OS (with hardware support) still provides isolation to other benign apps
    * However some exploit can lead to privilege escalation
  - **Trusted Execution Environment (TTEs)** usuallly aim to reduce the TCB needed to execute applications
- *Primitives*: Isolation, Bootstrapping trust, Sealing storage
- **Isolation** (defense in depth and confidential computing)
  - CPUs traditionally enforce isolation berween permission levels
  - More privileged levels control and can modify less privileged levels
- **Intel SGX** *Isolate 'small' Apps into enclaves*
  - Design choice
    * Create isolated environments at the application level, called **enclaves**
    * Enclaves are isolated from all the other software in the system OS/Hypervisor
  - Resists a physical attacker (cold boot attack, bus tapping, etc)
    * Everything outside the CPU die is untrusted
    * The CPU die is assumed to be secure
  - The OS and the Hypervisor are still in charge of managing virtual memory and interrupts
  - The CPU keeps track of whether it is currently executing in enclave-mode and which enclave is executing
  - Memory reads to enclave memory when not in enclave mode always return `0xff`
  - Trying to read another enclave memory returns `0xff`
  - Similarly, not authorized memory writes fail silently
  - Enclaves can read/write the memory to their untrusted app
  - New component on the CPU takes care of securing memory, the **Memory Encryption Engine (MEE)**
  - The CPU is trusted, so data resides in the cache in plain text
  - While the content of memory pages is protected by SGX and the MEE, page metadata is not, metadata includes:
    * Access permissions of a page
    * Accessed bit: whether a memory page was recently accessed
    * Dirty bit: whether a memory page was recently written
  - **Controlled Channel attacks**
    * The os can remove execute or read/write access to memory page
    * This trigger an exception which contains the address of the memory page that was being accessed
    * This reveals enough information
    * Information from the accessed/dirty bits are also enough to leak information
  - **Side Channel attacks**
    * Enclaves share resources with other applications in the system: cache, core execution units, branch prediction structure
    * Monitoring utilization of these resources from another application leaks information
- **AMD SEV**: *Isolate entire VMS*
  - Design choices:
    * Isolate Virtual Machine from Hypervisor. VMs can execute code on ring 3,2 and 1
  - Resists low skilled physical attacker (cold boot, but not bus tampering)
    * Everything outside the CPU die is untrusted
  - Comparison with SGX

    * SEV encrypts data that leaves the CPU die
    * However, it does not store a MAC tag together with the data (data in memory is not authenticated)
    * Like SGX, SEV keeps track of the different isolated environments with HW prinitives
    * While unauthorized writes cannot happen from the CPU, if memory is corrupted some other way VMs will read the wrong value from memory
  - Key management
    * SEV uses an ARM co-processor to manage the different encryption key for each VM
    * At VM creation a new key is created in the co-processor to encrypt the VM memory
    * Each new VM has a different encryption key
- **ARM TrustZone**: *The tale of two world*
  - Design choices:
    * Have two separate isolated execution environments. These are referred to as the Non secure and the Secure world
  - No physical attacker
    * Communication to peripherals is assumed to be trusted
  - The memory is partition between the Secure and Non-Secure world
  - The Secure world can read/write the Non-Secure world memory, but the Non-Secure world is restricted to its own memory
  - The TrustZone Address space is controller (TZASV) is a hardware component that is used to configure which ranges of memory belong to which world
  - Only the secure world can configure the TZASV
  - Memory is separated at the physical layer. That is there are two separate physical address ranges, one for the secure world and one for the non-secure world
  - The CPU keeps track of which world in currently executing
- **Trusted Isoltation Environments** Trust Model summary



- **Bootstrapping trust** (Attestation/Secure boot) (confidential computing)
- **Sealing storage** (defense in depth and confidential computing)