

# Secure software development guidelines

## What are the many categories for security requirements?

Security requirements: CIA (+3A for software security) »

- Confidentiality – read access
- Integrity – modification
- Availability – access to the system
- Authentication – who is the entity?
- Authorization – can the entity do that?
- Auditing (aka non-repudiation) – proof of performed activities

## What is the goal of threat modeling?

- Goal: understand risks and threats maximum protection while still providing functionality, usability

## Draw a diagram of a banking app as the Target of Evaluation during threat modeling!

...

Identify the Target of Evaluation

- Define the boundary between the system and its environment
- System is a number of easily identifiable assets decomposition later

## How can we identify the assets of a system?

- Anything that has value to the organization, its business operations and its continuity

## What is the difference between use cases and abuse cases?

- Use case: how the system should be used
- Abuse cases: how the system should not be used

## What does STRIDE stand for?

- Spoofing - másnak adom ki magam
- Tampering - meg nem engedett módosítás
- Repudation - letagadhatóság, egy entitás képes letagadni vminek az elvégzését
- Information disclosure- Olyan információt osztunk meg amit nem lenne szabad
- Denial of services - szolgáltatás megtagadás
- Elevation of privilege - olyan tevékenységeket hajtunk végre amit nem lenne szabad





## What kinds of threats can a(n) external interactor / internal process / data store / data flow face?

Téglalap: External interactor

Kör: Internal process

Szaggatott téglalap: data store

Nyíl: dataflow

				
Spoofting	X	X		
Tampering		X	X	X
Repudiation	X	X	X	
Information disclosure		X	X	X
Denial of service		X	X	X
Elevation of privilege		X		

### How do we measure risk in security?

- Occurrence likelihood
- Impace of threats  $R = PT * DT$
- Not an actual number...

### Describe the secure design principles!

- Economy of mechanism (KISS) – simple, small
- Fail-safe defaults – restoration to a known good state
- Complete mediation – skepticism
- Separation of privilege – modularity, robustness
- Least privilege – bare minimum
- Open design – don't depend on security by obscurity
- Least common mechanism – sharing is a channel
- Psychological acceptability – human in the loop

### What is a UML profile? What purpose does it serve?

- UML profile: lightweight extension mechanism
- Structure diagram
- Defines custom stereotypes, tagged values, and constraints

### What is a UML stereotype?

- Stereotypes: extending existing metaclasses
- new kinds of building blocks
- Cannot be extended by another stereotype
- Stereotypes are classes
  - they can have properties
  - they can be inherited from

### How do the different UMLsec stereotypes influence the interpretation of different UML model elements?

- UML profile with security-related information
- Critical stereotype: Labels a (sub)system or object as highly important
  - Tagged values types:
    - Secrecy
    - Integrity

- High: information flow, can only be influenced by other high elements highlights prevention of leakage
- Data security stereotype: Ensure that data is secured with respect to a defined adversary model

#### **What is the difference between CVE and CWE?**

- **CVE** (Common Vulnerabilities and Exposures): publicly available database of known vulnerabilities. Tehát az egyes szoftverek konkrét hibái.
- **CWE** (Common Weakness Enumeration): formal list of software weakness types, tehát a lehetséges hiba típusok általános leírása.

#### **List 3 techniques for secure handling of inputs!**

- Normalization / Canonicalization: egyszerűsítés (pl. strings to unicode)
- Filtering: szűrés (pl. remove <> chars)
- Validation: data is reasonable (pl. megfelelő formátum)

#### **What kind of checks are involved in input validation?**

- existence
- types
- limits
- consistency

#### **What is the connection between neutralization and escaping?**

- Neutralization: ensure that the output satisfies some security criteria
- Escaping: ensure that the output contains no harmful elements
- Tehát mindkettő az output biztonságosságát ellenőrzi.

#### **What sort of vulnerability is exploited by command injection attacks?**

- Improper Neutralization of Special Elements used in an OS Command

#### **What is the difference between whitelisting and blacklisting?**

- Whitelist: engedélyezettek listája
- Blacklist: tiltottak listája

#### **Describe the Time-of-Check to Time-of-Usage (TOCTTOU) bug!**

- Time-of-Check to Time-of-Usage
- Idő telik el az erőforrás ellenőrzése és használata között, nem biztos hogy ugyanazt az erőforrást használjuk amit ellenőriztük.

#### **What is the timing windows of concurrency attacks?**

- time slot in which the concurrency error may occur

#### **How can concurrency attacks influence the timing window?**

- Retry many times: increased probability of success
- Enlarge timing window: pl. komplex inputtal nagyobb számítási idő

#### **Why is reflection a security risk?**

- Reflection in java: Runtime létrejövő osztályok

- Attacker can instantiate any class that implements the Config interface and follows the naming convention

## Security testing

### What is the goal of security testing?

- Goal: identify vulnerabilities, ensure security functionality
- Positive requirements: what is expected
- Negative requirements: what is unacceptable

### Which testing technique can be performed in which lifecycle phase?

- **Requirements:** threat and risk analysis
- **Design:** formal (model-based) techniques
- **Implementation:** static/dynamic analysis, formal techniques, pentest

### Which aspects of the testing process must be planned beforehand?

- Objective: purpose for executing a test
  - Functional: works, as intended?
  - Non-functional: doesn't work, as intended?
- Scope: granularity of the SUT, determines the test bases
  - Component/Unit
  - Integration
  - System
- Accessibility
  - White-box
  - Grey-box
  - Black-box
- resources, schedule, features to be tested, exit criteria

### What kinds of security testing approaches do you know? What are the advantages and disadvantages of each?

- **Static Analysis:** Instructions are only interpreted, not executed
  - Pro:
    - Scalability: large code bases can be handled
    - Can be applied early in the development lifecycle for source code
  - Con:
    - No runtime information, often produce false positives
- **Dynamic analysis:** Analyzes software as it executes in real life
  - Pro:
    - Access to runtime information more precise results
    - Vulnerabilities reported are definitely in the software
  - Con:
    - May require program instrumentation (not trivial for compiled code)
    - Low code coverage
    - Can't reason about behavior which has not been observed
- **Penetration Testing:**

- Pro: Setup is comparable to an actual attack
- Con: Time consuming?
- **Formal techniques:** Mathematics-based techniques
  - Pro: Greatly increase confidence, best choice for assurance
  - Con:
    - Security properties may be hard to formalize
    - Expertise required to translate information descriptions to models
    - For real-world systems, model may not be complete (but often enough)
    - Different analyses require different formalisms

### What are the main characteristics of manual/automated code review?

- **Code Review:** Detect vulnerabilities by „reading“ the instructions
  - Pro: Find
    - Access control violation vulnerabilities,
    - Web pages not sufficiently protected against forced browsing
    - Inconsistent security checks in the Android framework
    - Missing input validation checks in SAML implementations
  - Pro: Manual code review
    - human creativity
  - Con: Manual code review
    - tedious process, expertise in many areas
  - Pro: Automated: SAST tool analyzes instructions and report potential problems, humans need only to reviews report
    - scalability
    - no expertise required as tool encloses it
  - Con: Automated
    - ,00000000000000000000000000000000,,0,0000000000000tool only reports what it is coded to look for

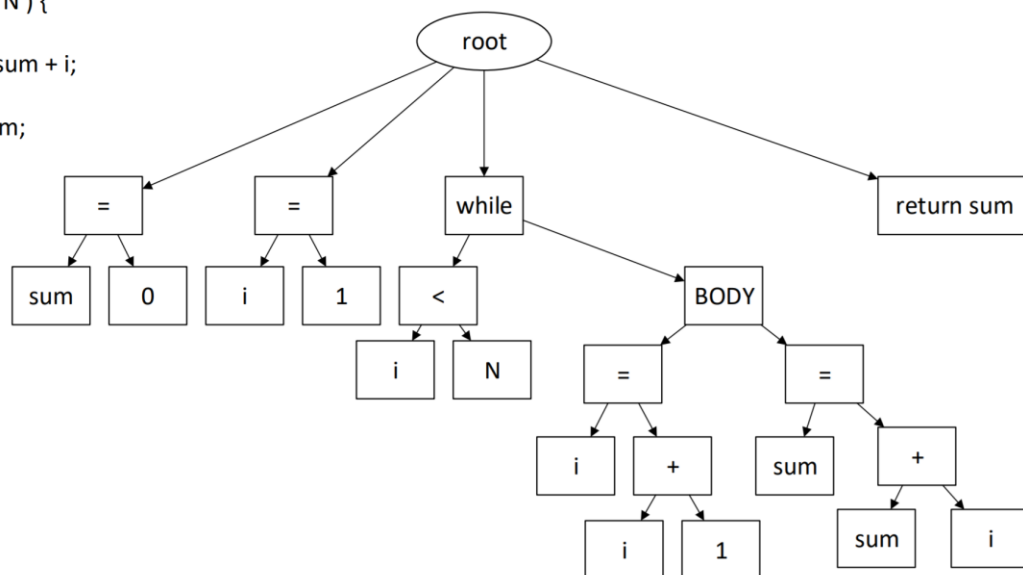
## What does an abstract syntax tree look like?

# Abstract Syntax Tree

```

unsigned int sum(unsigned int N){
    unsigned int sum = 0;
    unsigned int i = 1;
    while( i < N ) {
        i = i+1;
        sum = sum + i;
    }
    return sum;
}

```



## What kinds of analyses can automated code review tools perform?

- control flow analysis
- dependency analysis
- type inference and type checking
- program slicing

## Define the control flow graph!

- Elemi utasítások végrehajtási sorrendje
  - Előre mutató nyilak
  - Vissza mutató nyilak ciklusnál
- Elágazások if-eknél

## Define the program dependency graph!

- Control flow graph + Data dependence graph + Control dependence graph mixed together

## What is data dependency?

- Egy utasítás végrehajtásához szükség van valamelyik adatra, változóra, ezt jelölhetjük a nyilakkal

## What kind of errors can an automated code review tool make?

- False positive: (false alarm), nem tényleges hiba
- False negativ: nem talált meg minden lehetséges hibát, tehát egy hibát jónak vélt

## Describe taint analysis! What is the taint?

- **Taint analysis:** taint data coming from user controlled sources, and ends in a security sensitive operation (sink), without sanitization during its path
- Taint= flag

### What are the challenges of taint analysis?

- Challenges:
  - Undertainting: certain values are not tainted
  - Overtainting: tainted but not derived from source
  - Taint sanitization problem: Taint only added not removed
  - Time of detection vs time of attack: only alert when unsafe values used, no guarantee that program integrity is still OK

### Describe symbolic execution!

- **Symbolic Execution:** Instead of concrete values, special symbols represent input values (can be anything initial, goal to reach all branches)
  - Pro:
    - Automatic generation of concrete test inputs
    - Provides high test coverage
    - Able to find errors in complex software
    - Can reason about higher-level program properties, e.g. complex program assertions
    - Tools exist which implement symbolic execution
  - Con:
    - Scalability issues: path explosion, resource constraints
    - Environment problem
    - Unsolvable constraints

### What is a symbolic variable?

- Instead of concrete values, special symbols represent input values (can be anything initial, goal to reach all branches)

### What is the path explosion problem?

- Minden ifnél plusz két ágat kell vizsgálni (legalább), egy idő után kezelhetetlenül sok path

### What is the environment problem?

- Csak olyan kódról tud állítást megfogalmazni, aminek az utasításai látja (library kell)

### Describe mixed concrete and symbolic execution!

- Uninteresting parts of the analyzed program are executed concretely, on the CPU
- Interesting parts are analyzed symbolically

### In what scenario would you use attack patterns?

- Automated vulnerability scanning tools can be built, e.g. for webapps and web-based APIs

### What are the components of a fuzzer? What are the tasks of each component?

- Generator: Teszteset bemenetek létrehozása
- Test executor: teszt végrehajtása

- Test component: a tesztelni kívánt komponens (instrumentációval információt ad az állapotáról)
- Monitoring system: Report generálás, test component megfigyelése

### **Describe the four main fuzzing approaches!**

- Random fuzzing: behavior for large and/or invalid input (teljesen tetszőleges bemenetek generálása)
- Mutation-based fuzzing: randomly change valid input (jellemzően egy-egy bit állítása)
- Generation-based fuzzing: random input of pre-defined format(pl. csak dátumok gen)
- Evolutionary fuzzing: random input of pre-defined format (a tesztek lefutása során változik az input, attól függően mekkora lefedettséget értünk el, a monitoring system szolgáltatja az adatot, folyamatosan változik az input, ez még elég gyerekcipőben jár)

### **What is the goal of penetration testing?**

- Egy létező/dummy rendszer tesztelése valós támadásokkal, majd reporting készítés

### **What is the difference between a hacker and an ethical hacker?**

- Ethical hacker: szerződéssel, engedéllyel rendelkezik a támadáshoz, így legális a tevékenysége.

### **What are the phases of penetration testing?**

- Planning: important side conditions and boundaries for the test
- Discovery: discover and enumerate all accessible interfaces, then do vulnerability analysis
- Attack: test identified interfaces through attack attempts → Additional discovery
- Reporting: simultaneously with other phases, document all findings and their estimated severity

### **What development artifacts can be used to create a formal model? How?**

- Source code, formal code, Design documentation → Formal model after formalization
- Security property -> Formalization
- Formal verification from formal model + security model → Verdict

### **How would you model an API?**

...

### **How does (bounded) model checking work?**

- formal model is only explored to a certain depth

## **Memory corruption I**

### **What steps could lead to a memory corruption?**

1. Make a pointer invalid (Go out of the bounds of the pointed object, When the pointed object gets deallocated)
2. Dereference that pointer to trigger the error



### What should happen to a pointer to trigger a memory corruption error?

- Spatial error: dereferencing an out-of-the-bound pointer
- Temporal error: dereferencing a dangling pointer → use-after-free vulnerability
- Read from or write to the address pointed by the invalid pointer
  - Examples: format string vulnerability, buffer overflows

### What could be the 4 goals of an attacker?

- Information leak: Leak information to trigger exploitation, pl. bypass ASLR
- Data-only attack: to gain more control, pl. variable overwrite
- Control-flow hijack attack:
  - Code pointer (e.g., function pointer) is overwritten to divert control-flow
  - Or, the attacker can only indirectly control the instruction pointer
  - Shellcode injection and execution is diverted
  - Or, code reuse attacks
- Code corruption attack: teljes kontroll, saját kód teljes mértékű futtatása, eredeti felülírása

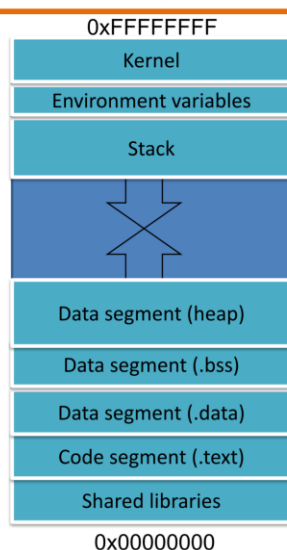
### Describe the registers in an x86 architecture!

- 32 bit
- General registers: eax, ebx, ecx, edx, esi, edi
- Special purpose registers
  - esp – stack pointer
  - ebp – stack frame (base) pointer, function context
  - eip – instruction pointer, stores the address of the next instruction to process

### What is memory segmentation?

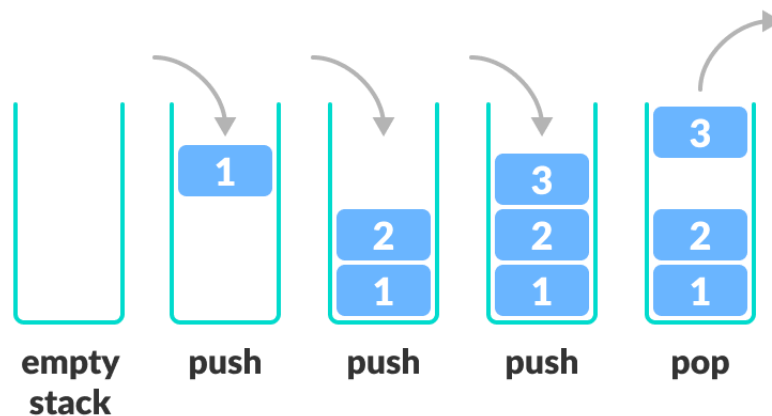
- A memória logikai egységekre osztása

#### Memory segmentation (ELF binaries)



- **Stack segment includes**
  - Local variables
  - Values required for procedure call
- **Data segment**
  - heap: dynamically allocated memory
  - .bss: Uninitialized global & static variables
  - .data: Initialized global & static variables
- **Code Segment:**
  - Executable instructions
  - Typically read-only

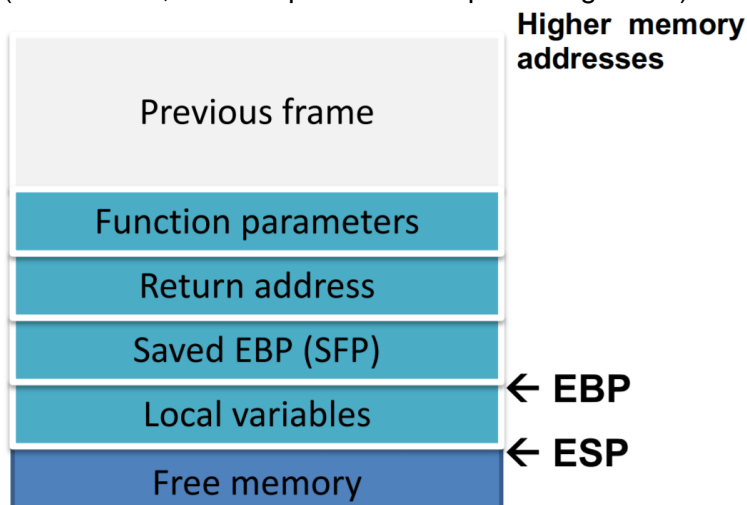
### Describe the behavior of the stack!



- LIFO:
- Stack is built up from several stack frames belonging to functions

#### What is a stack frame? What does it contain?

- Stack is built up from several stack frames belonging to functions, each frame:  
(Saved EBP, =Frame pointer of the preceding frame)



#### What types of buffer overflows exist?

- Occurs when the boundary of a buffer is exceeded by data which overwrites adjacent memory locations
- Stack overflow / Heap overflow

#### Describe a simple stack overflow!

- Stack overflow occurs – when a procedure copies user-controlled data to a local buffer on the stack without verifying its size
- Local data overwrites other values on the stack up to return address
- When the procedure returns, EIP is set to the address residing at the location of the return address --> control flow can be changed
- Insert code to that modified address --> „will be executed”

#### What is a shell code? Where should it be placed?

- Shellcode: egy olyan kód amit a támadó futtatni akar (történelmi okokból, régen ált. shellt akartak szerezni)

- Hova? Into the local buffer with a proceeding nop sled
  - Nop sled: NOPokból álló egymás utáni blokk
    - Put in front of the shellcode and jump into that area
    - Instructions should always reach the beginning of the shellcode
    - – Reason to apply: Bigger chance to find our shellcode
    - – On local systems the position of return address can be calculated (no ASLR)
    - Azért használjuk, hogy (sled, slide)“ rácsússzunk a kódunkra, illetve helykitöltési célokból is (a buffernek tele kell lennie, hogy overflowt érjünk el)
- Into environment variables
- Address of a function inside the program

### How could a buffer overflow be mitigated?

	Preventive	Detective	Anti-exploit
Hardware	–	Virtual memory management (VMM)	NX (Never Execute) memory protection
Operating system	Safe libraries	Antivirus, IDS	ASLR randomization Application-level access control
Programming language	Safe operations	Strict type and boundary checks	–
Compiler options	Detection and prevention of dangerous operations	Detection of common mistakes, Stack Smashing Protection	Generating more secure binary code
Source code	Development rules, source code review	Strict input validation	Software ASLR
Application	White-box / black-box testing	Black-box testing Input filtering	Patching

### How does Stack Smashing Protection work? What does Stack Smashing Protection do to function arguments?

- Fordítási időben bekapcsolható funkció, ami futásidőben detektálja a buffer overflowt
- Random da lokális változók elé, amit a támadó nem tud. Csak úgy lehet beleírni az SavedEBP, Return Addresshez, ha ezt is felülírja. Ha nem jó érték, akkor detekció történik az epilógusban a leaveret előtt.
- Függvényargumentumok a lokális változók után lemásolásra kerülnek, így ezeket nem tudjuk módosítani overflowval. A függvényünk ezeket az átmásolt változókat fogja használni.
- DE, a lokális változók felülírhatják még így is egymást, még akkor is, ha a non-buffer változókat felülre írjuk és marad több buffer value.

### How does ASLR work?

- Address Space Layout/Load Randomization

- Control-flow hijacking prevention
- OS-level ASLR: Randomly arranging the position of specific data memory regions
  - Base address of executables
  - Position of libraries
  - Stack
  - Heap
- Ez azért jó, mert a memória címek randomizálásával a támadó nem tudja, hogy pl. milyen return adresst adjon meg pontosan. (Ezért szükség lehet, egy information-leak attackra, amivel az ASLRt bypassolhatja)
- Con: 64biten van elég (~48bit) hely randomizálásra, ASLR does not protect against data corruption

### What does the NX bit (or DEP on Windows) do?

- Data Execution Prevention
- Non-Executable Stack
- Beállítható egy memóriaterületre (pl. stack), hogy onnan ne lehessen futtatni a támadó kódját. (overflow még lehetséges így is)
- Hardveresen NX bit, Szoftveresen emulálható,

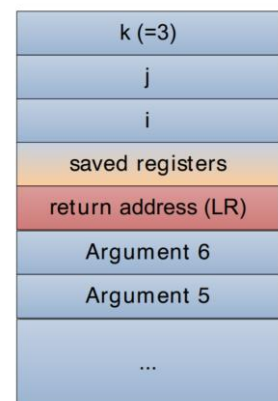
### Describe the idea of a Return to LibC attack!

- A return address egy olyan függvényre mutat(overflownál), ami a libc-ben már implementálva van. Ekkor, ha kellő paraméterekkel hívjuk meg a libc-s függvényeket, akkor elérhető, hogy látszólag eredetileg is akaratlagosan fusson le a számunkra jó kód (pl. shell indítása).
- Ez jó lehet NX bit bekapcsolása mellett, hiszen a futtatható kód, egy “amúgy is” futtatható részen van.
- Prevenció: Csúszátsuk le a libeket amit használunk vagy runtime Control-flow integrity bekapcsolása,
- Van fake frame-es return to libc, amivel a memória struktúra rendbetartása mellett, több funkciót is végre tudunk hajtani, lásd előadás

## Memory corruption II

### How does function calling work on ARM?

- Loading the first 4 arguments into registers R0-R3 (if there are more arguments, placing them on the stack)
- Calling the function (return address stored in LR)
  - Saving (push) the used registers (including LR) to the stack
  - • Allocating space for local variables (and initialization)
  - • ... function execution ...
  - • Disposing local variables
  - • Restoring (pop) caller state from the stack + returning
- Clearing any arguments from the stack



### How does a buffer overflow look like in memory on ARM?

Ugyan úgy, kicsit más a memória képe (R7? vagy bármi más), LR regiszter megjelenik a stacken, ami célpont maradhat.

Túlírásnál is ugyan az (kivéve pl. a kernel hibaüzenetében, egy bit ARM-Thumb mód közötti váltás miatt más ((75-helyett 74 volt a példában))

### What is the difference between the x86 and the ARM function calling?

- Argumentumok átadása, x86-nál stacken, ARM-en regiszterbe, majd aztán kerül stackre.
- ARM-Thumb mód utolsó bitje megjelenik a memórián
- LR regiszter és a saved registers megjelennek a stacken

### Why is the Return-to-LibC attack more difficult on ARM?

- NX (through PaX) prevents the attacker from running the injected code on the stack (or on the heap)
- The calling convention used by ARM programs passes the first four parameters in registers
- Thus, if a function has at least one parameter
  - The R0 register will be used as the first parameter
  - The R0 register will not be pushed to the stack
  - The end of the function will not pop R0 value from the stack
- Tehát, hogy Libc-t hívjunk paraméterekre van szükségünk, ahhoz hogy paraméterünk legyen olyan függvényt kéne hívni, ami a regiszterbe tölti a kért paramétert, de ehhez függvényt kell hívni, így jön a RoP.

### What is the main idea behind a ROP attack?

- Allows an attacker to execute code even if we have, Non-executable memory (NX / PaX), ASLR
- Executes carefully chosen machine code instruction sequences – called gadgets – by returning to them

### What is a ROP gadget?

- Each gadget ends in a **POP {\*,PC}** instruction to continue the execution with the subsequent gadget (address on stack)
- These gadgets can be potentially found in the codebase
  - Shared libraries are randomized
  - But most ASLR implementations cannot randomize your (non-Position Independent Executable) executable
  - gadgets can be found, and their location is stable

### How does a "real ROP scenario" look like?

- pl. `/bin/nc -l -p 1337 -e /bin/sh`, akarunk indítani egy backdoor shellt
- ekkor `cben, az execve(argv[0], argv, env);` ahol az `arg` a backdoor kódja, akarnánk futtatni
- ez pár egyszerű assembly utasítás, ami az `r0, r1, r2` regiszterekbe betölti ezeket az értékeket, majd elindítja a futtatást

- Tehát olyan gadget kell, amiben van registry fill, system call, memory write. Ilyen műveletek általában vannak a támadni kívánt kódban, amit különféle programokkal tudunk scannelni és kinyerni a támadáshoz használt gadgeteket

#### How can ROP be mitigated?

- **Pointer Encryption:** Memóriában pointer titkosítva tárolódik, közvetlenül használat előtt decrypt
- **CFI: Control Flow Integrity:** Validates program's control flow by checking at runtime all indirect branches (e.g. BX R1) against a whitelist
- **SFI: Software Fault Isolation:** Creates fault domains (sandboxes) for each untrusted module – Code running in one of these domains cannot jump to (or access) data outside its sandbox
- **Shadow stack (Safe stack):** Legyen két stack: Separating the stack to control and data (safe and unsafe) – Preventing incorrect control flow by storing the return addresses on the safe stack
- **CPI: Code Pointer Integrity:** Pointerekbe plusz információ (64 biten elfér mellé), hogy valid vagy nem valid, ellenőrző összeg. pl Pointer Authentication Code (pl. Iphone)

#### What happens in case of a format string vulnerability?

- Uncontrolled format strings can override the normal behavior of format functions
- Program crash – Viewing process memory – Writing to arbitrary memory locations

#### How can a process be crashed with a format string exploit?

- By accessing invalid pointers, the program crashes which can be useful for various reasons
- Example: `printf("%s%s%s%s%s%s%s%s%s%s")` displays the content of memory address stored in the stack, It is likely to fetch an address that we don't have privileges, the program crashes

#### How can the stack be dumped with a format string exploit?

- `%x`, Fetching parameters from the stack and displaying as hexadecimal numbers

## Malware 1

Ehhez nincsen official kérdéssor, szóval az előadásból gyártok kérdéseket, ami vizsgán nyugodt szívvel kérdezhető.

#### Mi a malware?

- malware = malicious software
- any code that can be added to a software system in order to intentionally cause harm or subvert the intended function of the system

### Milyen malware típusokat ismersz? Jellemezd őket!

- **virus:**
  - when executed, replicates itself by inserting its own copies (possibly modified) into other computer programs, data files, or the boot sector of hard drives
  - in order to function, viruses require their hosts
  - besides replicating, the virus may perform some harmful activity
- **Worm:**
  - standalone computer program that replicates itself in order to spread to other computers » unlike a virus, it does not need to attach itself to a host program/file/ medium
  - often, it uses a computer network for spreading, relying on exploitable security vulnerabilities on the target computer to infect it
  - besides replicating, the worm may perform some harmful activity
- **Trojan horse:**
  - standalone computer program that appears to perform some useful function, but it (also) performs some harmful activity

### Mi a ransomware? Hogyan működik, mi a célja? Hogyan lehetséges ellene védekezni?

- hard disk of infected computer can be encrypted and decryption key can be revealed only after some payment
- védekezés antivirussal.

### Milyen attack vectorokat ismersz?

- **e-mail attachment:** exe, zip, akár office, pdf
- **drive-by-download**
  - **drive-by-email:** malicious active content in the e-mail body
  - **link in an e-mail points to a malicious site**
  - **watering-hole attack:** attacker places malicious content on web sites likely to be visited by potential victims
- **file sharing**
- **portable media (USB drives)**
- **exploiting vulnerabilities in network services**

### Mi a malware packer? Mikor használjuk?

- Generally detection based of a known „binary sequence” of the code
- Authors of malware try to avoid easy detection They try to make the code „change itself” to avoid detection
- Packer: Most of the code is packed (compressed and/or obfuscated) and only the packer code is left unchanged

### What are the main types of malware analysis process?

- **Behavior analysis:** Using some sandbox, or real infected device and check activities by normal or specialized tools to see what is happening on the computer
- **Static analysis:** Using a disassembler check the contents of some malware. The malware is NOT executed.
- **Dynamic analysis:** With the help of tools (debugger, etc.) we execute the code, but take control of the run

### **Describe malware hunting!**

- You might want to find malware similar to your malware sample (hint on author, slightly different functions, hard coded Command and Control servers)
- An important, very specific part of the malware needs to be isolated (special code for obfuscation, special crypto)
- Signature on the very specific part should be extracted

### **What is Yara? How the rules look like?**

- YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a boolean expression which determine its logic

### **What is Conficker? Describe its functionality!**

- Conficker is a DLL
- Using the vulnerability it inserts itself into the system as a system service, also USB drives
- Update: Time-seeded random domain names are used to download encrypted binaries by HTTP
- Vulnerability: NetpwwPathCanonicalize() in netapi32.dll.

### **How Conficker update works?**

- Update: Time-seeded random domain names are used to download encrypted binaries by HTTP
- Domain flux: For the update, conficker A/B generates 250-250 random domain names, daily. C version 50.000 domain names daily
- Time synchronization: downloads web pages (google, yahoo,...) and uses the time data
- Updates are protected with RSA signatures – public key is in the bot itself – 1024 bit long in Conficker.A, 4096 bits for the other variants – The public key is a good signature to search for (bot identification)

### **What are Conficker Blacklists?**

- Conficker uses blacklist of network addresses (IP numbers) to avoid identification – And to avoid scanning low-yield networks (expecting that most of the computers are patched here)

### **How can Conficker be removed?**

- Conficker detects removal tools and tries to avoid removal
- Conficker code is packed (polymorphic) on the network or in the file system However, on the target computer the code is unpacked while running
- Easier to detect running processes
- The code is stored under random file names – not fully random (depends on the variant)
- Special flags and security settings on the file are used
- Every instance should be removed to avoid re-infection



- A trick: Conficker uses OS mutexes to avoid running multiple instances. The mutex generation is based on CRC. Might be used to avoid re-infections.

## Malware Detection

### What do we call malware?

- malware = malicious software
- any code that can be added to a software system in order to intentionally cause harm or subvert the intended function of the system

### What is a malware signature?

- Signature: short sequence of bytes, unique to each known malware

### What are the main components of a traditional anti-virus product? What are the tasks of each component?

- **Scanner Engine:** Scanning a futtatott fájl
- **Alert generator:** Alert küldése a felhasználónak malware esetén
- **Quarantine:** A fájl elzárt területre kerül, nem lehet futtatni
- **Signature Database:** Az ismert malware minták leírásai, scanning engine innen olvassa ki a signatureokat

### What is the process by which new signatures are created?

1. Malware authors: New malware
2. Users: Machine infected
3. AV vendor: Sample submitted to AV vendor
4. AV vendor: Sample analyzed
5. AV vendor: Signature generated
6. Users: Signature database update
7. Users: New detections

### What are the challenges of signature-based malware detection?

- AV vendors must have access to new malware
- Manual program analysis is tedious and time-consuming
- Automated malware development toolkits
- Encryption, packing
- Obfuscation
- Polymorphism, metamorphism
- Instruction virtualization, emulation
- Signatures can't detect new malware
- Existing malware may evade detection due to slight variations in its binary form

### What are the main components of a cloud-based anti-virus product? What are the tasks of each component?

- Users:
  - Signature db
  - Blacklist

- Whitelisted
- Quarantine
- Lightweight scanner:
- Alert generator
- AV vendor:
  - Sample database: Meglévő malwarek, signaturek hatalmas dbje
  - Intelligent malware detector: Datamining, Machine Learning

**Compare traditional and cloud-based anti-virus products! What are the key similarities and differences?**

- Similarity: Userside signatureDB, Quarantine, Alert Generator
- Diff: Cloud: AV vendor side “heavy computation” scanner, meanwhile user-side only lightweight scanner, larger DB on AV vendor side, more samples, Intelligence on cloud, more computational power.

**What kind of features can we use in order to train ML algorithms for malware detection? Name at least 5!**

- linux file tool
- static/dynamic linking
- debug symbols/Stripped binaries
- n-gram analysis
- linux strings tool
- Instruction = opcode + operands + extras

**Describe an analysis process to extract features for malware detection using ML!**

- Goal: capture the characteristics of samples using program analysis techniques
- Sample database -> Feature extraction -> Feature selection → Classification, Clustering

**What is a "controlled environment"? How would you construct one?**

- Controlled environment: interaction with environment is constantly monitored
- File system accesses
- Spawned processes
- Network communication
- Invoked system calls
- construct: Running from VM?

**Describe the machine learning problem of classification and clustering! What are the differences between their inputs?**

- **Classification:** Supervised learning, tanuló - teszhalmazok a cél a labelezés az ismeretlen mintákra, pl. káros vagy nem.
- **Clustering:** Unsupervised learning, Goal: identify groups (clusters) of similar samples without labels – Not just malicious / benign, but malware families and variants as well
- Differences: A klasszifikációnál egy előre labelezett halmazt adunk meg, amire rá tud tanulni az ML, tudja mi a jó mi a rossz és ez alapján állítja a belső layereit. A klaszterezésnél nincsenek előre beírt labellek, ott a hasonlóságot valami alapján keresi.

**What is the high-level process of training and using ML models for classification?**

- **Training phase:** Existing samples → Feature processing → Model training → Model
- **Testing phase:** Unknown sample → Model → label

**What are the characteristics of a good classifier (ML model trained for classification)?**

- A model lehet underfit, ekkor nem eléggé érzékeny a bemenetekre, ezért ismeri fel a malwaret
- Overfit: a modell rátanult a tanítóhalmazra (a hibáira), így egy új malwarehez nem elég általános
- Appropriate: Van némi hiba, de nagy valószínűséggel találja el a malwaret

**Name algorithms for constructing ML models for classification and clustering, 3 for each!**

- Classification algorithms: Decision tree, Neural network, Support vector machine, k-nearest neighbor
- Clustering: Partitioning: k-means, k-medoid, Spectral, Density based- DBSCAN

## OS security

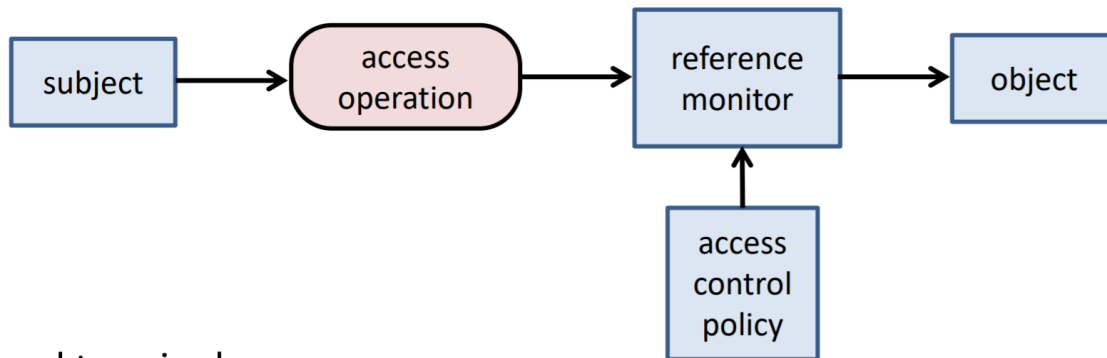
**What are the main roles of operating systems?**

- Abstraction of hardware
- Providing standardized interfaces
- Ensuring performance
- Multitasking
- Providing security

**What is the difference between monolithic kernels and microkernel architectures?**

- **Monolithic kernels:**
  - the privileged part of the kernel is large, including hardware drivers and file system services
  - this has drawbacks related to reliability and security
- **Microkernel:**
  - try to minimize the risk of crashing and getting compromised by making the privileged part of the kernel as small as possible
  - drivers can be separate processes running in user space
  - this has advantages related to reliability and security
  - but it also has some performance drawbacks

### What is the basic model of access control?



- **Subject:**
  - an active entity that tries to perform some access operation
  - typically a process running on behalf of a user or some other principal
- **Object:**
  - a passive entity representing the resource being accessed by the subject (file, channel programs etc)
- **Access operation:**
  - defines the nature of access (read, write ...)
- **Reference monitor:**
  - guards the resource by enforcing some access control policy (defines who can access what and under which circumstances)

### How this model applies in case of Linux file access control?

- objects are files – in Linux, every resource (except memory) is handled as a file
- subjects are processes running on behalf of a user (each process is associated with a real UID/GID and an effective UID/GID), user can be root for full access
- access to files is limited by the identity associated with the accessing process and the permissions assigned to the file

```

total 16
directory → drwxr-xr-x 2 boldi u
              drwxr-xr-x 3 boldi u
file →      -rw-rw---- 1 root  u
              -rwxr-xr-x 1 boldi u
              ↓      ↓      ↓
              permissions for owner
              permissions for group
              permissions for other
  
```

- File permissions rwx,
- **Reference monitor:**
  - must ensure that all security-sensitive operations are authorized by the access enforcement mechanism

### What are processes and threads? How are they created and represented by the OS?

- A running (instance of a) program is called a process
- Each process gets its own section of memory --> address space
- Processes should be confined to their address spaces
- **Threads** are “mini-processes within processes”, They are created by cloning the process state (registers, stack, and kernel state) but otherwise keeping the memory shared between the threads

- The OS represents processes internally in order to be able to manage them
- Process Control Block : Data structure used by the kernel to store information about a process
- New processes are created by special systems calls – e.g., fork() and exec() on Linux

### How does a system call work? How does the privilege level of the executing code handled when making a system call?

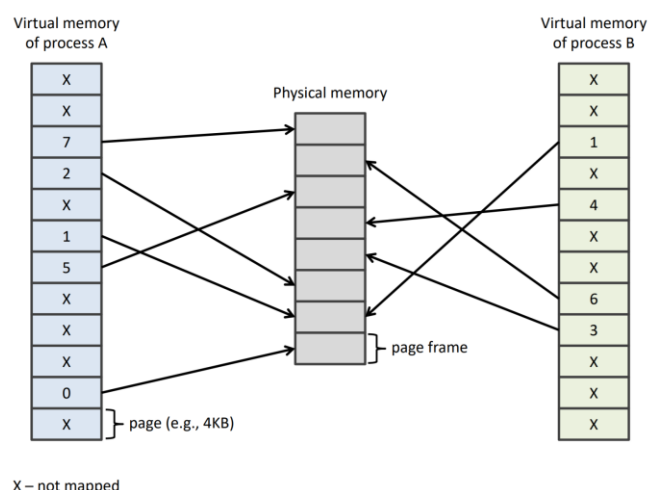
- System calls provide the mechanism by which user space processes interact with the kernel
  - Each system call is identified by a system call number, calls can have arguments and return values
- Application Binary Interface (ABI) – defines how the system call is actually invoked
- Sysenter, syscall: force the processor into a predefined privilege level 0 (sysenter) and into a predefined privilege level 3 (sysexit)
- Control is transferred to code at a fixed location in the kernel that uses the system call number to dispatch the call to the appropriate system call handler function § The system call handler is executed § The return value is put in the appropriate register(s) § Control is given back to the caller in user space
- The processor needs to keep track of the current privilege level of the code that it executes § Typically, some bits in a special register is used for this purpose § These bits cannot be set directly, but it is ensured at hardware level that only certain instructions

### What does the virtual memory abstraction mean?

- Virtual memory is an abstraction of the physical memory, which ensures that processes perceive their address space as a contiguous range of addresses § The term virtual refers to the fact that memory addresses used by processes are not real physical addresses

### How does paging work?

- Paging means that both the virtual address space and the physical memory is split into small chunks, called pages and page frames, respectively
- Pages are mapped to page frames in physical memory, but not all pages really need to be in memory at the same time



### How is memory protection realized in case of paging?

- Protection bits in the page table entry
- When a page is being accessed, the processor checks these bits, and any violation leads to a page fault exception
- In addition, processes can be isolated by ensuring that their pages do not map to the same physical memory

## Container security

Ehhez sincsenek official kérdések, azért párat gyártok.

### What is Chroot and FreeBSD Jail?

- Chroot jail is for “Change Root” and it’s considered as one of the first containerization technologies. – It allows you to isolate a process and its children from the rest of the operating system.
- FreeBSD Jail: It was intended to bring more security to the simple Chroot file isolation. – Unlike the Chroot, FreeBSD implementation isolates also the processes and their activities to a particular view of the filesystem

### Which resources can the CGroups limit and isolate?

- CPU – Memory – Disk I/O – Network, etc.

### Why LXC is important?

- LXC (Linux Containers)
- LXC is a userspace interface for the Linux kernel containment features.
- Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers.

### What are the four main components in Docker architecture?

- Docker engine
- containerd
- containerd-shim
- runC

### What's the difference between OS- and App containers?

- OS Containers where the operating system with the whole application stack is packaged
- Applications Containers that usually run a single process per container.

### What are the four major areas to consider when reviewing Docker security?

- the intrinsic security of the kernel and its support for namespaces and cgroups
- the attack surface of the Docker daemon itself
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers.

**What are Kernel namespaces?**

- Namespaces provide the first and most straightforward form of isolation: – processes running within a container cannot see, and even less affect, processes running in another container, or in the host system
- Each container also gets its own network stack – a container doesn't get privileged access to the sockets or interfaces of another container

**What is Docker Content Trust?**

- Content trust gives you the ability to verify both the integrity and the publisher of all the data received from a registry over any channel.
- Docker Content Trust (DCT) provides the ability to use digital signatures for data sent to and received from remote Docker registries.

**What is vulnerability scanning?**

- Vulnerability Scanning allows developers and development teams to review the security state of the container images and take actions to fix issues identified during the scan, resulting in more secure deployments.

## Virtualization Security

**What is virtualization? What security-related issues does virtualization introduce?**

- Virtualization: the creation of a software-based (virtual) representation of something
- Complexity issues – Virtualized systems are more complex –
  - More software -> more code -> more bugs -> more vulnerabilities
  - More updates to install (patch management)
- Management woes – Who manages what? – It becomes easier to violate the separation of duties principle as systems become more intertwined
- The security of the core environment becomes more important – Breach = bigger damage, bigger outage

**What kinds of attacks exist against virtualized environments?**

- Hypervisor detection
- Network/environment monitoring
- DoS
- VM escape
- Hyperjacking

**What is hypervisor detection? How can hypervisors be detected?**

- Goal: to detect whether the program is running in a hypervisor
- Why? – Providing better compatibility/performance – Analysis evasion (e.g. malware) – Enforcing licensing restrictions
- Via APIs – Environment-based detection – Hardware-based detection – Side-channel attacks

### **How may an attacker cause a denial of service in virtualized environments?**

- An attacker may attempt to overload or disrupt the host to cause performance degradation or service outage affecting multiple virtual machines
- CPU starvation
- Memory starvation
- Disk performance degradation
- Data loss/corruption: wasting all the space
- Network disruption

### **How can someone gain unauthorized access to virtualized environments? What are the risks? How can they be mitigated?**

- Individuals may access data without permission on a VM host – Disgruntled employees – Hackers with stolen (admin) credentials – Through vulnerabilities in the virtualization software
- DoS: turning off or deleting VMs and virtual hard disks
- Data theft
- Data manipulation
- Countermeasures – Proper physical and virtual access control – Auditing – Regular backups – vTPMs (virtual TPMs) – Shielded VMs (Microsoft Hyper-V), Encrypted VMs (VMware ESXi)

### **What is a VM escape attack? How does it work?**

- Attacks that aim to escape from the virtualized (and controlled) environment to gain access to other vms or to the host
- Via – Exploiting bugs in the VMM or its components – Exploiting bugs in the underlying hardware

### **What is hyperjacking?**

- An attack that aims to – (Silently) take over an existing hypervisor – Install a second hypervisor below (or above) the legitimate one ▪ Similar to rootkits, but even more difficult to detect ▪ May follow a virtual machine escape

## **Firmware security**

### **What are the goals of a platform firmware?**

- The firmware is software that runs before the OS for the purpose of – initializing the hardware – loading the OS
- Some firmware components remain in memory after the boot process and they are used by the OS to interact with the hardware (runtime services)

### **Why is firmware security important?**

- OS security measures and anti-virus solutions do not apply here (they are not yet loaded)
- Compromised firmware can provide an environment in which the loaded OS (no matter how secure it is) cannot run safely



- – Firmware is just software, and therefore, it is vulnerable to the same kind of threats that typically target software (e.g., buffer overflow)
- Modern firmware is a large piece of software --» likely to have bugs
- Attackers started to target firmware --» e.g., bootkits

#### **What does UEFI stand for? What is standardized by the UEFI Forum?**

- EFI (Extensible Firmware Interface) » new firmware spec developed by Intel for the first 64-bit CPUs (Intel Itanium) » it was meant to replace BIOS
- UEFI (Unified EFI) » industry alliance (UEFI Forum) formed by large companies
- Platform boot flow standardized?

#### **What are the phases of the UEFI boot process?**

- **SEC:**
  - Very first code executed by the CPU
  - minimal code fetched directly from an SPI flash
  - The SEC phase is responsible for – Handling all platform reset events (power-on, wake-up) – Executing microcode patch update to the CPU – Configuring the CPU Cache as RAM (CAR)
- **PEI:** Pre-EFI Initialization
  - PEI Dispatcher invokes PEI Modules (PEIMs) that perform early hardware and memory initialization (CPU, chipset, board init)
  - Last PEIM called is DXE IPL (Initial Program Load), which decompresses FV\_Main into main memory (RAM) and transitions to the DXE phase
- **DXE** = Driver Execution Environment:
  - Code is executed from RAM
  - DXE Dispatcher dispatches DXE and Runtime (RT) drivers from FV\_Main (in RAM) into main memory (RAM)
  - The DXE phase is responsible for –
  - Additional hardware initialization and configuration performed by DXE drivers
  - System Management Mode (SMM) set up
  - Secure Boot enforcement
  - Firmware update signature checks
- **BDS** = Boot Device Selection
  - The Boot Manager consults configuration information to decide where the OS should be booted from
  - It has access via the UEFI interface to all UEFI Boot Services that the DXE phase set up --» it can use them to access the file system on the hard drive in order to find an OS bootloader
  - If UEFI Secure Boot is turned on, it also checks the integrity of the OS bootloader before starting it
- **TSL** = Transient System Load:
  - This is typically the OS bootloader loaded from the HDD
  - The OS bootloader loads the OS kernel into memory

#### **What is the main idea of UEFI Secure Boot? What are the caveats?**

- Verify if an executable (e.g., OS bootloader) is permitted to load and execute during the UEFI boot process

- Verification is based on checking digital signatures on code (or checking code hashes)
- **Caveats:**
  - Secure Boot is optional; the way it is managed, enabled or disabled is a decision of the platform manufacturer and the system owner
  - Security boils down to managing authorized keys and hash databases
  - Flash based UEFI components (SEC, PEI, DXE Core) are not verified

#### **How are images verified by UEFI Secure Boot before loading and executing them?**

- Pass signature check by dbx (Forbidden)
- Pass signature check by db (Authorized)
- checks image hash

#### **What keys and databases are used by UEFI Secure Boot? How are they managed?**

- **Platform Key (PK):**
  - Set by the OEM when system is built in the factory
  - Its purpose is to protect the KEKs from uncontrolled modification
  - Stored on the SPI flash in the Authenticated Variables region
  - Can be updated by a secure flash update
- **Key Exchange Key (KEK)**
  - Its purpose is to protect the Secure Boot Databases (db, dbx, dbt) from unauthorized modifications
  - There can be multiple KEKs provided by the OS and other trusted 3rd party application vendors
  - New KEKs are authenticated by the PK
  - A holder of a valid KEK can insert or delete information in the Secure Boot Databases (db, dbx, dbt)
  - Stored on the SPI flash in the Authenticated Variables region

#### **What is Intel Boot Guard technology? Why is it useful?**

- Az UEFI secure boot abból indul, hogy a flashben lévő memória megbízható, ezt hivatott ellenőrizni az Intel Boot Guard
- ACM – Authenticated Code Module
  - verifies PEI code (IBB) with a key whose hash is stored in a read-only CPU register (one-time programmable fuse) --» key cannot be updated!
  - signed by Intel and signature is verified by CPU microcode using an embedded Intel key

#### **What is UEFI Secure Capsule Update and Intel BIOS Guard?**

- firmware updatelésre, Entire new firmware binary is signed by the OEM's private key
- Verification of the signature and writing the new image into flash must happen in a secure execution environment
- Additional protection is provided by Intel BIOS Guard technology
  - allows only the Intel BIOS Guard Authenticated Code Module (ACM) to program the SPI flash
  - this module performs firmware verification and update in an isolated environment (Authenticated Code RAM)

### **Why can compromising firmware flash regions be dangerous?**

- Malware injected into the firmware flash regions will not be detected by anti-virus that scans memory and hard disk
- will run on every subsequent boot
- will survive OS re-install and hard drive re-formatting
- can compromise PEI modules to disable flash updates
- can compromise DXE code to disable enforcement of UEFI Secure Boot
- can potentially compromise anything on the hard disk (from DXE)
- Authenticated Variable region of the flash, which contains keys and security databases, can also be compromised – bypass of UEFI Secure Boot verifications may be possible

### **What are SMM and SMRAM, and what concerns do they raise?**

- System Management Mode
  - Highly privileged processor mode
  - Entered through a System Management Interrupt (SMI)
  - SMM code has full visibility of the entire memory and all devices, it can modify memory contents, and even overwrite critical system files and data on storage media
  - Transition to SMM is transparent to the rest of the system
- SMRAM
  - Only accessible within an SMI
  - “Invisible” to the OS
  - Malware resident in SMRAM cannot be detected or removed by traditional anti-virus software
  - Malware resident in SMRAM has full access to all system memory and devices

## **Trusted Computing**

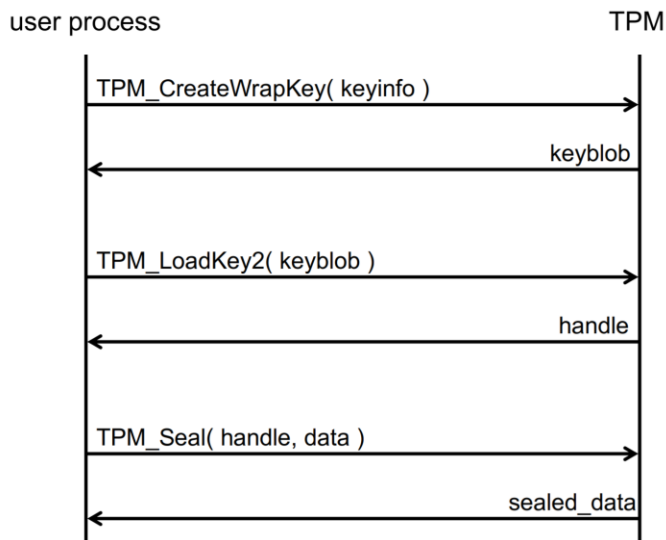
### **What is a TPM? What types of roots of trust does it provide?**

- Trusted Platform Module (TPM), embedded crypto-processor
  - the TPM enables an entire platform to ...
  - authenticate itself – using public key cryptography, involving a private key protected by the TPM and strongly tied to the platform itself – authenticate its current configuration and running software – using cryptographic hashes of code and other mechanisms
- Root of trust for measurement (RTM): a trusted implementation of a hash algorithm, responsible for the “measurement” of the platform
- Root of trust for reporting (RTR): a shielded location to hold a secret key representing a unique platform identity – the endorsement key (EK)

### **How are commands on TPM objects and resources authenticated?**

- each TPM object (e.g., a key) is associated with an authdata value, a 160-bit shared secret between a user process and the TPM – think of it as a password that has to be cited to use the object
- authdata may be a weak (guessable) secret, may be based on a user-chosen password (e.g. in Microsoft Bitlocker) – the TPM resists online guessing attacks of weak authdata by locking out a user that repeatedly tries wrong guesses
- user processes issuing a command that uses a TPM object must provide proof of knowledge of the associated authdata

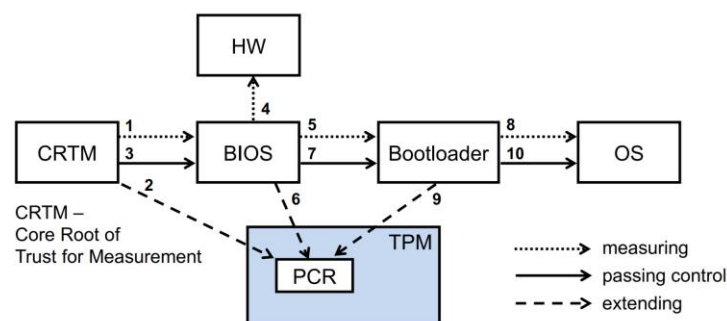
### How can the TPM be used for secure storage? (overview figure)



### What does platform measurement mean?

- measuring a component = computing its hash
- a component can “measure” another component and insert the measurement result into a Platform Configuration Register (PCR) in the TPM

### How does the TPM support measured booting? (figure)



- every component A that loads another component B and passes control to it first measures B and extends a PCR with the measurement
- the PCR then represent an accumulated measurement of the history of the executed code from power-up to the present

Every component verifies the next integrity by using TPM measure function

### What does platform attestation mean and what is its limitation?

- attestation: – the TPM can prove the state of the platform to a third party by signing the value of the PCR that represents the accumulated state of the booting process
- attestation only tells a remote party what executable code was launched on a platform, but it does not tell what is currently running – no guarantees that the launched programs are bug-free and reliable – in particular, programs could have security related bugs, and could be

#### **Why shouldn't the endorsement key be used directly for platform authentication?**

- – however, if EK was used to sign messages from the TPM, then privacy would be breached...
- EK is intended to be a long-term key and can be thought of to be the identity of the TPM

#### **What principles does the ARM TrustZone technology realize?**

- All SoC hardware and software resources are partitioned such that they exist in one of two worlds: the Secure world for the security subsystem and trusted services, and the Normal world for everything else.
- A single physical processor core can safely and efficiently execute code from both worlds in a time-sliced fashion
- This removes the need for a dedicated security co-processor, and allows for high performance security operations.
- Hardware logic present in the TrustZone-enabled system bus fabric ensures that no Secure world resources can be accessed by code running in Normal world.

#### **How are the Secure and the Normal worlds distinguished within the processor?**

- There's an extra control signal for each of the read and write channels on the main system bus ---» NS-bit (non-secure bit) § Non-secure masters set the NS bit high, which makes it impossible for them to access secure slaves

#### **How does switching between the two worlds work?**

- Setting the NS bit directly by writing the SCR is not possible § Switching between worlds is possible only via entering to monitor mode § This can be triggered by executing the Secure Monitor Call (SMC) instruction, or by a subset of hardware exceptions (interrupts)
- The trusted monitor mode software then performs the context switch securely

#### **What is a Trusted Execution Environment (TEE), and how is it different from a secure element (e.g., a TPM chip)?**

- A TEE is an isolated execution environment that provides security features such as integrity of applications running within the TEE and confidentiality of their data
- are implemented on the main processor (---» performance), but ...
- – use hardware supported isolation from the rich OS
- can have privileged access to device peripherals (e.g., fingerprint scanner) and secure elements
- device attestation

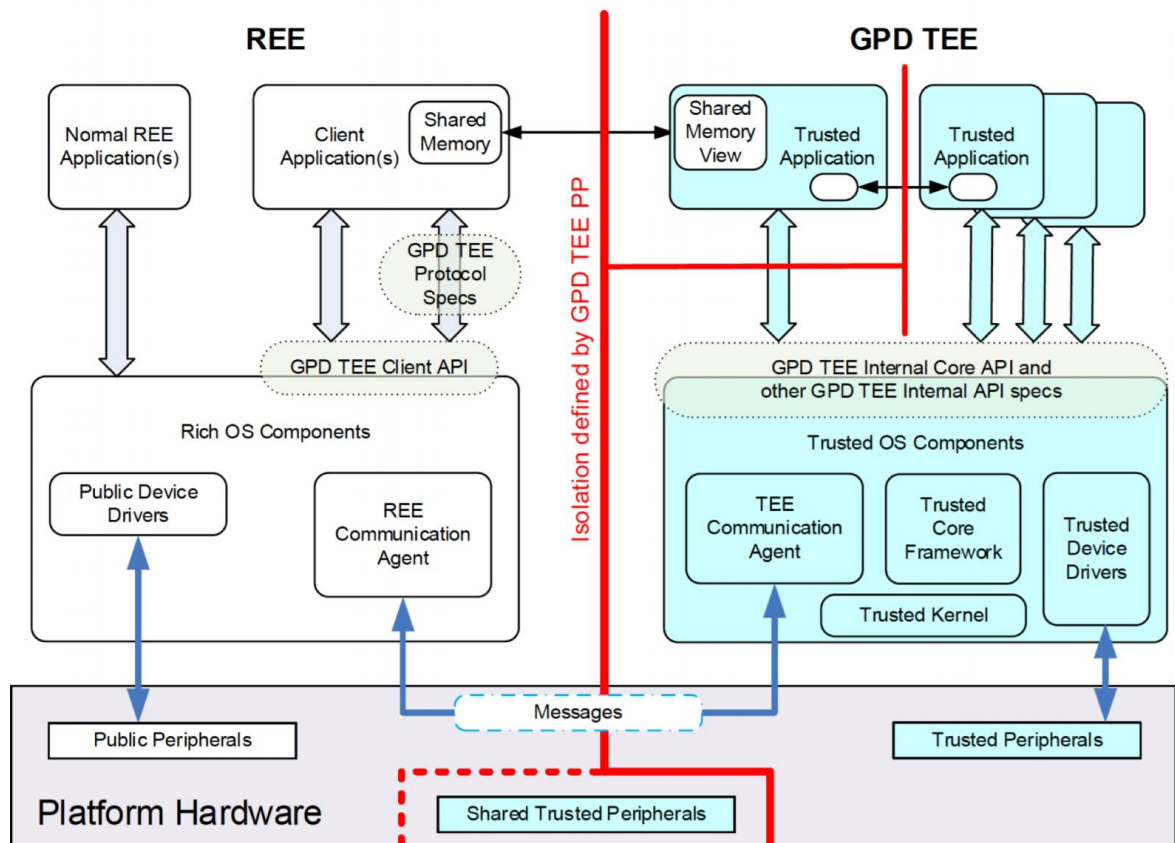
#### **What trade-off does a TEE realize? Compare the guarantees provided by a TEE to those provided by a normal OS and a tamper resistant secure element!**

Property	SE	TEE	OS with Software Protection
Level of code and data protection	Best Tamper-resistant	Better Hardware secured	Good
Memory and computation performance	Limited	Maximum	Maximum*
Executed on the main processor	No	Yes**	Yes
Secure peripheral access	No	Yes	No
Provides device attestation	Limited	Yes	No
Software ecosystem	Limited	Yes	Yes
Use case support	Limited	Unlimited	Unlimited

\*Some software protection mechanisms impact performance.

\*\*May run on a separate processor

### How does the GlobalPlatform TEE software architecture look like?



### How can Client Apps communicate with Trusted Apps?

- TAs can start execution only in response to an external command
- A session is used to logically connect multiple commands invoked in a Trusted Application
- When a Client Application creates a session with a Trusted Application, it connects to an instance of that Trusted Application

- The TEE Client API is used to establish a session with a Trusted Application, set up shared memory, send TA specific commands to invoke a trusted service, and then cleanly shut down communications

#### **Why shared memory is advantageous as a communication means?**

- Large amounts of data can be quickly and efficiently communicated between a CA and TA via a memory area accessible to both the TEE and the REE
- Care has to be taken with the security aspects of using shared memory!

#### **What does secure boot mean? How is it different from measured boot?**

- Secure boot puts the device into a known good state after reset
- the device may have been compromised at run-time --> secure boot should fix this by re-loading uncompromised software components
- software components may have been modified at run-time or while the device was powered down (e.g., overwritten flash memory) --> secure boot should detect this
- Secure boot is more than “measured boot” (see TPM slides) – loaded software layers are not only measured, but their integrity is checked and the boot process is interrupted if verifications fail – integrity verification is based on code signing (digital signatures)

#### **How can crypto operations with sensitive secrets be implemented with a TEE?**

...

#### **What is an Intel SGX enclave and what security guarantees SGX provides for enclaves?**

- It allows an application to instantiate a protected container, referred to as an enclave, in the application’s address space
- § Enclaves provide confidentiality and integrity even in the presence of privileged malware
  - Enclave memory cannot be read or written from outside the enclave regardless of the current privilege level and CPU mode. – The enclave environment cannot be entered through classic function calls, jumps, register manipulation, or stack manipulation. The only way to call an enclave function is through a new instruction that performs several protection checks. – Enclave code and data are kept encrypted in memory. The key is stored within the CPU and not accessible outside.

#### **What do enclaves contain within their address space?**

- The enclave contains a unique SHA256 hash value that identifies the code and data that was loaded into the enclave during creation
- Each enclave is also signed using a 3072 bit RSA key and the signature is stored in the enclave along with the SHA256 hash of the signer’s public key

#### **How are enclaves protected at run-time?**

- The EENTER function is the method to enter the enclave under program control § To execute EENTER, software must supply an address of a TCS (Thread Control Structure) that is part of the enclave to be entered

- The used TCS is considered busy until execution exits the enclave, and any attempt to enter an enclave through a busy TCS results in a fault
- When called, EENTER... – Checks if the specified TCS is not busy – Changes the mode of operation to be in enclave mode – Saves the old stack and base pointers and sets their new value to point in the enclaves stack – Sets the TCS busy – Transfers control to the location specified by the TCS

#### **How can enclaves protect their persistent data?**

- When loaded in memory, enclaves are encrypted and integrity protected by the Intel Memory Encryption Engine (MEE)
- When the MC accesses a page in memory that needs to be encrypted/decrypted, the crypto processing is done by the MEE transparently
- Keys are randomly generated at reset by a hardware RNG and accessible only to the MEE
- All this protects enclave code and computing state (heap and stack), which can contain sensitive secrets

## Physical Attacks and Tamper Resistant Devices

#### **What do we mean by a tamper resistant device?**

- tamper resistant computing devices are designed for carrying out operations in strict isolation from their environment – they can store and process data such that neither the data nor any execution state can be observed or modified by an attacker, even with direct physical access to the device – stored data typically include cryptographic keys that never leave (in unencrypted form) the physically secure environment provided by the device – such devices eliminate the need to physically protect an entire computing system

#### **Give some examples for application areas of tamper resistant devices!**

smart cards – TPM chips – hardware security modules

#### **How can attacks against tamper resistant devices be classified?**

- **invasive attacks:**
  - direct access to the internal components of a tamper resistant chip – often results in permanent damage
- **semi-invasive attacks:**
  - access to the chip surface, but without damaging its passivation layer – the chip remains operational – influence on internal operations by using some physical phenomenon – electrical contacts through authorized interfaces only
- **non-invasive attacks:**
  - no physical penetration to the chip –
  - local non-invasive attacks: observation or manipulation of the interaction between the device and its physical environment,



- remote non-invasive attacks: observing or manipulating only the normal input and output of the device (

**For each attack class, explain – how do those attacks work? – how can those attacks be prevented or detected?**

- **Invasive attacks:**
  - getting access to internal components of a device by removing its cover or drilling holes
  - in case of chips (integrated circuits), such access requires chemical processes to remove the chip's cover
  - probing chip internals directly
  - microscope, laser
  - Defense: sensor mesh implemented in the top metal layer, consisting of a serpentine pattern of sensor, ground and power lines
  - making it more difficult to visually analyze the chip surface
- **Semi-invasive attacks:**
  - § attacks that involve access to the chip surface, but which do not require penetration of the passivation layer or direct electrical contact to the chip internals
  - optical probing techniques to inject faults into digital circuits
  - detection of active attacks and performing some suitable alarm function, such as erasing key material
  - opaque top-layer that shields the chip from external light
- **non-invasive attacks**
  - side-channel attacks: timing attacks and power analysis
  - : Differential Power Analysis (DPA): – measure the power consumption of a chip while it performs cryptographic computation on different data à power traces
  - Defending:
    - eliminate conditional execution of steps
    - make timing and power consumption of different execution branches identical
    - use constant time elementary operations
    - make leaked information unrelated to secret data

**What levels of physical protection are defined in FIPS 140-2?**

- Level 1 – no physical security
- Level 2 – tamper evidence
- Level 3 – needs tamper resistant
- Level 4 – active tamper detection and response

**Give some examples for high-end, mid-range, and low-end tamper resistant devices!**

- High: IBM 4758 coprocessor
- Mid: smart cards, TPM chips
- Low: cheap microcontrollers

**What are the main security features of contemporary smart cards?**

- top-layer conductor meshes

- randomized ASIC-like logic design
- internal bus hardware encryption to make data analysis and access to internal memory more difficult
- light sensors to prevent an opened chip from functioning
- voltage sensors to protect against under- and over-voltages used in power glitch attacks
- clock frequency sensors to prevent attackers slowing down the clock frequency for static analysis and also from raising it for clock-glitch attacks

### **What are API attacks and how can they be prevented?**

- non-invasive attacks that can be carried out even without physical access to the device
- the API of a tamper resistant module is a software layer through which the module's functions are exposed to the external world
- attacking the API means exploiting design weaknesses of the API for extracting secrets from the device (or just increasing the efficiency of cryptanalytical attacks)
- formal analysis techniques developed for key exchange protocols may be amenable to the analysis of crypto APIs

## **IOS security**

### **How does secure boot work on iOS?**

- Ensure integrity of the software components » Lowest levels of software are not tampered with – Proceed only after verifying the chain of trust
- Secure Enclave coprocessor: separate secure boot – The Boot Progress Register (BPR) is used by the Secure Enclave to limit access to user data in different modes and is updated before entering the next boot step (processor dependent)
- Secure boot chain: – Boot Rom » read-only » hardware root of trust » Apple Root CA key – iBoot – iOS Kernel

### **What is the Secure Enclave?**

- security coprocessor
- It uses encrypted memory, Includes a hardware random number generator
- Provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised
- Runs the „Secure Enclave OS” based on an Apple-customized version of the L4 microkernel
- When the device starts up, an ephemeral memory protection key is created by the Secure Enclave Boot ROM
- The Secure Enclave memory is also authenticated with the memory protection key

### **Low level protections**

- **Kernel Integrity Protection:**
  - After the iOS kernel completes initialization, KIP is enabled to prevent modifications of kernel and driver code

- After boot completes, the memory controller denies writes to the protected physical memory region
- **System Coprocessor Integrity Protection**
  - System coprocessors are dedicated to a specific purpose, and the iOS kernel delegates many tasks to them
  - SCIP uses a mechanism similar to Kernel Integrity Protection to prevent modification of coprocessor firmware
- **Pointer Authentication Codes**
  - Pointer authentication codes (PACs) are used to protect against exploitation of memory corruption bugs.
  - System software and built-in apps use PAC to prevent modification of function pointers and return addresses

#### **What biometric authentications are available on iOS devices?**

- **Touch ID** – Allows the use of stronger passwords – Chance of random match: 1:50.000 (after 5 mismatches it is disabled) –
- **Face ID** – Allows the use of stronger passwords – Chance of random match: 1:1.000.000 (after 5 mismatches it is disabled)

#### **Describe file system encryption on iOS.**

- Data Protection is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies built into each iOS device
- File system key: generated at iOS install, constant for all files
- By setting up a device passcode, the user automatically enables Data Protection

#### **What can code signing guarantee on iOS?**

- All executable code must be signed with Apple-issued certificate
- Extends the concept of chain of trust
- Prevents the load of external code or self-modifying code
- Apps can be traced back to developers
- In-house app development with Provisioning Profiles (enterprise apps)
- Code signature checks at runtime as well

#### **What runtime process security features are available on iOS?**

- Third party apps are sandboxed
- Unique random home directory for every app (assigned at install)
- Access to any other information is only possible through iOS services
- OS partition is read-only
- Majority of iOS, as well as third party apps run as non-privileged user
- Unnecessary tools (such as remote login services) are removed
- Built-in apps and third party apps (by default) are compiled with ASLR turned on (exploitation of memory corruption is harder)

## **Android**

**Why must applications be signed?**

- All apps must be digitally signed in order to be installed

**How can applications be signed? What schemes are used? Explain and compare them.**

- The signing certificate does not need to be issued by a trusted CA – That is, the certificate may be self-signed
- v1 scheme – a standard JAR signing method – A hash is computed for each file separately, and the list of hashes is signed
- v2 scheme (APK Signature Scheme) – The entire APK is treated as one data blob
- v3 scheme - Adds support for the rotation of signing keys
- v4 scheme - Supports incremental downloads (sign parts only)

**What is rooting? How can a device be rooted?**

- Rooting is the act of gaining root permissions over the device
- The factory-installed ROM may have the option to enable root access
- Via a kernel exploit ("soft root")
- Unlocking the boot loader and installing a custom ROM with root support ("hard root")

**Name at least two advantages and disadvantages of rooting a device.**

- Interface moddability • Over/underclocking the CPU/GPU
- Unlocking the boot loader may void your warranty
- Installing a non-compatible recovery/system ROM may brick your device

**What is Google SafetyNet's Attestation API used for? Give an example**

- Attestation API – makes it possible for apps to check how 'safe' a device is
- Custom root

**What is Safe Mode (in Android)?**

- In the Safe Mode, only core applications are available – No third-party apps can be started in this mode

**What are Device Administrators?**

- Apps with Device Administrator permissions have the ability to – Set rules regarding how the lock screen works – Require strong passwords – Require that storage be encrypted – Disable the camera – Wipe the device after X failed login attempts – Remotely wipe the device

**What would you do if you lost your phone?**

- cry
- Find My Device app, Lets you – See where your phone was seen last

**What do you know about the Android Debug Bridge?**

- An interface that makes it possible to install, run, and debug applications on a physical device, as well as upload and download files and invoke device features from a computer via USB

**What is the purpose of Verified Boot?**

- Each component, starting from a hardware root of trust, verifies the next component in the boot chain
- Provides a means of verifying whether anything from the boot loader to the operating system was tampered with

**As a developer, why is it important to protect your mobile apps?**

- Loss of revenue (hacked/pirated apps) – Unhappy users (cheating in games) – Bad publicity, fines (security breaches, user data leaks)

**List some of the best practices that can help you mitigate risks.**

- Follow best practices – Attempt to detect tampering and hostile environments – Apply obfuscation – Perform regular security audits
- Know the language, the framework, the libraries, and the platform you're using
- ▪ Follow coding standards and guidelines
- ▪ Do not store anything sensitive in the application

**Why would you want to detect hostile environments? How can you do so?**

- Root/bootloader unlock detection
- ▪ Tamper detection

**What is obfuscation? How does it work?**

- Obfuscation is the process of making it difficult to decompile, analyze, and understand machine code