PERSONAL - Security Engineering

Security Requirements

- **confidentiality**: protect sensitive data from unauthorized access
- **integrity**: information has not been altered without proper authorization
- **availability**: maintain operational readiness of a system
- **non-repudiation**: impossibility to inappropriately deny an action
- **auditability**: ability to reconstruct earlier states of a system
- **accountability**: ability to hold an entity responsible for its actions
- **privacy**: security of personal information; a person has control over which information is generated, stored, processed, deleted and by whom
- **anonimity**: the identity of an entity is hidden

Security Principles

- **security analyses**: hypothesize threat / attacker and systematically try to counter that threat
- **security requirement**: the specific *needs or conditions* that must be met to ensure the protection of a system (e.g. confidentiality, availability, integrity...)
- **security mechanisms**: *tools and methods* used to enforce security requirements (e.g. cryptography, policies...)
- **security principles**: *abstract guidelines* that provide high-level direction for designing and analyzing security mechanisms and their trade-offs (*wisdom + best practices*, but *not direct design solutions!*)

  - **least privilege**: every subject should *not have more privileges than necessary* to complete its job
  - **complete mediation**: access to every object must be *controlled* in a way not circumventable (i.e. checked to be sure it's allowed)
  - **implicit deny**: security measures should start in a *secure state* and return to a *secure default state* in case of failure (variant: default decision is lack of access)
  - **compartmentalization**: organize resources into *isolated groups* with limited means of communication
  - **minimum exposure**: *minimize* the "*attack surface*" the system presents, by e.g. minimizing external interfaces, limiting information given etc.
  - **open design**: the security of a mechanism should *not* depend on the *secrecy of its design / implementation* (i.e. no security through obscurity)
  - **economy of mechanism**: security mechanisms should be as simple as possible ($\rightarrow$ less bugs)
  - **defense in depth**: employ multiple layers of mechanisms to hinder any potential attacks (the more layers, the more difficult to compromise, the less likely attackers try to break it)
  - **least common mechanism**: mechanisms used to access resources should not be shared
  - **psychological acceptability**: mechanisms should not make the resource more difficult to access (than if they didn't exist)

Security, Usability, Psychology

- **weakest link**: the security of a system is "a chain; it is as strong as its *weakest link*" (typically *humans*)
- **password problem**: *high usability* means *easy to remember*; *high security* means *difficult to guess*; can passwords be *both*?
  - **authorization**: grant access to resource
  - **authentication**: prove claim of being someone using *authentication factors*
    - **knowledge factors**: something you *know* (e.g. password, security questions)
    - **ownership factors**: something you *have* (e.g. physical passkey, ID)
    - **biometric factors**: something you *are* (e.g. retinal scan, fingerprints)
    - **location-based factors**: *somewhere* you are (e.g. current location for car keys)
    - **time-based factors**: *sometime* you are in
  - **multi-factor authentication**: use more than one factor to *authenticate* user
  - **password manager**: solution, store (long and complex generated) passwords securely using master password

Social Engineering

- **social engineering**: manipulate victim to perform bad actions and / or reveal information

Principles (Cialdini)

- **reciprocity**: people tend to return favors
- **commitment and consistency**: if people openly commit to something, they are more likely to honor that commitment
- **social proof**: people will do things they see others doing
- **authority**: people tend to obey authority figures
- **liking**: people are easily persuaded by other people whom they like / know
- **scarcity**: perceived scarcity generates demand

Taxonomy

- **type**: what?
  - **phyiscal**: attacker performs a physical action
  - **social**: attacker relies on socio-psychological persuasion tactics (see *Cialdini's principles*)
  - **technical**: attacker performs technical actions, typically over the internet
  - **socio-technical**: social + technical
- **channel**: how?
  - **e-mail**: phishing, reverse social engineering
  - **instant messaging**: phishing, reverse social engineering, identity theft
  - **telephone, VoIP**: where a victim might delivering sensitive information
  - **social networks**: fake identities
  - **cloud**: situational awareness of a collaboration scenario
  - **websites**: watering hole, phishing (fake websites)
- **operator**: who?
  - **human**: attack conducted by a person on a limited number of targets
  - **software**: (automatic) attack

Attack Vectors

- **dumpster diving**: sifting through trash to find sensitive information
- **shoulder surfing**: direct observation techniques, e.g. looking over someone's shoulder at their screen
- **reverse social engineering**: establish trust between attacker and victim; convince victim to reach out to attacker
- **waterholing**: compromise a website that is likely to be of interest to the chosen victim(s)
- **advanced persistent threat (APT)**: long-term, internet-based espionage attack
- **baiting**: malware-infected storage medium left in a location where it is likely to be found by the targeted victim(s)

Examples

- **phishing**: impersonate legitimate organizations or individuals to trick victim into revealing sensitive information
  - **spear phishing**: focus on an individual rather than a wide collection of people (*general* phishing)
  - **dynamite phishing**: additionally use personal information to make it more convincing
  - **smishing**: phishing via text messages (SMS)

Threat Modeling

- **power**: what can an attacker do?
- **cost**: how expensive is the attack for them?
- **incentive**: what do they get from attacking us?
- **actions**: what are they doing to reach their goal?
- **countermeasures**: what can we do to fight them?

ISG Terminology

- **vulnerability**: a flaw or weakness in a system, process, or control that could be exploited to cause harm (i.e. what *could let something bad* happen, what could *enable* a threat)
- **attack**: intentional act by which an entity attempts to evade security services and violate the security policy of a system
- **attacker**: the person performing the attack
- **attack vector**: path or means by which an attacker gains access to a system or network
- **threat**: a potential cause for security violation (i.e. what *could* happen if the vulnerability is exploited)
- **asset**: anything of value to an organization (e.g. data, systems, personnel...)
- **risk**: expectation of loss when a threat exploits a vulnerability
- **countermeasure**: measure that opposes a threat, vulnerability or attack

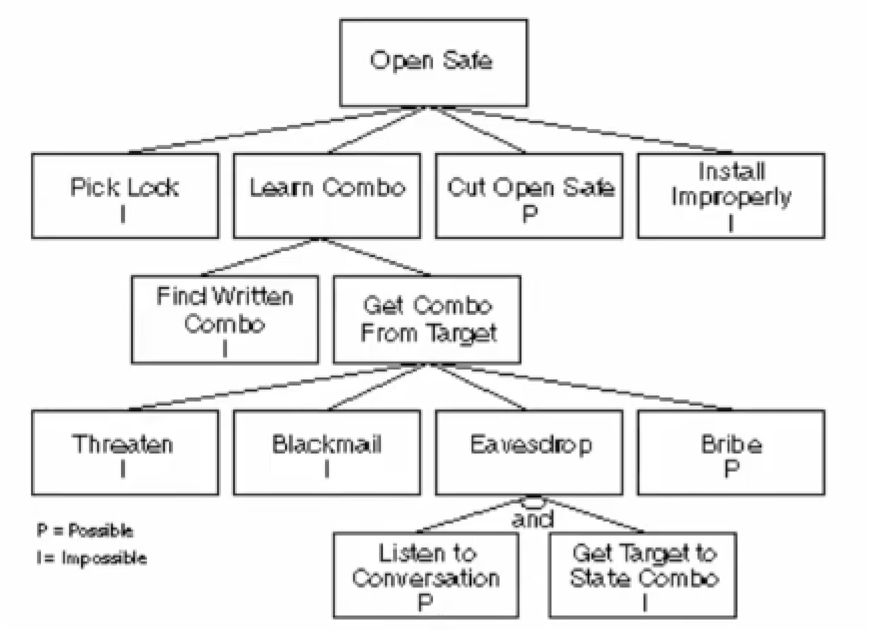
Attacker Models

- **crypto systems**
  - **known-plaintext (KPA)**: attacker knows one or more pairs of plaintexts and their corresponding ciphertexts, using these to attempt to deduce the key or learn enough about the cipher to decrypt other messages
  - **chosen-plaintext (CPA)**: attacker submits arbitrary plaintexts of their choosing to an encryption oracle and obtains the resulting ciphertexts
  - **ciphertext-only (COA)**: attacker only knows the ciphertext (weakest model), attempt to exploit redundancy or structural weaknesses

- **interaction protocols**: between 2 parties
  - **honest-but-curious users**: follow the protocol *honestly* and *obey rules* but may collude to learn more than they should
  - **malicious users**: *break rules* to get more information
- **eavesdropping**: *listen in* on a shared channel; can be blocked using encryption
- **man-in-the-middle (MITM)**: someone who can *eavesdrop, modify, delete and inject* information in a "conversation"
  - **format**: $\text{defender} \xleftrightarrow{\text{message}} \text{attacker} \xleftrightarrow{\text{message}} \text{defender}$
- **remote attacker**: exploits vulnerabilities
  - **format**: $\text{attacker} \xrightarrow{\text{inputs}} \text{defender}$
- **man-at-the-end (MATE)**: the adversary has *full control* over or visibility into the end-point environment (e.g. application); can observe and manipulate code, perform reverse engineering and extraction, bypass protocols etc.
  - **format**: $\text{defender} \xrightarrow{\text{software}} \text{attacker}$
- **malware**: a malicious program which the attacker wants to run on a victim's machine which can have a different functionality based on the goal of the attacker
- **side-channel attack**: attacker uses *any* source of information to get information about a victim, typically external factors (e.g. power consumption, time taken etc.)
- **social engineering**: manipulate victim by exploiting human psychological factors

Attack Trees

- **attack tree**: representation of an attacker's plan to achieve a goal
  - **nodes**: threats, with assigned attributes (probabilities, possibilities, estimated cost)
  - **top**: goal; **lower**: specifics (how?)



Risk Analysis

- **risk analysis**: consider crucial assets $\rightarrow$ what happens if CIA of these assets is violated $\rightarrow$ how could this have happened $\rightarrow$ think about and deploy countermeasures

STRIDE

- **STRIDE model**: a model for identifying computer security threats
  - **spoofing**: attacker identifies themselves as another person / entity
  - **tampering**: attacker can manipulate data which they shouldn't be able to
  - **repudiation**: even if an attacker is caught, we cannot prove they have done it
  - **information disclosure**: attacker can read private / confidential data
  - **denial of service**: attacker can stop system from working
  - **elevation of privilege**: attacker can get more rights than they should
- **steps**:
  1. **identify objectives**: what assets? what compliance requirements? what quality requirements? etc.
  2. **create application overview**: identify application's key functionality and characteristics
  3. **decompose application**: identify trust boundaries, data flows etc.
  4. **identify threats**: identify threats and attacks that might affect application and compromise your security objectives
  5. **identify vulnerabilities**: authentication, authorization

ATT&CK

- **ATT&CK matrix**: knowledge base of adversary tactics and techniques based on real-world observations

Web App Security

- **OWASP Top 10 (2021)**: top 10 most *critical* vulnerabilities (2025 TBD)
  1. **broken access control**: restrictions not properly enforced
  2. **cryptographic failures**: no proper protection
  3. **injection**: send untrusted data as part of command or query
  4. **insecure design**: broad
  5. **security misconfiguration**: insecure default / incomplete / ad hoc configurations
  6. **vulnerable / outdated components**: self-explanatory
  7. **identification / authentication failures**: authentication and session management improperly implemented
  8. **software / data integrity failures**: insecure CI/CD pipeline

9. **security logging / monitoring failures**: attackers can maintain persistence and expand their domain of influence
10. **server-side request forgery (SSRF)**: web app is fetching a remote resource without validating the user-supplied URL

- **vulnerable and outdated components**: pay attention to dependencies; vulnerability in *one* dependency $\longrightarrow$ vulnerability in *entire* software
  - **solutions**: remove unused dependencies, keep bill of software, regularly check CVEs
- **broken access control**: usually misconfiguration that allows directory listing and / or traversal
  - **insecure direct object reference (IDOR)**: attackers can access or modify objects by manipulating identifiers used in a web application's URLs or parameters (e.g. user `123` can view `124`'s data on `example.com/users/124`, no further checks)
  - **solutions**: properly enforce access rights, use UUIDs
- **security misconfiguration**: directory / path traversal (a.k.a. `../` attack)
  - **solutions**: restrict user access, use hardcoded paths, surround user input with own path code
- **server-side request forgery (SSRF)**: trick a server into making a request on the attacker's behalf (attacker sends a crafted URL to an application endpoint that fetches remote resources; because the server performs the request, it can access internal-only systems or bypass network firewalls)
  - **solutions**: input sanitation, don't send raw responses to clients, use whitelists when accessing internal IPs
- **cross-site request forgery (CSRF / XSRF)**: trick a logged-in user’s browser into making unwanted actions on a web application in which they’re authenticated
  - **solutions**: XSRF tokens, `SameSite` attribute
- **clickjacking**: overlay fake website with hidden frame of legit website
  - **solution**: disallow embedding of content by potentially hostile pages
- **(BONUS) SQL Injection**: pass malicious SQL queries to unchecked inputs
  - **blind SQL injection**: iterative SQL injection using an oracle which only reports back *true* or *false*
  - **solutions**: input sanitation, *prepared statements*
- **(BONUS) cross-site scripting (XSS)**: force website to execute malicious JavaScript code client-side
  - **non-persistent (reflected)**: one-time (e.g. as parameter in URL)
  - **persistent (stored)**: stored on website (e.g. comment with JavaScript code in it)
  - **DOM-based**: payload executed client-side due to unsafe / unchecked HTML manipulation (e.g. raw `innerHTML`)
  - **solutions**: content-security-policy (CSP), `HTTPonly` cookies, input sanitation

- **problems with input sanitation**: whitelist / blacklist approach, encoding (HTML vs. URL vs. JavaScript)

System-Level Security

- **buffer overflow**: a program reads from or writes data to a buffer *beyond* the buffer's allocated memory
    - **problem**: Von-Neumann architecture $\rightarrow$ program and data share the same memory $\rightarrow$ data beyond this buffer can be interpreted as code
    - **classic `char` buffer example**: `b...b|f...f|r...r` (high $\rightarrow$ low, `b`: buffer, `f`: stack frame pointer, `r`: return address $\leftarrow$ overwrite this!)
    - **solutions**: use memory-safe langauges, defensive programming (safe functions)
- **supply chain attacks**: the injection of malicious code into a software package (library) in order to compromise dependent systems further down the chain
    - **vulnerable package**: a package that *unintentionally* contains a security vulnerability (e.g. Log4J)
    - **malicious package**: a package that *intentionally* contains a security vulnerability (e.g. `xz`)
    - **problem**: just because a piece of software is open-source, doesn't mean that people will *actually* look at / check the code for vulnerabilities
    - **solutions**: write the code yourself, keep log, don't use outdated packages, check CVEs etc.
- **attacks on self-made cryptography**: don't roll your own crypto!

Obfuscation

- **obfuscation**: transform program P into P' from which it is harder to extract information than from P
    - **static obfuscation**: remain fixed at runtime, make static analysis more difficult, can be attacked through dynamic techniques
    - **dynamic obfuscation (self-modifying code)**: programs keep changing at runtime
    - **targets**: layout (identifiers, code layout), data (data structures), control flow (algorithms)
- **reverse engineering**: extract data or model by inspecting low level description and / or behavior

Static Obfuscation

Intellectual Property (Code, Algorithms)

- **scramble identifiers**: replace identifiers (variable names, function names etc.) with random strings (e.g. `sum` $\rightarrow$ `f8df3e0b12a`)
- **instruction substitution**: replace binary operation with something functionally equivalent but more complicated (e.g. `a = b + c;` $\rightarrow$ `r = rand(); a = b + r; a = a + c; a = a - r;`)

- **garbage code insertion**: opposite of dead code removal; insert code which functionally changes nothing
- **merging and splitting functions**: combining / dividing code of multiple / one function into a single / more function(s)
- **opaque predicates**: a boolean expression whose value is known to the obfuscator but is hard for an attacker to infer
  - **purpose**: insert bogus control-flow (dead / superfluous branches)
  - can be broken using **abstract interpretation** (systematically analyzing predicates using mathematical reasoning and symbolic manipulation)
  - P^T: opaquely true predicate (always true) (e.g. `(x * x + x) % 2 == 0`)
  - P^F: opaquely false predicate (always false) (e.g. `x * x < 0`)
  - $P^?$: opaquely intermediate predicate (either true or false) (e.g. `x % 2 == 0`)
- **control flow flattening**: remove control flow structure of functions
  1. put each basic block as a case in a switch statement
  2. wrap the switch in an infinite loop

  - **basic block**: a straight-line code sequence with *no branches in* except to the *entry* and *no branches out* except at the *exit*
  - **optimization**: keep tight loops as one switch entry, use gcc's address of labels (`&&`)
  - **attack**: rebuild original CFG
  - **solutions**: opaque expressions
- **virtualization obfuscation**: obfuscate algorithm using own (random) ISA and emulator
  - **prep**: split program into basic blocks (`goto`s)

  1. generate random bytecode ISA L covering all instructions in P
  2. translate P to L
  3. create emulator to interpret L on host machine

  - **pros**: random ISA, implicit control flow flattening, flexibility to add other obfuscation techniques on top

Example: Control Flow Flattening

```c
/* before */
int gcd(int a, int b) {
    while (a != b) {   // B0
        if (a > b) {   // B1
            a = a - b; // B3
        } else {
            b = b - a; // B4
        }
    }
```

```
    return a;           // B2
}

/* layout */
B0: while (a != b) goto B1; goto B4;
B1: if (a > b) goto B2; goto B3;
B2: a = a - b; goto B0;
B3: b = b - a; goto B0;
B4: return a;

/* after */
int gcd(int a, int b) {
    int next = 0;
    while (true) {
        switch (next) {
            case 0:  if (a != b) next = 1; else next = 2; break;
            case 1:  if (a > b)  next = 3; else next = 4; break;
            case 2:  return a;
            case 3:  a = a - b;  next = 0; break;
            case 4:  b = b - a;  next = 0; break;
            default: break;
        }
    }
}
```

Example: Virtualization Obfuscation

```
/* before */
void foo(int x) {
    int y = 10;         // B0, integer assignment
    y++; y++;           // B0, integer increment
    if (x > 0) {        // B0, branch if > 0
        y++;            // B2, integer increment
    }                   //
    else {}             // B1
    printf("%d\n", y); // B3, call to printf with integer argument
}

/* step 1.1: ISA  */
/* integer assignment */  52, LHop, RHop   ;
/* integer increment  */  03, op           ;
/* branch if > 0      */  08, op   , offset ;
/* call to printf     */  18, op           ;
/* halt               */  00               ;
```

```
/* step 1.2: data */
int data[] = {
    00, // x
    00, // y
    10, // const. 10
    05  // jump offset (in bytes)
};

/* step 2: translate */
int code[] = {
    52, 01, 02, // y = 10;
    03, 01,     // y++;
    03, 01,     // y++;
    08, 00, 03, // x > 0...
    03, 01,     // y++;
    18, 01, 00  // printf(...);
};

/* step 3: emulator */
/* see virtualization techniques */
```

Secret Data

- **opaque expressions**: generalization of opaque predicates to arbitrary values
    - $E^{=v}$: opaque expression of value v*table-lookup*
    - **array aliasing**: replace data with equivalent data with respect to a *statically initialized array* where certain invariants hold (e.g. every 3rd value starting from 0 is $\equiv_7 3$)
- **white-box cryptography**: implementing *cryptographic algorithms* in such a way that the *secret keys remain hidden* even when an adversary has *full access* to the implementation
    - hide keys by transforming the cipher into a giant *table-lookup* so that there's no obvious key material in memory or code

Example: Array Aliasing

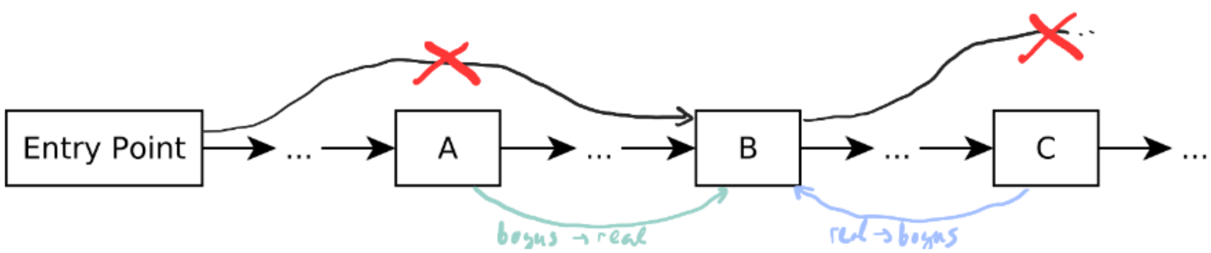
```
/* every third cell starting from 0 has a value 3 mod 7  (*) */
/* every third cell starting from 1 has a value 5 mod 11 (!) */
/* every other cell contains some fixed value            (X) */
int g[] = {10, 5, 13, 3, 27, 5, 24, 38, 0, 73, 115, 3, 66, 60, 17, 31};
//         *   !  X   *  !   X  *   !   X  *   !    X  *   !   X   *
//         0   1  2   3  4   5  6   7   8  9   10   11 12  13  14  15

int gcd(int a, int b) {
    ...
```

```
    next = g[3] % g[11] - g[8];           // 3 % 3 - 0            = 0 - 0 = 0
    next = 3 * g[11] - 4 * (g[4] % g[5]); // 3 * 3 - 4 * (27 % 5) = 9 - 4 * 2 = 9 - 8 = 1
    next = g[5] - g[3];                   // 5 - 3 = 2
    next = g[2] % g[1];                   // 13 % 5 = 3
    next = g[15] - g[4];                  // 31 - 27 = 4
    ...
}
```

Dynamic Obfuscation

- **software diversity**: every customer gets a different "version" of a software
  - **pre-distribution software diversity**: *different binaries* generated and distributed by developer (static)
  - **post-distribution software diversity**: all users get same binary, which contains *self-modifying code* (dynamic)
- **self-modifying code**: two phases
  1. **compile-time**: transform program into *initial configuration* and add *runtime code transformer*
  2. **runtime**: interleave execution of program with transformer calls (changes code segment at runtime)
- **replacing instructions**: prevent code recovery via memory snapshot
  - **idea**: replace real instruction with bogus instruction; right *before* execution, replace bogus with real; right *after*, replace real with bogus
  - **implementation**: choose 3 points A, B, C in CFG, all paths *to* B must flow through A and all paths *from* B must flow through C
    - A replaces bogus instruction in B with real
    - C replaces real instruction in B with bogus



- **dynamic code merging**: keep code in constant flux
  - **idea**: two or more function share same location in memory, create template; before a function is called, patch memory using *edit script* to load it

- **implementation (f_1, f_2)**: create template T with same size as largest function
  - T contains *values* at memory offsets which are *common* for f_1, f_2
  - T contains *wildcard values* at memory offsets which are not common
  - edit scripts e_1, e_2 *replace wildcards* of T to load f_1, f_2 respectively
- **dynamic decryption and reencryption**: at some point in basic block, decrypt next block, jump, then encrypt previous basic block

Example: Dynamic Code Merging

f_1				
0	1	2	3	4
`b7`	**`48`**	**`a0`**	`53`	`fa`

f_2			
0	1	2	3
`e9`	**`48`**	**`a0`**	`33`

T				
0	1	2	3	4
?	**`48`**	**`a0`**	?	?

$e_1 = \{0 \rightarrow \texttt{b7}, 3 \rightarrow \texttt{53}, 4 \rightarrow \texttt{fa}\}$
$e_2 = \{0 \rightarrow \texttt{e9}, 3 \rightarrow \texttt{33}\}$

Collberg's Taxonomy

- **Collberg's taxonomy**: metric for evaluating code obfuscation techniques
  - **potency**: comprehensibility of code by humans
  - **stealth**: identifiability of obfuscated code
  - **resilience**: resistence against automatic deobfuscation
  - **cost**: performance and resource overhead of obfuscation

Software-Based Integrity Checking

- **software-based integrity checking**: routines integrated into program itself, obfuscated
- **program signing / signature verification**: only effective against *static* patching attacks (i.e. *before* program is even loaded into memory)

Code Integrity

- **self-checksumming**: check integrity of *code* by hashing and verifying segments (with *expected* hashes) periodically at runtime
    - **post-compilation**: addresses unknown before compilation, architecture-dependent...
    - **Chang / Atallah**: harden with a network of *checkers* (check eachother) and *repairers* (repair tampered regions) and *hide hashes*
    - **cyclic checkers**: $B_1 \xrightarrow{\text{checks}} B_2 \xrightarrow{\text{checks}} B_3 \xrightarrow{\text{checks}} B_1 \xrightarrow{\text{checks}}$ ... (hash values in cycles replaced with placeholders to be patched)
    - **attack**: pattern matching
    - **solution**: obfuscation
    - **attack**: memory split attack
        - **constitute two memories**: *untampered memory* for reads (self-checksumming checks) and *tampered memory* for fetches (execution)
        - **alter OS kernel** such that all *code segment* reads read the *untampered memory*, while *instruction fetches* read the *tampered memory*
    - **solution**: self-modifying code (*if* the program code pages are writeable), OH + SROH
    - **attack**: dynamic taint propagation (to detect self-checksumming routines)
        - **taint addresses of instructions**, taint *backwards* to see *where* the code came from, trace *forward* to see how the code is being *checked*

State Integrity

- **state inspection**: check program states
    - **function return values**: analyze return values of functions to ensure they are correct
    - **stack trace**: check stack trace to verify history of function calls (protects against *changeware* by checking trace that leads to crypto function calls)
    - **hardware performance counters**: two-phase
        - **protection phase**: capture profiling information and inject verifiers with *thresholds*
        - **execution phase**: collect runtime information and evaluate
    - **oblivious hashing (OH)**: hash execution traces (instruction sequence + memory references) (~0.3%)
        - **problem**: only works for computations which are independent of the input
        - **solution**: short-range oblivious hashing (SROH) (~3.1%) to protect *data independent instructions*
            - **data dependent instruction (DDI)**: at least one *operand* depends on input data
            - **control flow dependent instruction (CFDI)**: the *condition*, which leads to the branch that the instruction resides in being taken, depends on input data

        - **data independent instruction (DII)**: may be control flow dependent but *not* data dependent **(!)**
- **virtual self-checksumming**: self-checksumming on virtual obfuscated code
  - **no post-compilation patching needed**: hashes computed on *(virtual) code array*

Hardware-Based Software Protection

- **protection pyramid (top $\rightarrow$ bottom)**: ensure that systems are intact at all times
  1. **attestation**: a mechanism to prove to a remote party that your operating system and application software are intact and
     trustworthy
     - **device identification**: only certain machines are allowed in a certain space (**note**: IP and MAC addresses do NOT count as secure attestation, because they can be spoofed)
     - **secure generation of cryptographic keys**: poor key generation can lead to system security violations
     - **secure key storage**: sensitive keys must be secured from external software
     - **device health**: check machines for possible compromises
  2. **secure storage**: prevent the disclosure of sensitive information
  3. **secure execution**: mitigate threats related to the exposure of systems
- **static security**: system components constitute a *hash chain* (i.e. if the system starts secure, it stays secure)
- **dynamic security**: CPU-level security mechanism protects programs through execution (i.e. even if the system starts correctly, something can be changed at runtime)

Static Security

- **trusted platform module (TPM)**: secure cryptoprocessor typically used for verifying that the boot process starts from a *trusted combination of hardware and software* and *storing disk encryption keys*
  - **platform configuration register (PCR)**: stores hash of part of the running hardware / software stack, to be used in checksums later via software
  - **hash chain**: method of providing attestation
    - TPM $\rightarrow$ boot loader $\rightarrow$ OS $\rightarrow$ application(s)
  - **features**:
    - **hardware RNG**: cryptographically secure hardware-based psuedo-random number generator
    - **key generator**: secure generation of cryptographic keys for limited uses
    - **remote attestation**: creates a nearly unforgeable hash key summary of the hardware and software configuration to verify that the hardware and software have not been changed

    - **binding**: data is encrypted using the TPM bind key; create cryptographic keys and encrypt them so that they can only be decrypted by the TPM
  - **keys**:
    - **endorsement key (EK / RSA)**: ID of TPM, generated upon production and used for establishing trust, *never used for signing*
    - **storage root key (SRK)**: root of trust used to encrypt other TPM-generated keys (hierarchical), generated upon installation
    - **attestation identity keys (AIKs)**: used merely to sign for attestation (proving system state via PCRs)
    - **other**: binding, sealing, signing...

Dynamic Security

- **enclave-based security**: CPU-level security enforcement protects programs throughout execution
  - **Intel SGX / ARM TrustZone**: CPU-enabled code enclaves, where the code and memory remain *encrypted* (*sealed*) before, during and after execution in RAM
    - **vulnerable** to side-channel attacks

Side-Channel Attacks

- **side-channel-attack**: any attack based on information gained from the *implementation* of a system rather than weaknesses in the design or algorithm itself
- **power consumption**: observe peaks based on power usage
- **timing attack**: analyze time taken by an algorithm; can differ based on input
  - **solution**: remove time dependencies
- **oracle attack**: manipulate input to extract information from an oracle (which can only answer yes or no)
  - **solution**: close side-channel (i.e. shut up)
- **search for keys in memory**: keys should have high entropy, so look for high randomness in memory dump
- **cache attacks**: exploit time differences when accessing memory
- **cold-boot attack**: cool down DRAM to preserve content after reboot, then look for key
  - **solution**: don't store keys in memory / encrypt keys in memory using some other key stored elsewhere
- **attacks on air-gapped systems**: not connected to any network
  - **optical**: powerful cameras
  - **Stuxnet**: worm, spread via USB sticks

    - **AirHopper**: malware on infected system generates FM radio signals, which are then decoded by an infected mobile phone
    - **BitWhisper**: use heat emissions as a communication channel between two infected computers in close proximity
    - **PowerHammer**: tap and analyze electromagnetic emissions of compromised computer
    - **LANTENNA**: encode data over radio waves
- **LLMs**: prompt injection, data leakage, inadequate sandboxing, training data poisoning

Privacy-Enhancing Technologies

- **privacy**: the right of users of preserve the *confidentiality* of certain data or actions, while maintaining the *functionality* of systems
    - **contradictions**: impact on *functionality* (e.g. hiding precise location renders finding nearest point of interest useless), *accountability* (allowing anonymous attacks), *efficency* (e.g. routing through TOR)

Anonymization

- **key attributes**: uniquely identifying information (e.g. name, address, phone number...)
- **quasi-identifiers**: combination of attributes that can be used to identify users (e.g. {ZIP, birth date, gender})
- **sensitive attributes**: as the name implies (e.g. medical records, salaries...)
- **k-anonymity**: the information for each person *cannot* be distinguished from at least $k-1$ other people in the same release
    - **formally**: a table is k-anonymous if *any quasi-identifier* present in the released table appears in *at least k records* (rows)
    - **generalization**: replace quasi-identifiers with less specific values (e.g. $47677, 47602, 47678 \to 476^{**}$)
    - **supression**: blunt the data ($\ldots \to *$)
    - **attacks**: k-anonymity doesn't work if sensitive values in an equivalence class *lack diversity* or if the attacker has *background knowledge*
        - **homogeneity attack**: the sensitive attribute is the same for every entry in an equivalence class
        - **background knowledge attack**: additionally use some other knowledge / educated guess
- **L-diversity**: extension of k-diversity, taking the previous attacks into account
    - **formally**: a *table block (equivalence class)* is L-diverse if there are *at least L different* values of the *sensitive attribute*
        - a *table* is L-diverse if *every block* in the table is L-diverse

- a background knowledge attack can only succeed if the attacker has $L-1$ background knowledge
- **probabilistic**: frequency of most frequent value bounded by $\frac{1}{L}$
- **entropy**: entropy of the distribution of sensitive values in each class is at least $\log(L)$
- **recursive (c, L)-diversity**: makes sure that the most frequent value does not appear *too* frequently in a block
- **attacks**: skewness attack, similarity attack
  - **skewness attack**: (drastic) skewness in data distribution (e.g. typically only 1% of the population tests positive on a test, but in one block, over *half* are positive $\longrightarrow$ if we know someone is in this block, there is a 50% chance they tested positive)
  - **similarity attack**: when all the sensitive attributes can be *generalized* (e.g. gastric ulcer, gastritis, stomach cancer $\longrightarrow$ the individual suffers from a stomach disease)

- **t-closeness**: mitigates L-diversity problems
  - a *table block (equivalence class)* is said to have t-closeness if the *distance* between the *distribution of a sensitive attribute in **this class*** and the *distribution of the attribute in the **whole table*** is no more than a threshold t
- **differential privacy**: carefully add *noise* to data to maintain a level of accuracy while minimizing the chances of identifying its records $\longrightarrow$ *plausible deniability*

Example: 4-anonymous table, 3 equivalence classes

- **homogeneity attack**: class 3; if we know someone is in their 30s and lives in ZIP 130**, we can conclude they have cancer, because all values of the sensitive attribute are the same
- **background knowledge attack**: class 1; if we know someone is under 30, lives in ZIP 130**, and we have *background knowledge* that they are vaccinated against hepatitis, we can conclude they have the flu

#	Zip	Age	Sex	Condition
1	130**	< 30	*	Hepatitis
2	130**	< 30	*	Hepatitis
3	130**	< 30	*	Flu
4	130**	< 30	*	Flu
5	148**	>= 40	*	Cancer
6	148**	>= 40	*	Hepatitis
7	148**	>= 40	*	Flu
8	148**	>= 40	*	Flu
9	130**	3*	*	Cancer

#	Zip	Age	Sex	Condition
10	130**	3*	*	Cancer
11	130**	3*	*	Cancer
12	130**	3*	*	Cancer

Inverse Transparency

- **"watch the watcher"**: give data owners oversight over the usage of their data
  - **in other words**: customer gives data to company, company gives oversight back to customer (how is that data being used?)
  - **goal**: make misusage unattractive

Passwords

- **preimage attack**: given $h(x)$, find x' such that $h(x') = h(x)$
  - **method 1**: list possible passwords (higher probabilities first), calculate $h(x)$ on the fly for each password
    - **+**: very simple, no space needed
    - **-**: high computational effort
  - **method 2**: list possible passwords, store them *alongside hash* in file or database, sort by hash
    - for a given $h(x)$, look for this hash
    - **+**: fast lookup
    - **-**: huge database
  - **method 3**: hash table with same reduction function
    - **+**: fast(-ish) lookup, some computation, less space
    - **-**: probability that chains merge (same end likely)
  - **method 4**: rainbow tables
    - same as method 3, but with *different* reduction functions
    - **+**: same as method 3 but fewer chains merge
    - **-**: lookup more expensive
- **salting**: add *random string* to password and hash them, *one per user* ($\rightarrow$ *same password* with different salt results in *different hash*)
  - lookup tables cannot be pre-computed
- **password hashing function**: functions specifically designed for that purpose
  - **MD5**: preimage attack exists

    - **SHA-1/2/3**: not designed for that purpose, fast $\longrightarrow$ can be brute-forced
    - **bcrypt**: iteration count can be increased
    - **Argon2**: resistant to GPU cracking attacks

Alternatives

- **single sign-on (SSO)**: authenticate using already existing account (e.g. Google, Facebook) on another platform
    - **+**: more convenient, less passwords to remember
    - **-**: if your SSO account gets cracked, you lose access to *all* related services
- **multi-factor authentication**: can be broken
    - **SMS / phone call**: can be intercepted, or the code generated might not be securely generated
    - **authenticator apps**: *MFA bombing* (bombard a user with multiple requests, hoping to overwhelm and trick them into approving a malicious login attempt)
- **passwordless**: replace passwords entirely
    - **magic links**: one-time URL sent to a user's email address
        - **+**: no password, easy to implement
        - **-**: email dependent, spam filter, *you still need a password to access your email account...*
    - **Fast Identity Online 2 (FIDO2)**: use *passkeys* (unique pair for every website, private key stored on device, public key registered by online service)
        - **-**: OS-dependent, no way to backup key, no defined recovery process if key is lost...

Malware

- **malicious software (malware)**: software designed to disrupt, damage or gain unauthorized access to a computer system (i.e. targets CIA) $\longrightarrow$ *automated MATE*
    - **virus**: malicious executable *code attached to another file*, spread when an infected file is passed from one system to another
    - **worm**: *standalone program*, can spread quickly by itself over the network
    - **trojan**: malware that carries out malicious operations *disguised as something else (desired)*
    - **spyware**: steal private information from a system (and send it to the hacker)
    - **ransomware**: encrypt data and hold it hostage for a ransom
    - **backdoor**: grant attacker future access to a system
    - **rootkit**: modifies OS to create backdoor
    - **keylogger**: records everything the user types
    - **adware (*)**: potentially unwanted application, (aggresively) displays advertisements

    - **riskware (*)**: not a malicious application by design, but an application which performs sensitive operations that can pose threats if compromised
- **detecting malware**: undecidable in theory; *arms race* between malware authors and malware analysts
    - **conventional**: scanning logic $\leftrightarrow$ engine $\leftrightarrow$ database
        - **string scanning**: find substring in string
            - **problem**: slow for large programs
        - **hash scanning**: match hash of string / program to malware database
            - **problem**: avalanche effect $\rightarrow$ malware author can simply change one byte and the whole hash is invalidated
            - **fuzzy hashing algorithms**: algorithms where minimal changes in input lead to minimal changes in digest
        - **malware-specific detection algorithms**: custom code for single malware family / variant (e.g. find section name `ATTACH`)
        - **heuristics**: generalize program properties (e.g. $\text{connects to web server \& contains decryption loop}$)
        - **emulation**: run program in virtual environment and analyze behavior
            - **problem**: inconsistent (how long to run for? what to check for? what if the malware knows it's running in a VM?)
    - **nextgen**: use machine learning to automatically recognize behavioral / code patterns
        - **thresholds**: for something like VirusTotal, use e.g. how many / what antivirus software detects the program as malicious to determine if it truly is malware
        - **labeling apps**: inaccurate labels leads to inaccurate models (e.g. is *adware* malicious?)
        - **performance decay**: model performance decays over time
        - **adversarial ML**: if attackers know how a machine learning model works, they can tweak their malware to confuse the model

Misconfiguration

- **common issues**: weak default config, no "best practice" guides, confusing config structure, too much background knowledge required
- [security content automation protocol (SCAP)](): open standards that are widely used to enumerate software flaws and configuration issues related to security, to (automatically) fix them
    - **problem**: need knowledge
    - **solution**: share knowledge
        - **common vulnerabilities and exposures (CVE)**: database of vulnerabilities

        - **extensible configuration checklist description format (XCCDF)**: describe how and check if a config is secure
        - **open vulnerability and assessment language (OVAL)**: automated checks
- **Center for Internet Security (CIS)**: where the knowledge (*"benchmarks", i.e. standard / best practices*) is collected

Problems

- **CIS**: no one size fits all (some rules may be omitted due to breaking legacy systems or simply being too specific)
    - **problem**: keep track of chages between CIS guide and personal guide
    - **solution**: version control
- **manual guides**: administrators have to go through these manually, which is error-prone (and some might not even do it)
    - could (?) be automated with LLMs or other projects (e.g. OpenSCAP)
- **exemptions**: sometimes, rules *have* to be avoided because they might break something, so we need to find a good middle ground
- **cyber insurance**: for some companies, it's easier to just buy insurance $\rightarrow$ companies don't share data about incidents / breaches
- **availability only**: most companies only care about availability (through backups), not on intellectual property theft, which is usually worse