Definitions

- **"Defender's dilemma":** attacking a system is much easier than defending it, this because to attack a system you only need to find one way to attack it. To defend it, you must find all the possible ways to attacking it and apply measures to protect against all attacks
- **Vulnerability life cycle** created and discover possible scenarios:
  - discovered by the developer (best case)
  - discovered by a bad guy that starts an attack (worst case)
  - discovered by a hacker that performs responsible disclosure giving the developer some days to fix it before he publishes the vulnerability.
- **Security Assessment:** it is necessary to look for its vulnerabilities and their related exploits. We must use a mix of testing and analysis. It's better to do it during development, because the cost of fixing grows with time. It's not necessary to do it when the product is ready, it's better to find bugs earlier.
- **Security Certification:** a formal attestation (a document) of some security properties of a system produced by an accredited third independent party.
- **Static Analysis:** you don't operate the system and you don't need to run the system. So analyse documentation (formal verification e.g. auditing). Using white-box mode, we can access the source code. In black-box mode, we use binaries (decompilers).
- **Dynamic analysis:** Test a running instance of the system, less exhaustive but it can show the presence of errors, not their absence (e.g. vulnerability assessment and penetration testing).
- **White box:** know the real topology, user credentials to login, can give more results but not all vulnerabilities will be found.
- **Black box:** no system's information, similar to an attacker view, it can disrupt the normal system behaviour.
- **Grey Box (hybrid solution):** have some information but not the full information.
- **False Positive:** Automated tools may report vulnerabilities but they are bugs that cannot be exploited or a software patched but not distinguishable from an unpatched one.
- **False Negative:** vulnerabilities that exist but are not found.
- **Security Auditing/Reviews**: There are a number of meetings at evaluating the security of some artefact before the product is analyse (execution phase) and there are preliminary phase:
  - Goal of the audit (to find which are the most critical components) and planning (how many persons are involved)
  - Preparation
  - Execution
  - Produce a report and share it
- **Debugger:** used for the purpose of finding vulnerabilities or used in the exploitation phase (P.T.). It's based on testing, it makes use of static analysis techniques.
- **Proof:** a sequence of well-formed formulas such that P_i is an axiom or a direct consequence of the predicting formulas, according to an inference rule.
- **Theorem:** a formula that can be reached in this way: there is a formula for who is a proof that terminates with that formula, so the theorem is true.

Techniques for Networked Systems

Vulnerability Assessment: It is the identification (and reporting) of vulnerabilities in a system. Port scanning phases: knowing hosts, OS identification, services and application (ports opened or not).
Vulnerability Assessment can lead to false positives. It could be white box, black box, grey box.

Penetration Testing: It is the identification of vulnerabilities in a system and the attempt to exploit them to assess what an attacker can gain from an attack (black-box technique), *metasploit* is a famous tool for PT. Testing never gives false positives. The first part is similar to Vulnerability Assessment. It can be static and/or dynamic. Stages:

- Pre-Engagement
- Information Gathering
- Threat Modelling
- Vulnerability Analysis
- Exploitation
- Post-Exploitation
- Reporting

VA vs PT

VA	PT
Reports all found vulnerabilities	Reports vulnerabilities that can be exploited and what can be gained
Mostly based on automated tools	Mostly based on manual activities
Easy to do	Requires high expertise
Often done by internal personnel	Often outsourced
Cheap	Expensive
Performed frequently (whenever significant changes occur)	Performed more rarely or when more accurate analysis is needed

Techniques for Software Application

Static Code Analysis (Static Application Security Test): An analysis that could be done is:

- Type checking
- Style checking: try to check if some style are used or not or are violated in the code

Possibility to navigate from a function to its definition code. Automated formal verification is based on:

- Control/Data-Flow analyser
- Model checkers
- Theorem provers

Symbolic execution is not a real execution but is what can happen when a program is executed. It can be:

- White box: read the code
- Black box: read the binaries

To find vulnerabilities, the techniques:

- Don't execute the code
- Check the behaviour of all inputs
- Partially automatic

It constructs a model from the code (formal model) and we also have informal requirements (what we are looking for) that have been extracted from the code. Having both, we can compare the two and verify the properties, if not satisfied a report is produced.

Dynamic Code Analysis (Dynamic Application Security Test): perform some particular test to discover vulnerabilities. Two techniques:

- Fuzzing: test a program generating randomised inputs
- Proxies: emulate MITM

To find vulnerabilities, the techniques:

- Testing check the behaviour of some inputs (not exhaustive)
- Fully automatically
- Execute the code

IAST (Interactive Application Security Testing): Recent concept that tries to combine static and dynamic analyses.

- Pros
  - Find many vulnerabilities (like SAST) but with few false positives
  - Very fast and scalable (good for dev-sec-ops)
- Cons
  - Not yet available for any language/framework
  - Not yet widely known and adopted

SAST	DAST
Find more vulnerabilities	Find less vulnerabilities
More false positives	Less false positives
Used in all development stages	Used in the last development stages
Can provide info on the cause of vulnerability (code line)	Cannot provide info on the cause of vulnerability (black-box)
Coverage of libraries is an issue	
Each tool applies to only some languages/frameworks	Independent of language/technology used to develop application
May be time-consuming	

Security Assessment and Certification Standards

Common Criteria (CC): Comes from similar national standards from many countries and it has been standardised by ISO.

- **Objectives**: They provide a common standard reference for evaluating/certifying IT system security. Permit comparability between the results of independent security evaluations.
- **Approach**: CC is just a criteria. An evaluation scheme is a regulatory framework defined by each nations/country. To perform an evaluation and a certification in according to the CC, it's necessary to have an evaluation methodology and an evaluation skill.
- **Document Structure**:
  - Introduction and General Model
  - Security Functional Requirements
  - Security Assurance Requirements

Security assurance: there is a problem of trust. You need to trust only to an authority that tell you that there is a certain that what it's telling you is true. A certification is needed. To classify Assurance Techniques, we have formal (use mathematical model), informal (no mathematical models at all) and semi-formal (a mix).

Key Concepts:

- **Target Of Evaluation (TOE)**: system or component under evaluation. It has Security Functional Requirements (SFRs) and Security Assurance Requirements (SARs).
- **TOE Security Functionality (TSF)**: parts of the TOE that must be relied on for the correct enforcement of the SFRs.
- **Protection Profile (PP)**: an implementation-independent set of security requirements for a category of TOEs that meet specific consumer needs.
- **Security Target (ST)**: a set of security requirements and specifications to be used as the basis for evaluation of an identified TOE, the easier case is that the ST provides the PP for the category of this specific TOE (it's obviously possible to add other requirements).
- **Possibility to evaluate PP or ST+TOE**: requirements themselves can be evaluated, even without a related TOE.
- **TOE + ST evaluation**: the aim is to evaluate sufficiency and correctness of TSF adopted to satisfy security requirements.
- **CC Recognition Arrangements (CCRA)**: since not all the countries have a regulatory framework to perform this kind of assessment and certifications. The other countries need to sign this agreement. Countries can be classified into:
  - **Authorising Nations (certificate producing)**: They have their evaluation schema to perform CC evaluation.
  - **Consuming Nations (certificate consuming)**: They don't have an evaluation schema but they want to recognize CC evaluation done by others.

Functional Requirements Classification: There is a classification of security requirements in the CC, done in this way:

- **Classes:** macro groups of security requirements. "*Identification and Authentication*" FIA.
- **Family (of requirements):** there are requirements that have the same goals but are different from others. It's the second level.
- **Components:** single requirements that you can put inside PT or ST by collecting this hierarchy of requirements. One component requires one or more functional elements. When you have a component evaluated, each functional element can be evaluated separately with the same result.
- **Functional elements:** inside a component, the smallest unit that can be evaluated.

Assurance Requirements: correspond to the intended Assurance Techniques. Assurance is achieved by:

- evaluating the evidence that the intended Assurance Techniques have been applied.
- performing independent verification/testing activities.

There are a number of techniques that are illustrated in the CC and correspond to all the different types of techniques that we can use in order to improve assurance level.

Assurance Effort Scale: for each one of these axes we have different assurance levels that determine assurance effort for each single assurance technique.

- **Rigor**:
  - Checking correspondence between specification
  - Designed and implementation

    - Design can be done by an informal way or a semiformal way and the level can be different and the level of assurance is achieved
- **Scope**: we can apply to a larger or smaller part of a system that is under evaluation, if large I will increase the assurance level.
- **Depth**: level of detail at which is done.

Assurance Requirements Classification: it contains the list of techniques in many different cases and it is classified in a way that is similar to how functional requirements are classified (**class, family, component, assurance elements**). The component can be put in ST or PP. The assurance elements correspond to different type of elements:

- **Developer actions:** something that the developer expected to do during development
- **Evidence:** what kind of evidence must be provided by developer
- **Evaluator action:** what evaluator is expected to do (e.g. extra testing).
- **Assurance Levels:** The assurance is quantified by means of discrete Evaluation Assurance Levels (EAL). The standard defines these discrete levels and it is also possible to define custom levels in between, this because if you change the mix of what is required by each standard assurance level (each EAL requires a set of components) you can get something in the middle between two assurance levels defined by the standard. A higher EAL is obtained from the previous one by including other components (other families) and replacing components with higher level assurance components (from the same family).
    - **EAL1 - functionality tested:** where the threats to security are not viewed as serious.
    - **EAL2 - structurally tested:** higher than lvl 1, we have real vulnerability analysis.
    - **EAL3 - methodically tested and checked:** higher but without substantial re-engineering.
    - **EAL4 - methodically designed, teste and reviewed:** suggested for conventional commodity TEOs, more focused VA and PT.
    - **EAL5 - semi-formally designed and tested:** introduction of formal methods, for systems that are critical.
    - **EAL6 - semi-formally verified, designed and tested:** for protecting high level assets.
    - **EAL7 - formally verified, designed and tested:** extremely high risk situations.

Formal Methods

Techniques used when we want to achieve a high level of assurance about security.

Formal specification techniques

Formal specification has to do with how we give an imprecise ambiguous description of a system or its security properties. It's based on some mathematical model as a model, it's something that introduces an abstraction with respect to reality. We can have specifications at the design level and then up to implementation. Features:

- **Unambiguous:** not have multiple interpretations
- **Consistent:** no internal contradiction
- **Complete:** all relevant information is represented in the model.

If the language that we use for formal verification is characterised by these properties we say it is a formal language and we have a formal specification using these languages.
We have two different formal models:

- **Structural models:** how the system is structured, how it is composed
- **Behavioural models:** There are two ways:
    - **Operational style (imperative style):** describe the behaviour specifying the operations and the action performed by the system:
        - **State Transition Models**: when we want to check some temporal properties that cannot be checked by means of Data Flow Analysis (ex. CFG or CG). An example where we need the state transition models is checking the absence of memory leaks deriving from not freeing allocated memory requires checking a TL formula.
            - **Transition System (TS) (S, init, rho)** model formally says that is a set of states and transition relation that is a subset of the cartesian product "SxS". In practice, it is a set of pairs of states. Usually, we can represent a transition system as a state diagram.
            - **Labelled Transition System (LTS) (S, init, L, rho):** is a TS in which there is a labelling of transitions used to specify events. Here the transition is a subset of "SxLxS".

        - **Control Flow Graph (CFG)**: it describes how the program behaves and the nodes of the graph are states of the program. This model captures only the control flow of the program but not the states of variables, to include it we have to augment the CFG with something else.
        - **Non-Determinism**: Abstract models use non-determinism for representing execution aspects that are not known a-priori, like for example the inputs of a program. It is also used to represent different possible implementation choices and in this case a TS is more complex, it could be better to use a CFG.
    - **Descriptive (declarative way):** we can compare it to declarative programming languages and describe the system properties without specifying how the system behaves. A formal model of a system property can be expressed formally by a logic formula.
        - **Propositional logic**: is in practice the boolean logic, formulas of atomic propositions, that can be true or false.
            - Tautology: formula that is always true
            - Contradiction: formula that is always false

          A formula is said satisfiable if it is true for at least one interpretation of APs. A formula is said valid if it is true for all interpretations of APs (tautology).
        - **Predicate (also called "I order") logic**: it is an extension of propositional logic where atomic propositions are replaced by predicate and the new concept of constant, variable, function, relation and the ∀ ∃ are introduced.
        - **Formal System (Theory)** which is defined by the combination of syntax (an alphabet of symbols and set of formulas) and semantics:
            - **A formal language**: an alphabet of symbols and a set of well-formed formulas. It defines if I can write a formula, if it is correct. "What formulas I write?"
            - **A deductive apparatus**: a set of axioms and inference rules "How do I give a truth value to a formula?"
              Theorem and Proofs, a sequence of formulas, where each one is the direct consequence of some of the preceding formula

      Temporal properties: proposition and predicate logics that describe facts related to a program execution. For dynamic system are time-varying, so a solution is using predicate logic with a variable *t* (continuous or discrete time).

        Temporal logics: it's an extension of classical logics (prepositional or predicate). They can be defined:
            - Prepositional vs I order logic
            - Discrete vs continuous, implicit vs real, linear vs branching time
            - Event vs state, instant vs interval, past vs future modalities

          These types of propositions use the **Linear Temporal Logic (LTI)** which is a simple logic that uses temporal operators ([], U, …)

System behaviour Classification:

- **Computational system:** case of square functions. The system produces some output given some input. Terminated when finished his task.
- **Reactive system:** usually in the distributed system. It interacts continuously with their environment. It will not stop and it's not possible to describe a mathematical relation between data, we have to describe other techniques for the description. From the operational point of view, we could use a state transition model. From descriptively, they can be described by temporal logic.

There is nothing to do with the presence of concurrency in the system, because it has to do the measure of requirements of the system. At the same time, I can have a reactive system that doesn't use internal concurrency. Reactive systems are a more general case of computational systems.

Formal Verification techniques

It is a static analysis technique that applies to system formal model. It certifies the model, not the system. By analysing the model, we can achieve a high assurance that this model is correct. It is possible to verify that a formal model satisfies some formal properties with a set of logic formulas (a syntactic check of formulas and by a satisfiability of the check). Different targets:

- Self-consistency of formal methods that we have obtained from formal specifications:
    - No internal contradictions
    - Well-formed
- Checking that the same correspondence exists between one model and another

- Boundedness of specifications
- Verify a system specification satisfies some properties

An algorithm that can answer correctly, takes finite time and uses finite memory. The fact that a simple problem is undecidable is somewhat that is discouraging because it seems that you cannot formally verify anything interesting. Undecidable doesn't mean that you never answer the question, but not always.
Algorithms have different properties:

- Sometimes to tell 'I don't know', because if I don't know the answer is right or wrong, this algorithm is not very useful so it's better say 'I don't know'
- Give wrong answer

Possible solutions:

- Semi-decision procedures: a decision (Y/N) but not for all input/phases
- Making abstractions: formal verification is an abstraction of reality
- Approximation and non-exhaustive modelling or analysis in a controlled way
- Different approach called 'Correctness by construction'

We have a logic language and a deductive system for it. We also have the concept of interpretation also called model. Two different approaches: model checking and theorem proving.
Model checking: there is the *model M* (interpretation) and the *property f* (a well-formed formula, in the formal system). We check if the formula *f* is true under the interpretation *M*. If it is false, give us a counterexample.

- **Techniques for Dynamic System:**
  - **State Exploration:** In this case, the Model Checking generates all the possible states/runs of the state-transition model and checks each one of them to see if the formula is true or false. The state exploration can be:
    - **Explicit:** you generate explicitly each state and each run.
    - **Symbolic:** there are more sophisticated techniques that represent sets of state/runs in a more compact way

    Special case:
    - **Reachability Analysis:** it's a special case of state exploration. We can use it when the property to check is a simple temporal property *[ ]P*, this formula means that P is true in all future states.

    Limitations of State Exploration: It can be applied only if the number of states/transitions is finite but is possible to build the model checker even when the number of states is huge (based on abstraction), the states that have been explored in this case are not all the possible ones. So we don't have the 100% certain that is true but this will give us more confidence about the correctness. The other problem is that for concurrent systems, the complexity grows exponentially.

Theorem proving: we can use it to prove mathematical theorems. Most of the properties are undecidable and they also use human assistance. The input are:

- Theory of formal system
- Formula that I want to improve tells me if *f* is a theorem or not.

Starting from the state transition model with the interpretation of transition and the tool generates a set of axioms and rules (it transforms the STM into a deductive system). These are put together with other axioms and rules that are those of temporal logic. In the end, we have a new theory that we give to theorem prover and the formula. If yes it gives me proof (*f* is in the system). If it doesn't find a proof, theorem prover doesn't know if the property is true or not. Theorem prover verifies validity of *f* for all interpretations by finding a proof.
The augmented theory that comes from state transition model is:

- **Sound:** with respect to K, if for each theorem *f* K |= f
- **Complete:** if the opposite implication that is true with K |= f, the formula is a theorem.
  If a theory is sound and complete respect to K: |- f ⇔ K |= f
  If a theory is sound but not complete respect to K: |- f ⇒ K |= f (the formula is true but *f* is not a theorem)

Techniques:

- **Logic Programming:** It is a way to write a formal system (a theory), using a program. When you write a program in logic programming, you cannot write normally a formal system as you want but you have some restrictions. You can tell if the formula is true and if there is a proof for the formula in this formal system (it is a theorem prover).

Model Checking	Theorem Proving
Provides a proof of non-validity (counter-example)	Provides a proof of validity
Can be applied directly to an interpretation	Requires generation of theory
Both can tell with certainty if property is true (validity)	
	If proof is not found, nothing is known

Security Verification Protocol

Security Protocols are the protocols that we use to achieve some security goals like authentication, secure key exchange, data integrity, confidentiality and so on. They use communication and cryptographic primitives (MITM attack on Authentication).

Challenges:

- Concurrent session: we can have many session as we want, no predefined bounds on the number of sessions, the system theoretically is infinity state
- Attackers can behave in any way (they can create any kind of message)
- Make this kind of analysis by hand is impossible (too many scenario to consider)
- When you implement you can introduce errors/bugs that can disrupt protocol security

If you have an implementation you can test and define some attacks on the protocol (testing), but testing doesn't reveal all attacks.

Security Proofs: A protocol is secure when no *reasonable* attacker can break a protocol, *in practice*. "Reasonable" and "in practice" are informal, we need a precise and unambiguous meaning.

- A *reasonable* attacker is what the attacker (we assume) can and cannot do.
- Break a protocol *in practice* when an attacker has succeeded in attacking the protocol.

Different approach:

- Symbolic**:** high level model, more abstract. It starts from the assumption that we use a particular type of cryptography which satisfies some properties.
- Computational**:** more precise but harder to verify, specifying what cryptography algorithm are used.

Rigorous Symbolic Modelling (Dolev-Yao): The idea is that we use abstract data types: data are not represented in a concrete way as they are in the real system, but they are represented symbolically. The crypto operations are modelled as algebraic operators (that operates on the symbolic terms) with ideal properties. Ex. *decrypt(encrypt(M,K),K)=M*. This is only theoretical, I could try to perform a brute-force attack to find *M* without knowing the key. In this kind of modelling, we make some ideal assumption on what an attacker can do:

- Attackers can read, delete, substitute, insert messages.
- Attackers can build messages from current knowledge.
- Attackers can execute crypto operations.
- Attackers cannot guess secrets, get partial knowledge.

The aim of the modelling is to let us perform verification in which we prove that attacks are impossible under these assumptions.

- With Model checking we search for possible attacks and, if one is found, we have counterexamples that the security properties do not hold.
- With theorem proving there is a search for formal correctness proofs, if a proof is found the property is true on the model.

Computational Model: models are more accurate in the sense that data are concrete ('bitstring') and also algorithms operate in bitstring. We model the algorithm and not the implementation because it will be implemented in a correct way. A reasonable attacker is a process that tries in polynomial-time. Each protocol is modelled by the probability with which this event may occur, not only events that are possible in the protocol. Nothing is impossible here, but we can consider some operations negligible.

- **Proofs:** *"If there exists an attacker that runs in poly time and has non-negligible success probability than there exist an algorithm that solves a hard computation problem in poly time with non-negligible probability"*

Proverif

It is an automated theorem prover for security protocols based on Dolev-Yao modelling. Protocols are modelled by a process calculus and then translated automatically to a logic program. It provides:

- The model that has to be written (description of the protocol) typed or untyped.

- The description of the security properties (called queries)

The resolution algorithm can succeed or not, so the possible results are *Property is true (there is a proof)* or *Property cannot be proved*. If the property cannot be proved, proverif starts another stage of the analysis called attack reconstruction (it tries to find an attack on the protocol). Remember that finding a potential attack is not proof of an attack, it is a potential one, so we have to check if it is a real attack or not (the attack may be a false positive).
Specifying protocols (2 possibilities):

- Horn clauses (low-level, only for experts)
- Typed or Untyped Extended pi calculus (internally translated to Horn clauses)

We can combine and apply function to variables and names (constant), classified in:

- Constructor
- Destructor

The input to Proverif is a file (script) composed of:

- Declarations of operations (e.g. cryptographic operations)
- Process macros (reusable definitions of processes)
- Main process
- Queries (properties to be verified)

The output is a report, for each query (property to be verified) the report. It specifies:

- Whether the property has been proved to be true or false (or property could not be proved)
    - If property proved false, an attack was reconstructed (not always lead to an attack)

Specifying Properties:
Secrecy: an attacker must not be able to get closed terms that are intended to be secret, it means that if there is a symbolic name that is intended to be secret an attacker should not be able to obtain this secret. An attacker, normally, could not get the secret but learn something about it (partial knowledge) if we don't want partial knowledge, we have **strong secrecy**. The process is the same, only the secret changes and an attacker cannot distinguish when it changes and also an attacker cannot distinguish P and Q. A strong secrecy can be verified by proverif. If you write *'noninterf x,y, ..'* means that the terms written x,y should remain strong secretly. P[x/N] ≈ S P[x'/N].
We have **forward secrecy**, even if the long term key gets compromised (the attacker gets it), the attacker has no possibility to learn something about the past (the secret is exchanged at each session).
Formal specification of secrecy: we define an adversary and also what is the initial knowledge of the adversary.

- S-Adversary: as any process closed (one that doesn't interact with another one)
- Trace T: it's a run of the process
- T outputs N: contains an output statements of N to a channel M that is in the set S
- Closed process P: preserves the secrecy of N from S-Adversaries

Approximation: The Horn clauses approximate the protocol behaviour specified in extended pi calculus:

- The number of times a message is sent is not represented by the Horn clauses
- Two fresh names could be represented by the same name

The logic theory described by the Horn clauses over-approximates the behaviour of the real protocol that's because false positives are possible.

Weak Secret: a secret should not be weak, we want a secret that is not subject to guessing.

- Phase 0: interaction phase. The attacker interacts (intercept and store) with the honest actor.
- Phase 1: offline guessing phase. The attacker got some information and process on the data.

We can find this error by means of proverif.

Correspondence Properties: other properties are about authentication and integrity of data. These properties, expressed by means of 'correspondence', require the verification using a temporal logic but proverif does not give a real temporal logic like linear temporal logic that gives some temporal operators (in particular correspondence operator).

- **Injectivity of Correspondence:** *inj_event(e(…))* ==> *inj_event(e'(…))* specific when we want to have one-to-one correspondence between events, so each occurrence of event e(...) corresponds to a distinct occurrence of event e'.

Phases: some protocols are divided into phases that are executed sequentially and proverif support the description of them.
Proverif supports the description of phases:

- Each phase is a section of the global run.

- Phases are numbered (0, 1, 2,...).
- By default each properties are in phase 0 (if not specified).
- Each process code is divided into phases: phase n; P .
- When the system enters phase n, only the processes in that phase can execute.

Phases and forward secrecy: the secrecy of a secret exchanged during a phase of a protocol (e.g., a session) cannot be compromised after the end of that phase, even if other long-term secrets are disclosed.

SOME/IP (Scalable service-Oriented MiddlewarE over IP):
IP-based middleware solution for automotive systems (for in-vehicle ethernet communications). The offerer apps provide services that can be accessed by requester apps. No security provided. So we have these problem (threat model):

- Potential attackers are: car owners, dishonest repair shops.
- Attacker goals:
  - unauthorised access to in-vehicle services and information.
  - disruption or alteration of in-vehicle services.
- In-vehicle ECUs are accessible through the network so a physical device can be connected to the in-vehicle IP-based network, thus acquiring full access to communications.
- In-vehicle ECUs can be replaced by fake ECUs

In order to solve these problem each ECU is equipped with a tamper-proof Hardware Secure Storage (HSS)

- used to store cryptographic material (secret keys) and execute cryptographic operations
- difficult to access/duplicate/move/remove

A security protocol is used to achieve authentication/integrity and confidentiality goals. Each app has its own public/private key pair:

- public key stored in app certificate (with access policies)
- private key stored in HSS

A symmetric key is used to secure service requests:

- one key per service instance
- a new key is established whenever the app is restarted

Software vulnerabilities

Vulnerability related to software: there is a difference between bug and vulnerability. There are bugs that are not vulnerabilities and bugs that are also vulnerabilities (difference in what the application is intended to do):

- Bug: a set of required behaviour larger than the possible behaviours, some of the required behaviours are not possible because of bugs.
- Vulnerability: not required behaviour. It's maybe an additional behaviour that is innocuous that has not created aims. We have possible behaviour more than required behaviour.

There are different ways that an attacker can use to access the software:

- When the software sends input and gets some outputs
- Inspection: software available in binary form using binary inspection tools and get some information about the software itself.

Classification:

- CWE: the Common Weakness Enumeration is a category system for software weaknesses and vulnerabilities.
- Fortify Taxonomy: also called the 7 Kingdoms. The idea is that the software vulnerability that we can find should follow in one of these 7 classes, comprehended classes and not perfectly disjoint.
- OWASP Top 10 (for web applications): working group that has the aim of providing tools.

Fortify Taxonomy: the 7 Kingdoms:

1. Input Validation and Representation**:** It includes security problems that are caused by trusting untrusted input (e.g., SQL injection, XSS)
2. API Abuse: Security problems arise from not respecting an API contract or from misinterpreting the behaviour of API functions.
3. Security Features: Security problems come from not using security features properly, for examples:
   - improper use of access control, cryptography, privilege management, etc.
   - leaking privileged information inside the code.

4. Time and State**:** Timing issues can arise with concurrency/interactions between threads, processes and there are unwanted forms of interactions, taking all cases of concurrency into account is difficult. Race conditions are introduced when we use asynchronous operations.
5. Error Handling: Security problems may arise from missing, poor, or improper error handling. Like:
    - a missing error handling causes the program to continue in case of error, with unpredicted behaviour.
    - a missing error handling may cause the generation of an exception that leaks sensitive information
6. Code Quality: Security problems are caused by poor code quality that lead to unexpected behaviours
7. Encapsulation: Security problems caused by not properly implementing strong boundaries, for example not protecting code in a browser from other code running in the same browser (e.g. cross-session contamination or e.g. client and server?)

OWASP Top 10: another ranking of vulnerabilities, because among all possible vulnerabilities OWASP identifies 10 classes, ranked based on security risk for web applications.

1. Broken Access Control: bad control on the authentication, or what is authorised and what is not.
    a. REST API for privileged operations made available to authenticated users
    b. Violation of the least privilege principle
2. Cryptographic Failures:
    a. Sensitive data transmitted or stored in cleartext
    b. Unsalted password databases
    c. Use of old or weak cryptographic algorithms for encryption
    d. Missing security certificate validation
    e. Improper key management
    f. Wrong use of cryptography and cryptographic APIs
3. Injection: Injection flaws occur when an attacker can send hostile data to an interpreter. Different types of code injection are possible:
    - SQL Injection: untrusted data are inserted into a SQL query without validation or filtering
    - PHP Injection: untrusted data are passed to the PHP interpreter
    - Script Injection (e.g., XSS): XSS consists of injecting HTML with embedded script code to bypass Same-Origin-Policies (SOP).
        - Reflected XSS: the vulnerability arises from the fact that a vulnerable website reflects some data that come from the user.
        - Stored XSS: it is even more dangerous than Reflected XSS because now the script injected by the attacker is not immediately reflected to the victim, but it is stored in a database. Later, maybe another application that uses the same DB may read the information and reflect it to another user.
        - DOM XSS: Same mechanism as reflected XSS, but the missing input validation is in the client-side code.
4. Insecure Design: security is not properly considered in the development process
    a. Absence or incompleteness of threat modelling which leads to neglecting some possible attacks. Detectable in the early stage of the development
    b. Wrong choice of security controls. If we use a security control not adequate to counter the problem, we have a possible vulnerability. This kind of issue will not be detected typically by code analysis, but they can be detected by other types of analysis such as proverif-like analysis.
5. Security Misconfiguration: Security Misconfiguration occurs when security features are not properly configured.
    a. Some development setting not changed to production settings.
    b. Directory listing not disabled on the web server
    c. Improperly configured permissions
    d. XXE
6. Vulnerable and Outdated Components: The software uses vulnerable/outdated components, third party components
7. Identification and Authentication Failures: Occurs when an attacker can break identification or authentication.
    a. Permitting credential stuffing
        i. Allowing weak passwords
        ii. improper invalidation of sessionID
        iii. Exposure of session ID in the URL

iv. User account id received from the user and not verified
v. Using weak credential recovery mechanisms

8. Software and Data Integrity Failures: these failures occur when integrity of software or data is not properly checked/ensured. These failures happen when software is transferred from one location to another ,for example not transferring software over HTTPS or not signing the software that is transferred.
    a. Updated downloaded without sufficient integrity verification
    b. Application downloading plugins or libraries from untrusted repository or CDNs
    c. Insecure deserialization: there are many languages EX. JAVA that can serialise objects and then restore serialised objects somewhere else. It is common in these applications to send serialised objects to other entities in the network. There should be protection, it should be signed.
9. Security Logging and Monitoring Failures: Insufficient logging or monitoring of the logs that lets attackers perform their attacks without being detected. For example:
    - absence of logging of security-relevant events.
    - unclear log messages.
    - logs not constantly monitored.
    - logs only stored locally.
    - PT or scans do not trigger alerts
10. Server-Side Request Forgery (SSRF): an URL is sent to a victim web application that doesn't check the URL before using it and it points to a non-trusted other application which can steal secret. Some example are:
    - SSRF to get access to sensitive information (e.g., file:///etc/passwd)
    - SSRF to perform unintended operations on resources protected by firewalls, VPNs or ACLs

OWASP Top 10 modification: Years go on and the order of the application vulnerability change due to the spread and the discovery of new vulnerability, others can be so known that they are no longer in the list. In 2017 the OWASP TOP 10 was a very precise type of vulnerability, while now it is a list of groups of vulnerability.

Relation Between 7 Kingdoms and OWASP Top 10: More classes can be mapped to the same kingdom and a single class can be mapped to more than one kingdom.

Finding Software Vulnerabilities with Static Code Analysis

Formal code models: all the main programming languages have formal syntax and informal semantics. If you want to extract a formal model from a code written in a programming language, you have to assign a particular interpretation to the code (similar to the work done by a compiler). The model is more abstract than the executable code, because it represents only partially the behaviour of the program. The model can be built from the source code or from the binaries (in this case we work like a decompiler) and depending on the type of model, different analysis techniques will be used.

Model Extraction Process (from source code): Source code → Lexical analysis → Parsing → Semantic Analysis → Intermediate code generation → Optimization. (The last two steps are not relevant for the static analyse process).

- Lexical extract tokens from the source code, these tokens are the single word/symbols that make the source code.
- Parser to build a parse tree that interprets the sequence of tokens according to the rules of the language, it will create the abstract syntax tree. In the tree you have only the reference to the symbol table:
    - The tokens, parser tree, the abstract syntax tree and the symbol table are structural models: they represent the structure of the code, how it is organised and what are the meanings of some parts of the code, it is not yet a representation of the behaviour.
    - Check:
        - Tokens: to find vulnerable functions (strcpy)
        - Type scanning to retrieve the type of data used, useful for some vulnerability (static stored info of a db)
        - Style checking: you know the structure of the code, so you can check if a particular style of programming is respected or not.
        - Code understanding/navigation is a feature provided by integrated development environment (IDE)

Control Flow Graph and Call Graph: behavioural models. They can be built from the last state (Abstract Syntax Tree + symbol table). The aim is to represent the possible control flow of the program (CFG) and what functions each function can call (CG). It's used for code generation and optimization. Both are shared by compilers (able to build both from binaries and from the source code) and decompilers (only binaries).

Data/Control Flow Analysis: this kind of techniques are used by compilers (optimization phase) to report errors and warnings to the user.

- **Control Flow Analysis:** the aim is to get information from the (real) control flow.
- **Data Flow Analysis:** the aim is to get some information about the values. We build the model *D* for each basic block, we compute a transfer $D=f_B(D)$ function that represents how the abstract values of variables are modified by the basic block. Then *D* depends on the output of preceding basic blocks.
    - **Precision:** may be better or worse according to how we model variables. Real values of variables introduce abstraction, looking for some aspects and it leads to an approximation.

Intra and Inter Procedural Analysis: many functions that call each other in different ways.

- **Inlining:** not always possible because we don't know at compile time what exactly the function that will be called are.
- **Intra (global) and inter (local) procedural analysis:** we alternate two different steps in the analysis, the inter is a local step while the intra procedure analysis is propagated to the call graph.
    - Local analysis perform for each function Control Flow Graph based on known assumptions about inputs.
    - Global analysis performed to propagate information from function to function according to Call Graph.

Vulnerability with Dataflow Analysis: assertions about variables can be checked by dataflow analysis, this means for example that I want that a certain variable has a particular value in a certain location of the program.
[] ((control state == X) => P(variable state)) → This means that it is always true *[]* if the program has arrived to a specific location X, then a certain predicate P holds it (the predicate is a boolean expression about the value of a variable in that particular location).

- **Taint:** The taint status depends on the type of injection. It's not enough for this type of analysis to have a single boolean value that is tainted or not.
    - **Taint Analysis (or Taint Propagation):** We can perform an analysis to find whether a source can propagate to a sink (if an input received in a certain location of the program can propagate itself to another one). With propagation, we mean that this input influences or arrives unchecked/unfiltered ecc. These analysis are relevant for security:
        - **Detect leakage** of secret information vulnerabilities
        - **Detecting Injection Vulnerability:** Statements can:
            - Taint a variable (input statements from an untrusted source).
            - Propagate the taint status, the status pass-through this statements
            - Propagate the taint status from one variable to another
            - Clean the taint status
            - Potential vulnerabilities (sink)
    - **Other vulnerabilities:**
        - Buffer Overflows: find overflows comes from an untrusted source
        - Format String: for an attacker to inject something used inside a formal string
- **Pointer Aliasing:** useful if two variables point to the same location. If we know this, the precision of our analysis increase
- **Initialization status of variables:** variables not initialised can take any possible value
- **Predicates about variable status:** we can track using data flow analysis

Abstraction: code complexity may prevent fully detailed behavioural analysis, if the program becomes too complex, the time taken to perform the analysis becomes too long. You should abstract all the details that are not relevant to check, so it's introduced subtraction without losing precision. A representation of all the possible runs of the model and with an abstraction, I can map the execution onto a reduced number of executions. I've neglected some details and all these executions became a single execution in the abstract model. Neglecting what is not necessary:

- **Strongly-preserving abstraction (sound ⇒ and complete⇐)**: equivalent to analysing the property in the original system without losing anything.

- **Correctness-preserving abstraction (sound ⇒):** false positive. If we can find that the property doesn't hold, he can be a false positive.
- **Error-preserving abstraction (complete ⇐):** false negative.

Approximation: an abstraction is exact if:
- For each run of K there is a corresponding run of K*.
- For each run of K* there is a corresponding set of runs of K.

Exact abstraction means that the runs of the concrete system are the same runs that are represented by the abstract system. Otherwise with approximation
- Some of the runs of K are not represented → (subset) under-approximation → false negatives.
- Some runs that don't occur in K are represented → (superset) over-approximation → false positives.

Program Slicing: to create over-approximations starting from the code of the program modifying the code itself (no state transition model). There are variables to track because they influence the property to check. Variables and instructions that don't modify variables are removed.

Symbolic execution: it is a way to analyse the behaviour of a program considering all the possible inputs, it's similar to Data Flow Analysis because it uses symbolic data and not concrete data. It is classified among the static analysis tools, but it is an execution, not a normal execution but it is a sort of simulation of the program execution using abstract models of data. The data model of a variable is a symbolic expression these symbols represent entire type domains and it can use assertion checking.
The state of a symbolic program execution is a triple (cs , σ, π) where:
- cs is the control state
- σ is the state of variables
- π is a path predicate (can be true or false and it represents a set of constraints that we have when we reach a particular point in the program execution)

When a statement is executed, the state is updated:
- Each time that a value is assigned to a variable, σ is updated
- Each time a conditional statement is executed, π is updated with the condition that has been found for the conditional statement.

There is also a feasibility check: it is possible that one of the branches of the conditional statement is never taken, because the condition is always true or false. So there is a check that is done by means of a satisfiability checker, more precisely a Satisfiability Modulo Theories (SMT) checker or solver.

Symbolic Execution Pitfalls: Ideally, it's possible that you loop forever because until you find that a condition is not unsatisfiable you have to continue. If there are solutions with an unbounded number of paths:
- It's possible to use lossless techniques to limit this
- Limit the execution path to a subnet

So, it's possible that there will be false negatives.

Limitations from STM solver: a problem that is decidable only under some conditions is an unsatisfiability problem. If we don't put so many constraints, it's an unsatisfiable problem.
- Decidability and undecidability are called *Theory*

Concolic Execution: Concolic execution and Symbolic execution are applied analysing source code and binaries. Symbolic execution decides if an assertion can be false and the STM solver tells what the values are. Concolic execution is half Concrete execution and half Symbolic execution. We combine them in order to make the analysis more precise. Two different forms:
- **Dynamic Symbolic Execution:** The concrete execution drives the symbolic execution. We start with a concrete execution with some random inputs and we execute in double mode (symbolic and concrete at the same time). Path feasibility is not necessary and when the execution terminates, I negate the last path constraints so I can check if the last decision that was made can be made in a different way, in this case I use a SMT solver to explore the other branch. So after this I repeat concolic execution with the new inputs found and so on.
- **Selective Symbolic Execution:** useful when the source code is not available, in this case I should do only a concrete execution and when it returns, it returns also value used in a symbolic execution. There are parts of code that are executed only symbolically and parts only concrete. From this combined execution, I can overcome some limitations of symbolic execution.

Detecting and Fixing Some Java Security Vulnerabilities

Injection Vulnerabilities: exploited by injecting something that can be countered by proper validation of the input/checking of the input. These kinds of vulnerabilities are due to input coming from the outside from an untrusted (who comes from "the network") source:

- Java interacts with the web using HTTP by means of servlets or sockets.
- Standard inputs if the user is untrusted
- Reading information from files

Possible sinks are:

- Run program in OS, common injections are possible in Java
- Interact with a remote DB
- Use log statements and operate in local files
- It's possible to look information and cross-site scripting can be exploited in this way

Socket: it is a possible source of untrusted data.

- Socket class that represents a TCP socket
- DatagramSocket class, which represents a socket by means of which you can send and receive datagrams with UDP

Servlets: with HTTPServlet class we receive HTTP requests and produce HTTP responses:

- doGet(HttpServletRequest, HttpServletResponse)
- doPost(HttpServletRequest, HttpServletResponse)

Data from untrusted sources is the first parameter while a possible sink is the response, because if data coming from the request are reflected to the response, we have a possible XSS.

Spring API REST implementation: some frameworks hide the presence in HTTP Servlets (the common is spring boot framework used to implement Rest API, HTTP methods). You don't see the servlets but there are servlets that run and you put annotations in the code.

SQL injection: it is a possible attack when parameters are directly added to the SQL string. A way to prevent SQL injection is to use prepared statements, but this is not the only way. Another one is to validate or sanitise the inputs so they don't contain dangerous characters.

XXE (XML External Entities) : This is another type of vulnerability that is possible when we use an XML parser for reading data and it is due to the way in which external entities are handled (reference to external file). If you provide an XML document with references to external entities it is possible that you provide unexpected input to the program. To detect these vulnerabilities, we want to consider possible sink functions.

XSS:

- **Reflected XSS:** data coming from an HTTP request can be reflected to the HTTP response
- **Stored XSS:** same mechanism, but more serious because it can affect different applications/users
- **DOM XSS**

Other forms of injections:

- **Format string:** Java class has some statements that can print output using format strings, memory can be accessed only to object pointers. An attacker can insert more parameters than arguments and the function throw an exception.
- **Regex DOS:** is exploited the fact that some regular expression are checked.
- **Evil Regex:** have particular inputs that will take a very long time to match.

Static code analysis tool

There are different techniques for static analysis and there are many tools that implement these techniques. OWASP collects information about all these tools, someone has tried to test these tools by creating benchmarks: a benchmark is a collection of programs that can be analysed by these tools and when you analyse them, you will find some of the vulnerabilities that are present into the program, they also notify the false positive and false negative of the tool.

Use: the tool itself will use some internal configuration (rules). Phases:

- Plan: decision of which are the tools to use, vulnerability to find and so on.
- Run: analysis of code in according with rules and produces raw results.

- Code review: (manual part) checks real errors and analyses reports. A report is more detailed, it's also good to try to estimate the risk. Some of the tools help you in providing this information.
- Fix code

Static Code Analysis Techniques: The most important techniques that these tools adopt are:

- Lexical scanning
- Structural Checking
  - type checking, style checking, etc.
- Control/Data-Flow Analysis
- Temporal Logic Checking
  - Model checkers
  - Theorem provers
- Symbolic Execution

Techniques Evaluation Metrics: the tools in according to 3 features:

- Precision: how the technique is precise in modelling the software behaviour.
- Depth/scope: how large is the context that the technique simultaneously considers (line, function, module, application). If we increase the precision and depth, the time also increases.
- Execution time/scalability: analyse very large code basis.

When the scope increases, execution time increases and increases more than linearity

Measuring Tool Accuracy :

- OWASP Benchmark Project: benchmark for Java (with scoring system)
- NIST SARD (Software Assurance Reference Dataset) created benchmark for other languages and it's a collection of open vulnerability cases
- WAVSEP

The OWASP Scoring System: is the scoring system that has been developed to compare the tools. For each test case, each tool produces a list of alarms classified as:

- True Positive (TP): the alarm points to a real vulnerability
- False Positive (FP): the alarm points to a false vulnerability

For each test case and tool we have:

- False Negative (FN): vulnerability not detected
- True Negative (TN): non-vulnerability correctly ignored

True Positive Rate (TPR) = TP / (TP+FN)
False Positive Rate (FPR) = FP / (FP+TN)

If we trace a graph (see slide 331), we can divide the area into two sub-areas. In the white one (the upper half), we have tools that behave "better than guessing", and in the other one (the lower half) we have tools that behave "worse than guessing".

Vulnerability assessment

Vulnerability Assessment is the process that evaluates all the vulnerabilities that may affect your system. Preliminary steps:

- Planning: you want to understand what is scoped and what is not in scope. There are different ways to structure the vulnerability analysis, mainly based on the data that you will access: one important aspect is the frequency of renovating the VA: if black-box, white-box and grey-box. You do the VA now, you cannot consider your system safe in one year from now.
- Information gathering: you acquire as much information as possible on the targets in the scope (topology, hosts, open services, operating systems and versions). You need to extract as much information as possible, this task is called *footprinting* and it is done with automated tools like Nmap, Nessus or OpenVAS.
- Scanning, detection and assessment of vulnerabilities: Using automated tools to perform scanning. Check if the current status of the network is compliant with the vulnerability management program. Draft the network vulnerability assessment results.
- Report results and identify countermeasures: report the consequence and the impact of them to the system starting from the most critical ones (prioritisation). PT is necessary for a complete analysis.

Risk Analysis approaches:

- Threat-oriented: start from the threats and you determine: the vulnerabilities, the assess impacts and probabilities based on the threats.

- Asset/impact-oriented: you start listing the assets that may be interesting for the attackers and what are the ones very important for you. Threats are derived from and associated to the assets (vulnerabilities play a slightly minor role).
- Vulnerability oriented: starting from a vulnerability, you search if there are some threats that can exploit it.

The NIST Risk Management Framework: it is one of the main results in the standardisation of the processes for the risk analysis. It is divided in four phases:

- Framing: you understand your system.
- Vulnerability and risk assessment: you try to understand what the risks are. So you identify threats, weaknesses, vulnerabilities and the possible consequences.
- Mitigate risks.
- Monitor

NMAP: It is a free and open-source tool for net discovery and security auditing. It can determine a lot of elements in the network like:

- Hosts.
- services (application name and version).
- operating systems (and OS versions).
- packet filters/firewalls in use.

With the port scanning it is assigned a tag to each port:

- Open
- Closed
- Filtered
- Filtered|open
- Filtered|closed

Scanning techniques:

- SYN scan
- TCP connect
- ACK scan
- UDP scan
- Version and OS detection

Greenbone Vulnerability Manage: it's open source, the main component to execute a vulnerability scanner are:

- Manager (gvmd?)
- GSA: is a web interface of the GVM that allows users to access the GVM features.
- The OpenVSA scanner

Metasploit: is not public. The idea is to have a framework able to execute payload.

CVE (Common Vulnerabilities and Exposures): A standard way of describing publicly disclosed cybersecurity vulnerabilities found in software.
CWE (Common Weakness Enumeration): a list of software and hardware weaknesses, the goal is to understand flaws and create automated tools that can be used to identify, fix and prevent those flaws. It is not guarantee that a CWE will result in a CVE: maybe it is not exploited or maybe it is just a false positive. CWE needs to be monitored.
CAPEC (Common Attack Pattern Enumeration and Classification): It is a community resource for identifying, categorising and understanding attacks. It's a dictionary of common attack patterns and for each of them:

- defines a challenge that an attacker may face
- provides a description of the common technique(s) used to meet the challenge
- presents recommended methods for mitigating an actual attack.

CVSS (Common Vulnerability Scoring System): to each vulnerability is associated a score, usually computed with the CVSS. The three metrics are:

- Base: score associated to a vulnerability depending on damages, exploitability, impact (range 0-10).
- Temporal: reflects the characteristics of a vulnerability that change over time.
- Environmental: characteristics of a vulnerability that are unique to a user's environment.

NISTIR 8011: a methodology for building a vulnerability management system into a company. Two point of view:

- End users: using a software that discovers vulnerabilities and wait patch.
- Developer: someone discovers a vulnerability in the software because if everyone knows that a software will be unpatched, this is not so good.

Vulnerability categorization:

- Patched: a patch exists and it has been already applied.
- Unpatched: a patch exists but it has not been applied.
- Zero-day: disclosed but no patches available yet (the most dangerous).

Build a vulnerability management system: the step zero is to know the system and collect into a database. NIST proposed to try to follow the standard workflow (CONOPS) taken from the documentation.

1. Actual states: determine the actual stage of exposure to vulnerabilities. If you know the system, you can list the version, use vulnerabilities tools and attach different labels.
2. Desired state: you have to define in a security policies your security requirements for what concerns vulnerability.
3. Find/Prioritize defects: you have to find the difference between the actual state and the desired state (CVEs and CWEs) and then list the thing you have to adverse (based on removing false positives) so take the appropriate response actions.
   Desired state software objects are the versions selected for lowest risk of unpatched vulnerabilities

Mitigations: You can decide to not apply that patch because it creates problems (NIST is more rigorous than others). One important idea that they propose is to identify two different roles:

- **Software Flaw Manager (SWFM):** this is on the side of the developer, the software vendor. He has to evaluate the severity, decide the reaction, create a new patch ecc.
- **Patch Manager (PatMan):** it's the corresponding role at the user side. He has to manage all the activities related to patching the system, so receives patches, applies patches to devices, tests patch interoperability, ecc.

Enterprise Patch Management: It is the management system who manages people, tasks, problems of the patches, ecc.
Talking about Planning Patch application, the important concepts are:

- Timing: decide a timing for reaction, all and now would be the ideal case.
- Prioritisation: decide what to patch first.
- Testing: don't apply patches without testing them, as patches can cause serious operational disruptions.
- Aggregate patches (vendor): solve more vulnerabilities at the same time.

Problem related to patches:

- Attacker may reverse-engineer patches, easy way to build exploits
- In some cases, after applying a patch the system doesn't work (so we lose money)
- Use automatic patch management tools instead of manual (if possible)

Technologies: means paying money for an integrated tool:

- Subset: a way to address this problem is to start from the subset of your information system
- Standard first: standard window for the user, server are not modified (best way to start)
- Difficult ones later: sophisticated, difficult ones not tested you leave them at the end

Types of patch management software:

- Agent-Based: requires an agent at each entity to patch.
- Agentless Scanning: a scanner determines what to patch.
- Passive Network Monitoring: scan the network to identify unwanted behaviour.
  (see slide 63 for more information and comparison)

SCAP (Security Content Automation Protocol): this is done by the NIST with this suite of information to manage the patch management system (verifying the installation of patches, checking system security configuration settings). SCAP provides:

- Languages: define common terms you will have the need to reporting formats
- Reporting formats: provide standard way to express collected information
- Enumeration: to say all possible cases, label, information
- Integrity protection
- Measurement and scoring systems

Binary exploitation

Shellcode: small piece of code used as the payload in the exploitation of a software vulnerability.

Mitigations:

- **Stack canaries:** random values added in the stack after each call, they are checked at function exit by the OS (performs a check between the original and the actual value of the canaries, if there is some mismatch a buffer overflow attack has been performed). Stack canaries are the same for all functions but different at each execution. The idea is to add a small piece of data before the FRAME and the return address. If you write and you want to change the return address, you have to write on top of these canaries (random data generated when the application is executed and stored in some part of the program).
- **DEP (Data Execution Prevention):** why should a program execute code from data segments? The stack is used to pass values to the functions, to store return addresses. Why should I take and execute instructions from the stack? Since you have to store data, I will not allow you to execute instructions, only code segments must be executable and segments must not be writable and executable at the same time, because you can't execute code on the stack.
- **ASLR (Address Space Layout Randomization):** we randomise the memory location where the system executables are loaded, so an attacker can't use a fixed address. The idea is to try to avoid putting the same segment in the same position all the time: we are not randomising the whole memory, we are randomising the starting address of each segment. This means that if you take static fixed addresses and then you execute again the application, your script will not work the second time that you perform the execution.

New Attack:

- **ROP:** is a way to bypass the problem of injecting your code and so bypass the DEP. So we don't execute code from the non-executable segments but we jump from an instruction to another in order to build our attack. The problem is that, when you jump, you execute the rest of the function, so we look only for instructions that are very close to a return (gadgets). The idea is that you first run a program like ropgadget (finds the gadget into the code and compile a list of them), then you build a ropchain, (creating a sequence of gadgets that does what we want to do).
- **RET2LIBC:** It is a technique effective as the ROP but easier to implement, where instead of looking for gadgets in the program you use the Standard C library. Here there are the most powerful functions to execute attacks.

Analysis tools

Reverse Engineering: it consists in understanding the way in which some components work starting from the black box. Everything that we extract with RE must be readable for humans and understandable, otherwise is not RE. The purpose of RE are:

- guess information on the source architecture, understanding high-level system description.
- reconstruct the source code and identify reusable components
- correct/adapt the binaries according to your needs
- check if software protections work, if programs you want to buy are secure enough, steal PI (Property Information)

Executable Binaries: the executable files (the binaries) include a lot of informations:

- Data
- Code (to be executed)
- Additional management information
  - Memory allocation
  - Symbol

Executable formats: describe how executables are structured and this structure depends on the execution platform. These ecosystem have a way to manage shared libraries, in the executable we have also information about dynamic linking (used by shared libraries)

- **ELF formats:**
  - **Header table:** contains information about all the info into ELF
  - **Symbol table:** symbols are name of functions/variables/libraries and all the information that have a name into the system

- **A section for dynamic linking**: contains info for the runtime loading of the shared library.

Each section could be Read Only, Write, Data, Code.
Tools to read the code: objdump or readelf.

Disassembler: from an executable binary, it generates a listing in assembly language. Disassembling is not a simple operation and it is not deterministic. The instruction set:

- Have variable-sized instruction set
- Is densely used (many codes are valid)
- Instructions may overlap.
- You may find data into the text section (hard for the disassembler)
- We need to consider jumps and calls, but function pointers, information by the linker and others may create desync. Solution: build the execution flow (hard problem)
- The disassembler not always knows the symbols
- Understanding what the parameters are to call the functions is hard
- Pointers may collide and you can have self-modifying code

There are two families:

- **Linear Sweep disassembler:** it's assumed that the instructions are in order. If you have overlapped instructions, data and code mixed, ecc, these things make the disassembler misaligned with the actual execution flow.
- **Recursive Traversal disassembler:** With a modified instruction, instead of going linearly, disassembles instructions following the (expected/reconstructed) execution flow constructed during disassembly. But it is clearly hard to determine the execution flow statically. So, there are usually pieces of code that are never reached by statically analysing the code because you need data that are available only at runtime.
- **Interactive disassembler:** the disassembler process works interactively with humans that resolves misinterpretations of data as code, but it means spending weeks in accepting or rejecting the hypothesis made by the tool.

Control Flow Reconstruction: It's a complex operation that reads the code, all the jumps, all the function calls, all the function pointers and all the other indirections that you can find into the program to build a Control Flow Graph (CFG). The nodes of this graph are called basic blocks Call Graphs, that is a CFG that only shows the function calls and returns.

Radare2: tool that can perform both static and dynamic analysis (the dynamic analysis means debugging). You can patch binaries, you have an advanced search for patterns, headers, and function signatures and it supports scripting (in python).

Decompiler: it is the opposite of compilers, so it reconstructs source files from the executables. Even the best decompilers are usually unable to perfectly reconstruct the original source code, but when they work they are better than disassemblers. An example of a decompiler is Ghidra.

Debuggers: there are debuggers that attach to the binaries and that are mainly intended for reconstructing the execution flow of the applications. You can perform different operation:

- Step, next, continue
- Insert breakpoint
- Examine the CPU status, the stack
- Write values in memory to force the behaviour of an application.
- ecc ..

One of the most famous debuggers is *gdb.*

Global Offset Table and Procedure Linkage Table: Lazy binding: you don't bind all the functions at the beginning when the program is loaded, the first time that you need a function, the OS with the dynamic linker will provide the precise address of the function that you are calling. If you are calling the printf, you don't know in the binaries, when these have been loaded in memory, where to find the actual code of the other functions. What you do is a very sophisticated procedure because of the **Position Independent Execution (PIE)**: the program can be loaded in any place in memory.

- **Produce Linking Table (PLT):** a table where you find the reference to all the dynamic functions available in our code. All the libc functions that you call will find a line into the PLT.
- **Global Offset Table (GOT):** the PLT contains the code that jumps to the GOT, a table where the code into the PLT will find the actual absolute address of the function to call. Inside the row

corresponding to the needed function, we have initially an address that points to a function of the address that is copied inside the row of the GOT. After this process is finished, the PLT returns the actual address and the system jumps to that address to execute the code. The second time, the PLT will find the correct address. **Attack exploiting the GOT:** there are different types of attack, to protect against them there is the RELRO and the full RELRO, that make the GOT read only.

Format strings attack: it is one of the most powerful attacks to overwrite pieces of memory when we control the strings that will be passed to the printf. The only way to perform this attack is to be in this situation: → *printf(buffer)* → so a printf without the "%" ecc.
To build an attack, we can prepare a very long string with ".%x", that means "*print one word (32 bits or 64 bits depending on the architecture) from the memory in hexadecimal format (dot is a separator)*".
How to perform the attack:

- python -c ("AAAA"+".%x"*number), you have to do this and search for '41414141' ('AAAA' in hex) so you can count the offset (the distance between the first word printed and our marker AAAA). "Number" is a number that we have to increment until we find the marker.
- python -c ("AAAA"+".%y$x"*number), y is the offset (a number), "$" in the printf asks to print what is directly pointed "y" words after the beginning.
- python -c ("addr"+".%y$s"*number), addr is the address that we want to read and understand what has, so we use %s and not %x. This attack doesn't work if the address is writable and not readable.

Knowing that, with the printf, we can use the %n option, that prints into the address passed as a corresponding parameter the number of characters that have been printed on the output of the printf before finding the %. So it's possible to make a more powerful attacks with format strings in this way:

- python -c ("addr"+"%numx"+".%y$n"*128);
  - y is the offset as before
  - addr is the address where we want to write (not read), so it has to be writable obviously.
  - num is the number of bytes to print - len(addr) → if we want to print in addr 9 and len(addr) is 4, num is 5 (so 5+4=9).

Tracing: list what is happening into an application. The main difference between debuggers and tracers is that the purpose of debuggers is to give full control to the user, the one of the tracers is instead to observe the application when it is executed without adding a lot of burden. For the tracer must not happen that the address of the variables in memory changes when you execute the application into the debugger or directly from the shell.

- **Ptrace (important tool):** final solution for everything that is executed in user space. You can do it automatically in a fork (the tracer must be a process), but in general is a system call that allows one process ("tracer") to observe and control the execution of another process ("tracee"). You can also use a trace to overwrite the values in memory to change the execution and force the process to go in other parts of the CFG. Ptrace is intended for making very strong debugging with multi-process applications. Protect ptrace:
  - Modules to avoid the attachment of tracer
  - You can have a look at the status of the PIDs.
  - To detect a ptraced process is using data structures (like EBF)
- **Strace:** based on ptrace. It traces and reports only the system calls. It doesn't have the full features and complexity of ptrace.
- **Ltrace:** based on ptrace, traces and reports calls to library functions.
- **Trace-cmd:** interface to configure the Ftrace tracer, designed to monitor what happens inside the kernel. Two phases:
  1. First collects raw information
  2. The buffers, where those information are prepared, are transformed in a more readable way.
- **SystemTap:** the most powerful tool at kernel level. There is a C-like language that you can use to generate the instructions for the tracer.

VM introspection (hints): some applications (malware) don't want to be traced so they check that they are not running on a virtual environment or in a sandbox before start.

Other dynamic analysis approaches

Fuzzing: invented to force the crash of an application by injecting fake/wrong inputs. What has been proved is that, a lot of times, the applications are vulnerable to unexpected inputs. A *fuzzer* is an automatic tool that generates a lot of inputs passed to the application. It's good for small applications. The idea is to inject fake

inputs and check for bugs (a sort of brute force). You spend a lot of time and memory and it's not guaranteed to find something.

Workflow:

1. Study the input format (what is valid or not)
2. Fuzz some data
3. Send the data to the application
4. Look for something strange in the execution
    a. If an error occurred, it is studied and reported to someone that can fix it
5. Repeat

It can be effective only to try all the possible inputs, but it's impossible so we need approximations. *Code coverage*: starting from a CFG, we know if a branch is entered or not, so we know how much code we are able to test. If the coverage doesn't grow, I have to change the input. We can fuzz files, network traffic, generic inputs.

Approaches:

- Depends on what you know about the application (white/grey/black box)
- Depends on the input they generates:
    - Generation-based: easy to configure but low coverage
    - Mutational: start from a valid input and mutate it
    - Model-based, grammar-based, protocol-based: require too much effort to setup
- Depends on the complexity of transformations (dumb, smart)

Pros and Cons:

- It allows detecting bugs and improves security testing
- Complements usual software testing procedures
- Effective because it's used by hackers
- Not able to describe bugs
- Requires resources and time
- Doesn't work in detecting unwanted behaviours
- Only detect simple faults or threats
- With non-white box approaches, you don't know how goods are these trials.

American Fuzzing Loop (AFL): is the most used fuzzing tool, you can setup it without having to waste months for preparing it. It is efficient, it uses genetic algorithms to trigger inputs that increase the code coverage. It is grey-box because it has some knowledge of the application. It is mutational because it works on the seeds. The major limitation is that it works in a single thread.

GCOV/LCOV: if we don't want to use AFL, we can directly use the gcc compiler. It is possible to inject some pieces of code that will save coverage information into GCOV files and then you can analyse them later manually or with LCOV

Fuzzing attack (evil fuzzing): finding bugs/crashes you can build a 0-days. If you make a server crash, you can perform a DoS attack. We cannot defend against fuzzing. You can only limit the consequences of crashes by making more complex understanding what is the real error of the crash.

Concolic analysis: The idea was to mix concrete and symbolic analysis, so instead of using only symbolic variables, you can force the value of some variables with some concrete values.

- **Angr:** it is a modular Python framework, the core is the simulation engine where we don't actually execute the application, we simulate its execution, where some operations are concrete and some others are symbolic. With Angr we can perform:
    - Binary Loading.
    - Static Analysis (CFG, BinDiff, Disassembly, Backward-Slice, Data-Flow Analysis, Value-Set Analysis, etc)
    - Binary rewriting.
    - Type inference.
    - Symbolic Execution.
    - Symbolically-assisted fuzzing (driller).
    - Automatic exploit generation.

    It is officially an analysis module:
        - exposes a control interface: the Project.
        - calls features, stores results, exposes interfaces to access all the data

How Angr works:

- **Binary loading:** is done with CLE (CLE Loads Everything) that is a generic loader able to support almost all the standard format for binaries. After this operation, we have a representation of the memory, of the registers and all the data that are typical of this application.
- **ArchInfo:** a set of classes in Python to describe all the architectures (registers, ARM processor, ecc)
- **SimEngine:** having the information about the architecture, the simulation engine knows what are the states and so can start the simulation. From now, every time that a single execution step is executed, the simulation engine predicts the changes without executing the data. It builds a lot of different branches, it simulates different parts.
- **PyVEX (Vector Extension)**: angr uses generic instructions, so the simulation engine doesn't need to know all the opcodes but just these generic instructions. There is also a *lifter* that is able to translate the specific opcodes into PyVEX instructions.
- **Claripy:** manages the concrete values and solves the symbolic expressions. It manipulates expressions for possible concrete values and can solve expressions using a SMT solver.
- **SimOS:** the generic framework for supporting different OSs but the only developed is for Linux. Angr provides SimLinux that defines the Linux specific objects that are mapped to the symbolic objects and provides symbolic representation of library functions so it is reasonably fast.

Software protection

Man At The End (MATE): An attacker that has full access and privileges on one endpoint, so he has physical access to devices where the software runs and/or unlimited access to analysis tools (static, dynamic, concolic, symbolic ecc).

- **Least resistance:** we have to consider that attackers want to make money in a short time. So what we have to do is try to delay the attackers. We have also to consider that we cannot protect all these versions very well on all the platforms and that an attacker will select the less protected one.
- **Economics of MATE attack:** The approach for software protection is to try to limit the revenues for the attackers. An attack will be convenient if it starts with the exploitation and the area that represents its gain is bigger than the area of attack building.

Software protection: Some mechanisms exist to limit the exploitation of some attacks. This mechanism can be useful against Man At The End. Software contains a lot of money of the company. It is also important to protect the data that are inside the application (GDPR)

Techniques:

- **Anti-reverse engineering:** limit the application of the tools. If you cannot attach a debugger, the disassembled code is not understandable or you have a lot of pieces not attached by the RT (recursive traversal) disassembler, you will waste a lot of time and resource
- **Obfuscation:** it makes the code much more complex for human beings to be understood. If it's not easy to understand I have to waste a lot of time trying to simplify the code. It's a simple wall that sooner or later will be destroyed (you need to stay up for at least a couple of days). It adds fake control flow, manipulating functions to hide their signature and avoid static reconstruction of the code (you are forced to do dynamic analysis). *Diablo* is one of the best working at the binary level, while *tigress* works at source code.
    - **Code obfuscation**
        - **Control Flow Flattering (CFF):** The CFF is transforming CF in a flat code. You will use gdb (not radare2). Use the debuggers to understand the code takes much more time compared to the static analysis. After some executions, you will have an idea for example the meaning of the state variables.
        - **Branch functions:** They substitute a jump instruction to a call of a function that assigns the value of some registers. This makes it difficult to find the CFG. This makes it very difficult to disassemble the code or at least it needs a lot of time to disassemble it by running the code multiple times.
        - **Opaque Predicates:** Predicates that are always true or always false. A part of the CFG will never be executed (only looking at the static code you will know). There are also scripts that disable some OPs manipulating the code, but you don't know which are really OP or real predicates and you will waste a lot of time to discover them.
        - **Merge:** The idea is to merge the atomic functions that design your code in a very big function, so all become more complex and the CFG is more difficult to understand.

        - **Split:** You can take more understandable functions and split them into independent functions at random places so again the CFG becomes complex and so an attacker wastes more time.
        - **Virtualization obfuscation:** We have a piece of code with some opcodes, that is the part that we want to protect, then from these opcodes you generate a new instruction set. You cannot execute the new opcodes, because they are not understood by the CPU. What you have to do is to instruct a virtual interpreter. We mainly do a sort of virtual interpreter, we generate randomly an instruction set and when the code arrives to the opcodes that we changed before, we start the virtual interpreter that will generate the real opcodes that we changed before that will be understood by the CPU. No static analysis will be possible.
        - **Just in time opcode generation:** translates a function F into a new function F' with some logical/mathematical operations. With this technique we force the attacker to execute the code.
    - **Data obfuscation**
        - **White box crypto:** since it is impossible to hide a symmetric key in the memory, due to the fact that it is possible to create a patch to dump it. There are some techniques that obfuscate the key in the encode and the decode. It's security-through-obscurity since it is very simple to break most of these techniques.
- **Anti-debugging:** Using ptrace we can control an application with a debugger attached to it. To defend our-self from this, the application is always run with ptrace, so no attacker can attach a new debugger to it. Since it is also easy to detach a debugger, the developer moves some part of the code to the debugger, so the attacker needs to reverse engineer the debugger and patch the application to make the application work.
- **Anti-tampering:** The application checks run time if there are some cracks or there were some changes not authorised to the code. Every time a change is found, the application will crash.
  Remote techniques:
    - **Software attestation:** It is a form of anti-tampering that uses a server to send to the client not only the application logic, but also an *Attester* , which has the objective to collect evidence that the application is working properly. There is a manager that from time to time sends *attestation requests*, the Attester responds and sends the collected data to a *Verifier.* To verify the collected evidence and in case of some strange behaviour there is a *Reaction Manager* that disconnects you from the server. In software attestation, we don't have any hardware on the client. Everything is in the software, so it is easy to attack it. The techniques that use hardware are called remote attestation techniques. The possible attacks are:
        - dynamic code injection (i.e., with debuggers)
        - parallel execution of an untampered version of the device

      Approaches to have a better execution corrections (but no one really work, there are always some problem):
        - measurement of the time spent to execute a particular piece of code (with tampered will be slower so we can notice it)
        - *Invariants* are properties in the code that are always true or false, so we check this properties, but an attacker can modify all except these, so we cannot notice the tampered code
        - We have some counters that check if you entered too many times in a part where you were not allowed to enter and so the crack is discovered.
    - **Remote attestation:** very similar to Software attestation but it is using hardware. You have the public key, the hash functions and secrets are not available in the memory. The attacker needs to be in the same place of the machine that he would like to attack. This is why the Trusted Computing Group developed the remote attestation idea.

  Local technique:
    - **Code guards:** It's only local, the developer adds a function that controls the correct function of the application by reading the memory run time or doing some check on the code. These control functions are hidden in the code so it is difficult for an attacker to find.
- **Code mobility:** In this case some parts of the application are downloaded from the server and once a while new versions are needed with different blocks online and others in local. The attacker will never have the full version local, only if he tries to reconstruct it from the different blocks that he has local in the different versions.
- **Client-Server Code Splitting:** Very similar to code mobility. The application locally asks the server for some parts that are executed on the server. On the server are saved a lot of small blocks instead of less but bigger ones. This way it is more difficult to reverse-engineer it.

- **Diversification:** If we have all the same copy, the crack will work for all of us. With diversification, developers produce thousands of different versions of the same application (the idea is to use a lot of protections and obfuscation techniques to generate different copies), the crack will work only with a small fraction of the applications.
- **Renewability:** updated the application very frequently, so we are limiting the impact of the previous cracks (previous crack won't work if we update the application)

How software is protected, nowadays: Most of the protections are patented and hidden, so you don't have the possibility to look at how your code is protected. This is security through obscurity that in general doesn't work but it can be small advantage if you have to resist for 20 days.

Software protections: categorization:

- **Prevention:**
  - **Anti-reverse engineering:**
  - **Obfuscation:**
    - **Code obfuscation**
      - **CFF**
      - **Branch functions**
      - **OP**
      - **Split/Merge**
      - **Virtualization obfuscation**
      - **Just in time opcode generation**
    - **Data obfuscation**
      - **White box crypto**
  - **Anti-debugging**
  - **Anti-tampering:**
- **Protection:**
  - **Offline (local):** are the ones injected in the binaries or in the source code of the application to protect.
  - **Online (remote):** are techniques that use remote entities, so not in your platform but maybe a trusted server located somewhere on the internet.
- **Abstraction:**
  - **Source code:** protections that take as input source code.
  - **Binaries:** protection that works at the level of binaries.

How to evaluate protections: we introduced the idea of potency that is not a formula to measure it, it is just a philosophic concept. Two different models:

- **Objective metrics:** can be used to estimate the potency, but only the 10/12 can be actually measured.
- **Empirical experiments:** controlled experiments that involve people to measure the times and the successes and derive evaluation of the effectiveness.

Limitations:

- High values of the metrics do not necessarily correspond to what people perceive as complex.
- Measuring effectiveness with experiments would require millions of experiments, which is not feasible.
- Open issues:
  - find meaningful measures of potency.
  - define formulas that work in practice.
  - mix objective metrics + empirical approach.
  - use predictive approaches.

Layered protection: It is a sort of defence-in-depth, so applying more protections has proven to be much more effective (more than one protection also on the same piece of code). We can also say that some protections have complementary behaviours, for example anti-tampering (delays modifications) works very well with some forms of obfuscation (delay comprehension).

Overhead: if we implement protection in our system, we will probably add several forms of overhead (sovraccarico), for example with obfuscation our application will become slower. All the possible overhead are:

- complex code is not as optimised as the original one

- pieces of bogus code.
- pieces of code for checking integrity
- new data added only needed for the protections

These overhead principally affect CPU, RAM and bandwidth.

Web attack

JavaScript and browser: The web attacks are very frequent and the two main categories involved into the web interaction are: the server and the client.

- **SOP (Same-origin policy):** It says that you can access data, cookies and all the stored values in the browser only if you come from the same domain, so someone from the internet cannot write a script to monitor your activities and steal your data. But SOP is too restrictive for a good user experience, so they started to relax the SOP in different approaches.
- **Sandboxing:** your web pages are executed in a sandbox, so they don't access your OS and resources without your permission. The idea of the sandboxing is that you are not completely isolated, but when there is a need for accessing the OS a window appears.
- **HTTPS stateless:** Since the HTTP is fully stateless, they invented the cookies: pieces that contain informations that are useful to the servers to perform their operations and are sent to the client. The problem is that they need protection.
- **Cookie:** size up to 4KB and are structured in this way:
  - Name (mandatory)
  - Value (mandatory).
  - Expiry.
  - Path.
  - Domain.
  - Need to be transmitted over a secure connection.
  - Cookie accessible through other means than HTTP (i.e., JavaScript)

  Types:
  - first party cookies
  - third party cookies
  - session cookies
  - permanent cookies
- **Injection:** it is whatever you can send to a web server and that is passed to an interpreter.
  - **SQL injection:**
    - Example 1 php: Username = *any_name* Password = *123' OR 'x'='x*
      So we have an OR with another statement that is always true. In this way, we can bypass the login procedure.
    - Example 2 php: write something more and add another statement using 'UNION'
  - **SQL map:** is an automatic tool to check for potential injection vulnerabilities. There are much more sophisticated queries that you can do and these are usually named blind queries, time-based queries and double-blind queries
    - Blind queries: you don't see the answer of your query, you see an answer made sophisticated by forcing the different output generated by the website. Implementing this kind of queries is extremely complicated, because you have to guess a lot of things about what's happening in the database.
    - Time-based: you notice that if the user is correct than the answer is returned after 1 second, otherwise after 1 millisecond, so, you can create a brute force attack by guessing the letters of the password looking at the time waited for the response.
  - **Detection:** you can use static analysis tools, dynamic analysis tools, PT to find the places where there is the risk of injection.
  - **Prevention:** you have to check the APIs, all the interpreters, all the commands, so for example using parameterized APIS and prepared statements.
  - **Secure Programming principles (for injection):**
    - Validate input.
    - Heed compiler warnings.
    - Architect and design for security policies.
    - Least privilege.
    - Sanitise data sent to other systems.
    - Adopt a secure coding standard.
- **Cross site scripting (XSS):** It forces a user to execute into the browser some scripts that can produce some outputs. So an attacker can steal the cookies, in other cases can redirect the user in a

malicious way to a malicious site and so on (normally only the victim's browser is affected by this attack).

- **Categories:**
  - Reflected XSS: attacker uses other alternative methods (ex. link sent via e-mail, redirect of Web pages) to deliver the script to the user.
  - Stored XSS: it is even more dangerous than Reflected XSS because now the script injected by the attacker is not immediately reflected to the victim, but it is stored in a database. Later, maybe another application that uses the same DB may read the information and reflect it to another user.
  - DOM XSS: based on altering the DOM environment in victim's browser, for ensuring that the (original) script is executed in some unexpected way.
- **Detection:** not a sophisticated operation
  - If the parameters that you pass to the APIs can contain the character to inject a script, a reflected XSS is possible to do.
  - If you write a script in an input text and it is inserted into the web page, we can say that is possible to do a stored XSS.
  - Need to check if the methods of the DOM objects can be freely accessible from some inputs that are controlled by the attacker (for DOM XSS).
- **Prevention:** you need to block any untrusted data and you don't have to save or process data from untrusted sources without any sanitization.
- **Secure programming principles:**
  - validate input.
  - architect and design for security policies.
  - sanitise data sent to other systems.
  - use effective quality assurance techniques.
  - adopt a secure coding standard.

- **Insecure deserialization:** attack based on the fact that when websites and clients exchange data, they usually use objects (ex. JSON). Serialisation is a deterministic process that produces as output a text string, if an attacker can perform insecure deserialization he can bypass authentication, control the OS and so on.
  - **Detection:** the two primary types of attacks are:
    - Object and data structure related attack
    - Data tampering attack
  - **Prevention:** isolate deserialising code and run on a low privileges environment.
  - **Secure programing principles:**
    - validate input.
    - heed compiler warnings.
    - architect and design for security policies.
    - least privilege.
    - sanitise data sent to other systems.
    - use effective quality assurance techniques.
    - adopt a secure coding standard.