- **Responsible disclosure** is an ethical disclosure policy in which after a vulnerability has been found, instead of immediately disclosing it publicly, the ones that found it inform the developers and give them a period of time, called *grace time*, after which the vulnerability will become public. This gives some time to the developer to release the patch. This can help reduce the risk, if the developers manage to create a patch in time.

- **0-day vulnerability** is a privately-known vulnerability (so, an exploit exists), not known to the developers/owners of the system.

- **Common Criteria** is a security evaluation standard for IT products. It aims at defining a common standard reference for conducting security evaluation and certification, while permitting comparability between independent evaluations. It does so by providing a uniform/standard approach to evaluation/certification, a way of expressing security requirements and assurance levels and constraints on the evaluation methodology (that are defined in the companion document *CEM*).

- **Target Of Evaluation** in the context of CC is the system or component under evaluation. It has a set of *Security Functional Requirements* and *Security Assurance Requirements*. To enforce the SFR, we need to rely on the TOE's *Security Functionalities*, *TSF*, (which are parts of the TOE).

- **Protection Profile** is a set of implementation independent Security Requirements for a category of TOEs that meet specific customer needs.

- **Security Target** instead is the set of SR to be used as a basis for the evaluation of a particular TOE. The evaluation aims at verifying whether the Security Functionalities of the TOE are sufficient to satisfy the requirements.

- **Evaluation Assurance Levels (EAL)** are a key concept in CC: they're a numeric rating that defines the level of assurance reached by a security evaluation performed on a given product. There are 7 levels, each of which requires a set of components, with the possibility of introducing intermediate ones. To pass from one level to the next, we either increase the number of assurance components or replace components with higher-level assurance ones of the same family. As we grow up the levels, the rigour increases.

-
- **Security assurance** is the measurable confidence that an entity meets its security requirements. It is obtained by applying different assurance techniques such as use of security controls and mechanism, use of a development methodology, use of security assessment techniques, etc. Those techniques can be applied at different stages of development (policy assurance, design assurance, implementation assurance, operational assurance) and with different levels of rigour (formal, semi-formal, informal).
  In the context of CC, assurance techniques correspond to the assurance requirements, and Assurance is achieved by evaluating the evidence that the intended Assurance Techniques have been applied and performing independent verification/testing activities. Some *assurance evaluation techniques* are *verification of proof*, *penetration testing*, *analysis of guidance documents*, *analysis for vulnerabilities*, *etc*. Those assurance techniques can be applied with different levels of rigour, scope and depth.

- **Assurance Effort** is defined by what assurance techniques have been applied.

- **Security assessment** is the process of evaluating the security of a system, network, process or product to find vulnerabilities and potential risks. It can be performed by internal or external staff/consultants and it has the goal of identifying potential vulnerabilities and weaknesses to provide possible mitigating solutions.

- **Security certification** is a process that aims at defining and certifying that a product, process or company meets a specific set of security standards or guidelines. It is based on the evidence of employed assurance techniques, results, evaluation methodology and independence/accreditation of the evaluator. It must be done by an independent third-party that reviews the system or product, performing testing and providing formal certification that the system meets the specified security requirements.

- **Evaluation Schema**, in the context of *Common Criteria*, is a framework/methodology of evaluation for IT products security. Each *Authorising Nation* signer of the CCRA has its own.

- **Formal specifications** are techniques used to define a unambiguous (no multiple interpretations), consistent (no internal contradictions) and complete (all relevant information is represented in the model) description of a system and its properties. They're based on mathematical models that introduce abstraction (of different possible levels). It can be used at different stages of development and for different specifications: requirements, structural or behavioural.

- **Formal verification** is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics. It can be done to check the self-consistency of a formal model obtained by a *formal specification*,

that a behavioural model satisfies its formal requirements or the cross-consistency of two formal models.

- **Program Slicing** is an over-approximation technique that is directly applied on the code, without the need of building its state-transition model. It works by defining a set *C* of variables that influence the property *f* we want to check. We then remove all the variables from the code that do not appear in *C* and all the instructions that do not influence the variables in *C*. Lastly, we substitute the decisions (branch conditions) of removed variables with non-deterministic choices.

- **Non-determinism** is a concept used in model representation to represent execution aspects that are not known a-priori, such as inputs and the scheduling of concurrent processes from the OS. It can also be used to represent different possible implementation choices.

- **Transition System** is a mathematical model used to represent a system and its behaviour in the form of states and transitions. It is formalised as a triple (S, init, ᵽ):
    - S is the set of states. In the state-transition model of a sequential program, a state is a pair <s,v> where s is the control state and v is the set of values of the variables.
    - Init is the initial states, and belongs to S
    - ᵽ is the set of transitions SxS

- **Control Flow Graph** is a direct graph that describes the control flow of a sequential program, what are its statements and how they are connected. In *code-reconstruction*, it can be intended as the over-approximation of all the possible sequences of program locations that will be executed by a given program.
  Its nodes are basic blocks.

- **Basic Block** is a set of uninterrupted instructions, with only one way in and one way out, without jumps inside.

- **Call Graph** is a CFG that only shows function calls.

- **Propositional Logic** is used to specify the behaviour of a system in a descriptive style, allowing to give a rigorous representation without specifying what happens in each state. PL is based on atomic propositions and operators that can be used to combine the propositions. A formula is the combination of AP and operators. To give a meaning to PL it is required to give an *interpretation*, the *semantics* that map atomic propositions to True or False and the operators (and, or, not) to their meaning (formalised using *truth tables*).

- **Satisfiable formula** is a formula that holds true for at least one interpretation of the Atomic propositions.

- **Valid formula** is a formula that holds true for every interpretation of the Atomic propositions (i.e. a *tautology*).

- **Predicate Logic (First Order Logic)** is an extension of propositional logic, where atomic propositions have been replaced by predicates. It also introduces the quantifiers 'for each' and 'exists' and adds the concepts of *constants*, *variables*, *relations* and *functions*. It is possible to define an interpretation but it is more complex than in the Proposition Logic, as we need to map the data (constants, functions, predicates...) to a Domain that is not just {T,F}. For *closed formulas*, the ones without free variables, the interpretation does map it onto {T,F} but for open formulas (with n free variables) it maps them to a relation on D^n, where D is the domain.

- **Temporal logic** is an extension of classical logic that also allows us to describe the temporal evolution of facts. Can be based on propositional or First order logic, the time in it can be continuous or discrete, linear or branching, implicit or real. It can also be defined as event or state, instant or interval, past or future. An example is the *Linear Temporal Logic* that introduces temporal operators such as [], ○, ♢, U, where given a predicate P
    - []P means that P must hold true for all future states.
    - ○P means that P must hold true in the next state.
    - ♢P means that P will eventually be true in a future state.
    - P U Q means that P will hold true until Q becomes true.

- **Formal system (Theory)** is a way to formalise a logic, used for formulas that are defined on large or infinite domains. It is the combination of
    - a *formal language* made of an alphabet of symbols and a set of well-formed formulas.
    - a *deductive apparatus/system* that is used to give meaning to the formulas. It is a set of *axioms* and *inference rules*.

  Using an apparatus to define the semantics gives us the possibility to do the *proof* of a theorem.

- **Proof** is a sequence of wff P1,...,Pn such that Pi is either an axiom or a direct consequence of some of the previous P according to an inference rule.

- **Theorem** is a wff P such that exists a proof that terminates with P.

- **Model Checking** is a formal verification technique that answers to the question 'does the property *f* hold true under the interpretation M?'. It uses a model checker that takes as inputs a model M (the interpretation)

and a property *f*, a wff in the Formal System. It answers True unless it finds a counterexample in which case returns False. It can be applied directly to any model, however as it works by means of *state exploration*, it can only answer 'True' if the model is a finite-state one. In case of infinite state models, the model checker is not able to answer. There are, however, *abstraction* techniques that can help reduce the infinite state model to a finite-state one. Another weakness of the state-exploration technique used by Model checkers is the *state explosion* that happens when working with concurrent systems. State exploration can be done either *explicitly*, generating each state and run, or *symbolically*, a more sophisticated technique to represent states and transitions in a more compact way.

- **Reachability analysis** is a particular case of state exploration, used by model checkers, when the property to verify is a simple temporal property []P. It is sufficient to generate all the states and check in each state if the formula holds true.

- **Theorem proving** is a formal verification technique that can be used to prove whether a given property is a theorem in a formal system. It takes as input a property *f* (a wff in the formal system) and a Formal System (a Theory), which means that is not directly applicable to any interpretation but it needs to be 'translated' into a deductive system made of axioms and rules. It returns, as answer, either:
  - 'Yes' and a proof, if it manages to find one.
  - Proof not found, which does not mean that the property is false (even if it didn't find it, it might still exist).

If the transformation is done correctly, when the prover returns a proof, it means that the property is true in the system. The transformation must be *sound* to guarantee that a property found as 'True' actually is valid in the original system. As the Theorem prover does not operate using state

exploration, it does not suffer from state explosion. Theorem provers can be automated or require assistance (Interactive TP).

- **Sound** means that if a property holds true in the formal system, it also holds in the interpretation K (|-f ⇒ K|=f). Is the most important part of 'correctness' for Theorem Proving (the other being *complete*). A sound abstraction is said to be *correctness-preserving*.

- **Complete** means that if the property *f* holds true in the interpretation K, then it also holds true in the formal system (K|=f ⇒ |-f). A complete abstraction is said to be *error-preserving*.

- **Proverif** is an automated theorem proving tool based on Dolev-Yao modelling. It allows us to verify the validity of certain properties (queries) of a security protocol expressed in the form of Pi-calculus (either typed or untyped), by means of a theorem prover. To do so, it first translates the pi-calculus model into a deductive system in the form of Horn clauses. This transformation is an *over-approximation* which means that it might introduce false positives (for instance, in the Horn clauses model the number of times a message is sent is not represented). Proverif then proceeds to find a proof of validity for the queries. If it fails, it starts an attack-reconstruction phase, which is not based on theorem proving, to find a potential attack (a counterexample). Note that even if it finds an attack, it might be a false positive (due to the over-approximation), and even if it fails to find one, it does not mean that it doesn't exist.

- **Correspondence** is a property that states that one event eB can only happen after an event bA has occurred. They are used to define order relationships between events, for authentication and data integrity properties. In *Proverif* it can be expressed as

  $event(eB(...)) \Rightarrow event(bA(...))$.

A stricter form of correspondence is the *injective correspondence* that also imposes that to each event bA corresponds at most 1 event eB (a.k.a. Each occurrence of eB corresponds to a distinct bA), whilst in the basic correspondence, 1 single bA may correspond to multiple eB. In *Proverif* it is expressed as

$$inj\text{-}event(eB(..)) \Rightarrow inj\text{-}event(bA(...)).$$

- **Secrecy** means that an attacker must not be able to get *closed terms* that are intended to be secret. Formally it can be defined as: given an S-Adversary Q with an initial knowledge, a Trace T of a Process P and a secret N, we say that the *closed process P preservers the secrecy of N from S-Adversaries* if $\forall Q,\ \forall T \text{ executed by } P|Q,\ T \text{ does not output } N$.

In *Proverif* the query to check the secrecy of the secret s is *query attacker(s)*.

- **Strong secrecy** is a property that defines that an attacker must not be able to get partial knowledge of a secret. More generally we can say that the attacker must not be able to distinguish between different runs of the processes with different values of the secret: P[x/N] (the process with 'x' as the secret N) and P[y/N] must be *observationally equivalent* for an S-Adversary ($P[x/N] \approx_s P[y/N]$).

The S-Adversary cannot acquire any information on N by interacting with P.

- **Weak secret** is a secret that is subject to offline guessing attacks. To verify that a secret is not weak (query: *weaksecret s*) *Proverif* uses a mechanism based on phases.

- **Forward secrecy** means that the secrecy of a secret exchanged during a phase of a protocol cannot be compromised after that phase has terminated, not even if other secrets are disclosed.

- **Taint analysis** is a form of data flow analysis that is used to identify and track the flow of data coming from an untrusted source (tainted data). It is used to identify the source, track the flow and find possible sinks where the tainted data are used. It also studies the condition in which the data reach the sink, as instructions along the way can either maintain the taint status (*pass trough*), sanitise it (*clean*) or propagate it to other variables. Every time a tainted value is used in a potentially dangerous way, such as being passed as part of a SQL query to an interpreter, the analysis will flag it as a potential vulnerability. There are 2 main applications for Taint analysis:
    - Detect Leakage of secret informations
    - Detect injection vulnerabilities

  Taint analysis can also be useful to find possible *buffer overflow* vulnerabilities (but it's not sufficient, as it alone does not allow to understand whether the buffer overflows or not) and *format string vulnerabilities*.

- **OWASP scoring System** is used to evaluate the performance of static analysis vulnerability detection tools. It works by computing 2 values:
    - X: FP_Rate = FP/(FP+TN), the rate of false positives identified by the tool wrt the total of actual non-vulnerabilities.
    - Y: TP_Rate = TP/(TP+FN), the rate of true positives wrt the total of vulnerabilities actually present.

  They are computed as rates rather than absolute values so that the actual number of vulnerabilities in the application does not count, thus making different analyses comparable. These two values are used on a cartesian plane to define a squared area. The diagonal line of the area represents a tool with 50% probability of correctly defining a vulnerability, a.k.a. a random-choice tool (while an ideal tool would reside on the Y axis). To be considered acceptable, a tool's point

(FP_Rate, TP_Rate) must reside above the line. The tool's score is computed by the distance of its point from the line.

- **Enterprise Patch Management System**: patch management is the process of identifying, collecting, applying and verifying patches for products and systems. EPM reduces the time spent by the company to deal with patches, allowing it to employ more resources on other security concerns. Patches are used to fix security and functionality problems, mitigating vulnerabilities thus reducing exploitation opportunities, and adding new features to the softwares. EPM Systems can employ automated tools for patch management. Those tools allow continuous monitoring and patch application, reducing the impact of human error. It is important to notice that those tools do introduce other vulnerabilities: an attacker could submit a tampered patch that would be then automatically installed by the tool, or the tool could be attacked and then used to apply malicious patches. However, it is usually better to have these tools rather than not.

- **Same Origin Policy** is a security policy that aims at reducing the possible interaction between web pages running on a browser, to avoid scripts from a page accessing sensitive data on another with a different origin. A web page with a given origin (protocol, host, port) cannot interact with the DOM, the data and the cookies set by another page with a different origin. Some HTML tags such as *img* and *script*, however, can issue GET requests to different origins. As SOP was considered too restrictive for a good user experience, nowadays it is usually replaced with a more relaxed form, such as Cross-Origin Resource Sharing or Cross-Document Messaging.

- **Vulnerabilities Assessment** is the process of identifying (and reporting) the vulnerabilities of a system. Can be conducted statically or

dynamically, with a white/grey/black-box approach. It produces a report with the list of found vulnerabilities.

- **Penetration Testing** is the identification of vulnerabilities in a system and the attempt to exploit them to assess what an attacker can gain from an attack and the damages it can cause. The first phases are comparable to a black-box vulnerability assessment. It goes through different stages: *pre-engagement*, *information gathering*, *threat modelling*, *vulnerability analysis*, *exploitation*, *post-exploitation* and *reporting*. In the end it will report a list of vulnerabilities that can be exploited and what can be gained. It's generally more expensive than *vulnerability assessment* as it requires high expertise.

- **Buffer Overflow** is a type of vulnerability that exploits a writable buffer whose boundaries are not checked. This allows an attacker to overwrite the stack and execute arbitrary code, alter the program's behaviour, skip pieces of code, set variables' values, etc., by providing an input bigger than the buffer's size. An example could be inserting into the buffer a shellcode containing the instructions to open a shell, overflowing the buffer so that we're able to overwrite the return address with the address to the first instruction of our payload (or the first of the buffer, if we encased the payload in NOP instructions).

- **ROP and Ret2libC** are two types of attacks that exploit a buffer overflow vulnerability and let the attacker circumvent Data Execution Prevention policies. Both work by using code that already exists in the program, by inserting the address of the pieces of code to exec in the buffer. The differences are:
    - Return Oriented Programming uses small sequences of instructions close to a ret called *gadgets*, that have been found by use of other tools such as *Ropgadget*. In the

modified stack, the attacker puts the addresses of these gadgets (it builds a rop-chain) and manages eventual parameters.
    - Ret2libC uses functions present in the standard C library. In this case, beside the address to those functions, the attacker also needs to set up the stack to contain the expected values of the called function. It is usually easier to do than ROP.

- **Data Execution Prevention** is an execution policy that forbids code execution from data segments of a program. A segment cannot be writable and executable at the same time, and if code is executed from a writable segment a segmentation fault is generated. This technique aims at preventing arbitrary code execution from zones where the code can be rewritten, like the stack.

- **Disassembler** is a tool that tries to translate a binary into assembly instruction. I said tries because disassembling is not a deterministic operation, due to multiple factors: variable-size instruction sets, densely packed instruction sets, compiler-optimised code that causes instructions' bytes to overlap, information known only at run-time.

- **Linear Sweep disassembler** is the simplest type of disassembler. It works based on the assumption that the opcodes are stored in adjacent positions. Although it is the easiest to implement, it does not work except for very small portions of code, as it cannot manage overlapping instructions, data in the code, and more generally whatever makes the disassembled code misaligned with the execution flow.

- **Recursive disassembler** is the most common type of disassembler. It works by conducting a depth-first search in the code, reconstructing the

Control Flow Diagram and only analysing the reached code by following the jumps that it found. This lets it ignore the data that might be between code, however it also leaves part of the code uncovered. Some of the more sophisticated tools, once finished the DFS from the entry point, start a new DFS from an uncovered address, and in the end it will then produce a CFG made by disconnected pieces that need to be manually composed. This type of disassembler has problems with indirect branches and calls that might refer to runtime-only information.

- **Software obfuscation** is a family of software protection techniques that aims at discouraging an attacker to attack our code by making the code harder to read and understand. It also tries to force dynamic analysis, which is more time consuming. We can divide it into data obfuscation and code obfuscation. For code obfuscation we have a series of techniques that:
    - Fragment a large function in many small ones or fuse many small functions, like getters and setters, into one (*split/merge*).
    - Add fake blocks/instruction that seems like they could be executed but actually will never be, e.g. because the branch condition is a tautology (*opaque predicates*)
    - Make the control flow unintelligible, by adding function calls in place of direct jumps, breaking the CF in the basic blocks, randomise their order and insert them in a complex-switch statement that ensures they are executed correctly but whose condition can be known only at runtime (*branch functions*, *control flow flattening*)
    - Hide the actual opcodes used, by producing a virtual instruction set (with an interpreter) that is translated to the actual instructions at runtime (*virtualization obfuscation*)

    - In general, force the dynamic analysis, avoiding static reconstruction

  For data obfuscation we can hide constants, by means of *systems of equations*, and variables, using *homomorphic transformations*, or perform *white-box cryptography* (in which the key is also hidden in the code). Software obfuscation does not prevent attacks, it makes them only harder to do; as said before, the aim is to discourage the attackers (by making so that the effort/money invested is not worth it).

- **Anti-tampering** is a category of software protection techniques that aim to avoid, detect and react to non-authorized changes in the code and in the program behaviour, or at least increase the complexity required to make those changes. Anti-tampering techniques can be categorised based on whether they resort to external components (*remote*) or not (*local*), and if they use secure hardware (as a *root-of-trust*) or not. It is implemented either as *application integrity* or *execution correctness*.

- **Software Attestation** is a set of techniques that are part of the Anti-tampering software protection. They work based on the use of a server as a root-of-trust, so without leaning on a secure hardware (like *Remote Attestation* techniques), which is better for mobile devices and IoT (and in general, for any device without SHW). During the execution, the software and the server communicate and the RA manager of the server periodically asks to the software's Attester an attestation proof, which could be a checksum of the binary/configuration files in the system or a checksum of the binary loaded in memory. The Server then compares the attestation response with the expected one and in case of missing correspondence, triggers some reaction in the software (crashes, slows down, cuts connection, etc.). This kind of protection is weak to cloning attacks and dynamic code injection. The SA works on the principle of *application integrity*, that assumes that if the binary are the

expected ones the code will behave accordingly. A better principle would be *execution correctness*, more sophisticated and requiring a formal model of the process. Unfortunately a working anti-tampering technique based on this concept does not exist yet.

- **Remote attestation** is a type of anti-tampering techniques that verify if a program running on another system is behaving as expected using secure hardware as a root-of-trust. Current RA methodologies have limits (besides ofc requiring a secure HW): they do not scale well for virtualization.

- **Code Guards** is a family of local anti-tampering techniques that employ constructs inside the application code to prevent and react to changes in the code behaviour. They are usually implemented as pieces of code that performs checksums on values in memory or executed instructions. Another variant are the so called *crypto guards* which decrypts the next block of instructions only if the previous one has been executed correctly. Code guards react by preventing the execution of the rest of the code, causing crashes, graceful degradation, etc. The reaction must be delayed, because the guards are inside the code, and immediately reacting when an error is found might give too much information to the attacker. Code guards are usually employed in a combination (guards that protect other guards) or in overlapping disposition, and often require the attacker to remove all of them. They are easier to implement, less expensive, but they are also subjected to cloning attacks and as the guards are inside the code they result less secure than Software or Remote attestation. They still have the advantage of not requiring an online connection.

- **Fuzzing** is a testing strategy that works by generating random inputs to feed to the application and logging eventual errors/crashes/faults in order to find bugs.

- **Fuzzers** are tools used to conduct Fuzzy Testing, able to generate the random inputs and log the possible errors/crash/faults. They are used to find bugs, perform regression testing and combined with other analysis tools to improve the analysis result. The generation of the inputs can be of different types: *generation-based* (random), *mutational* (starting from a seed/example/file and then applying techniques to generate more) or *model-based* (which requires a mathematical model, a formal representation of the inputs). Fuzzers can work with a white-box, grey-box or black-box approach, but without WB it is difficult to find boundaries (and compute coverage, which is normally used to define interesting inputs).
  Fuzzing is technique that can give insight on bugs (and possibly, vulnerabilities) that might not trigger under standard testing, but it is not free from disadvantages:
    - it is expensive both in terms of computation and time
    - it might not give enough information about the bugs it finds
    - can only detect errors that lead to crash/error messages/memory leaks etc. and does not cover unwanted behaviours that do not cause these reactions.

  Some tools allow to speed up the process by running multiple instances with different generation seeds; single-thread tools like AFL do not work well in parallel as many of the inputs would be duplicated.
  Fuzzing is also used by attackers to find bugs and potentially vulnerabilities, to perform DoS or other kinds of attacks.
  You can't really avoid fuzzing attacks but one can mitigate the threat by means of firewall, least privileges and releasing software with less bugs.

- **Corpus minimization** is a strategy adopted by some *Fuzzers*, such as the *American Fuzzy Lop*, that consists in reducing the set of initial files/seeds (called *corpus*) removing the unneeded ones (e.g. the ones that do not increase coverage). The fuzzer then works like "fuzz, reduce, fuzz again". This operation speeds up the fuzzing process by removing uninteresting values.

- **Symbolic execution** is a technique that consists in testing a program by executing it with symbolic data, that corresponds to testing several inputs simultaneously. Whilst it is called 'execution' it is more appropriately a *simulation* using abstract models of the data.
  The state of symbolic execution is a *triple*(cs, σ, π) where:
    - *cs* is the control state
    - σ is the variables state
    - π is the path predicate, the set of constraints that allows us to reach a certain point of execution. Each time it is updated, it gets evaluated by a *Satisfiability Modulo Theory* solver that performs a *feasibility check*.

  Symbolic execution has issues with loops and path explosions, but some lossless techniques and techniques that limit the execution to a subset of the paths (but might introduce false negatives) are available. It also has problems with external libraries and function calls (that can be solved by inlining but introduces more complexity).
  Other limitations of symbolic execution are linked to the SMT, as the satisfiability problem is not always decidable if not enough constraints are given, and the analysis might be time consuming.
  Symbolic and Concolic execution can be applied to both source code and binaries (in that case you first need to build the model from the binaries) and can be used to mount attacks by finding possible vulnerabilities by use of assertions and the SMT.

- **Concolic analysis**: is a type of analysis based on *concolic* (*Concrete + Symbolic*) *execution*. It tries to find a trade-off between precision and performance moving towards testing (which has the advantage of never introducing false positives), and it is used to analyse complex programs for which Symbolic execution would introduce approximations. It comes in two forms:
    - *Dynamic Symbolic Execution*: in this case the concrete execution drives the symbolic one. We start with concrete random inputs and execute in double mode. Path feasibility is granted by the concrete execution so there is no need to use SMT solver at each branch. Once an execution terminates, we negate the last path and use the SMT solver to find new inputs to explore the other branch, repeating the concolic execution with the new inputs.
    - *Selective Symbolic Execution*: it is used when there are functions whose code is not available. When a call is reached, we continue in concrete mode only and once we get back the return value is used to resume concolic execution. We divide the code into regions that are executed only symbolically, only concretely and concolically.

- **Angr** is a modular python framework that performs binary loading, static analysis, binary rewriting, type inference, symbolic execution, symbolically-assisted fuzzing and automatic exploit generation. It is used to perform *concolic analysis* on software. It is composed by different modules:
    - CLE (*CLE Loads Everything*): turns an executable file into a usable address space. Extracts data from the binary, guesses the architecture and produces a representation of the memory map as if the real loader had been used.

    - *Archinfo*: a collection of classes containing information on different architectures (#registers, endianness...). After CLE guesses the architecture, data contained in archinfo is used to build the execution engine.
    - *SimEngine* (Simulation Engine): interprets the code and simulates the execution. Starts the execution from an entry state and moves it forward from one state to the other, generating the set of successor states. When it finds a branch, it collects the set of constraints that allows it to enter the branch and passes them to *Claripy*.
    - *Claripy* is the component that interprets the results of the simulation engine, managing concrete/symbolic values and expressions, adding constraints and evaluating them using an SMT solver (by default, Z3).
    - *PyVex*: an abstraction layer for the instruction set. Angr translates the opcodes used in the binary in a VEX representation using a *Lifter*, since it is impossible to execute the code on any platform. In this way the SimEngine does not need to know the actual opcodes but just the PyVex.
    - *SimOs*: the Angr components that models the operative system: its files, library functions, procedures, etc.

- **Simulation State**: is a snapshot of the program (or rather, of the simulation) current registers content, stack, heap, and other ArchInfo attributes. The SimManager updates the simulation moving from one state to the other based on the computation of a basic block.

- **XSS (Cross-Site Scripting)** is a type of injection attack that aims at bypassing SOP to execute arbitrary code on the victim's browser. It is used to steal credentials, perform session hijacking, website defacement,

install malicious plug-ins, redirect to malicious sites, etc. It can presents itself in one of the following forms:

    - *Reflected* XSS: the attacker sends a link to the victim to a vulnerable website. A vulnerable website in this case is one that *reflects* data coming from the user without validating/sanitising them, that is, the sent data is used to render the response. The link will contain a script inside the query parameter, that will be executed once the page will be populated with the website's response. As *script* and *img* tags in HTML can issue GET requests to other origins, the execution is permitted.
    - *Stored* XSS: the most dangerous form, in this case the script is stored in the website server. In this way not only the first user but also others will receive and execute the malicious script, propagating it without the need for a link sent to the other users.
    - *DOM* XSS: in this case the missing validation is not on the server side, but directly on the client. For instance, a webpage that uses a query parameter to render some part of itself (without issuing a request to the server, or it would be *reflected*); by having a malicious link to this page it is possible to inject code.

- **SQL injection** is a form of injection attack in which the attacker exploits unsanitized parameters passed to a sql interpreter to execute malicious queries. This form of attack is possible when a program accepts user inputs and uses them to build a SQL query without properly sanitising/validating them. Depending on how the query response is shown to the user, the attacker could read sensitive data, insert and modify values or simply damage the DB by removing tables and entries.

There are various tools that can be used to check if this kind of vulnerability is present and produce exploit payloads, like *SQLmap*. Possible mitigations are:

- Sanitise user inputs
- Use parametric queries, safe APIs
- Limit the informations given by error messages to the users
- Use interpreter-specific escape characters
- Apply a least privilege policy

More in general, it is always a good idea to validate the inputs, keep untrusted data separated from queries and commands, architect and design for security policy, least privilege, and adopt secure coding standards.

- **Format String attack** is a type of attack that exploits an uncontrolled format string vulnerability, to crash the program, read values in memory, or execute arbitrary code. It stems from the use of user-controller input as the format parameter in certain C functions (it is also possible with some JAVA functions but the only effect there is to crash the system). When a format string requires more parameters than the available ones, e.g. printf("%s %s %x", a, b), the function tries to read the remaining data from the rest of the stack (where the parameters are), thus accessing memory that should not be available to the user. Moreover, using the '%n' format specifier, it is possible to write values in the target memory address. The attack is usually conducted first by using a marker sequence to find the position of the values we're interested in, and then use a payload like ("addr" + ".%y$s") to read from the address with offset y.
  A code example:

```
int main(int argc, char *argv[]){
    char buffer[100];
```

```
    printf('%s', argv[1]); //not vulnerable, the user-controlled data are not the formatter
    printf(argv[1]); //vulnerable
    sprintf(buffer, argv[1]) //vulnerable, as in sprintf the formatter is the second string
}
```

- **Fortify Taxonomy** is a software vulnerabilities classification that defines 7 categories (*kingdoms*) that are comprehensive (but not disjointed) of all possible vulnerabilities. They are:
    1. *Input Validation and Representation*: problems are caused by trusting untrusted input.
    2. *API Abuse*: depends on not properly using APIs.
    3. *Security Features*: not using security features properly.
    4. *Time and State*: unexpected interactions between threads, processes, etc or from bad timing. Like race conditions.
    5. *Errors and Error Handling*: the way in which errors are handled.
    6. *Code Quality*: security problems caused by poor code quality.
    7. *Encapsulation*: caused by not properly implementing strong boundaries.

- **Insecure deserialization** is a form of attack that consists in deserializing hostile/tampered objects supplied by an attacker. The impact is severe since there can be privilege escalation, replay, injection, and in the worst case remote code execution. There are two types of attacks:
    - *Object and data structures related attacks* that require classes changing behaviour during/after deserialization.
    - *Data Tampering attacks* in which the data structure is used but its contents are changed.

  To prevent this kind of attacks one must implement integrity checks on serialised objects, enforce strict types constraints, do not accept serialised objects from untrusted sources, monitor deserialization activity.