*******disclaimer****: this file is a collection of all (i hope) previous questions of the past exam calls for the SVT course. It should be a better version wrt the previous one. The answers could contain some mistake: please, if you notice one, add your correction with color orange, not editing the answer already written (just to understand what is the error).***

P.s.: Feel free to add anything you think could be useful for passing the exam.

SVT - exam questions

1) In the context of the Common Criteria, what are Evaluation Assurance Levels? In what cases an EAL is higher than another one? (sample 1)

   Evaluation Assurance Levels are metrics to quantify the level of assurance achieved. Each EAL requires a set of components. A higher EAL is obtained from the previous one in two ways:
   - including other components (from other families)
   - replacing components with higher level assurance components (from the same family)

   An EAL is higher than another one when we have a higher risk situation.

2) What are security evaluation standards? What are the Common Criteria? (sample2)

   Security Evaluation Standards are references for security evaluation of products and processes. Common Criteria are an example of standards for product evaluation. In particular, they permit comparability between the results of independent security evaluations, providing a common standard reference for evaluating/certifying security of an IT system.
   CC are just criteria, they don't define any development process, evaluation methodology or regulatory framework. Instead, they provide
   - a uniform approach to evaluation/certification;
   - a way of expressing security requirements and assurance levels
   - a set of constraints on the evaluation methodology

3) CCRA (08/02/2021)

CCRA stands for Common Criteria Recognition Arrangement. All nations signer of the CCRA recognize the results of the evaluations done by the Authorizing Nations:

- Authorizing Nations (certificate producing): they developed their own evaluation scheme to accredit laboratories to perform CC evaluations.
- Consuming Nations (certificate consuming): they don't have their own Evaluation Schema, but they want to recognize CC evaluations done by others.

CC are criteria to measure how secure a system is. An evaluation method needs to be associated with these criteria. This method is called evaluation schema and it's different from country to country.

4) In the context of Common Criteria, what is an Evaluation Schema (31/01/22)

An evaluation schema is an administrative and regulatory framework, associated to CC, that each authorizing nation has developed to accredit laboratories to perform CC evaluations. Consuming nations don't have their own Evaluation Schema, but they want to recognize CC evaluations done by others.
These definitions are given in the CCRA (CC Recognition Arrangement), a document that must be signed by all countries that want to recognize CC evaluations.

5) Security Assessment vs Security Certification (30/08/2021)

A **security assessment** is a form of security evaluation which requires looking for vulnerabilities and their related exploits in a computer based system (hw, sw, cyber-physical, networked system). The assessment can be done with different techniques

- Vulnerability assessment and penetration testing
- Code analysis (static and/or dynamic)
- Formal verification

SA can be done under different perspectives: during development, done by the user, with a security certification.
In particular, security certification responds to the need of having

1) a way to certify security by means of independent trusted third parties, providing evidence that security has been achieved.
2) metrics for security assurance, namely the confidence that an entity meets its security requirements

Certification is based on evidence of employed assurance techniques (formal, semi-formal, informal), evidence of evaluation methodology and accreditation of evaluators

6) What is security assurance in the context of security assessment and certification standards? Give a precise definition.
What are the main techniques that can be used to achieve assurance? Make at least 3 concrete examples of such techniques (of different types) (14/02/22)

- Examples:
  - Penetration testing
  - Analysis of guidance documents

- Analysis for vulnerabilities
- Verification of proofs
- checking the processes and procedures are being applied

The security assurance is a process which checks if a product meets security requirements defined at the beginning of the development. With security assurance and with some metrics it is possible to provide credible evidence that security has been achieved as defined in the security requirements, and it certifies its security. It is normally done by third-party operators using formal, semi-formal or informal techniques. At the end, the security certiofication gained is the evidence of the security. But to make certifications and evaluations comparable, some security evaluation standards are defined, which can be related to the product (like CC) or to the process.
3 of the main techniques are:

- Check the TOE design representation against the requirements
- Analysis of functional tests developed and of the results provided
- Analysis of vulnerabilities and penetration testing

7) SAST, DAST and IAST: what are their definitions? Explain pros and cons

- **SAST** - Static Application Security Testing: White box static analysis of the source code.
  - **Pro:**
    - Find more vulnerabilities
    - Used in all development stages
    - can provide info on the cause of vulnerability (code line)
  - **Cons**
    - More false positives
    - Coverage of libraries is an issue
    - Each tool applies to only some languages/frameworks
    - may be time-consuming
- **DAST** - Dynamic Application Security Testing: Black-box dynamic analysis (vulnerability scan)
  - **Pros:**
    - Less false positive
    - independent of language/technology used to develop application
  - **Cons:**
    - Find less vulnerabilities
    - Used only in the last development stages
    - Cannot provide info on the cause of vulnerability (black-box)

- **IAST:** Interactive Application Security Testing: combination of static and dynamic, due to complexity of applications, made of many third party components
    - **Pros**:
        - Find many vulnerabilities but with few false positives
        - very fast and scalable
    - **Cons**:
        - not yet available for any language
        - not yet widely known and adopted

8) Make a comparison between static and dynamic techniques for security analysis, highlighting pros and cons of each one

Static analysis consists in analyzing the code without executing it, using techniques which are partially automatic, mainly executed in white-box mode.

While the dynamic one is more like real testing which needs to have an executable code and it checks its behavior for example using different inputs.

With static analysis, it is possible to perform type checking, style checking, formal verification modeling the system and also symbolic execution. The pros are that we don't need an executable code and it finds more vulnerabilities and also the code line, it gets a better code coverage, and it can be used in different development stages. But it also has some cons, for example it finds more false positives which need to be checked with dynamic analysis, the tools are specific to a language, and it is not able to test third-party libraries if the code is not available.

While with dynamic analysis it is possible to test the executable code or also the binary with different techniques like debugging, fuzzing but also penetration testing. It has pros, for example it gets less false positives and it is independent of the language, but also some cons like it finds also less vulnerability as it could not get a good coverage, it can be used only in the last development stages and it requires good expertise. It is also mostly black-box based.

---

9) List at least 5 of the OWASP top-10 vulnerability classes

OWASP stands for Open Web Application Security Project. It's a periodic publication that identifies the top 10 most critical security risks for web applications.

1. broken access control: violation of least privilege principle
2. cryptographic failures: use of weak algorithms for confidentiality
3. insecure design: security is not properly considered in the development process. (e.g. no threat modeling)

4. injection: untrusted data are inserted into a SQL query without validation or filtering.
5. security misconfiguration: security features are not properly configured. XML external entities (XEE) attack due to weakly configured XML parser, which can lead to disclosure of sensitive data
6. identification and authentication failures: when integrity of software or data is not checked/ensured.

10) Non determinism

Non-determinism is used by abstract models for representing execution aspects that are not known a-priori, but also for representing different possible implementation choices. They are not known a priori:

- The inputs of a program
- how concurrent processes are scheduled by the OS

11) What is a transition system and what is its formalization?

A transition system is a model (state-transition) to describe operationally a reactive system. Namely, a system that interacts with its environment in a predefined way, respecting temporal constraints. It is formalized as follows: (S,init,p), where

- S: it is a set of states
- init: it is the initial state (init belongs to S)
- p: it is the transition relation (p is included or coincident in SxS)

If labeled: (S,init,L,p). Where L is a set of labels, i.e. the events. In this last case p is in S x L x S.

A transition system is formalized with a control flow graph. In particular, a control flow graph describes the flow of a sequential program execution.

12) Explain what is a control graph and what is a basic block

A control graph is an example of a state-transition model of a sequential program execution. Or, alternatively, it is a way to formalize the transition system (for a concurrent program, each sequential part is represented by a TS). It is made as a directed graph including the following vertices:

- entry, exit, assignment, test

The full model of the program is a TS: (S,init,p) where:

- V: set of all possible contents of variables
- States (S): set of pairs <s,v> where
  - s: it is a control state (an edge of the Control Flow Graph)
  - v: contents of variables (an element of V)
- initial state (init): <s0,v0i> where
  - s0: it is the edge outgoing from the entry vertex
  - v0i: an element of V (the element of V corresponding to “all variables not initialized”)

- state transitions (p): determined by the semantics of program statements

A basic block is a sequence of instruction executed without interruptions or jumps.

13) What is a responsible disclosure? Provide a precise definition and an example scenario of responsible disclosure

It's a vulnerability disclosure model in which vulnerability is disclosed only after a period of time, giving the possibility to the developers to create a patch for solving the issue. In this case the time with high risk is shorter. An example of scenario could be that a group of researchers discover one or more exploitable vulnerabilities, but before publishing them to the public they communicate the issues to the owner of the system, giving an amount of time pre-defined to developing patches. Like the most famous bug-hunting program.

14) 0-day vulnerability

It is a vulnerability which is disclosed (privately) but no patches are available yet, so it is exploitable. An unknown vulnerability after discovery becomes a zero-day vulnerability.

15) What is a temporal logic formula, make an example

It is an extension of classical logic formulas that describes the temporal evolution of the program executed. It can be defined in various ways based on:

- logic (propositional vs I order),
- time (Discrete vs continuous, Linear vs branching),
- modalities (Event vs State, Past vs Future)

The main temporal operators of Linear Temporal Logic are:

1) O f: f is true in the next state
2) [ ] f: f is true in all future states (always)
3) ♢ f: f is true at least in one of future state
4) f1 U f2: f1 keeps true until f2 becomes true

Example: Assuming that a variable "x" has positive value during the whole program execution → " [ ] x > 0 "

16) Propositional Logic and Predicate Logic (I Order Logic)

A formal model of a system property can be expressed formally by a logic formula.

Propositional and Predicate Logics describe static facts, immutable in time. Semantics of propositional logic:

- interpretation of atomic prepositions:
  - it can be formalized as a function I: AP → {F, T}
- interpretation of operators
  - it can be formalized as boolean functions (truth tables)

Abstract reasoning. The interpretation of operators lets us reason regardless of the interpretation of AP:

- **Tautology**: formula that is *always true* (regardless of how APs are interpreted)
  - Q => (P => Q) , P ∨ (¬ P)
- **Contradiction**: formula that is always false (it is the negation of the tautology):
  - P ⋀ (¬ P)
- **Satisfiability:** a formula is said satisfiable if it is true for at least one interpretation of APs
  - duality of validity and satisfiability: *f is valid* ⇔ ¬*f is not satisfiable*
- **Validity:** a formula is valid if it is true for all interpretations of APs (i.e., it is a tautology)

Predicate Logic (I order logic). It is an extension of propositional logic where:

- Atomic propositions are replaced by predicates
- Are introduced the new concepts of constant, variable, function, relation and the universal and existential quantifiers
- For closed formulas (without free variables)
  - interpretation maps each formula onto an element of {F,T}
- For open formulas (with n free variables)
  - interpretation maps each formula onto a relation on D^n
- Example: ∀k ((1≤ k ≤n) => (v(k) < v(k+1)))

17) How to formalize a logic: A formal system - Lukasiewicz

A Formal system (theory) is a way to formalize a propositional logic defined by:

- A **formal language**
  - An alphabet of symbols
  - a set of well formed formulas (sequence of symbols belonging to the language)
- A **deductive apparatu**s (or deductive system) that give the meaning to the formulas
  - a set of axioms (formula to which the true value is assigned axiomatically)
  - a set of inference rules (expressing that a certain formula is a direct consequence of certain other formulas)
- An **example** could be: Łukasiewicz
  - *Formal language*: propositional logic syntax, with the only two primitives operators (=> , ¬)
  - *Deductive apparatus*: **Axioms**
    - A1) P => (Q => P)
    - A2) (P => (Q => R)) => ((P => Q) => (P => R))

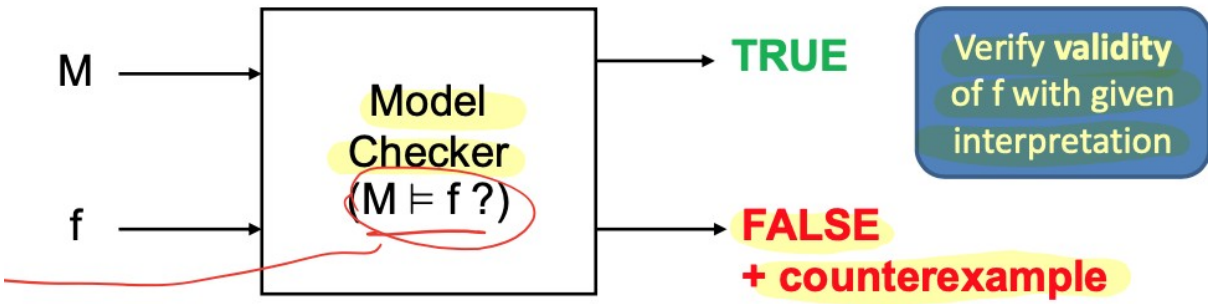
        - A3) (¬ Q => ¬ P) => (P => Q)
    - Deductive apparatus: **Inference rules**
        - I1) $\frac{P, P=¿Q}{Q}$ (modus ponens) : if P is true and P=>Q is true, then also Q is true
- **Proof:** a sequence of wff P1,...,Pn such that, for each i, Pi is an axiom or it is a direct consequence of some of the preceding formulas, according to an inference rule
- **Theorem**: a wff P such that there exists a proof that terminates with P.
    - we write ⊢P (P can be proved) to mean P is a consequence of the Formal system axioms and rules (a theorem)

18) What is model checking and how does it work? In this context, what is a counter example?

Model checking:

- Provides a proof of non-validity (counter-example)
- Can be applied directly to an interpretation
- Can tell with certainty if property is true (validity)

Model checking is one of the possible approaches for verifying that a formal model satisfies some formal properties. In this approach model M represents an interpretation under which a property f (a wff in the Formal System) must be verified:



- (M ⊨ f ?) : is the formula f true under the interpretation of the model M?

The model checker returns TRUE if the property is considered valid, otherwise it returns FALSE + counter-example. A counter-example is some evidence that the interpretation M does not satisfy f.

19) Techniques for Model checking Dynamic systems: State Exploration

This technique consists of exploring all states/runs of the model looking for violations of the property:

- if violation **is found**: property is false. and the state where violation has been found is the counter-example
- if violation **is not found**:
  - if states/runs have been explored exhaustively => property is true → that's possible just for finite-state models
  - if states have not been explored exhaustively => property may be true (but we have no certainty)

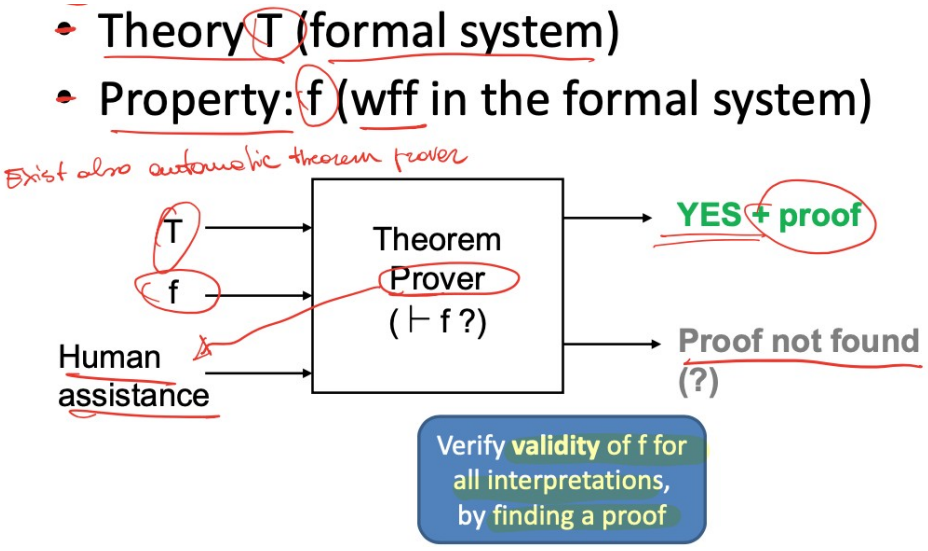
There are two kinds of state exploration techniques:

1) **Explicit**:
   a) states and transitions are enumerated one by one
   b) states/transitions to explore can be reduced by applying various techniques
2) **Symbolic**: states and transitions represented symbolically (by boolean functions)

20) Explain what Theorem prover is and give a formal definition about the proof which it provides

Theorem proving:

- Provides a proof of validity
- Requires generation of theory
- It can tell with certainty if a property is true (validity, like the Model checking approach)
- if proof is not found, nothing is known
- Theorem proving does not suffer of state explosion



There are two techniques for Theorem Proving

1) Proof assistants (interactive TP): just check the correctness and completeness of the proof developed by the user
2) automated Theorem Provers: that can find a proof autonomously by applying tactics and other strategies (e.g. resolution), but they may not terminate or succeed

Moreover, TP becomes easier if the logic system is restricted to have only Horn clauses (implication causes and facts). In a logic made only of Horn clauses and propositional logic the problem of proving that a conjunction of facts is a theorem can be solved automatically in linear time by means of a resolution algorithm.

Proof: a sequence of wff P1,...,Pn such that, for each i, Pi is an axiom or it is a direct consequence of some of the preceding formulas, according to an inference rule

Soundness/Completeness

The augmented theory is:

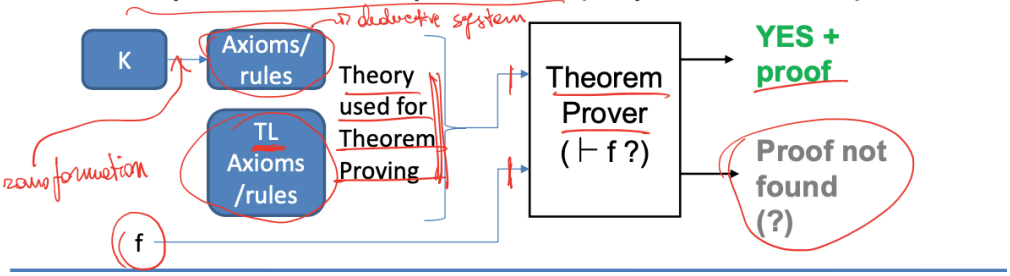
- **sound**, w.r.t. interpretation K if for each theorem f, K ⊨ f
- **complete**, w.r.t. interpretation K if for each f such that K ⊨ f, f is a theorem

Hence,

- if the theory is ***sound and complete*** w.r.t. K: $\vdash f \Leftrightarrow K \models f$: proving that if f is a theorem is equivalent to say that the formula f is true with K interpretation
- if the theory is ***sound (but not complete)*** w.r.t. K: $\vdash f \Rightarrow K \models f$ → practical implication: sometimes you are not able to say that a formula is true even if a problem is decidable

Theorem Proving for Verifying Dynamic Systems Properties (TL)

- The behavioral model of the dynamic system is expressed by **specific axioms/rules** added to the theory
  - In practice, the addition is an alternative way to express the interpretation (Kripke structure)



21) What is reachability analysis? How does it work? What are its limitations?

In the context of model checking of dynamic systems, reachability analysis is a simplified form of state exploration. More precisely, reachability analysis is done through explicit model checking tools and it's used to verify the temporal logic property [ ]P, i.e. analyze all states and verify that P holds in each state.

- Limitations:
  - It works well only if the number of states is finite, because this technique implies the exploration of all states. Hence dealing with infinite states models some abstractions are needed
  - For a concurrent system there's the risk of **state explosion**, i.e. the complexity grows exponentially with respect to the number of parallel components.

22) Injective correspondence e correspondence normale

Injective correspondence: requires (to be true) that each occurrence of event e(...) corresponds to a **distinct** occurrence of event e'(...):
inj_event(e(...)) => inj_event(e'(...))

basic correspondence: it is **non-injective** → it is true even when the same execution of event e'(...) corresponds to **more** executions of event e(...): ***event(e(...)) => event(e'(...))***

23) Can Proverif find false attacks? Explain why.

Yes, proverif can find false attacks. Because the logic theory described by the Horn clauses over-approximates the behavior of the real protocol, false positives are possible. In fact the approximations have been proved sound, but not complete. The set of traces that can be derived according to the Horn clause model is greater than the set of traces that are really possible.

24) Explain the meaning of the following Proverif output

Query inj-event(eC(x_1,y_1,z)) ==> inj-event(bS(x_1,y_1,z)) is true.

The query (proven to be true) aims to check the correspondence property, an order relationship that should trace events, in this case eC and bS , used to specify authentication and data integrity properties on a model. Correspondence means that when the event eC happens, the event bS must have happened before. This correspondence is also injective, i.e each occurrence of event eC must correspond to a distinct occurrence of event bS. It is used to prove authentication using a set of keys and nonces as parameters and the injectivity tries to prove if the model is resistant to replay attacks. In this case, as it is proven to be true, we can say that at each eC instance corresponds one different bS which happened before.

25) secrecy property in proverif. Formal definition

A closed process P preserves the secrecy of N from S-Adversaries if:
For each S-adversary Q, for each trace T executed by P|Q (P combined Q)
T does not output N.
S-adversary is a close process where the adversary initially knows the terms present in S.

26) Strong secrecy property in proverif. Formal definition

The secrecy property is strong when an attacker cannot even get partial knowledge from the system. Queries about strong secrecy are written as **noninterf x,y:** where x,y are the free terms that should remain strongly secret.

≈ OBSERVATIONAL EQUIVALENCE

Closed process **P** preserves the strong secrecy of **N** (the secret) from S-Adversaries (**noninterference**): **P[x/N] ≈s P[x'/N]**. **≈s** means that the two processes are **externally indistinguishable for an S-adversary**. [x/N] -> the secret N substituted with secret x.

Namely, it cannot find S-adversary As, which outputs a different message according to whether it interacts with **P[x/N] or P[x'/N]. important for secrets that have low entropy. (level of entropy directly proportional to the number of bits of secret)**

P and P are two instances of the same process. for an S-adversary there is no way to know if a process exchanges x or x'. An S-Adversary is not able to see when the secret changes.

27) What is the meaning of the security properties defined in the script? How could the process definitions be fixed to make the second property true? (write the fixed version of the process)

```
free channel (Be public channel)
free bitstring [...] (shared secret)

("Cryptography hash")
query attacker(s)

event send(bitstring)
event receive(bitstring)
query x bitstring.event(receive(x))==> event(send(x))

let p(A) =
        row ds bitstring
        event send(ds)
        out(c ,(ds, hash(ds,s)))
        0

let p(B) =
        in (c, y bitstring)
        let ( ds_in bitstring,  hash_in bitstring) == y in
        let hash_ds = hash(ds_in,s) in
        if hash_ds == hash_in then ok()
        event receive (ds)
        0

process
        {
        !pA | !pB
        }
```

28) What is Taint Analysis?

It is an information flower problem that can be solved by Dataflow Analysis. We can perform an analysis to find whether a source can propagate to a sink.
source: it is an input received by the program in a certain CFG location

sink: a use of a variable in a certain CFG location.
propagation: it can have different meanings (influence, arrive, unfiltered or unchecked)
This kind of analysis is relevant for

- detect leakage of secret information vulnerabilities
- detect injection vulnerabilities

EXAMPLE: For Detecting Injection Vulnerabilities we are interested about a set of in-scope tainted variables. Tainted means coming from an untrusted source. Transfer functions can

- taint a variable (taint source): statements that make tainted data enter the system (e.g. fgets(buf,sizeof(buf),stdin) taints buf)
- let taint status of a variable pass-through: statements that don't change the taint status of a variables

statements can

1. propagate the taint status from variable to variable (e.g. strncat)
2. clean the taint status of a variable (check or sanitize input)
3. be potential vulnerabilities (sink) - statements that use a variable that should not be tainted

DISCUSSION: The taint status is modeled as a collection of flags. there can be different ways of being tainted:

- different types of sources taint variable differently
- filtering functions can filter only some types of taint
- sinks can be sensitive only to a certain taint statuses
- different severity levels can be associated with a sink according to the type of taint status

29) What does the percentage on the X represent? How can it be computed? What does the percentage on the Y represent? How can it be computed? Explain why the tools usually have scores that lay in the left upper area of the diagram (the white one)
OWASP scoring system spiegare il grafico dicendo cosa c'è sull'asse delle y e cosa su quello delle x, scrivere le formule, e spiegare perché di solito si hanno punti nella parte superiore del grafico invece che in quella inferiore.

True Positive (TP): the alarm of the test case points to a real vulnerability
False Positive (FP): the alarm points to a false vulnerability
False Negative (FN): vulnerability not detected
True negative (TN): non-vulnerability correctly ignored

The percentage on the X represents the False Positive Rate (FPR) = FP/(FP+TN)
The percentage on the Y represents the True Positive Rate (TPR) = TP/(TP+FN).

Because usually are used tools that have a score that is better than guessing

As the area is divided in two parts by the diagonal, if a tool is placed in the upper part, it means that its performance is better than guessing, otherwise it is worse than guessing.

- ***Exercises for Sisto's part: flawfinder_lab4, proverif_lab2, spotbugs_lab5***
- ***Exercises for Basile's part: VA/PT_lab1, binary exploitation_lab3, angr and simulation manager_lab6g***

30) EPM (Enterprise Patch Management System) che rischi mitiga e che rischi aggiunge

Patch Management is the process for identifying, acquiring, installing and verifying patches for products and systems.

EPM minimizes the time companies spend dealing with patching and increases resources for addressing other security concerns.

It is also useful to mitigate risks deriving from software vulnerabilities, reducing the opportunity of exploitation. Moreover a patch can correct problems in software and add new features and security capability to software and firmware.

there are problems related to patches:

- security risks: attacker may reverse-engineering patches, to build exploits
- cost: patched service can face restarting and interruptions
- a risk associated with the use of an automatic tool. A tool is a software, so vulnerabilities in the tool can be exploited
    - Mitigating additional risk is just application of standard risk analysis: so, tool must contain built in security against threats

To

31) Same origin policy

SOP aims to have no interaction between two pages in the same browser, in order to not let the scripts from one website accessing sensitive data from the DOM of another page (a different website).

Indeed, scripts contained in a first web-page can access data in a second web-page if and only if:

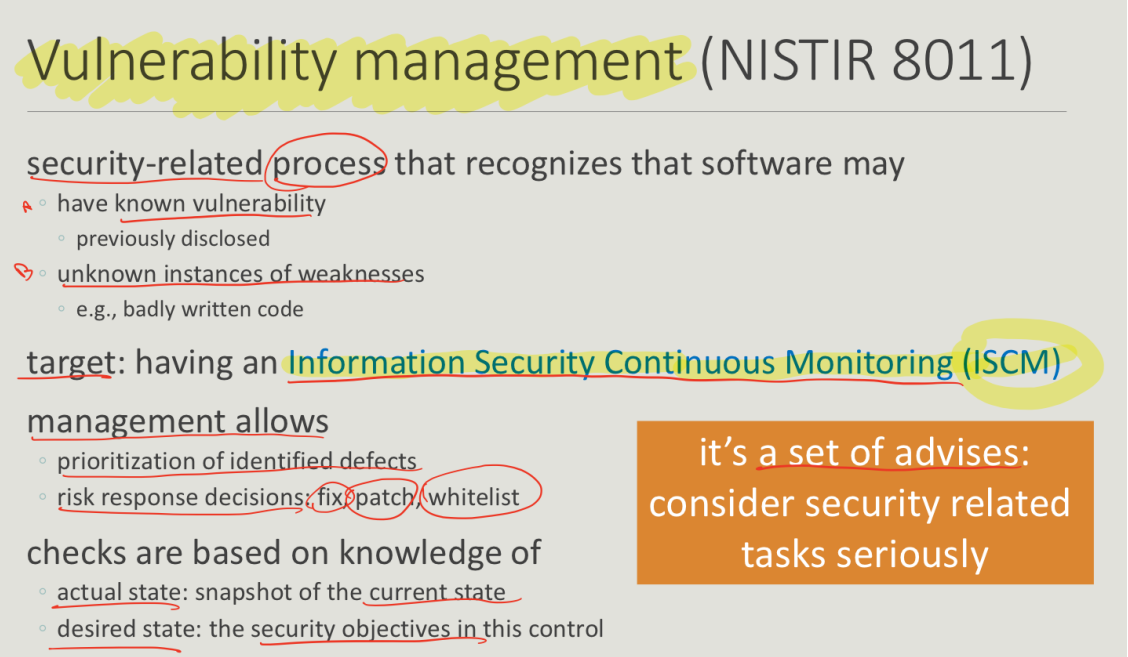
- both pages have the same origin (checking URL)
- protocol, port and host must be the same

The most common JS-related security bugs are caused by breaches of either the SOP or the Sandbox. Notwithstanding, SOP is too restrictive for a good user experience (think about the necessity of third party cookies to classify your browsing data or the experience with sites that have multiple

subdomains). For this reason, modern browsers implement a partially relaxed SOP with different approaches:

- cross-origin resource sharing
- cross-document messaging
- web sockets

32) A company has performed a Vulnerability Assessment and identified several vulnerabilities. Propose a strategy to prioritize the correction of the vulnerabilities (hint: follow the NIST suggestion).



A possible strategy is, first of all, based on building a DB of all the elements that could contain vulnerabilities, like code and installed software.
Then, understanding the actual state of the system and its status of exposure, performing vulnerability scans, and compare this actual state with the one that we want to achieve, after having defined security requirements.
To address the vulnerabilities, a possibility is starting from the ones with a score higher than 7.0 and solve them in the next 24 hours. This can be done identifying mitigation methods like installing patches if the vulnerability is well known, but sometimes it's a 0-day and we don't have a patch.

33) Describe what is a buffer overflow vulnerability. Furthermore, explain how a buffer overflow in a function can be exploited to execute a remote shell.

BOF consists in exploiting the weakness present in a program when the boundaries of an input buffer are not checked. (e.g. when using the "gets" function without any further check). Providing an input greater than the buffer size, previously

allocated, I am able to overwrite the stack, leaving room for different actions. I could set variable values, alter the program behavior, bypass controls, jump anywhere in the program, skip pieces of code that I don't like for some reason.
Shellcode (available on the web) is a small piece of code used as the payload in the exploitation of a software vulnerability. Opening a shell on a remote host (and possibly setting root privileges) is possible by overwriting on the stack, exploiting BOF, the return address with a new one that returns to a shellcode, encompassed between NOP instructions, because there is no need to be really precise in returning the begin of shellcode.

34) Rop Vs ret2libc

ROP stands for return oriented programming, and it is a way to bypass the problem of injecting your code, so to bypass the DEP. We are able to perform a ROP attack using gadgets, namely a sequence of meaningful instructions followed by a RET. The goal is to create a gadget-based instruction set from the code you have in a program. Then create the shellcode by chaining gadgets (ROP chain). Not always possible.

Ret2libc is a way to execute the ROP attack, borrowing the code from the application, exploiting libc (if present) functions instead of using gadgets. We need to find the base address of the libc and write it into the memory or stack and then find the address of the functions you want to call (system, open, read, exec, write…). It is easier than building a ROP chain.

Answer from Basile:

1) ROP and ret2lib: they are methods to exploit BOF when you cannot put shellcode in the stack.
2) Reason: deployed DEP (no canay, no ASLR, both may be an issue also fro them)

ROP:

3) Def ROP: build payload (e.g. malicious code) by chaining “ROP gadgets”
4) Def ROP gadgets: (small) sequences of instructions ended by a RET that are already present in the application code segments. How to find gadgets: ad hoc tools
5) something about how to build a chain:
    - write in the stack the address of gadgets to execute instead of the legitimate return address (by exploiting the BOF)
    - manage the stack (store and remove function parameters) and use additional gadgets to prepare the stack + use gadgets to prepare the registers with proper values

ret2libc:

6) Def: ret2libc: build the payload by resorting to calls to the libc library (which is almost omnipresent)
7) something about how to mount it
    - prepare the stack to return to a libc function (overwrite the proper return address)
8) Relative comparison:
    - same purpose
    - ROP usually much more difficult
    -

35) Describe the "Data Execution Prevention" protection and indicate which kind of attacks it aims at countering.

DEP protection consists in making executables only code segments and, consequently, these segments must not be executable and writable at the same time. If code is executed from write-only segments, a segmentation fault is generated. Hence, data segments must be R&W, like stack, heap… The code segment must be R&X, like .text, .plt . Where R stands for readable. Its purpose is preventing arbitrary execution of the code, especially from a zone that can be overwritten.

36) Recursive disassembler (che dovrebbe essere il recursive traveled) spiegare cos'è, vantaggi e svantaggi

A disassembler is a software for translating binary machine code into assembly instructions using lookup tables. And for tracing the control flow to decode sequences and breaches of code. Disassembly is not deterministic.

Recursive traversal disassembles instructions following the expected execution flow constructed during disassembly. The most serious disassemblers are RT.

They start at the entry point of the program and when they find an instruction that modifies the execution flow (JMP), they follow the actual executed code. Thus RT disassemblers decode the program with depth-first search, translating bytes actually reached.

Advantages:

- not confused by data embedded in the code sections
- skip over data bytes as they are never reached by the traversal

Disadvantages:

- determining the execution flow is hard statically, due to indirect jump and calls, and due to missing run-time information
- may not process all the bytes in the executable, because not all code locations are accessed through direct (static) branches from the entry point, there are also indirect branches like function pointers, callbacks

37) linear sweep disassembler

A disassembler is a software for translating binary machine code into assembly instructions using lookup tables. And for tracing the control flow to decode sequences and breaches of code. Disassembly is not deterministic.
A linear sweep disassembler (e.g. objdump) is implemented under the assumption that instructions are stored in adjacent positions.
It aims to sequentially decode bytes into instructions, from the beginning of the first section of an executable until the end of the section or until an illegal instruction is reached.
It has the advantage of being simple and easy to implement. However, it has some shortcomings because:

- In case of a dense instruction set, it is easy to desync. Thus, it may work for small pieces of code.
- there could be dealignment if data and code are mixed
- it can generate mistakes, in case of overlapping instructions.

38) Describe what is software obfuscation, how does it work, and which risks mitigates.

obfuscation: make the program much more difficult to understand for human beings. Also a form of anti-static analysis protection.

- -

A technique that aims at reducing the understandability of the code. The purpose is to delay the attacker. The perfect obfuscation is impossible because there are functions that cannot be obfuscated. There are code obfuscation and data obfuscation.
For the code obfuscation you can

- make the control flow unintelligible (control flow flattening, branch functions, hide external calls)
- add bogus control flow (opaque predicates)
- hide the signature of the functions (split/merge)
- forcing dynamic analysis, avoiding static reconstruction of the code (just-in-time techniques, virtualization obfuscation, self modifying code)
- analysis (anti-taint, anti-alias)

For data obfuscation:

- you can hide constants and values
- WB cryptography to hide secret keys in the code.

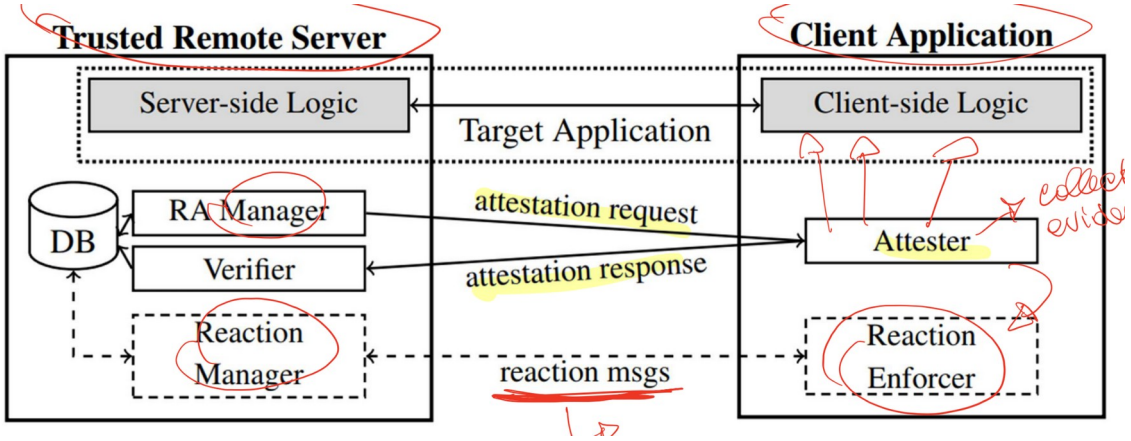
risk: static reconstruction of the code, reverse software engineering

39) Software Attestation and Code Guards, definitions, pros and cons. When is it better to use one over the other?

Software attestation is a form of anti-tampering.

- -

Software Attestation: It's a verifier that checks the evidence to understand if an application is working properly. A form of anti-tampering that uses a server to send to the client not only the application logic, but also an Attester (who receives an attestation request from RA Manager). This one collects the evidence and sends it to the Verifier. In case of strange behavior, there is a Reaction Manager that disconnects you from the server.



- **pro:** implemented as application integrity: checksum of the binary or configuration files, checksum of the binary loaded in memory.
  No hardware is used.
- **cons:** vulnerable to several attacks (dynamic code injection, parallel execution of an untampered version of the device)

Code Guards: they are pieces of code injected into the application to check other pieces of code of the same program

- **pro:** if checks are OK, the program is assumed to be uncompromised.
  deploy layers of guards (guards that protect other guards) to increase effectiveness.
- **cons:** the correct values are somewhere in the code, and they can be read and defeated with static/dynamic attacks.
  vulnerable to attacks that change the execution environment. e.g. the cloning: two copies of the program in memory, one is correct and thus used for redirecting the attestation request.

It is better to use software attestation, instead of code guards, for embedded systems, IoT, and portable devices.

39bis) **Consider the Software Attestation and Code Guards protections.**
Describe what is the primary protecting objective for applying them in a target application.
Discuss the advantages and disadvantages for each protection.
Finally, present one case when one protection should preferred over the other one. (*31/01/2022*)

Objective: The objective of these techniques is to delay the time in which an attacker can break an application. These techniques are "anti-tampering" techniques, with the aim to avoid that an attacker can create modified (unofficial) versions of applications.

Software attestation: inside the application there is an attester which periodically computes the evidence that the program is behaving correctly and the server (manager) periodically sends some requests in order to check the integrity. The attester sends the response to a verifier that will check the integrity. If a mismatch is found, then it is possible to send reaction messages in order to stop or crash the application. The advantages are that the attestation is performed remotely and it does not require a lot of code on the app. The disadvantages are that it is vulnerable to cloning attacks (running an unmodified version of the app in which to redirect the requests) and also to dynamic code injection (e.g., debuggers).

Code guards: pieces of code injected in the app that verifies that another piece of code is not compromised. These are typically used in chains, which means that it is much more effective to use many code guards instead of only one, so that the attacker needs to break all of them before it is able to crack the app. Code guards can react to manomissions by crashing the application. Again, this is vulnerable to cloning attacks and to the fact that code guards are put inside the app, so they can be defeated.

Comparison: An example of preferring software attestation is the case of videogames (online apps), in which if any cheat is detected from the server, it can "ban" the client. Code guards, instead, are preferred to software attestion for social apps (e.g., skype used 1000 guards) but also for all those apps that can be used offline.

40) Anti-tampering

anti-tampering: avoid, detect or even react to non authorized changes to program code or behavior.

Anti-tampering is a category of protections that aim at making changes to the code more complex in the sense that changes should come with a cost.
The purpose is to preserve the integrity of the code and execution correctness (much more complex to obtain).
We have different families of protections:

- local or remote: local if components are in the program, remote if resort to external components (e.g. servers as root of trust)

- with or without secure hardware: when available some computations can be offloaded to pieces of hardware that cannot be tampered without local intervention.

41) Describe the advantages and limitations of the fuzzers when used for security verification and testing.

Fuzzers: are tools to cause the crash of an application by injecting fake/wrong (maybe random) inputs.

Advantages: what kind

- fuzzing allows detecting bugs and improves security testing
- complements usual software testing procedures
- used by hackers, who observe crashes, memory leaks, unhandled exceptions, for mounting attacks

Disadvantages:

- may not be able to give enough information to describe bugs
- requires significant resources and time
- does not work in detecting unwanted behaviors (e.g. security threats like malware [viruses, worms, Trojan] that do not cause program crashes)
- can only detect “simple” faults or threats
- when using non-white box approaches: difficult to find boundaries

42) [TODO] Explain what is the corpus minimization used by the fuzzers (e.g. AFL), the purpose, and the results it permits to achieve.

Moreover, illustrate one more technique that can be used for improving the performance of fuzzers, shortly indicating purposes and results.

Finally, discuss why and how fuzzers can be used by a company developing software for vulnerability assessment, what they help to discover, and who benefits from using them.

Corpus minimization: the corpus is the set of initial files, the seed that is given in input to the fuzzer; corpus minimization means to remove the unneeded files, the ones that are useless since they don’t increase the coverage of the code but only slow down the fuzzing

Purpose: ***Reducing number of inputs to use, removing the ones that does not give good coverage and are useless.***

Result: ***The analysis is faster as it uses less inputs without losing too much accuracy.***

Technique name and short description: ***Two possible techniques are:***

1. using shellphish which helps to parallelize more AFL instances and uses ML methods to generate new inputs.
2. using custom dictionaries to help AFL generating inputs with keywords specific to the target.
Purpose:
Result:

Fuzzers for VA, why: ***It helps the testing phase during the entire development process***
Fuzzers for VA, how:
what discovers: bugs, crashes and invalid inputs not considered before
who benefits: both the developers and the attackers, since they both can use fuzzing to analyse the software

43) [TODO] Explain what kind of software analysis can be performed by means of the 'angr' tools. Describe the functions of the loader (CLE), explain what are the simulation states and how they are used by the Simulation Manager. Finally, explain how to configure the execution of the unwanted branches.

1. *what kind of software analysis can be performed:*
   binary loading, static analysis, binary rewriting, type inference, symbolic execution, automatic exploit generation, symbolically-assisted fuzzing (driller)
2. *CLE, simulation states, and how they are used in Sim Manager:*
   The binary loader provides the ability to turn an executable file and libraries into an usable address space. ***It is a generic loader which is able to guess the architecture, extract the executable code and data of different formats and create a representation of the program's memory map.*** A simulation state is a snapshot of the registers' status, memory and other arch info attributes ***which is the current program state. The simulation manager changes the state from one to another depending on the instructions to execute, for example adding constraints when a branch is taken.***

3. *how to configure the execution of unwanted branches*

   If during a symbolic execution we want to avoid a specific branch, in the explore command we need to add an "avoid" parameter followed by the address of the branch to avoid. For example:
   sm.explore(find=0x08048709, avoid=0x0804871e)

44) Xss cos'è, varie tipologie e cosa può ottenere un attaccante e fare un esempio

XSS stands for cross site scripting and consists of forcing the user to execute into a specific browser sandbox you are interested in, some scripts (bypass the SOP) that can produce some outputs. you can use xss:

- to steal the cookies
- to redirect the user to a completely different site
- to perform unauthorized operations (especially if other vulnerabilities are present)

There are 3 categories of XSS attacks

1) stored XSS: when source code of the attack script is stored on the vulnerable server. Most dangerous one because when the user requests data, also receives the malicious script. Several users can be victims.

2) Reflected XSS: when the attacker uses alternative methods to deliver the script, like a link via email. *?? The attacker sends a link to the victim, who then opens it and a GET is automatically made to a vulnerable site. The vulnerable site responds to the victim with a POST in which there is a malicious script. The vulnerable site does not check the ??*

3) DOM-based XSS: when the DOM environment is altered in the victim's browser, to ensure that the script is executed in some unexpected way. *(missing input validation at user's side)*

45) Explain how SQL injection vulnerabilities work. How can it be fixed/prevented?

SQL injection is an injection attack that makes it possible to execute malicious SQL queries. SQL injection can be used to bypass security measures. This attack occurs through untrusted data sent to SQL interpreter for executing unintended commands and for accessing data without proper authorization. A malicious user could exploit SQL syntax and boolean algebra to bypass authentication, or to download some whole tables in a DB full of sensitive data. Also, an attacker could

- check if a table exists
- check if a column exists
- check if the value of a field is ‘a’
- use timing when the output is not differentù

It can be detected through static analysis, pen test, or dynamic analysis.
Prevention could be done by:

- keeping untrusted data separate from commands and queries
- using safe APIs, which avoid the use of interpreters or providing a parameterized interface (and sanitize parameters properly before using them)
- using an interpreter-specific escape routine on all parameters
- use positive or “white list” input validation on ALL input

Some secure programming principles can be also considered:

- validate always the inputs
- heed compiler warnings
- architect and design for security policies
- least privilege policy
- sanitize data sent to other systems
- adopt a secure coding standard

46)format string attack

It is an attack that exploits the wrong use of format functions. Like printf. It is possible when we control the parameters that will be passed to the printf. For instance, to a printf(“%d %c %s”, num, c, buf) we pass the format specifiers string, and then the name of the variables (num,c,buf) corresponding to those parameters. The function retrieves them from the stack. If the number of the format specifier is equal (or lower) to the number of the variable it is ok. At most one of the variables without format string is ignored. But, if we have more format strings than variable names, the last one will read the stack!

`E.g. printf (“%d%c%s”, num, c) → very dangerous, because %s will display information present on the stack!`

The purpose of this attack is double:

1) I can read what I want from the stack, inserting in the buffer to print the following string: “addr” + “.%y$s”*128; where addr is the memory address of the memory location i want to read (should be readable!); y is the offset ***the format string should be “addr” + “.%y$s”, the one above should be be a mistake on the slides***
2) i can write what i want on the memory, if writable. , inserting in the buffer to print the following string: “addr” +”%numx” “.%y$n”*128;

where num is the number of bytes to print + len(addr)
the format string should be “addr” +”%numx” “.%y$n”, the one above should be a mistake on the slides

https://secgroup.dais.unive.it/teaching/security-course/format-strings/

---

other exercises relatives to LABs

1) **When Flawfinder runs on the following C code**

```
#define MAXLINE 1024
#define MAXWORD 512
int processFile(FILE *fp) {
   char line[  MAXLINE];
   char word1[MAXWORD], word2[MAXWORD];
   while( fgets (line, MAXLINE, fp)!=NULL ) {
         if (sscanf(line, "%s %s", word1, word2) != 2) {
         return 1;
         } else
         process(word1,word2);
    }
    return 0;
}
```

it produces the following output:

FINAL RESULTS:

test1.c:9: [4] (buffer) sscanf:

The scanf() family's %s operation, without a limit specification, permits buffer overflows (CWE-120, CWE-20). Specify a limit to %s, or use a different input function.

test1.c:5: [2] (buffer) char:

Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

test1.c:6: [2] (buffer) char:

Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger the maximum possible length.

Considering the code and the Flawfinder output, answer the following questions:

1. **Does the code contain buffer overflows? If so, at what lines? If so, write a fixed version of the code.**

   The code contains buffer overflow at line 9 because one of the substrings could be longer than MAXWORD (e.g: line is composed by one string of 1000 chars and one of 23 chars). The problem is fixed if a limit in substring dimension is specified. Fix:

   sscanf("%511s, %511s", word1, word2);

2. **Are there any false positives in the warnings produced by Flawfinder? If so, at what lines?**

   There code contains one false positive at line 5 (declaration of line[MAXLINE]), because the string is filled in the fgets() function, which will never insert more than MAXLINE characters because the limit is specified.

3. **Are there any false negatives in the warnings produced by Flawfinder? If so, at what lines?**

   The code doesn't contain false negatives

---

2) **Here, f*or your convenience, you can find a definition of shared-key cryptography and signatures in the typed Proverif syntax:***

(Shared-key cryptography *)*

fun senc(bitstring, bitstring): bitstring.

reduc forall x: bitstring, y: bitstring; sdec(senc(x,y),y) = x.

(Signatures *)*

fun ok():result.

fun sign(bitstring, skey): bitstring.

reduc forall m:bitstring, y:keymat; getmess(sign(m,sk(y))) = m.

reduc forall m:bitstring, y:keymat; checksign(sign(m,sk(y)), pk(y)) = ok().

Following the same approach and language, write a definition of the symbolic functions that represent the computation and the verification of Message Authentication Codes (MAC). A MAC is a code computed from a message and a symmetric key, with the purpose of enabling integrity verification. The integrity of a message with its MAC can be verified by using the same key that was used to generate the MAC.

(* MAC *)

type mkey.

type result.

fun ok():result.

fun MAC(bitstring, mkey):bitstring.

reduc forall m:bitstring, k:mkey, y:bistring; checkMAC(MAC(m,k),y) = ok()

Please, check if this solution is ok :)

---

3) **Consider the following code, for which Spotbugs has produced two warnings at lines 18 (execute Query, can be vulnerable to SQL injection) and 25 (printin, could be vulnerable to XSS).**

```
@WebServlet(value=*/test2*)
public class Test2 extends HttpServlet{
    private static final long serialVersionUID =1L;
@Override
public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, I0Exception{
    response. setContentType("text/html;charset=UTF-8”);
    String user « request.getHeader(“USER”);
    String pass request. getHeader (“PASSWORD”);
    user - java. net. URLDecoder. decode(user,”UTF-8”);
    pass • java. net. URLDecoder. decode(password,”UTF-8”);
    String sql:
    if (user. equals(“adm”) && pass.matcher(“^[a-zA-Z0-9_]+$”)){
        sql = "SELECT from USERS where USERNAME =’+user+’and
        PASSWORD=’+ pass+’’’:
        try{
            java.sql. Statement statement
            statement. executeQuery( sql ):
            response. setStatus(200):;
            }catch (java. sql.SQLException e){
        response. setStatus(500);
        response. getwriter().printin(“Error processing request. “):
    }else{
        response. setStatus(400):
        response.getWriter() .println(‘Bad request for user’+ user):
    } //end if
} //end doPost
}
```

For each one of the two warnings raised by Spotbugs, tell whether it is a true positive or a false positive, and explain why.

The warning at line 18 is a false positive. Even if a prepared query is not used, both “user” and “pass” are sanitized. User can be only “adm”, while password is properly checked using the regex “^[a-zA-Z0-9_]+ which does not allow to inject sql code because characters like spaces, semicolons and apex are not allowed.

While the warning at line 25 is a true positive as in the else statement, the user string could assume any value different from “adm”, for example a script which will be executed by the user browser (reflected XSS).

---

4) **Describe what is a buffer overflow vulnerability. Furthermore, explain how a buffer overflow in a function can be exploited to execute a remote shell.**

When Flowfinder runs of the following code:

```
#include <stdin.h>
#include <string.h>
#define MAXWORD 512
int process();
int processInput(char *input, int off) {
        char word[MAXWORD];
        int length = strlen(input);
        if (off < 0) {
                off =-off;
        }else if (length > MAXWORD) {
                return -1;
        }
        if (off >= MAXWORD) {
                strcpy(word, input);
        }else{
                strncpy(word, input, off);
                word[off] = '\0';
        }
        return process(word);
}
```

It produces the following output (4 hits):

1) **test2.c:20:[4] (buffer) strcpy:**
   Does not check for buffer overflows when copying to destination (CVE-120).
   Consider using strcpy_s, strncpy or strlcpy.
2) **test2.c:9:[2] (buffer) char:**
   Statically sized arrays can be improperly restricted, leading potential overflows or other issues (CVE-110; CVE-120).
   Perform bands checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.

3) **test2.c:11: (buffer)**
   Does not handle strings that are not '\0' terminated; if given one it may perform an over-read (CVE-126).
4) **test2.c:22:[1] (buffer) strncpy:**
   Easily used incorrectly; doesn't always are not '\0' terminated a check for invalid pointers (CWE-120).

Consider the code and the Flowfinder output, answer

1) **Does the code contain buffer overflows? If so, at what lines? If so, write a fixed version of the code.**

2) **Are there any false positives in the warnings produced by Flowfinder? If so, at what lines?**

Buffer overflow is a kind of vulnerability where a program tries to write more data to a buffer than it can contain. Normally it happens when the input size is not checked and it can cause a crash of the program or the execution of arbitrary code. When the program executes a function where an external input is written into a buffer without checking its length, it is possible to inject a shellcode + a specific number of random bytes + an address where the address points to the shellcode. In this way, the address will overwrite the return address of the function and once reached by the program it will execute the shellcode.

1) A buffer overflow is possible at line strcpy(word, input); because input length is not checked when off is negative and, after it becomes positive, it is >= MAXWORD.

```
if (off >= MAXWORD && length < MAXWORD) {
        strncpy(word, input, length);
}
```

2) The warning 4 is a false positive as the it writes only “off” bytes and its value is properly checked. Then the null character is also added correctly.

---

5) Consider a scenario where you have SSH access, as a normal user, on a remote machine you want to attack

You find a program, running on it as root, which is only accessible from localhost on a specific port. By decompiling the code, you notice that the input is managed by a function, named func() and reported below:

```
func()
{
   int var2;
   char var1[128];
```

```
    int var3;
    get_input_from_socket(var1);
    print_on_socket(var1);
}
```

You also disassembled:

- get_input_from_socket, which takes up to 1024 bytes from the socket and copies in the memory location pointed by the var1 pointer
- print_on_socket, which redirects on the socket the output of the printf (i.e., using the sprintf functions)

Given this code shown before:

1. What kind of attack looks very promising, as it does not depend on any protection enabled on the program/server?
2. Which security property is easy to endanger with this attack?
3. What other attack is possible, whose success depends on the absence of protections that may have been enabled by who compiled the application?
4. Imagine these protections are not active on the running program, what do you have to send to the server to try a privilege escalation? Describe the structure of the payload to send (assume it has been compiled for 64-bit architecture). Use an informal syntax but be very precise on the number of the bytes (you can use python-like or any other syntax provided its behaviour is unambigous or you explain it)
5. Which protection(s) may render you attack impossible?

1. *Format string attack, since if there is good protection buffer overflow is not possible.*
2. *You can read from the stack in memory but you can also write some bytes in memory. For example, it is possible to overwrite the function address of a row in the GOT in order to execute an unintended function.*
3. *Buffer overflow. If no protection is applied, it is possible to overwrite the content of var2 but, moreover, it is possible to overwrite the return address in order to jump in any other point of the program, but also in any point of our buffer that may be filled with a shellcode.*
4. *We need to exploit buffer overflow. We need to put in our buffer a shellcode which is able to perform privilege escalation (hex version of setreuid()) but we also need to overwrite the return address in order to jump on the shellcode. Our stack is composed as below:*

   STACK ADDR | Variable
   0x00000000 | var2 (4 bytes)

0x00000008 | var1 (128 bytes)
xxxxxxxxxx | var3 (4 bytes)
xxxxxxxxxx | FRAME (suppose 8 bytes)
xxxxxxxxxx | ret(id)

I suppose that my shellcode that is able to perform privilege escalation is 64 bytes sized.

*python -c 'print("\x90"+myShellcode+"A"*75+"x00\x00\x00\x08")'*
75 = 140 - tpx64 (shellcode len) -1(number of NOP)
140 = 128+4+8

5. *It is possible to avoid this kind of attack by using Data Execution Prevention which does not allow data sections to be executable.*
   Stack Canaries
   Address Space Layout Randomization

47) After having initialized the project *p* and execute the following commands:
es = p.factory.entry_state()
sm = p.factory.simulation_manager(es)
sm.explore (find=lambda s : b"Correct result:" in s.posix.dumps(1), avoid=0x0804871e)
The output of the tool is:
<SimulationManager with 3 active, 1 found, 10 avoid>
First, discuss what tool has been used and what kind of analysis has been performed.
Then, identify the component in charge of doing this analysis, and provide (informal yet precise) details on how this analysis is been conducted.
Finally, explain the meaning of the output produced by the tool.

The tool used: Angr
The analysis performed: Concolic analysis
The component of the tool actually used: Simulation Manager
More details on how the analysis is performed: First of all, the entry state of the program is found with the first instruction and it is passed to the simulation manager as its starting point. Then, an exploration is performed forcing the simulation manager to reach the output "correct result" and avoid a specific branch at line 0x0804871e to make the execution faster. Once executed, the sim manager found a possible input (1 found) useful to get the wanted output.
Explain the meaning of:

- active: There are still three active states from which one may want to continue the execution which haven't found the wanted output yet.
- found: In one branch, the program reached the output we passed.
- avoid: number of branches excluded, as they taken the branch to avoid.

48) Assume that “value” is tainted while “name” is not. Real vulnerability or not? Why? Real vulnerability or not? Why?

```
void function (char *name, char* value){
        char buffer[100];
        if (strlen(name)>94)
                return -1;
        strcpy(buffer,name)
        strcat(buffer,":%s\n");
        print(buffer,value);
}
```

The yellow one is not a vulnerability as a fixed-size buffer, if correctly managed and its boundaries are respected, does not generate vulnerability as buffer overflows. The instruction which copies the value of “name”, which is not tainted, respects the size of the buffer. Also with the concatenation of “: %s\n” the size is respected.
While the blue instruction is a real vulnerability as the “value” string is tainted and could not contain the end of string character “\0”.

Consider the following definitions of processes in Proverif language:

```
type key.
type host.
type nonce.

free c:channel.
free s:bitstring [private].

fun encrypt(bitstring,key): bitstring.
reduc forall x: bitstring, y: key; decrypt(encrypt(x,y),y) = x.

let pA(A: host, B:host, kS:key) =
    new Na: nonce;
    out(c, (A, B, Na));
    in(c, y:bitstring);in(c,m: bitstring);
    let (=Na,=A,=B, k:key)=decrypt(m,kA) in; 0.
    out(c, encrypt(s, k)).

let pB(B: host, kS:key) =
    in(c, x:bitstring);
    let (xA:host, =B, xNa:nonce)=x;
    new k:key;
    out(c, encrypt((xNa,xA,B,k),kS));
    in(c, x2:bitstring);
    let sec:bitstring = decrypt(x2,k) in
    0.
```

Answer the following questions:

1. What kind of encryption is represented by functions encrypt and decrypt?. What are the properties of this kind of encryption in the symbolic model?

2. Describe the behavior of the protocol specified by these process definitions with your own words

---

1. Functions encrypt and decrypt represent symmetric key enc/dec, since the same key (y) is used to encrypt and then decrypt the secret (x).