Software Un-security Exploitation Evaluation

Christèle Faure, christele.faure@safe-river.com

***Abstract*—The elaboration of a software attack is a long and fragile process because each piece of information must be "stolen" from the software under attack without being noticed. Moreover any added protection can disallow preliminary attacks necessary to get these data. Existing tools search for vulnerabilities but give no way to evaluate the security impact. We have defined a methodology to study different aspects of security from the source code of a piece of software. It may take as input the vulnerabilities computed by another tool and allows for the investigation of their possible exploitation. But it can also be used to answer other security questions such as "is my asset impacted by a given software input". We intend to automate this methodology using static analysis based on abstract interpretation.**

I. Introduction

Software security analysis is supported by different types of tools: static tools (textual analysis, pattern matching, abstract interpretation) detect potential vulnerabilities by symbolically executing the software or dynamic tools (penetrating tools, fuzzing tools, testing tools) find true vulnerabilities by performing aggressive tests.

This paper focuses on the static aspects of security analysis as advocated in [1], [2], [3]. More than two hundred static tools claim to tackle software security. They search for software vulnerabilities such as the well known buffer and stack overflows, but they also detect the presence of calls to dangerous functions such as formatting functions (`fprintf`[1]), memory copy functions (`memcpy`), input functions (`scanf`).
These tools can be classified with respect to the errors they focus on: memory errors are detected by `Clousot`, `Sparrow` or `C Global Surveyor`, runtime errors are computed by `Astrée`, `PolySpace`, `Frama-C`, `Inspector`, `Lintplus`, and dangerous calls are tackled by `Vulncheck (gcc option)`, `CodeAssure (RATS)`, `Flawfinder`, `ITS4`, and `Security Analyst`. Static security tools also check for the respect of usual or user defined coding rules (or principles) such as "filter input value before use". `Security Analyst`, `Fortify`, `CodeAssure (RATS)`, `ITS4` do offer this functionality to some extent. Each of these tools searches for several classes of vulnerabilities, but none of them detect all kinds of security vulnerabilities.

Moreover, static tools detect only potential errors because they cannot always prove that each error truly occurs at runtime. Amongst static tools, the syntactic ones detect only simple errors from patterns because they have no clue about the potential executions: they generate false negatives (true errors not detected) and try to limit false positives (false errors detected). The semantic tools symbolically execute the piece of software and generate no false negatives, but may generate false positive due to the over-approximations necessary to assure the termination of symbolic execution.

Finally, whatever kind of tools is applied, the user is left with the difficult challenge of proving that the potential error can be used in true attacks. This is currently done manually by experts who know how to built an attack from a vulnerability. We have developed a methodology that helps evaluate if an error can be used in an attack: on the first hand find out if it could be exploited within attacks, and on the second hand evaluate if the implemented defense means disable the corresponding attacks. Our methodology takes both aspects of security [4] into account: the attack elements present in the piece of software but also the defense elements.

[1]The examples given in this paper are based on the C language but can be mapped on most programming language.

II. Our methodology

At the software level, the elaboration of an attack is an iterative process that goes deeper and deeper into the piece of software: one preliminary attack gives a piece of information, and the next attack uses it to get a deeper piece of information. When the hacker gets enough information to built a rewarding attack, he elaborates and performs it. The obtained pieces of information are of several kinds: execution time, runtime error, output value, rejected input, error message ...

But in any case, the hacker makes use of the piece of software itself to elaborate the attack: he queries the software from its input channels, forces the execution throughout vulnerabilities to change the intended behaviour and gets the stolen information from the software output channels. The same idea is used through the notion of attack surfaces [11], [12] for security risk analysis. From this, we got the idea that an analysis of the software itself (without knowledge about the environment) is the first step towards security throughout: *(1)* the identification of attack paths, and *(2)* the evaluation of the defense means effectiveness.

We propose a four steps method to analyze a piece of software for security:

1) Location of attack means,
2) Location of defense means,
3) Search for attack paths,
4) Search for unprotected attack paths.

Location of attack means

This step aims at generating the attack map i.e. locating instances of attack elements into the piece of software. We explained above that an attack is built from elements present in the piece of software itself. We classify attack elements as: $\bar{I}$ input functions (`gets, scanf ...`), $\bar{V}$ classes of vulnerabilities (buffer overflow, ...), $\bar{O}$ output functions, $\bar{A}$ assets (file, variable ...), $\bar{C}$ rights (root, user ...) or privileges (high, low). The elements of classes $\bar{I}, \bar{O}, \bar{C}$ are defined by the development language and library specifications, those of class

$\bar{A}$ are defined by the user and the elements of class $\bar{V}$ are either known before hand (computed by static tools, dynamic tools, test ...). The attack means located in the piece of software form the software attack map described as (I, O, C, A, V) where I is the set of input points present in the piece of software, V is the set of located potential vulnerabilities, O the set of output points, A the set of located accesses to assets, and C the set of rights or privileges change locations.

Location of defense means

This step aims at generating the defense map i.e. locating instances of defense elements into the piece of software. The defense elements are actual implementations of general security principles such as for example: "prefer white testing (test of correct possibilities) to black testing (test of incorrect possibilities)", "filter the value imputed from the environment before using it", "protect the accesses to assets". The defense elements are user defined and should be known beforehand. In practice, the defense means are implemented as call to functions from user defined dedicated libraries: the filtering functions check for the accepted patterns (white testing) within a string or the accepted value for a numerical variable, and the protection functions encode/decode messages, check for the rights or privileges. Within a piece of software, the calls to defense functions form the software defense map.

Search for potential attack paths

This step aims at computing the attack paths that go through the software execution and cross attack means. But all the executions path linking attack means cannot be the base of an attack. For example, an execution path which does not first encounter an input points cannot lead to an attack because it cannot be activated. The hacker needs to control the piece of software and can only do it from the input channels.

We define attack patterns as sequences of attack means from I, O, A, V. For example, $I \rightarrow V \rightarrow A \rightarrow O$ represents the set of attacks that start from an input point, end at an output point, and cross a potential vulnerability before an access to an asset. An attack paths is denoted by $i \rightarrow v \rightarrow a \rightarrow o$ where $i \in I$, $v \in V$, $a \in A$ and $o \in O$. Potential attack paths that match a given attack pattern are searched for by forward analysis on the piece of software. All attack paths matching the chosen patterns are considered: if a potential execution path cross all these program points, then the actual path is considered as an attack path otherwise it is rejected.

Search for unprotected potential attack paths

This step aims at computing the potential attack paths that do not cross defense means as expected. The defense elements mainly protect the inputs, outputs and assets and their usage is specified from coding rules that lead to defense patterns. From the software defense map and the software potential attack paths, this step compute the subset of unprotected potential attack paths. The elements from the defense map executed along a path are identified for each path. If all the defense elements specified by the defense patterns are present on the path, then it can be considered as protected. If one defense mean is missing, then the attack path is considered as unprotected.

This methodology enables to study different aspects of security. Amongst the possible questions, one can answer for example the question "could a vulnerability be exploited" by investigation the pattern $I \rightarrow V \rightarrow A \rightarrow O$ or $I \rightarrow V \rightarrow O$, the question "could an asset flow out of the software" by investigating the patterns $A \rightarrow O$, or the question "could an asset be impacted by a vulnerability" by studying pattern $V \rightarrow A$.

The first two steps of this method are easy to perform manually because most of the elements can be identified at a glance in the source code of the piece of software. The two other steps are a lot more complex because they require the identification of certain execution paths within the software. The execution paths are in general over-approximated by syntactic paths present in the source code. The search for syntactic paths is tedious, error prone and fairly unprecise but is the only possible manual way to realize this step.

III. Conclusion

We have manually tested our methodology on small examples and we want to automate it to be able to evaluate it on industrial applications.

We have specified a tool to support this methodology. The tool should apply static analysis based on abstract interpretation [6] to be aware of execution paths and *(1)* identify attack and defense cartographies from a knowledge base of defined attack and defense elements, *(2)* compute execution paths that cross sequences of attack and defense elements as defined by attack patterns and defense rules.

We will start the development of the tool from a naive notion of what attack and defense elements are, and some very simple attack patterns and defense rules. But we intend to elaborate more sophisticated data by studying known attacks, as well as development rules for security. At the end of this work, the tool will be extended with new attack and defense elements, but also with more complex attack and defense patterns.

We want to prototype first a tool for C, but the objective is to support C, C++ and Java. We do not want to develop standard components such as the frontend, the alias analyzer, the value analyzer, and have therefore chosen to develop our tool as a plugin into an existing framework. We intend to evaluate four frameworks for static analysis: `Frama-C` [7], `PAG` [8], `SATIRE` [9], `gcc` [10].

References

[1] *Secure Programming with Static Analysis*. Chess B. and West J. Addison-Wesley Software Security Series. 2007.

[2] *Software Security*. Building Security In. McGraw G. Addison-Wesley Software Security Series. 2006.

[3] *Static Detection of Exploitable Vulnerabilities in Input Dependencies*. Robin Van Schendel, Master thesis, 2007.

[4] *Low_level Software Security: Attacks and Defenses*. Ulfar Erlingsson, Microsoft Research, 2007.

[5] *Common Criteria for Information Technology Security Evaluation*, Part 1: Introduction and general model, September 2006.

[6] *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. Patrick Cousot & Radhia Cousot. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[7] Frama-C, `http://frama-c.cea.fr/`

[8] PAG, `http://www.absint.com/pag/`

[9] SATIRE, `http://www.complang.tuwien.ac.at/satire/`

[10] GCC plugin, `http://gcc.gnu.org/wiki/plugins`
[11] *Fending off future attacks by reducing attack surface*, M. Howard, 2003.
[12] *An Approach to Measuring A System's Attack Surface*, Pratyusa K. Manadhata, Kymie M.C. Tan, Roy A. Maxion and Jeannette, CMU Technical Report CMU-CS-07-146, August 2007.