Towards automatic detection of vulnerabilities in software-intensive systems

Delphine Beaulaton[1], Jean Quilbeuf[1], Salah Sadou[1], and Régis Fleurquin[1]

Univ. Bretagne-Sud, UMR 6074, IRISA, F-56000 Vannes, France
name.surname@univ-ubs.fr

Abstract

We propose an approach for detecting security vulnerabilities in software-intensive systems. Our approach builds a model describing the system, vulnerabilities are then discovered by analyzing that model. We rely on the CWE database for identifying key concepts in order to build a metamodel for representing systems. We illustrate our approach on an example.

1 Introduction

Programmers and designers traditionally focus on the performance and functional correctness of the software that they are producing. Performance and correctness are not sufficient in an industrial context, where the software should also foster maintainability, re-usability or interoperability. The field of Software Engineering provides tools, methods and good practices for building software with these required properties. However, security is another property of interest for software. Security has received a growing attention over the last decades and is now a major concern for industries, but is still not systematically taken into account in the design of new systems.

The security of a system is its ability to maintain the confidentiality, integrity and availability of its assets against potential attacks [3]. In the frame of software-intensive systems, the assets are pieces of information. Confidentiality states that only authorized users of the system may access a given information. For instance, only the holder of a bank account can access its balance. Integrity states that only authorized users can modify a particular piece of information, i.e. only the holder of an account can order transfers from that account. Availability means that the information should be available at all times to authorized users. A *security policy* specifies the confidentiality, integrity and availability properties expected for each asset of a system.

Some systems are specifically designed to be secure such as banking applications or military communication systems. In that context, "secure" means "reasonably secure for their intended use" [5]. A classical approach for building complex systems is to interconnect simpler systems. Such an approach is not guaranteed to build a secure system, even if the simpler systems are secure themselves. Indeed the interconnection of several systems may introduce new vulnerabilities. A vulnerability can be seen as an unintended data or execution path in the system that allows an attacker to access, modify or make unavailable some assets. The Common Vulnerabilities and Exposures[1] is a database that lists known vulnerabilities. Note that a system with vulnerabilities is possibly functionally correct, but unable to enforce its security policy.

Detecting vulnerabilities is challenging, in particular when they arise from the composition of several systems. Several methods exist for detecting specific vulnerabilities [4], however all possible vulnerabilities are not covered. Therefore our goal is to provide a methodology allowing

[1] https://cve.mitre.org

the detection of weaknesses, indicating to the designer which parts of the system are probably vulnerable. Thanks to this more abstract level (weakness vs. vulnerablity), we hope to discover more vulnerabilies than existing approaches.

In order to tackle this problem, we propose to build a model of the system that helps detecting its possible weaknesses. Such a model should state the policy security of the system as well as its assets. Furthermore the model should include a description of the system, either generated from existing code, or other documents, that enables to state vulnerabilities as structural properties of that model. In order to build this model, we rely on CWE[2], a database that classifies weaknesses. The idea is to be able to model systems such that each weakness class from CWE is transformed into a property on the model.

This paper is organized as follows: Section 2 presents our general approach for detecting vulnerability in systems. We detail in Section 3 the current version of our meta-model for describing systems to be analyzed. We present our constraint based analysis in Section 4 and an example in Section 5. Finally, we conclude in Section 6.

2 General Approach

Most of system representations target functional aspects. Here, we focus on security, which is a non functional aspect. So we need a representation that allows us to highlight the vulnerabilities. The difference between a weakness and a vulnerability is that a weakness can lead to one or several vulnerabilities. Indeed a vulnerability is directly usable by a hacker to attack a system whereas a weakness represents one or several potentially dangerous situations (future vulnerabilities). Then, it seems more efficient to detect weaknesses within the system in order to be able to avoid a larger amount of risks.

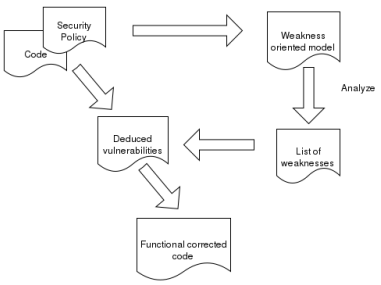

Figure 1: Representation of the process

Our proposed approach relies on the transformation of a functional representation of system into a one that allows vulnerability identification. The complete process is sketched in Figure 1. From the Security Policy and the code of a software-intensive system we build its corresponding weakness-oriented model. The analyze of the model will give us a list of potential weaknesses. From this list and in accordance with the code we identify the potential vulnerabilities of the system. The final step will be to go through the code of the system and correct the vulnerabilities in order to have functional more secure code.

The model of the system that we build is weakness-oriented, which means that the choice of elements we are using to describe the system is influenced by the hypothetical existence of weaknesses. Actually, the model represents a given state of the system. So if there are no potential risks, it will be directly reflected on the model.

We build our metamodel according to a bottom-up approach. The first step consists in iteratively selecting weaknesses from the CWE database. From each weakness we identify some general principles that define it, in order to have an overall vision of its effects on the system. Then, if necessary, we modify the metamodel to integrate the newly identified principles reflecting the current weakness. This iterative process allows us to define the vulnerability-oriented metamodel described in the next Section.

[2] https://cwe.mitre.org

3 A Vulnerability Oriented Model Representation

In order to detect vunerabilities, we need to represent on one hand the security policy and on the other hand the information an attacker can actually access and maybe even modify. Our metamodel (see Figure 2) encodes these two types of information.

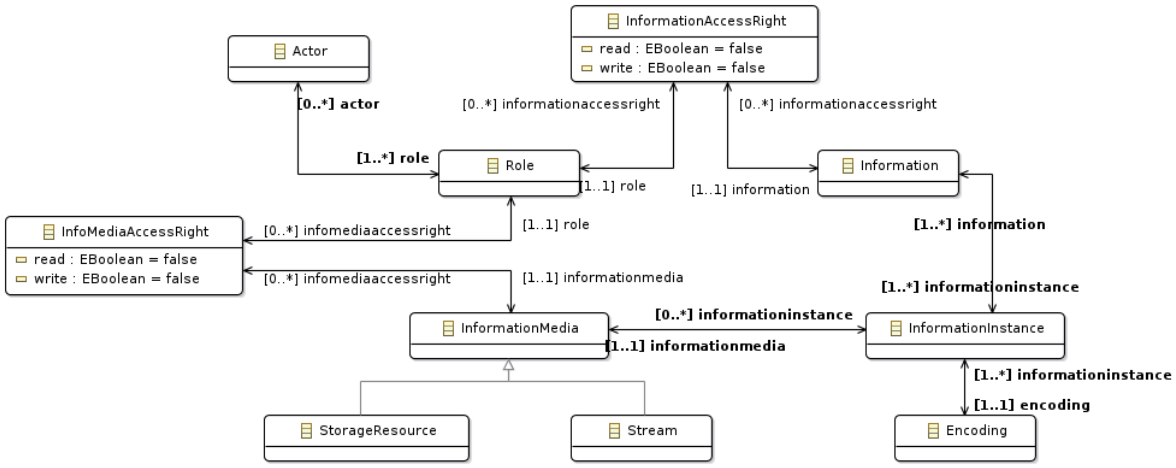

Figure 2: metamodel representing the state of a system

First, we model the actors and roles of the system. The Actor metaclass stands for a person who interacts with the system. They can be an internal user of the system such as an administrator. They can also be an external attacker with no particular access rights. The way they are supposed to interact with the system is defined by the role(s) they own.

The Role metaclass defines some control spheres. For example an actor that owns an administrator role will have a higher access level and will need different information access than a basic user. Access levels are embodied by the security policy of the system.

The security policy is an important part to describe. It describes who can access each information and the modalities of this access. Comparing the security policy and the actual behavior of the actors is the key to detect confidentiality or integrity violations. In the metamodel, the security policy is modeled through an association between the metaclasses Role and an Information. The type of access rights (read and/or write) are defined in the metaclass InformationAccessRight. The absence of link between a Role and an Information means that actors with that Role are not allowed to read nor write that information.

The Information metaclass represents the data that belongs to the system. Each Information is contained in one or several information instances (metaclass InformationInstance) and every one of them exists in an encoded version.

Each information instance is accessible through an information media (metaclass InformationMedia) that can be a storage resource or a stream, that allows them to be transmitted. The actual access of the actors to the information is represented by the link between a role and an information media through an instance of the InfoMediaAccessRight metaclass.

```
context Role:
  let policy-read-allowed : Set(Information) =
    self.informationaccessright->select(read).information
  let system-read-allowed : Set(Information) =
    self.infomediaaccessright->select(read).informationinstance.information
  inv confidentiality:
    policy-read-allowed->includesAll(system-read-allowed)
```

Figure 3: OCL invariant characterizing confidentiality

4 Detection of Weaknesses through Constraints

The metamodel developed in the previous section integrates the security policy of our system and the information that a user can actually access and modify according to the system code. We can check that both are consistent. An inconsistency between the security policy and the actual access granted by the system indicates that the system contains a potential weakness.

As expressed earlier, the goal of our metamodel is to be able to express weaknesses as OCL constraints. Figure 3 shows an invariant that characterizes confidentiality. First, we define the set of information that a Role is allowed to access (policy-read-allowed), then the set of Information that a Role can actually access through the system (system-read-allowed). The invariant is broken if system-read-allowed is not included in policy-read-allowed, that is the Role can access an information that it is not allowed to access.

5 Examples of Weakness-Oriented Models

We consider a web application and two of its actors: Jim, a developer of the system, and John, an external user. The security policy states that actors with the role developer, such as Jim, can access debug information, in order to correct the existing bugs. Conversely, actors with the role "external user", such as John, are not allowed to access debug information, because that information could give a malicious user indications about the architecture of the system for further interaction.

Consider a scenario where an exception occurs, and an error page presenting debug information is shown to the actor. The system allows the actor to access the debug information. We model this scenario by a InfoMediaAccessRight instance linking the role of the actor and the web page, which is a storage resource also linked to the debug information. From that scenario, we extract two situations.

The Figure 4 models a first situation, where the actor is Jim. Since the security policy of the application allows Jim to access the debug information, there is an instance of InformationAccessRight that links Jim's role and the debug information. Consequently, the invariant from Figure 3 is respected.

The Figure 5 models a second situation, where the actor is John. The security policy forbids external users to access debug information. This is reflected in the model by the absence of connection between his role and the Information debugInfo. Here the invariant from Figure 3 is not respected, which allows us to detect a potential breach of confidentiality. This situation is indeed listed in CWE as the weakness "CWE-600: Uncaught Exception in Servlet". John could cause a crash of the system by, for instance, entering an unexpected value. As the crash is not handled safely, John can read the exception information, including sensitive structural information about the system and use it for a later attack.

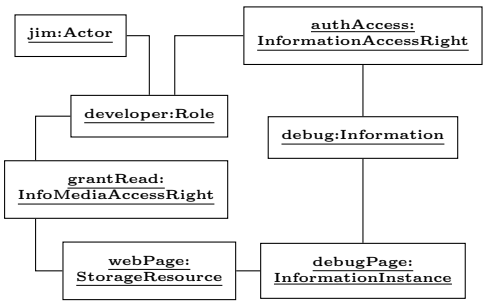

Figure 4: The instance diagram representing a normal situation of the system

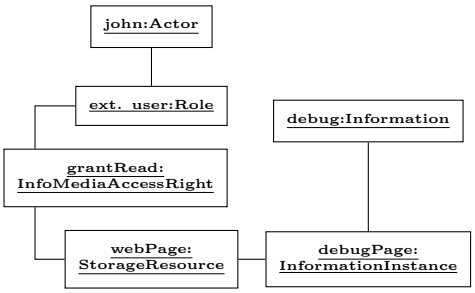

Figure 5: The instance diagram representing an abnormal situation of the system

6 Related Works and Conclusion

Detecting vulnerabilities using a metamodel has been used before. For example Almorsy et. al. in [1] use formal vulnerabilities definitions in OCL and a system description class diagram. The goal is to find vulnerabilities going though the code and finding pattern that matches the formal signature. Then in their second publication on this topic [2] they step up into abstraction by focusing on the architecture to detect attack signatures.

Our approach for detecting weaknesses is to include in a single model a representation of the system and a representation of the security requirements. Currently our metamodel represents a state of the system and we are able to express its secure situations by using OCL invariants, as illustrated by our example. The non-compliance with this invariant implies possible weaknesses. In the above approaches, the focus is on detecting a particular vulnerability, based on user-defined signature. We can also write OCL constraints targeting specific vulnerabilities. In this case, a system validating one of these constraints contains the corresponding vulnerability.

As future work, we plan to model the structure of the system, in order to represent how information flows through it. Furthermore, we plan to incorporate further analyzes, to provide more details about a breach of the invariant. For instance, we could precise which role is able to breach the security policy. Then, we could analyze the path that leaks a confidential information into a location readable by unauthorized roles. The final goal is to indicate to the designer which part of the code is vulnerable, allowing them to mitigate the detected vulnerability. Ideally, such a model would be built incrementally, by relying on external tools to discover hidden channels.

References

[1] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *IEEE/ACM ASE'12*, pages 100–109, 2012.

[2] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *ICSE '13*, pages 662–671, 2013.

[3] M. Bishop. What is computer security? *IEEE Security Privacy*, 1(1):67–69, Jan 2003.

[4] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*, pages 521–524, Dec 2010.

[5] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.