Towards Automated Penetration Testing for Cloud Applications

Valentina Casola*, Alessandra De Benedictis*, Massimiliano Rak†, Umberto Villano‡,
* *Università di Napoli Federico II, DIETI, Napoli, Italy*
alessandra.debenedictis@unina.it, casolav@unina.it
† *Università della Campania Luigi Vanvitelli, DIII, Aversa, Italy*
massimiliano.rak@unina2.it
‡ *Università del Sannio, DING, Benevento, Italy*
villano@unisannio.it

***Abstract*—The development of cloud applications raises several security concerns due to the lack of control over involved resources. Security testing is fundamental to identify the existing security issues and is particularly powerful when carried out by means of penetration testing techniques. Unfortunately, penetration testing requires a deep knowledge of the possible attacks and of the available hacking tools and is very energy demanding. In this paper, we present a methodology that allows to easily carry out a coarse-grained security evaluation of a cloud application by automating the set-up and execution of penetration tests. The methodology relies on the knowledge of the application architecture and on the availability of a catalogue including security-related data collected from multiple sources and properly correlated.**

***Keywords*-Cloud Penetration Testing, Security Service Level Agreement**

I. Introduction

The development of cloud applications, relying on the orchestration of services possibly distributed over multiple Cloud Service Providers (CSPs), raises several issues, especially in terms of security. Evaluating the security of cloud applications is not easy at all, and also Security Service Level Agreements (Security SLAs) from public CSPs offer very few guarantees [?]. Recent results from some European projects, such as SPECS (www.specs-project.eu), MUSA (www.musa-project.eu) and SLA-Ready (www.sla-ready.eu), promote the adoption of Security SLAs based on security metrics and quantitative service level objectives that allow both negotiation and monitoring by means of suitable tools and systems [?]. Available monitoring systems, however, are not effective in evaluating security. They are mainly based on traditional network-like monitoring tools with probes/agents and correlating back-end systems to detect any anomaly. These systems may potentially monitor security features at any level of the cloud infrastructure, but their adoption may be limited by their intrusiveness in the providers' infrastructure, which may violate CSP policies.

Indeed, security testing is fundamental to identify the threats and vulnerabilities affecting cloud applications and to assess their level of security. Among existing security testing techniques, penetration testing appears to be one of the most relevant and powerful means to identify exposed vulnerabilities of a generic system. Unfortunately, it is a typically human-driven procedure that requires a deep knowledge of the possible attacks to carry out and of the hacking tools that can be used to launch the tests, and it is not within everyone's means.

These considerations led us to investigate the introduction of automated procedures to generate and execute penetration tests. In this paper, we present a penetration testing process that allows to configure and launch automatically a testing environment to obtain a coarse-grained evaluation of the security level provided by a cloud application. On the one hand, the proposed process leverages the adoption of specific models, taking into account the features of the cloud application to test and the possible risks it is subject to, along with the relationships existing among the well-known concepts of threat, attack, vulnerability and weakness. On the other hand, it relies upon the configuration and usage of existing security tools and systems, which is partly carried out based on the above modelling activities.

The remainder of the paper is structured as follows: in Section II, we introduce our penetration testing methodology and discuss the main inputs to the automated penetration process. In Sections III, IV and V, we describe in details the three phases devised by the methodology, respectively related to the preparation, scanning and execution of the tests on the system. A running example is used throughout the paper to better illustrate the activities carried out in each of the above phases. In Section VI, some related work on penetration testing is presented. Finally, in Section VII, we draw our conclusions and discuss our plans for future work.

II. Penetration testing methodology

The penetration testing methodology presented in this paper enables to obtain a coarse-grained evaluation of the exposed vulnerabilities and existing security risks of a cloud application (referred to as the System under Test - SuT - hereafter), by means of an automated process supporting the set-up and execution of penetration tests starting from a description of the application.

The whole process leverages a complex knowledge base used to extract the information needed in all the phases of the penetration testing set-up and execution. This knowledge base is represented by a *catalogue* that brings together threats, exploits, attacks, weaknesses and vulnerabilities deriving from multiple sources (both standard and not). We started building the catalogue in the context of two recently closed European projects, i.e., SPECS and MUSA, which have investigated the management of the life-cycle of secure cloud and multi-cloud applications, and we are still working on its development. In the catalogue, threats, attacks, weaknesses and vulnerabilities are classified based on the type of application components to which they relate: three component types are currently considered, namely *web application*, *storage service*, and *identity provider system*, which represent some of the most common logic building blocks of complex applications. Threats listed in the catalogue belong to multiple sources, including the *OWASP Top 10 2017* classification (for web applications) and the *RFC6819 - OAuth 2.0 Threat Model and Security Considerations* document (for authorization servers based on OAUTH). Vulnerabilities are those listed in the Mitre *Common Vulnerabilities and Exposures (CVE)* system (http://cve.mitre.org), while attacks are built from well-known exploits and attack patterns. Exploits are taken from the Metasploit Framework's *Exploit Database* (https://www.rapid7.com/db/modules/), while attack patterns are those classified by the Mitre *Common Attack Pattern Enumeration and Classification* (CAPEC) initiative (https://capec.mitre.org). The concept of weakness has been highlighted by Mitre in its *Common Weakness Enumeration (CWE)* project (http://cwe.mitre.org). Weaknesses are defined as "mistakes regardless of whether they occur in implementation, design, or other phases of the software development life-cycle" that may introduce vulnerabilities. The catalogue currently lists CWE weaknesses plus additional weaknesses that have been identified by analysing specific design or implementation flaws affecting common types of components, even referring to their role in the most common communication protocols.

In the catalogue, threats and weaknesses are directly mapped to attacks. In particular, threats are linked to the attacks that realize them, while weaknesses are mapped to the attacks that may be carried out to check whether they actually lead to some vulnerability exposure. Vulnerabilities, instead, are directly mapped to exploits. Providing a detailed description of the data model behind our catalogue is out of the scope of this paper, but the interested reader can refer to the public deliverable D2.3 of the MUSA Project [**?**] or to the paper [**?**] for further explanation. Moreover, the data model is publicly available as an open source project on BitBucket at the address https://bitbucket.org/cerict/sla-model.

A. Penetration testing phases

Our penetration testing methodology devises three main phases, namely *Preparation*, *Scanning* and *PenTesting*. As sketched in Figure 1, each phase includes both *model-based* activities and *system-based* activities. The former activities require a modelling effort and are based on the catalogue introduced in the previous section. They allow to obtain the information needed to configure systems and tools that are deployed and/or launched during system-based activities.

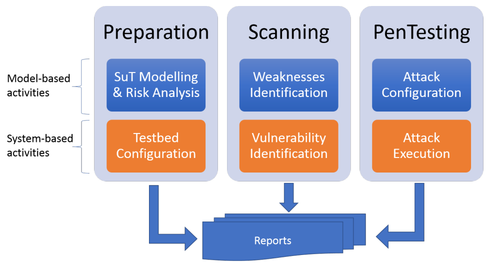

Figure 1. The automated penetration testing process

As clarified later, the methodology relies on the adoption of a graph-based formalism that allows to describe the SuT architecture in terms of its logical components and of their interconnections, in addition to providing deployment information. Based on the SuT model, during the *Preparation* phase, the analysis of the existing threats and of the related security risks is carried out as part of the model-based activities, while system-based activities focus on the testing environment configuration.

The *Scanning* phase is mainly devoted to identifying the *weaknesses* and *vulnerabilities* affecting the SuT, in order to verify whether existing threats can be actually exploited and harm the system. In particular, model-based activities focus on the weaknesses identification, while system-based activities are devoted to detecting vulnerabilities.

Finally, the *PenTesting* phase consists in building and executing suitable attacks aimed at exploiting the weaknesses and vulnerabilities identified in the previous phase in order to evaluate the level of security of the SuT.

As shown in Figure 1, each of the above described phases is based on a continuous *Reporting* process, which allows to log all the executed steps and generate reports.

B. SuT model

The proposed methodology assumes as an input a model of the application under test, which will be used to drive the above mentioned activities. In order to better illustrate the whole process, let us consider a very simple application that will be used as a running example in the remainder of the paper. The example application is built by the orchestration of two software components represented by commercial-off-the-shelf (COTS) products, namely a web application `W` running on top of the Apache Tomcat servlet container, which uses a database (based on MySql) `DB` for the management of persistent data. The subject interested in performing the penetration test is the cloud application

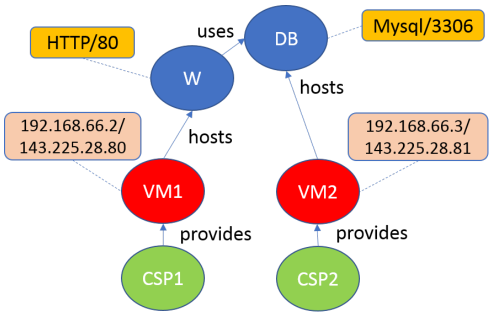

Figure 2. An example application - MACM formalization

provider or developer that aims at evaluating the level of security of the application before releasing it publicly. In this case, a grey-box testing strategy is considered, which assumes a partial knowledge of the internal structure of the application and of its components, and the knowledge of the protocols possibly used for component communications.

The application can be modelled by means of the *Multi-cloud Application Composition Model (MACM)* formalism introduced in [?], which allows to describe in a simple and immediate way the logic components of a cloud application, their mutual relationships and the information related to their deployment and to their security properties. In MACM, components are modelled as graph nodes of type *SaaS service*. Other types of nodes considered by the formalism are the *IaaS service* type, which models the infrastructure resources (i.e., the VMs) used to deploy the components, and the *CSP* type, which models the providers offering the VMs. The relationships among these nodes are represented as directed edges and belong to three categories. A CSP *provides* one or more VMs. An IaaS service (i.e., a VM) *hosts* an SaaS service. Any service, either IaaS or SaaS, *uses* an SaaS service. Figure 2 reports the MACM representation of the simple cloud application introduced above, where the web application component `W` is hosted by a virtual machine `VM1` offered by the provider `CSP1`, and the database component `DB` is hosted by a virtual machine `VM2` offered by the provider `CSP2`. The `W` and `DB` nodes are annotated with the name of the protocols used to communicate with them (i.e., HTTP and mysql, respectively) and with the related ports. The `VM` nodes, instead, are annotated with their IP addresses, both public and private (the latter are the IP addresses assigned to the machines in the SuT internal private network, used to set-up the testbed as explained later).

III. Preparation phase

As outlined before, *Preparation model-based* activities aim at determining existing threats and at analysing the security risks posed by such threats, while *system-based* activities are devoted to configuring the testing environment.

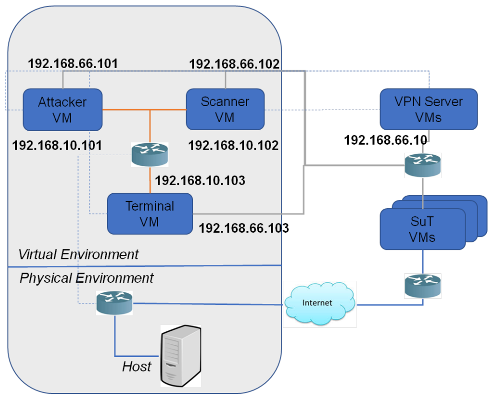

Figure 3. The testing environment

A. Model-based Preparation phase: SuT risk analysis

The risk analysis process performed in the *Preparation* phase has been illustrated in [?] and relies upon the catalogue introduced in Section II. Starting from the SuT model based on the MACM formalism discussed before, an automated process is carried out that allows the user to identify the main threats each component is subject to based on its nature and its interactions with other components, and to identify, classify and rank existing risks. Based on the type of involved components and on the information on how components are implemented and interact with one another, the main existing threats are identified. Afterward, the tester is guided through a risk classification process based on the OWASP Risk Rating Methodology [?], which allows to identify and rank the existing risks. According to the priorities set by risk levels, a set of threats of interest is selected as the result of the model-based *Preparation* phase.

Referring to the example application modelled in Figure 2 for instance, assuming that no information is available on the internal behaviour of the SuT components, all the threats associated to the web application component type are selected for `W`, and all the threats associated with the storage service component type are selected for `DB`. Moreover, assuming equal risks for all the threats, these threats will be considered all of interest for the SuT, and will thus represent the outcome of the model-based activity.

B. System-based Preparation phase: Testbed configuration

The automation of the penetration testing process is enabled by the availability of a pre-configured testing environment that can be easily launched based on the information extracted from the SuT model. The testing environment consists of a virtualized environment composed of a pre-configured network (VPN) connecting three virtual machines that represent, respectively, the *attacker machine*, the *scanning machine* and the *terminal emulator machine*.

The attacker machine is a virtual machine equipped with a

Kali Linux distribution. Kali Linux (https://www.kali.org/), based on the Debian operating system, offers more than 600 penetration testing tools including the Metasploit Framework (https://www.metasploit.com/) and is highly customizable, thus representing a powerful security platform for professionals and practitioners. Metasploit is a penetration testing platform that enables to find, exploit, and validate vulnerabilities and is the main tool used in our penetration testing process to launch attacks against the SuT and verify its level of protection.

The scanning machine is a virtual machine hosting a vulnerability scanning tool. Here we consider the Open Vulnerability Scanner Assessment System (OpenVAS) [1], an open source project providing a framework for the monitoring and management of vulnerabilities. In order to perform vulnerability detection, OpenVAS carries out specific tests that exploit the vulnerabilities to verify whether they affect or not a system (in a penetration testing-like fashion). Such tests can be configured and tuned, which is very useful to manage different monitoring policies. OpenVAS is a framework whose architecture consists of three main components, namely the OpenVAS Scanner, the OpenVAS Manager and the OpenVAS Client. The Scanner is the component that performs the vulnerability tests (NVTs) on the target machine, and it is configured and controlled by the OpenVAS Manager. The Client instead provides an interface to operate the Manager.The scanning machine of our tesbed is configured with the three above components, and the OpenVAS Client is launched automatically during the *Scanning* phase.

Finally, the terminal emulator machine, which aims at reproducing the common usage of the SuT, is a virtual machine equipped with Gatling [**?**], a well-known stress tool born for load and performance testing that allows to record user interactions with applications and to build usage scenarios that can be launched in simulation during the tests. Usage scenarios are recorded during the system-based *Preparation* phase, in order to enable their automated execution during the *PenTesting* phase. Note that such scenarios will be used mainly to test DoS attacks.

The virtual network among the above testing machines is bridged over the physical network hosting the virtual environment and can directly access the Internet and the services publicly offered by the SuT. Moreover, a VPN server is installed within the SuT internal network to enable the connection between it and the testing machines. This allows to simulate a physical connection between the two systems and enables to perform the tests in conditions very similar to those of the production environment. Figure 3 shows the network configuration of the testbed for our running example: the testing machines are connected in a pre-configured VPN, identified in the figure by the addresses belonging to the class 192.168.10.1/24. At the same time, they are connected with another VPN established within the SuT network, where the VPN Server is deployed. Note that the VPN Server is deployed and configured automatically based on the information specified in SuT model regarding the addresses used in the SuT internal network. As reported in Figure 2, the VMs hosting the `W` and `DB` components of our example application have been assigned private addresses belonging to the class 192.168.66.1/24, therefore the VPN Server will be configured with an IP address belonging to the same class.

At state of art, the testing environment runs on top of VirtualBox, using a single physical machine, even if we are currently working to port the solution onto an OpenStack infrastructure, in order to offer it *as-a-Service*.

[1]http://www.openvas.org/

IV. Scanning Phase

The *Scanning* phase is devoted to identifying the *weaknesses* and *vulnerabilities* affecting the SuT According to our approach, weaknesses are identified as part of the model-based activities performed in the *Scanning* phase by accessing the catalogue introduced before. Vulnerabilities, instead, are identified by carrying out a vulnerability scanning process aided by existing scanning tools, and thus represent a result of the system-based activities of the *Scanning* phase.

A. Model-based Scanning phase: Weakness identification

Weaknesses identification is straightforward since, as said in Section II, the catalogue maps weaknesses to component types. Hence, the weaknesses of interest for the SuT can be easily identified by means of a set of selective queries over the catalogue according to the component types involved in the SuT.

Let us consider again the simple application introduced in Section II-B and let us recall that `W` represents a web application component adopting the HTTP protocol for communication. One weakness reported in the catalogue and associated with the web application component type regards the lack of validation of the inputs passed by means of the POST method. If no validation is performed by the web server, in fact, the component may be subject to DoS attacks carried out by simply posting big files. This weakness, along with the others related to `W` and `DB`, is simply extracted from the catalogue as a result of the model-based scanning activities.

B. System-based Scanning phase: Vulnerability Scanning

While weaknesses are related to the application architecture, vulnerabilities depend on the involved technological stack and on the specific software used in the system. As anticipated, in order to identify the known vulnerabilities, we adopt the OpenVas tool, which is installed and configured on the scanning machine of our testbed. As part of the system-based activities of the *Scanning* phase then, we simply launch OpenVAS and collect the reports of a full scan. In the case of our running example, the tool highlights immediately many of the common vulnerabilities affecting the adopted software, mainly related to the presence of bugs in the used version of Tomcat and of the hosting operating system.

V. PenTesting Phase

The *PenTesting* phase consists in building and executing suitable attacks aimed at exploiting the weaknesses and vulnerabilities identified in the previous phase. The attack preparation is a model-based activity and consists in the proper orchestration of the exploits and attacks reported in the catalogue, while the test execution relies on the use of the Metasploit tool deployed on the attacker machine of the testbed.

A. Model-based PenTesting phase: Preparation of the attacks

The result of the *Scanning* phase is represented by a list of vulnerabilities and a list of weaknesses. Vulnerabilities are listed in the report produced by OpenVAS, which can be directly used to identify the exploit modules to launch in Metasploit since the latter are directly mapped to standard vulnerabilities in the catalogue. Once identified, the exploit modules are usually configured manually, based on the information on the concrete system deployment, including for example the IP addresses of the target machines. Note that, according to our approach, such information is already available in the MACM representation of the SuT, which makes it possible to configure the exploit modules automatically. To summarize, the exploit modules are identified automatically, based on the vulnerability reports produced in the *Scanning* phase, and are also configured automatically, based on the information provided by the SuT model.

For what regards weaknesses, as explained in Section II, they are explicitly mapped to attacks in the catalogue. Attacks are represented by sets of tasks (for now very limited) in Metasploit: for each task, we currently provide the ordered list of Metasploit modules to launch to perform the attack. Even in this case, the information specified in the MACM model of the SuT is used to automatically configure the Metasploit modules.

Related to the weakness mentioned in the previous section for our example application, i.e., the one related to the lack of validation of HTTP POST input size, we associated to it a specific DoS attack in the catalogue consisting in attempting to upload a very big file to the web server interface by means of the HTTP POST method. The attack is realized by means of a task corresponding to a custom exploit module developed for Metasploit, which is selected and added to the list of tasks to launch in the execution phase. The module is configured with the connection parameters extracted from the SuT MACM representation and no further action is required to enable its execution.

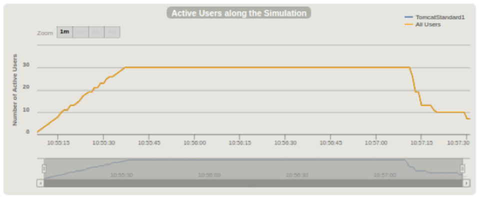

Figure 4. Number of active users in normal conditions

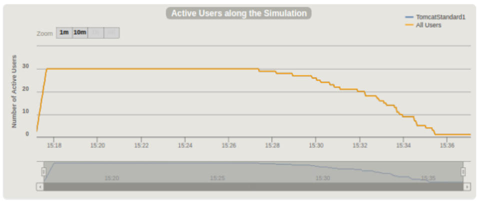

Figure 5. Number of active users under a DoS attack

B. System-based PenTesting phase: Execution of the attacks

The final result of the model-based activities is a list of attacks, expressed in terms of ordered lists of Metasploit modules to launch, whose configuration is defined according to the MACM model. The system-based activities, then, consist in simply executing the attacks in the pre-determined order. After each attack, the system is cleared (reset) and a new attack takes place. For each attack, our tools report the success/failure of the attack itself.

Referred to our running example, Figure 4 shows the number of active users registered in normal conditions during a given time frame by the Gatling tool, as part of the system-based *Preparation* phase. As expected, when the *PenTesting* phase is executed, the number of active users degrades due to the DoS attack mentioned before, as shown in Figure 5.

VI. Related work

Penetration testing is an approach to security widely studied in the literature [**?**]. The majority of available work aims at discovering vulnerabilities at the network level and mostly relies on procedures executed manually by a security expert. As discussed in the paper, compared to existing research, our approach to penetration testing focuses on the application level and relies on an automated process.

Although with peculiar characteristics, the contribution in the literature more similar to our proposal is Potassium [**?**], where penetration testing is offered as-a-Service. The actual application running on a cloud is automatically cloned by migration techniques and penetration testing is performed on the clone, and not on the production system. Other works exploring the automation of the penetration testing process are [**?**] and [**?**]. In particular, the latter paper proposes a system (Nemesis) that queries a database of known vulnerabilities and uses Metasploit to execute the tests. Another paper proposing penetration test automation by scripting is [?].

An approach similar to ours, in that it is based on the use of a cloud as testing environment for applications, is suggested in [?] and [?]. The problems linked to the penetration testing of applications exploiting cloud elasticity, an issue currently out of our scope, is considered in [?], where applications are monitored in different scaling up/down execution states.

VII. Conclusions

In this paper, we presented an automated penetration testing process thought to obtain easily a coarse-grained evaluation of the level of security of a cloud application. The process leverages (i) the information on the system under test (SuT) specified in a model based on the MACM formalism and (ii) the information included in a catalogue collecting threats, weaknesses, vulnerabilities, exploits and attacks classified based on the type of affected component. Our penetration testing methodology devises three phases, namely *Preparation*, *Scanning* and *PenTesting*. Each phase includes both model-based activities, relying on the information provided by the catalogue, and system-based activities, carried out by means of suitable tools. The testing environment is pre-configured and can be set-up and launched very easily, and the tests are run automatically based on the risks and security issues identified previously. We showed that, compared to traditional approaches to penetration testing, our process can be completely automated and is able to cover a large set of vulnerabilities and weaknesses. As the catalogue is fundamental to our approach, we are involved in a continuous process aimed at collecting new information and at correlating existing information. Related to this, we are particularly interested in refining the attack preparation strategy as our future plan, in order to improve the efficacy of the approach.

Acknowledgment

This research is partially supported by the grant FP7-ICT-2013-11-610795 (SPECS) and H2020-ICT-07-2014-644429 (MUSA).