Towards continuous security certification of Software-as-a-Service applications using web application testing techniques

Philipp Stephanow and Koosha Khajehmoogahi
Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany
{firstname.lastname}@aisec.fraunhofer.de

***Abstract*—Continuous security certification of software-as-a-service (SaaS) aims at continuously, i.e. repeatedly and automatically validating whether a SaaS application adheres to a set of security requirements. Since SaaS applications make heavy use of web application technologies, checking security requirements with the help of web application testing techniques seems evident. However, these techniques mainly focus on conducting discrete security tests, that is, mostly manually triggered tests whose results are interpreted by human experts. Thus these techniques are not per se suited to support continuous security certification of SaaS applications and have to be adapted accordingly. In this paper, we report on our current status of developing methods and tools to support test-based, continuous security certification of SaaS applications which make use of web application testing techniques. To that end, we describe major challenges to overcome and present experimental test results of using SQLMap to continuously test for SQL injection vulnerabilities.**

***Keywords*-cloud services; security testing; test-based certification**

I. Introduction

From a customer's perspective, Software-as-a-Service (SaaS) applications are deployed on remote infrastructures accessible through interfaces such as browsers where customers' control is confined to configuring user-specific application settings [1]. Naturally, providing SaaS applications involves web application technologies, such as JavaScript, JSON, HTML and CSS. Therefore, SaaS applications can possess web application vulnerabilities, e.g. can be vulnerable to SQL injections or session hijacking [2].

Vulnerable SaaS applications may violate security requirements of both the SaaS customer and the SaaS provider, thus threatening either business model. In order to mitigate this risk, security certification of SaaS applications seeks to check whether the application satisfies a set of security requirements. These requirements can be derived from control catalogues such as CSA's Cloud Control Matrix (CCM) [3] or guidelines, e.g. NIST SP 800-53 [4]. If the SaaS application satisfies the requirements, then a report called *certificate* is produced, stating *compliance*.

Traditionally, executing a security certification process is a discrete task producing results which are valid for some interval, e.g. one year. This implies that certification process' results are stable during the interval, that is, any other audit executed during the interval should produce identical results. With regard to SaaS applications, this assumption of stability does not hold: Attributes of SaaS applications can change over time where these changes are not predictable or detectable by customers or even by the SaaS provider herself. As an example, consider auditing a SaaS application revealed SQL injection vulnerabilities. Assuming that – as one possible countermeasure – data sanitization is implemented at the data base layer using stored procedures. Yet if the SaaS application uses frameworks like Ruby on Rails[1], then altering the data base used by the application's controller is a matter of simple configuration changes. If the new data base is not equipped with the stored procedures to sanitize user input, then such configuration changes can reintroduce previously fixed vulnerabilities. Moreover, opposed to traditional web application development, a SaaS provider does not need to possess her own resources but may leverage a Platform-as-a-Service (PaaS) provider, e.g. Google App Engine[2], to create and provide her SaaS application. This creates another layer of abstraction where changes in the back end possibly rendering the SaaS application vulnerable are not even detectable by the SaaS provider.

Security audits of SaaS applications thus require a different approach capable of *continuously*, i.e. automatically and repeatedly detecting ongoing changes and assessing their impact on security requirements. To that end, recent research proposes *cloud service certification* which aims at validating security requirements through continuous monitoring or testing, and thus produce meaningful certificates to increase the trust of customers towards cloud services [5][6][7][8][9][10]. Other work presents similar, but more general methods to perform automated *security audits* which aim at determining the security level of cloud services, e.g. [11][12][13]. However, none of these approaches focus on continuous security certification of SaaS applications. Moreover, most of current research ([5][7][12][13][14]) require tight integration with cloud infrastructures, introducing further challenges to security of the certification system itself [15].

In this paper, we outline major challenges of continuous security certification of SaaS applications (Section II). Then,

[1]http://rubyonrails.org/
[2]https://cloud.google.com/appengine/

we report on the current status of developing methods and tools to support continuous certification of SaaS applications using web application testing techniques (Section III). A prominent tool for web application testing is SQLMap[3]. Since these tools treat a web application under test as a black-box, our approach is *non-invasive*, that is, requires neither knowledge about the application's internal structure nor changing the infrastructure used to provide SaaS applications. We present comprehensive experimental test results of using SQLMap to continuously test for SQL injection vulnerabilities (Section IV). Concluding this paper, we describe necessary next steps to meet the identified challenges (Section V).

II. Challenges

In this section, we describe challenges to develop test-based methods to support continuous security certification of SaaS applications.

A. Definition of tests

Security requirements derived from CSA's Cloud Control Matrix (CCM) [3] or NIST SP 800-53 [4] are generic, oftentimes inherently ambiguous making automatic validation infeasible. Thus supporting continuously checking whether a SaaS application satisfies a set of security requirements requires to extract underlying security properties which can be automatically tested, thereby bridging the *semantic gap*. Reasoning about these properties requires collecting and evaluating *evidence* [16], i.e. observable information of a cloud service, e.g. monitoring data or source code. *Test-based certification models* produce evidence by controlling some input to the cloud service and evaluating the output, e.g. calling a cloud service's RESTful API and checking responses. Therefore, a central challenge lies in providing a method which allows for consistent definition of SaaS applications' security properties from which test designs can be derived.

B. Accuracy, precision and completeness of tests

The adoption of test-based continuous security certifications hinges on the accuracy of the tests: False positives, i.e. a SaaS application not satisfying a security requirement is not detected by the test, i.e. the test incorrectly passes, will reduce the trust of customer and providers in test results. Further, false negatives, that is, the test incorrectly fails and thus incorrectly indicates that the provider does not satisfy a security requirement, will provide room for the provider to dispute tests' results. Describing the accuracy of test-based methods which support continuous security certification based on false positives and false negatives, e.g. in the form of a confidence level for proportions, is essential to foster the adoption of these methods.

[3]http://www.sqlmap.org

Furthermore, when repeatedly executed with the same context, tests have to produce consistent results. Imprecise test results will undermine customers' and providers' trust.

Adoption also requires completeness, i.e. in case a security requirement is not satisfied, there needs to be a high probability that a test is defined which detects this violation. Note that, naturally, security tests are never complete since it is hardly possible to test for unknown vulnerabilities.

C. Vulnerability simulation

A *simulation* manipulates a SaaS application under test to mock vulnerabilities which specific test-based certification methods aim to detect. When determining accuracy, precision, and completeness, simulations are essential to establish the ground truth to which results produced by test-based security certification methods are compared. Since test-based security certifications of SaaS applications are continuously executed, vulnerability simulations need to be executed continuously as well.

The design of a simulation is driven by the security property that should be tested. For example, a simulation may publicly expose sensitive interfaces to mimic vulnerable service interfaces, or make a weak cipher suite available to simulate insecure webserver configurations. In this context, a method needs to be developed which allows to design vulnerability simulations based on real world requirements, e.g. rare events.

D. Verifiability of test results

Supporting continuous certification of SaaS security properties requires that the test results are verifiable, i.e. an independent third party has to be able to verify the correctness of test results. Only verifiable test results provide accountability of the provider as pointed out by [17].

E. Overexposure through test results

Mechanisms seeking to increase trust and transparency can leak critical information which can be used by attackers to trace vulnerabilities of a cloud infrastructure [18]. It is evident that results of tests which aim at detecting violations of security requirements can contain critical information. Thus, it is vital that the system implementing continuous security certification of SaaS applications is trustworthy as well [15].

F. Measurement overhead

Naturally, the strict interpretation of *continuous* security certification of SaaS applications is hardly applicable in practice, since, as pointed out in [19], uninterruptedly testing cloud providers can incur intolerable overhead. Thus one central challenge is to develop a measurement methodology which allows to reason about the overhead incurred by continuous testing SaaS applications, especially when facing multiple continuous tests, and choose test configurations accordingly.

III. Prototype

In this section, we present the status quo of our prototype to support continuous security certification of SaaS applications. After having described the vulnerability we continuously test for (Section III-A), we describe the tool selected for our test (Section III-B1), how to trigger its execution automatically and repeatedly (Section III-B2 and III-B3), the exemplary service under test (SUT) (Section III-B4), and, finally, how to simulate vulnerabilities of the SUT (Section III-B5).

A. *Continuously testing for SQL Injection vulnerabilities*

OWASP classifies the top 10 categories of vulnerabilities found in web applications based on their frequencies. In that classification, *injection* leads the list and is mentioned as the most prevalent type of vulnerability. *Injection* is a broad term which refers to different types of attacks such as SQL, OS commands and LDAP injection. Among all types of injection, SQL injection (SQLI) is the most common type of vulnerabilities existent in today's web applications.

Continuously testing for SQL Injection vulnerabilities can support the certification of security requirements derived from, e.g., Controls *SI-10 Information Input Validation* and *RA-5 Vulnerability Scanning* of NIST SP 800-53 [4], *Section 6.3.1. Software Assurance and Section 6.3.6. SAAS – Application Security* of ENISA IAF [20], controls *A.9.4.1: Information access restriction* and *A.12.6.1 Management of technical vulnerabilities* of ISO/IEC 27001:2013 [21], or *AIS-01: Application Security* and *TVM-02: Vulnerability & Patch Management* of the Cloud Control Matrix (CCM) [3] upon which the CSA STAR certificate is based.

B. *Environment and setup*

Figure 1 provides an overview of the components involved in our current setup: The *Clouditor* which continuously triggers tests' execution, *SQLMap Connector* which interacts with SQLMap through its RESTful API, the *Service under Test (SUT)* as well as the *Simulator* which modifies the SUT to be either vulnerable or secure. The next sections explain the components' function in detail.

1) SQLMap: There exist plenty of tools which aim at discovering vulnerabilities including SQLI. One popular tool which specifically and solely targets SQLI discovery is SQLMap: An interactive tool that features different attack methods including UNION query-based, stacked queries, out-of-bound, boolean-based blind, time-based blind and error-based attacks. A comprehensive introduction into different types of SQL injection attacks can be found in [22].

Given a URL as the attack target, SQLMap tries to make an educated guess about the back end database management system (DBMS) in action. Subsequently and accordingly, SQLMap tries to attack the underlying DBMS through different injection vectors that are more suited for that specific underlying DBMS by sending requests through a particular HTTP method such as GET or POST.

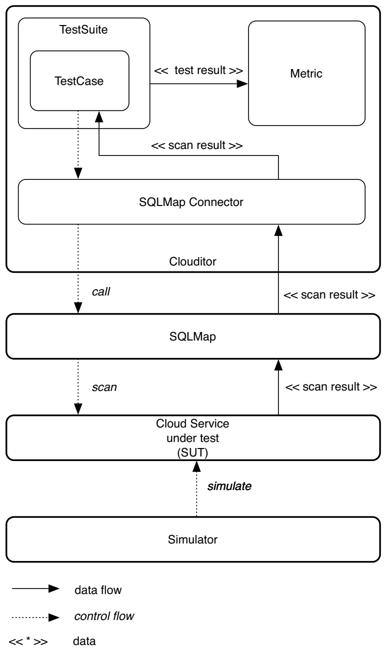

Figure 1: Interactions among different components of the experimental setup

SQLMap provides a wide variety of options to fine-tune and optimize SQLI discovery scans. As an example, a set of options can be utilized to command the SQLMap what HTML fields (e.g. GET or POST parameters) to target for its attacks. For instance, suppose a form submission sends its data via an HTTP GET method to http://www.example.com/submit.php?id=12&user=34&submit=submit. In this case, we are certain that the value `submit=submit` is always fixed and would not be a potential target for SQLI. Since, in practice, form submissions mostly include a number of parameters, excluding parameters like this can accelerate the scan process. In contrast to black-listing parameters, it is also possible to target some particular parameters for scans and neglect the rest (white-listing).

In addition to attempting to assess whether an application is vulnerable to SQLI or not, in case of vulnerabilities, SQLMap also reveals the attack vector, i.e. payloads that it has used in order to exploit the vulnerabilities. Therefore, it is possible to verify the truth or falsehood of the report generated by SQLMap manually.

Furthermore, in case of successful exploitation, SQLMap can fetch information from the underlying DBMS. For instance, it is able to fetch the name of current DBMS user, the name of current database in action, the database schema or it can enumerate the tables of a database.

Although SQLMap is predominantly an interactive tool which requires some user's involvement while operating, it also exposes most of its functionality through a RESTful

API. As a result, an external program can make use of the RESTful API and trigger scans programmatically. This feature of SQLMap leads us to the next component of our system called *SQLMap Connector*.

2) SQLMap Connector: The SQLMap Connector is a library developed in Java which calls the SQLMap RESTful endpoints, parses the returned data from SQLMap, and represents this data as Java objects.

The RESTful API of SQLMap associates each scan task with a unique ID used as a reference to a particular scan in subsequent API calls. Thus, it is possible to run multiple scans simultaneously. The sequence of actions required to run a scan is implemented as the following sequence of API calls: Initiate a new scan task, set the options for that task, run the scan, check periodically if the scan is still running, fetch the scan result data when the scan is over, and, eventually, delete that scan to release system resources.

3) Clouditor: We use Clouditor [10], a framework to support continuous test-based certification of cloud services, to continuously tests our SUT. To that end, we use the plugin architecture of Clouditor to implement a test case which calls the SQLMap API through our SQLMap Connector (see Section III-B2) to run scans on our SUT (see following section). The test commands SQLMap to retrieve the current DBMS user, a list of tables and a list of database users in case of successful exploitation. If no information is returned after a SQLMap scan has completed, then we conclude that the configuration of the SUT is not vulnerable and, hence, our test passes. If, on the other hand, any of the above information is retrieved, then we mark the SUT as injectable, i.e. vulnerable and the test fails.

4) Service under Test (SUT): Our service under test (SUT) is an instance of Damn Vulnerable Web Application (DVWA)[4]. DVWA is a PHP-based web application which is intentionally vulnerable to different types of security issues, including SQLI for educational purposes, giving the opportunity to users to exploit the platform. The weaknesses incorporated in DVWA are configurable so that one would need different levels of sophistication for exploitation. This configuration setting is set through an HTTP cookie. However, our setup requires to control the levels of sophistication through the configuration file of DVWA. To that end, we had to alter the source code of DVWA. Note that we also modified the source code to neglect anti-CSRF tokens as that is not our primary focus at this point.

5) Simulator: The purpose of the simulator is to determine the performance of our continuous tests, that is, how close are produced test results to their true values? To answer this question, the simulator establishes the ground truth to which results produced by the tests are compared.

Since our tests are executed repeatedly, simulating vulnerabilities has to be executed repeatedly as well. In our current setup, the simulator edits the configuration file of DVWA to mock SQLI vulnerabilities for some defined period. Thereafter, the simulator edits the configuration again, making the DVWA secure for some time. The vulnerable and secure configuration alternate where both the duration of the vulnerable configuration and the duration of the secure configuration can be randomized.

[4]http://dvwa.co.uk

IV. Experiments

This section presents experiments we conducted using our current prototype described in the previous section. We begin with describing the test and simulation configuration of our experiments (Section IV-A). Then we analyse the performance of using SQLMap to continuously test for SQLI vulnerabilities based on four universally applicable test metrics (Section IV-B).

A. Test and simulation configuration

To evaluate the performance of *SQLTest*, we conducted three different experiment where we run *SQLTest* every 10, 30 and 60 seconds. This means that once a test has completed, the Clouditor waits 10, 30 or 60 seconds until the next run of *SQLTest* is triggered.

To evaluate the performance of the three different configurations of *SQLTest*, we simulated 500 SQLI vulnerabilities for each configuration. Each vulnerability event lasted at least 60 seconds plus selecting $[0, 10]$ seconds at random. The interval between consecutive downtimes was at least 60 seconds plus selecting $[0, 30]$ seconds at random. Table I shows the distribution parameters of the vulnerability simulation for each configuration of *SQLTest*.

Table I: Distribution parameters of DVWA vulnerability simulation

test configuration	**total duration (sec)**	**mean (sec)**	**sd (sec)**	**min (sec)**	**max (sec)**
10	32566234	65.13	3.24	60.0	72.47
30	32546632	65.09	3.12	60.0	70.0
60	32474676	64.95	3.11	60.0	70.0

B. Performance SQLTest

In this section, we introduce four universally applicable test metrics to evaluate the performance of *SQLTest*. Each of the following sections describes one of these test metrics and how to use that metric to construct performance measures to evaluate *SQLTest*. Further, using the test metric and derived performance measures, we present and discuss our experimental findings.

1) Basic-Result-Counter (brC): This test metric counts the number of times a test failed or passed. Recall that in the case of *SQLTest*, a failed test indicates a vulnerable service under test. We consider a test result to be true negative if it failed while the SUT was vulnerable (brC^{TN}). If a test passes or fails when the SUT is vulnerable or secure, then the test result is considered false positive (brC^{FP}) or false negative (brC^{FN}), respectively.

Based on these oberservations, we compute four standard performance measures for binary classification: True negative rate tnr, false positive rate fpr, false omission rate for, and negative predictive value npv:

$$tnr^{brC} = \frac{brC^{TN}}{(brC^{TN}+brC^{FP})}$$
$$fpr^{brC} = \frac{brC^{FP}}{(brC^{TN}+brC^{FP})}$$
$$for^{brC} = \frac{brC^{FN}}{(brC^{TN}+brC^{FN})}$$
$$npv^{brC} = \frac{brC^{TN}}{(brC^{TN}+brC^{FN})}$$

Table II shows the results for performance derived from the test metric brC: As expected, the number of tests executed during each simulation decrease with increasing interval length between tests, i.e. from *SQLTest*^{10}, *SQLTest*^{30} to *SQLTest*^{60}. Furthermore, the mean duration and standard deviation of each executed test are similar for each test configuration of *SQLTest*.

If we only consider true negative test results (br^{TN}), then we can see that the mean duration of br^{TN} decreases with increasing interval length while the standard deviation increases. In turn, if we only consider false positive test results (br^{FP}), i.e. a test passed although the DVWA was vulnerable, then we observe that the mean duration and standard deviation of br^{FP} are similar. Lastly, we did not observe any false negative test results (br^{FN}), that is, no case where *SQLTest* incorrectly indicated that the DVWA was vulnerable to SQLI.

Furthermore, the total count of true negative test results (brC^{TN}) decreases with increasing interval length between tests. The count of false positive test results (brC^{FP}) is highest when running *SQLTest* every 30 seconds. We presume that the reason for *SQLTest*^{30} having the highest brC^{FP} lies in the specific simulation configuration we choose for our experiment (see Table I).

When computing the performance measures for binary classification introduced above, we can see that the negative predictive value (npv) as well as the false omission rate (for) are perfect for all test configurations since we did not observe any false negative test result. Further, *SQLTest*^{10} performs best because it has the lowest false positive rate (fpr) and the highest true negative rate (tnr).

2) Fail-Pass-Seq-Counter (fpsC): A *failed-passed-sequence* (fps) is a sequence of test results which, after a test passed, starts with a failed test and ends with the next occurrence of a passed test. Consider, for example, scanning for vulnerabilities of the DVWA for ten times in a row. The first three times the test succeeds (p), then for four times, the test fails (f) and for the remaining three times the test passes again. The fps in this example is $fps^{10}_{DVWA} = \langle f, f, f, f, p\rangle$. Drawing on this definition,

Table II: Test results of *SQLTest* based on brC

Test results	**SQLMap**		
	10	30	60
number of tests	2248	1337	843
mean duration test (sec)	19.25	20.25	20.91
sd duration test (sec)	21.07	19.54	20.22
min duration test (sec)	6.02	6.02	6.02
max duration test (sec)	142.2	101.14	159.2
mean duration of br^{TN} (sec)	56.42	45.79	33.66
sd duration of br^{TN} (sec)	11.32	14.73	20.92
mean duration of br^{FP} (sec)	64.92	65.57	65.1
sd duration of br^{FP} (sec)	2.72	2.95	3.06
mean duration of br^{FN} (sec)	0.0	0.0	0.0
sd duration of br^{FN} (sec)	0.0	0.0	0.0
brC^{TN}	489	390	377
brC^{FP}	85	190	72
brC^{FN}	0.0	0.0	0.0
true negative rate (tnr)	85.19	67.24	83.96
negative predictive value (npv)	1.0	1.0	1.0
false positive rate (fpr)	14.80	32.76	16.04
false omission rate (for)	0.0	0.0	0.0

the $fpsC$ counts the number of occurrences of fps of a particular test.

To evaluate the performance of *SQLTest*, we check if and how any fps overlaps with simulated vulnerability of DVWA. If a fps starts after a simulated vulnerability starts and starts before the simulated vulnerability ends, then fps is considered a true negative (fps^{TN}). If a fps starts after the last simulated vulnerability ends and ends before the next simulated vulnerability starts, then this fps is considered a false negative (fps^{FN}). Lastly, a missed simulated vulnerability is considered a false positive (fps^{FP}).

Based on $fpsC^{TN}$, $fpsC^{FN}$, and $fpsC^{FP}$, we compute tnr, fpr, for, and npv. The calculation of these measures is analogous to those for brC which were presented in the previous section.

Table III shows the results for the test metric $fpsC$: The count of true negative fps decreases with increasing interval length between tests, that is, *SQLTest* detects more simulated vulnerabilities of DVWA when intervals between repeated tests are shorter. Furthermore, since *SQLTest* did not produce any false negative test results (see Section IV-B1), we did not observe any false negative fps. The count of false positive fps increases with increasing intervals between tests. Note that for *SQLTest*^{10} the sum of fps^{TN} and fps^{FP} equals 500 which is the total amount of simulated vulnerabilities of DVWA (see Section IV-A). Yet in the case of *SQLTest*^{30} and *SQLTest*^{60}, the sum of fps^{TN} and fps^{FP} only equals 498 and 401, leaving 2 and 99 simulated SQLI vulnerabilities unaccounted for, respectively. The reason for this gap is that a fps^{TN} can cover multiple successive, simulated vulnerabilities. In this case, a test correctly fails, i.e. correctly determines that the DVWA is vulnerable at some point t. The vulnerability is fixed in $t+1$ and in $t+2$ another vulnerability of DVWA is simulated which is fixed in $t+3$. Again, in $t+4$ the next vulnerability is simulated but only then the next test is executed which produces a true negative result. If the next test correctly passes, then a true negative fps is produced which covers three simulated vulnerabilities of DVWA. The probability that a fps^{TN} covers multiple simulated vulnerabilities depends on the ratio between the interval length of successive tests and duration of each simulated vulnerability plus the duration of test execution.

Based on $fpsC$, running *SQLTest* every 10 seconds performs best since it has the highest true negative rate tnr and the lowest false positive rate fpr. All test configurations of *SQLTest* perform perfect with regard to the negative predictive value npv and the false omission rate for since we did not observe any false negative fps.

Table III: Test results of *SQLTest* based on $fpsC$

Test results	**SQLMap**		
	10	30	60
$fpsC^{TN}$	489	395	288
$fpsC^{FN}$	0	0	0
$fpsC^{FP}$	11	103	113
true negative rate (tnr)	97.78	79.32	71.82
negative predictive value (npv)	1.0	1.0	1.0
false positive rate (fpr)	2.2	20.68	28.18
false omission rate (for)	0.0	0.0	0.0

3) Fail-Pass-Seq-Duration (fpsD): This test metric captures the time elapsed between the start of the first failed test, i.e. first element of a fps, and the start of the next subsequent passed test, i.e. last element of a fps.

If we observe a true negative fps, then we compute the *absolute* difference between the duration of the fps and the duration of the simulated vulnerability of the DVWA which gives us $efpsD^{TN}$. In case of a false negative fps, the entire duration of the fps is erroneous since it incorrectly indicates a duration of a simulated vulnerability ($efpsD^{FN}$). Finally, if we observe a false positive fps, i.e. the absence of a fps despite a vulnerability was simulated, the error of the missed duration equals the duration of the simulated vulnerability ($efpsD^{FP}$).

During the simulation of vulnerabilities of DVWA, we observe instances of each type of error on $fpsD$. We treat observations of each type of error on fps as separate distributions and compute standard descriptive statistics, i.e. $mean$, standard deviation sd, min and max.

Table IV shows the results for test metric $fpsD$: With increasing interval length between successive tests, the mean, standard deviation, minimum and maximum of true negative $fpsD$ increase.

The average error *SQLTest* makes on estimating the duration of a simulated vulnerability (mean $efpsD^{TN}$ (sec)) increases with increasing interval length between successive tests. As expected, running *SQLTest* every 10 seconds performs best, i.e. has the lowest mean and standard deviation $efpsD^{TN}$ (sec).

Further, the $mean$, sd, min and max of $efpsD^{FP}$ (sec) of *SQLTest*^{10}, *SQLTest*^{30}, and *SQLTest*^{60} are similar. This is expected since $efpsD^{FP}$ describes the duration of missed simulated vulnerabilities and we use the same simulation configuration for each experiment (see Section IV-A).

Lastly, we did not oberserve any false negative test result (see Section IV-B1). Therefore, all distribution parameters for $efpsD^{FN}$ are zero.

Table IV: Test results of *SQLMap* based on $fpsD$

Test results	**SQLMap**		
	10	30	60
mean $fpsD^{TN}$ (sec)	69.43	78.62	128.61
sd $fpsD^{TN}$ (sec)	9.98	16.31	87.19
min $fpsD^{TN}$ (sec)	31.06	40.03	70.03
max $fpsD^{TN}$ (sec)	110.15	215.23	893.73
mean $efpsD^{TN}$ (sec)	7.98	16.27	30.45
sd $efpsD^{TN}$ (sec)	6.36	10.64	20.52
min $efpsD^{TN}$ (sec)	0.07	0.05	0.03
max $efpsD^{TN}$ (sec)	41.17	53.13	82.87
mean $efpsD^{FN}$ (sec)	0.0	0.0	0.0
sd $efpsD^{FN}$ (sec)	0.0	0.0	0.0
min $efpsD^{FN}$ (sec)	0.0	0.0	0.0
max $efpsD^{FN}$ (sec)	0.0	0.0	0.0
mean $efpsD^{FP}$ (sec)	64.09	65.49	64.14
sd $efpsD^{FP}$ (sec)	2.43	2.98	3.05
min $efpsD^{FP}$ (sec)	60.0	60.0	60.0
max $efpsD^{FP}$ (sec)	68.0	70.0	70.0

4) Cumulative-Fail-Pass-Seq-Duration (cfpsD): This test metric builds on $fpsD^{TN}$, $fpsD^{FN}$ and $fpsD^{FP}$ by accumulating any measured duration until a particular point in time, thus providing a global value. This metric can, for example, be used to describe the total duration of the vulnerable DVWA.

To describe the overall performance of *SQLTest*, we calculate the error of the cumulative duration of true negative, false negative and false positive $fpsD$, that is, $ecfpsD^{TN}$, $ecfpsD^{FN}$ and $ecfpsD^{FP}$. While $ecfpsD^{FN}$ and $ecfpsD^{FP}$ are obtained by summing over any observed $fpsD^{FN}$ and $fpsD^{FP}$, respectively, $ecfpsD^{TN}$ is computed by summing over the durations of any fps^{TN} observed during the vulnerability simulation and comparing the result to the total duration of the vulnerability simulation.

Table V shows the results for the test metric $cfpsD$: *SQLTest*^{30} has the lowest and *SQLTest*^{60} has the highest cumulative duration of true negative $fpsD$.

Regarding the overall performance, *SQLTest*^{10} has the lowest cumulative duration of false positive $fpsD$, that is, in total, *SQLTest*^{10} missed the least duration of simulated vulnerabilities. Since we did not observe any false negative $fpsD$, the cumulative duration of false negative $fpsD$ is zero for each test configuration. Considering the overall error *SQLTest* makes when measuring the total duration of simulated vulnerabilities ($ecfpsD^{TN}$), then *SQLTest*^{10} performs slightly better than *SQLTest*^{30}. Note that $ecfpsD^{TN}$ (sec) of *SQLTest*^{30} is negative which means that running *SQLTest* every 30 seconds leads to an underestimation of the total duration of simulated vulnerabilities.

Table V: Test results for *SQLTest* based on $cfpsD$

Test results	**SQLMap**		
	10	30	60
$cfpsD^{TN}$ (sec)	34016352	31055834	37039253
$ecfpsD^{FN}$ (sec)	0	0	0
$ecfpsD^{FP}$ (sec)	705015	6745146	7248156
$ecfpsD^{TN}$ (sec)	1450118	-1490798	4564577
$ecfpsD^{TN}$ (%)	4.45	4.58	14.06

V. Future work

This section outlines next steps we are going to take to develop test-based methods to support continuous security certification of SaaS applications. We explain how each step is linked to the challenges described in Section II.

A. Identifying security properties for continuous testing

We are currently investigating common web application vulnerabilities, e.g. CSRF, SQL Injections, XSS, and how continuously testing for these vulnerabilities can support security certification of SaaS applications. In this context, we will address the challenge described in Section II-A, i.e. *how to consistently define security properties of SaaS applications and derive corresponding tests*. An important aspect of this work consists of describing how to determine if a specific security property requires continuous testing, that is, how changes of the SaaS application can be detected. In this context, we will consider the notion of *hybrid certification* [23] where tests' execution is triggered on the condition of having observed some defined monitoring events.

Further, we have implemented a lightweight framework to continuously simulate vulnerabilities of SaaS applications (*Challenge of vulnerability simulation*, see Section II-C). In this context, we will investigate how to simulate different distributions of occurrences of vulnerabilities, e.g. infrequent vulnerabilities having long durations or frequent, short-lived vulnerabilities.

We will also develop methods to describe how confident we are that test results are correct as well as stable, i.e. when repeatedly executed, a test's results remain sufficiently similar (*Challenge of Accuracy and Precision*, see Section II-B). This will also allow for validation of test results by an independent third party which can use our method to assess and approve tests before deployment (*Challenge of Verifiability*, see Section II-D).

B. Adaptive continuous testing

An open question is how to adapt to variations of application specific protocols and setting which may occur over time, i.e. how to ensure that repeated tests will execute correctly and reliably if the SaaS application's behavior changes. Naively executing non-invasive tests is prone to false positives, e.g. testing a webserver's TLS configuration may fail not because of a vulnerable configuration but because the webserver cannot be reached. Thus assumptions made about the environment of the cloud service under test, i.e. *preconditions*, need also to be tested.

Further, we are currently considering to include the notion of mutation-based (e.g. [24]) as well as generation-based fuzzing (e.g. [25]) to generate variants of a specific test. This will allow us to increase the number of tests executed for a specific security property which, in turn, translates to a higher test coverage and thus higher probability that the security property holds (*Challenge of Completeness*, see Section II-B).

C. Measuring overhead of continuous testing

Since continuous security testing will introduce some overhead on the SaaS application under test, we will provide methods to describe the additional load incurred. To that end, we are currently investigating how benchmark suites such as *Apache JMeter*[5] can be used to compute performance penalties resulting from continuous security testing.

This will permit us to reason about intervals and scope of tests, i.e. test configurations which can be applied in practice (*Challenge of measurement overhead*, see Section II-F).

D. Secure the certification system

The results of security tests can contain critical information, thus a suitable security model for the system implementing continuous security tests has to be provided. We will extend the work in [15] to identify threats to our certification system, derive security requirements, and determine suitable security mechanisms to build trustworthy certification systems (*Challenge of Overexposure*, see Section II-E).

Acknowledgment

This work was partly funded by the Federal Ministry of Education and Research of Germany, within the project *NGCert* (http://www.ngcert.eu), Grant No. 16KIS0075K, and by the EU H2020 project *EU-SEC*, Grant No. 731845.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *NIST Special Publication*, vol. 800, no. 145, p. 7, 2011.

[2] B. Grobauer, T. Walloschek, and E. Stöcker, "Understanding cloud computing vulnerabilities," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 50–57, 2011.

[5]https://jmeter.apache.org/

[3] Cloud Security Alliance (CSA), “Cloud Control Matrix: Security Controls Framework for Cloud Providers & Consumers.” https://cloudsecurityalliance.org/research/ccm/, 2013.

[4] National Institute of Standards and Technology (NIST), “Security and privacy controls for federal information systems and organizations,” *Special Publication 800-53*, 2013.

[5] M. Krotsiani, G. Spanoudakis, and K. Mahbub, “Incremental certification of cloud services,” in *7th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, pp. 72–80, IARIA, 2013.

[6] M. Krotsiani and G. Spanoudakis, “Continuous Certification of Non-repudiation in Cloud Storage Services,” in *13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 921–928, IEEE, 2014.

[7] M. Anisetti, C. A. Ardagna, E. Damiani, F. Gaudenzi, and R. Veca, “Toward Security and Performance Certification of OpenStack,” in *8th International Conference on Cloud Computing (CLOUD)*, IEEE, 2015.

[8] P. Stephanow and N. Fallenbeck, “Towards continuous certification of Infrastructure-as-a-Service using low-level metrics,” in *12th International Conference on Advanced and Trusted Computing (ATC)*, IEEE, 2015.

[9] M. Anisetti, C. Ardagna, F. Gaudenzi, and E. Damiani, “A certification framework for cloud-based services,” in *Proceedings of the 31st Annual Symposium on Applied Computing (SAC)*, pp. 440–447, ACM, 2016.

[10] P. Stephanow, G. Srivastava, and J. Schütte, “Test-based cloud service certification of opportunistic providers,” in *9th International Conference on Cloud Computing (CLOUD)*, IEEE, 2016.

[11] B. Albelooshi, K. Salah, T. Martin, and E. Damiani, “Experimental Proof: Data Remanence in Cloud VMs,” in *8th International Conference on Cloud Computing (CLOUD)*, pp. 1017–1020, IEEE, 2015.

[12] X. Wang, J. Zhang, M. Wang, L. Zu, Z. Lu, and J. Wu, “CDCAS: A Novel Cloud Data Center Security Auditing System,” in *International Conference on Services Computing (SCC)*, pp. 605–612, IEEE, 2014.

[13] Z. Birnbaum, B. Liu, A. Dolgikh, Y. Chen, and V. Skormin, “Cloud Security Auditing Based on Behavioral Modeling,” in *9th World Congress on Services (SERVICES)*, pp. 268–273, IEEE, 2013.

[14] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, “Validating cloud infrastructure changes by cloud audits,” in *8th World Congress on Services (SERVICES)*, pp. 377–384, IEEE, 2012.

[15] P. Stephanow, C. Banse, and J. Schuette, “Generating Threat Profiles for Cloud Service Certification Systems,” in *17th High Assurance Systems Engineering Symposium (HASE)*, IEEE, 2016.

[16] S. Cimato, E. Damiani, F. Zavatarelli, and R. Menicocci, “Towards the certification of cloud services,” in *9th World Congress on Services (SERVICES)*, pp. 92–97, IEEE, 2013.

[17] A. Haeberlen, “A case for the accountable cloud,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 52–57, 2010.

[18] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, “Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services.,” in *USENIX Security Symposium*, pp. 175–188, 2012.

[19] A. Li, X. Yang, S. Kandula, and M. Zhang, “CloudCmp: comparing public cloud providers,” in *Proceedings of the 10th SIGCOMM Conference on Internet measurement*, pp. 1–14, ACM, 2010.

[20] D. Catteddu, G. Hogben, *et al.*, “Cloud computing information assurance framework,” *European Network and Information Security Agency (ENISA)*, 2009.

[21] International Organization for Standardization (ISO), “ISO/IEC 27001:2013 Information technology – Security techniques – Information security management systems – Requirements.”

[22] W. G. Halfond, J. Viegas, and A. Orso, “A classification of SQL-injection attacks and countermeasures,” in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1, pp. 13–15, IEEE, 2006.

[23] S. Katopodis, G. Spanoudakis, and K. Mahbub, “Towards hybrid cloud service certification models,” in *11th International Conference on Services Computing (SCC)*, pp. 394–399, IEEE, 2014.

[24] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, “KameleonFuzz: evolutionary fuzzing for black-box XSS detection,” in *Proceedings of the 4th Conference on Data and application security and privacy (CODASPY)*, pp. 37–48, ACM, 2014.

[25] D. Yang, Y. Zhang, and Q. Liu, “Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs,” in *11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1070–1076, IEEE, 2012.