Fuzzing of Mobile Application in the Banking Domain: a Case Study

Martin A. Schneider, Marc-Florian Wendland
Fraunhofer Institute for Open Communication Systems
Berlin, Germany
martin.schneider@fokus.fraunhofer.de, marc-florian.wendland@fokus.fraunhofer.de

Abdurrahman Akin
Research and Development (R&D) Center
Kuveyt Türk Participation Bank Inc., Kocaeli, Turkey
abdurrahman.akin@kuveytturk.com.tr

Serafettin Sentürk
Cyber Security Institute
Tübitak Informatics and Information Security Research Center
Kocaeli, Turkey
serafettin.senturk@tubitak.gov.tr

Abstract—Mobile applications are today ubiquitous, and everybody uses them on a daily basis. This applies also to security-critical mobile applications such as online banking apps. In today’s architectures, these mobile applications are usually fed from the same source as mobile applications on smart phones, i.e. web services. This makes security testing of web services inevitable. Furthermore, regulation increases and requires stronger security mechanisms as with the strong customer authentication from the Revised European Payment Services Directive (PSD2). Automated security testing is a way to cope with the increasing requirements on assuring the security of such web services and their implemented security controls whilst dealing with decreasing resources for such efforts. In this paper, we present our experiences from a case study provided by Kuveyt Türk Bank performed within the ITEA-3 project TESTOMAT where we introduced automated security testing in terms of fuzzing to complement manual security testing.

Keywords—web services, security testing, automation, fuzz testing

I. INTRODUCTION

For a long while, smart phones have become widespread in our individual and business lives. The International Data Corporation (IDC) Worldwide Quarterly Mobile Phone Tracker forecasts worldwide smartphone shipments to 1,339 billion units in 2020 [1]. The fast growth of the global smartphone market in the upcoming years will be speeded up by the increasing use of smartphones both in personal and business area. People use smart phones broadly in many areas such as keeping track of their emails, following social media, making video calls, tracking their work. There are wide range of applications that can run on the smart phones and give its users many advanced features to benefit from. These mobile applications are usually referred to as apps. Mobile apps have been extensively used in security sensitive scenarios. The users heavily rely on them to handle personal or corporate data (e.g., contacts, financial data and geographic location) and use services (e.g., online banking, online shopping, messengers, like WhatsApp, Telegram, Signal to name a few). Moreover, mobile apps are playing an increasingly important role in enterprise, government, military offices, safety-critical sectors, financial and banking sectors.

Nonetheless, various security issues of mobile apps are continually being discovered and discussed, ranging from sensitive data leakage [2], [3], [4], to privilege escalation [5]. Smartphones are interesting for security attacks because they store personal and sensitive data in their storage. In particular, more and more users carry out financial transactions such as shopping from their smartphones by means of NFC payment technology and making payments by using QR codes. Especially online banking transactions through web and mobile delivery channels and financial processes by the help of ATM machines together with the mobile banking applications are gaining popularity each and every year. Since the personal data and sensitive financial data are stored and transmitted in these transactions, security and privacy protection of such data is becoming extremely important. From this perspective, the three basic security properties confidentiality, integrity and availability should be sufficiently taken into account during design and development of web and mobile applications. More specifically, confidentiality, integrity, authentication, accountability, non-repudiation and authorization play crucial roles within security of mobile systems. For this reason, security testing of such properties should be performed to assure mobile banking systems are sufficiently protected against sophisticated cyber-attacks.

This paper focusses on security testing of a mobile banking application platform especially in terms of identification and authentication perspectives since regulation increases and requires stronger security mechanisms as with the strong customer authentication from the Revised European Payment Services Directive (PSD2). Particularly, in order to avoid identity theft cyber-attacks and for ensuring the secure login and authentication mechanisms by means of two factor authentication, fuzz testing of mobile banking features will be performed, first on GUI level and secondly in the service level. Fuzzing is a black-box testing technique, which basically consists in finding defects using malformed/semi-malformed data [6]. Fuzz testing, which is a type of security testing, is chosen for the banking scenario because it is able to
find the serious security defects effectively. Fuzzing tools, also called fuzzers, are known to work very well for identifying vulnerabilities, e.g., buffer overflow, DoS (Denial of Service), SQL injection, and cross-site scripting [7].

In the context of a widely-used mobile application developed in a large financial (banking) corporation in Turkey, there was a need to experience and benefit from fuzzing to find the security vulnerabilities more efficiently. Motivated by that need, the authors and their colleagues developed and adopted automated fuzzing.

This paper organized as it follows: Section II presents the related works; Section III introduces the case study which has been conducted in a mobile banking application scenario. In Section IV, we discuss the goals and the procedure to apply automated security testing for the backend web services. Section V conveys the results of security testing on the application. We conclude with Section VI where we also present an outlook on future directions.

II. RELATED WORK

This section discusses previous works related to test automation and fuzzing that are useful to understand the context in which the case study has been placed.

A. Automating the Dynamic Test Process

Test automation is getting more relevant together with the rise of the DevOps concepts [8]. In general, test automation is defined as “the use of software to perform or support test activities” [9]. Although this definition is not restricted to test execution, the term test automation is in the industry usually understood as the automation of the test execution. Anyhow, test automation per se has a wider scope than test execution, it strives to automate various activities of a test process, though. Predestined for being carried out in an automated manner are the activities of the dynamic test process such as test design, test implementation, test execution, comparison and verdict arbitration. The industry currently employs automated test execution in particular for functional test testing. Functionality is often well understood by testers and domain experts as opposed for non-functional testing such as security or performance. In addition, the system interfaces over which functional test cases can be executed in an automated manner often already facilitates automation. This holds in particular true for graphical user interfaces (GUI). The main purpose of a GUI is to grant users access to the functionality of the underlying system. Whether this functional access comes along with high aesthetics for a good user experience does not matter for automated test execution.

ISO 29119-4 [10] has standardized the common activities a dynamic test process consists of, but it neither prescribes which of those activities ought to be automated nor how they should be automated. The International Software Testing Qualifications Board (ISTQB®), in turn, introduces a generic Test Automation Architecture (gTAA) as parts of its Test Automation Engineer (TAE) syllabus [11]. The gTAA comprises logical architectural layers and the logical interfaces among them. These layers are the Test Generation Layer, the Test Definition Layer, the Test Execution Layer and the Test Adaptation Layer as depicted in Figure 1.

Based on gTAA, concrete test automation architecture (TAA) shall be derived that supports the test automation needs for specific projects. The ISTQB does not make any assumptions or suggestions how a TAA shall be effectively specified. One possible option is to utilize the UML Testing Profile (UTP) [12] for the specification of the layers and the concretization of the interfaces and data among the different layers. After the required components, interfaces and data structures of a TAA are clear, one or many test automation solutions (TAS) might be realized according to the TAA. Whereas a TAA would not prescribe which tool or programming language to use for automating testing activities, a TAS is a collaboration of concrete tools that carry out the respective activity. The independency between a TAA and a TAS allows for reusing the very same TAA for technically different solutions by relying on a shared test automation strategy. This may increase the cohesion, comprehensiveness and sustainability of the test automation approach across multiple projects and teams.

![Figure 1: The generic Test Automation Architecture](image)

The Test Generation Layer comprises components and tools that support the design of logical test cases and test data, either manually or automatically. Tools in this layer may be as straightforward as Word and Excel (for manual test design) or as formal as Conformiq or UPPAAL for state-based automated test design.

The Test Definition Layer offers tools and components to cope with test implementation. Test implementation is about making test cases, test data and test suites actually executable. In a gTAA, the Test Definition Layer is responsible to implement test cases, provide test data, link potentially existing test libraries, creating test suites and specifying test procedures. Executable test definitions – i.e., test cases and test data implemented on the Test Definition Layer so that
they are ready for execution – must not necessarily mean that they are immediately executable. It is still possible to use a logical representation of the test cases and test data, but with an operational semantics underneath so that test execution tools are able to read and interpret the test definitions. Examples for such logical but executable test definitions are keyword test cases or test cases expressed by means of modelling languages such as UTP or TDL. Another standardized testing language that is built on the principle of logical test definitions is TTCN-3 [13].

Test Execution Layer eventually executed the test definition against the SUT. It is able to read the information from the Test Definition Layer and translate (or compile) it into a technical representation that can be used for communicating with the SUT. Furthermore, it accepts the responses from the SUT and uses it for value comparison and logging. Eventually, the Test Execution Layer is responsible for verdict arbitration and reporting.

Although the specification of test cases may be present in a logical way, the actual interaction between the TAS and the SUT has to happen on a technical level. It is the responsibility of the Test Adaptation Layer to bridge that gap. Test adapter implementations connect the TAS (or more precisely, the test execution tool that resides inside the TAS) with the SUT, it encodes and decodes the logical test definitions so that the test execution tool is able to submit test data as well as accept and interpret the response of the SUT. Information are exchanged over interfaces such as the GUI, an API, the command line or services and protocols. Test cases may demand the execution over different interfaces. Thus, different adapters implementation might be required for the Test Adaptation Layer. Furthermore, the Test Adaptation Layer provide means to monitor the SUT to facilitate test verdict arbitration, an important aspect for fuzzing that will be discussed in more details in the subsequent section. The Test Adaptation Layer is usually implemented manually or gets support from tools for specific testing types or interfaces, e.g. GUI testing, and specific types of applications, e.g., web services based on SOAP and WSDL.

The implementation and introduction of TAS usually happens bottom-up to ensure immediate executability of the automated test cases. Thus, it is most important that the Test Execution Layer and Test Adaptation Layer ensure compatibility of and connectivity between the TAS with the SUT. In practice, the utilized test execution tool on the Test Execution Layer has a huge impact on the test definitions as well. For example, a TTCN-3 test execution tool requires TTCN-3-based test definitions, JUnit requires Java-based test definitions and so forth. Thus, the degree of the independency of the higher Test Definition Layer is usually not as high as the gTAA suggests.

As already said, the gTAA neither prescribes nor assumes how it is further developed into a TAA and TAS eventually. A TAS might be as integrated as a single holistic tool that offers all or some capabilities described by the gTAA, or as loosely coupled as a collaboration of distinct tools or services. Similarly, the gTAA is not restricted to specific testing type. It may be utilized for functional testing, security testing, fault tolerance testing or performance testing.

B. Fuzzing

Fuzzing is a technique that stimulates the SUT with invalid or unexpected inputs aiming at discovering weaknesses that result from missing or faulty input validation. Originally, fuzzing was invented by Barton Miller from the University of Wisconsin coincidentally while using UNIX command line tools over a modem connection. Due to a noisy dial-up line transmitted characters were randomly modified and led to crashes of many of the used tools (cf. [14], foreword by Barton Miller). During further investigation, they developed a random character generator that fed UNIX command line tools over a pipe. Although this approach is extremely simple, about one third of the tools on seven versions of UNIX crashed [15].

Since invalid inputs are sent to the SUT via its provided interface, fuzzing is a negative testing approach and is also referred to as interface robustness testing [16]. The stressed interfaces are also called attack surfaces [16] because their code is tested for robustness and may allow a successful attack.

Over the years, fuzzing has been evolved along several dimensions. These are for example the employed protocol knowledge [17], the usage of source code, i.e., white-box fuzzing [18], the usage of fuzzing heuristics, that capture knowledge used to mutate inputs or generate input from mutated specifications. Furthermore fuzzing has been evolved from the syntactic level to the semantic, i.e. fuzzing on a binary level, syntactic level, that is fuzzing with respect to input data, and the semantic level that comprises dependencies between fields of the same inputs, data flows, states, and messages flows.

Basically, fuzz testing tries to break the interface of the test item by stimulating it with invalid inputs. Hence, fuzzing tests the input validation and input sanitization mechanisms of the interface and probes it for possibly security-relevant faults. In contrast to vulnerability scanners, fuzzing aims at discovering 0-day-vulnerabilities that are previously unknown. These vulnerabilities may appear when invalid inputs are accepted instead of being rejected, e.g., due to the deviations in the implementation with respect to the specification. Fuzzing aims at discovery of vulnerability by so-called semi-valid input data. Semi-valid input data is invalid only in small portions and thus, have the chance to pass the test item’s parsers and may cause problems in the subsequent processing components of the SUT [19], [16].

While simple approaches generate data purely randomly, model-based fuzzers were considered state of the art until a few years ago. These generate, for example with the help of interface specifications, semi-valid input data, which can reveal safety-relevant errors in the input validation and the underlying application logic by subtle deviations from valid data. Complementary to this approach is the fuzzing of behavior that can detect errors in the processing of calls and call sequences [20], [21].

With the success of American Fuzzy Lop (AFL) [22] and LibFuzzer [23], fuzzing with random mutation of existing input data has reappeared. These tools evaluate feedback at runtime for the generation of further test cases in order to increase code coverage. The use of search-based methods is common, especially genetic and evolutionary algorithms,
which use mutation, recombination and selection to replicate natural evolution for finding candidate solutions. Individuals of a population are developed over several generations towards a solution, in the case of fuzzing are these test cases or test data: while recombination models the mixture of two tests, mutation modifies individual test cases at individual positions. A fitness function determines the quality of single individuals and thus supports their selection. The fitness function measures the code coverage achieved by a single test case or the coverage of previously unachieved code components by the previously generated and executed test cases.

Since fuzzing is a technique to generate test inputs, it provides no test oracle for assessing the SUT to decide if a vulnerability has been triggered. Therefore, fuzzing has to be used together with monitoring techniques to observe the SUT’s behavior for anomalies. The test oracle is the necessary source of information to determine the test result during or after test case execution [24]. A test oracle can decide whether the SUT reveals erroneous behavior. It provides the expected results or the test verdict (pass, fail). In the best case a specification can serve as a test oracle. The test oracle can be implemented manually by skilled personnel who know the correct system behavior, or in an automated form. Especially for highly automated tests such as fuzzing, an automated test oracle is indispensable to be able to perform it efficiently. Hence, the expected non-functional system behavior must be available in such a formalized form that it is possible to determine whether weaknesses exist, fast and with sufficient precision. The test oracle is therefore just as decisive for the quality of the tests as the abstract test cases generated in the test design and the associated test data. The two main challenges for test oracles are completeness and correctness. Completeness means that all errors that have occurred are reliably detected, thus minimizing the false negative rate. Correctness means that a behavior classified as an error is actually an error and thus the false positive rate [25] is minimized.

Barr et al. [25] distinguish test oracles into specified test oracles (e.g. in model-based testing), derived test oracles (e.g. from a specification or from an existing system such as a previous version or alternative implementation), and implicit test oracles that can detect obvious defects such as crashes. For security testing, specified test oracles are not practical because of the scope of the anomalous behavior to be specified [26]. Implicit test oracles for the detection of crashes are therefore the simplest and most common form of potentially vulnerability in the SUT resulting from the execution of fuzz tests [27]. These can be used as connectivity checks after executing one or more test cases [28]. Valid Case Instrumentation goes a step further and executes functional test cases according to a fuzz test case to detect abnormal behavior of the test object [28].

C. Fuzzino

Fuzzino\(^1\) is a fuzzing library that has been specifically developed to extend existing test automation solutions with fuzzing capabilities by providing smart fuzzing heuristics from well-established open source fuzzing tools. It provides generational as well as mutational fuzzing heuristics for primitive and complex data types. It provides a set of predefined heuristics and an automated mechanism for selecting heuristics based on provided type information. Thus, it enables automated test data generation for low-level test cases and test scripts that has been developed for functional testing. Section IV provides more information how Fuzzino is working and its usage.

III. Case Study

A. Description of the Mobile Banking Application and its MFA

Kuveyt Türk Participation Bank Inc. is a private financial (banking) institution in Turkey which started its operations in Turkish finance market in 1989. It has over 430 branch locations and delivers a wide array of financial products and services to customers. Those products and services are internet banking, mobile banking, ATM and XTM channels, gold and silver buying and selling to name a few. As of year-end 2019, the bank employed about 5,900 personnel [29].

The system under test (SUT) in this paper is the most used mobile application used by bank customers. It is called Mobil Şube (in Turkish), meaning Mobile Branch, and provides user interfaces in Turkish, English, and Arabic. The application gets the most traffic among the other alternative delivery channels and is a significant product of the bank. The app provides typical features for personal banking, e.g., transfer funds, invoice payments, and additional features such as currency exchange, gold buying and selling, viewing financial reports, and credit card payments. The app is available for both Android and iOS platforms. Android and iOS applications have the same functionalities. Both of these applications are communicating with backend API services and with the core banking business database. In terms of codebase size, the service layer is about 100 KLOC, iOS version of the app is about 308 KLOC and the Android version is about 949 KLOC. The app has more than 80 GUI screens (pages) in total, one of which is shown in Figure 2, which is the app’s login page. As of this writing, the app has more than one million active users, and a monthly traffic of about 36 million logins.

Particularly, authentication in the mobile banking application is established through two-factor authentication (2FA). The app provides different approaches to achieve 2FA. The first approach uses SMS-based authentication in which the system sends the user a one-time password (OTP) as a second factor. The second approach makes online verification in order to complete the 2FA without sending any OTP to the user. Instead, it sends the user a push notification that the user confirms to check if the user is the one who claims to make the login into the system. The second solution has been developed by the bank itself to fight against the SMS interception attacks [30].

\(^1\) https://github.com/fraunhoferfokus/Fuzzino/
This study has been conducted fuzz testing on the GUI level as well as on the web service layer to perform security test of the login process and above-mentioned authentication methods. To achieve this, some fields in the login phase of the mobile banking app are tested with fuzzing heuristics. Detailed information on this are given in Section IV.

B. Description of Backend Webservices

The backend services for the mobile banking app are based on Windows Communication Foundation (WCF) and provides HTTP endpoints to iOS and Android mobile branch applications. It provides dedicated endpoints and service operations for the different function of the mobile application. We will focus here on the login function of the authentication service and the corresponding login endpoint consists of 41 different service operations. Some of these operations are login, logout, password control, secret word setting, activating the user, SMS OTP. Other operations are collected under another endpoint. The second endpoint has around 500 service operations, like money transfer, invoice payment, foreign exchange, precious metal buying and selling, credit card operations, Account activities, QR code withdrawals, transaction listing.

C. Current State of Security Testing at Kuveyt Türk Bank

Kuveyt Türk Participation Bank has implemented manual tests in the SDLC process as well as automated regression tests, performance tests, and security tests. The bank uses professional tools for requirement management, test management, and defect management. There are automated regression tests on GUI and service level. The GUI level automated regression tests are running for iOS, Android and web browser. Kuveyt Türk Participation Bank is using Open Source test frameworks, e.g. the Appium test framework. The Selenium test framework is used to write the automated regression tests in Java for the web browser GUI applications. Kuveyt Türk Participation Bank has a regression test suite with a device farm. Getting fast feedback about change impact is the bank’s main purpose. Continuously improving to the next level of regression testing is quite significant. Regression tests are executed automatically in a given period with high coverage and fast feedback is given about the change impact [31].

At the service level, automated regression testing is done for REST and SOAP services. For the automated regression tests, Kuveyt Türk Participation Bank is using Citrus test framework and the test code is written in Java. Citrus is an Open Source Test Framework which can be used for integration tests and by which those tests can easily be added to continuous integration. Security testing is done periodically with use of advanced tools and processes, with manual and automated security tests. Comprehensive scans are performed to reveal local and remote vulnerabilities and security issues. Security testing is also applied to detected vulnerabilities related to fraud.

In service level testing of the mobile banking application, JSON request and response bodies are used. As the subject of this case study is fuzz testing of authentication features, JSON requests for login and authentication services have been selected. Those requests contain login properties like customer identification number, password, some unique properties of the device and some clarifying information about the type of multi-factor authentication. With SMS-based authentication, OTP codes sent to the user become critical. That is why fuzzing of OTP code plays a crucial role to find the security vulnerabilities. In case of authentication method with online verification, some unique information about the device becomes essential and will be subject to fuzzing. Figure 4 shows a sample of JSON request body.

![Sample JSON Request Body](image)

```java
1: StringSpecification stringSpecification = RequestFactory.INSTANCE.createStringSpecification();
2: stringSpecification.setEncoding(StringEncoding.UTF8);
3: stringSpecification.setType(StringType.STRING);
4: stringSpecification.setIgnoreLengths(true);
5: stringGenerator = new AllBadStringsGenerator(stringSpecification, SEED);
```

Figure 3: Initialization of Fuzzino
IV. INTRODUCTION OF AUTOMATED SECURITY TESTING FOR THE BACKEND WEB SERVICES

The test data generator Fuzzino developed by Fraunhofer FOKUS has been chosen for introducing automated security testing. Fuzzino is not a fully-fledged test generator and execution tool as most fuzzing tools are. Fuzzino’s test generation facilities can be integrated into the existing test infrastructures and thus, reusing the test infrastructure as well as existing test cases and thus accelerates the integration of basic fuzzing capabilities into already existing test environments and processes.

The integration of Fuzzino works by adding the dependency to the project. To initialize Fuzzino, five lines of code as depicted in Figure 3 need to be added as part of the setup procedure of every test suite. Since Fuzzino provide smart fuzzing heuristics, it is able to evaluate type information for the generation process.

Figure 3 depicts the whole process of initialization of Fuzzino for a simple string. Line 1 create the specification for the string that is encoded in UFT-8 (line 2), line 3 provides the information it is a plain, unstructured string. Fuzzino also offers a large set of predefined types based on string that can be specified instead, such as e-mail-address and IP address. In fact, Fuzzino provides some assumptions for type information that can be changed by modifying the type specification. Hence, line 3 is not necessary and only here for illustration purposes since Fuzzino’s assumption is that a string is a plain string. To use one of the predefined string primitives, StringType.STRING can be replaced, e.g., by StringType.PATH.

Based on the provided type information, Fuzzino does both, automatically select appropriate fuzzing heuristics for data generation and adapt the selected fuzzing heuristics to include the type information in the data generation process. For example, the heuristic can take into account the length of the string or its encoding (e.g., ASCII or UFT-8) to generate those values that comply to this specification or explicitly violate this.

We introduced fuzzing by augmenting the existing functional test suite with Fuzzino. Starting on the requirements of the mobile banking app, we selected the authentication process of the mobile banking app against the backend web services – on GUI frontend level as well as on HTTP request level – as most promising target for fuzzing. Therefore, we selected several fields of the corresponding request, created the Fuzzino heuristic initialization and finally augmented the test cases with general fuzzing heuristics.

V. RESULTS

A. Improvement of Security Testing

Kuveyt Türk Participation Bank and Fraunhofer FOKUS have a collaboration within the ITEA-3 TESTOMAT (The Next Level of Test Automation) project. Fraunhofer FOKUS has developed an open source fuzzing tool named Fuzzino and Kuveyt Türk wanted to experience the tool in its own environment to identify potential weaknesses of the system.

First, we applied Fuzzino on the GUI frontend (cf. Section III.A) of the application to find out if vulnerabilities can be identified that might be exploited by some end users. After installing and integrating Fuzzino, several fields, e.g., customer identification number, password, and OTP were used to send the fuzzed values, as part of the HTTP request resulting from the interaction of an end user with the GUI frontend of the banking app. As expected, the GUI frontend blocked most of the fuzzed values using input validation and sanitization techniques. For example, the customer number field on the GUI is only accepting digits with a maximal length of 11 characters. It was impossible to bypass these limitations using fuzzing heuristics that generate longer values and demonstrated that the web frontend of the bank web application does not inhibit unskilled adversaries to reveal any vulnerable behavior.

In the next step, we applied fuzzing to the web service backend to find out if a motivated, skilled adversary might take advantage of vulnerabilities that are not accessible through the web frontend. To do so, we implemented Fuzzino on the service level and experimented the fuzzed values on some properties in the JSON request body (cf. Section III.C). First, we integrated Fuzzino to our existing test framework Citrus. However, problems occurred when receiving the unexpected responses such as HTML error messages sent from the web application firewall (WAF). Citrus stopped the test execution with a fail when the response was not a valid HTML conforming the standards. Hence, we switched to Apache HttpClient for test execution.

Fuzzino has been applied to customer identification number, password and two unique device properties and two unique application properties. It showed us possible improvements on different parts in the service like checking the length of the provided properties, checking the names of the provided properties as well as their values.

In addition to applying fuzzing to existing JSON property values, some property names have been renamed. For instance, we changed the customer identification number field name to customer and renamed the ‘password’ field name to a ‘pass’. Also, some new properties have been added, provided by Fuzzino. Additionally, several delimiters instead of a single one have been used between properties.

B. Test Results

Fuzzino generated more than 4,200 fuzzed values using a single heuristic for bad strings. These values have been used for 6 different properties of the login request for 2 different authentication types one of which is SMS based and the other one is online verification method In total, more than 50,000 test cases have been generated using Fuzzino. The results

<table>
<thead>
<tr>
<th>Property</th>
<th>#JSON responses</th>
<th>%</th>
<th>#Requests rejected</th>
<th>%</th>
<th>#Request errors</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceOS</td>
<td>2579</td>
<td>61%</td>
<td>1129</td>
<td>27%</td>
<td>496</td>
<td>12%</td>
</tr>
<tr>
<td>DeviceNumber</td>
<td>2035</td>
<td>48%</td>
<td>1827</td>
<td>44%</td>
<td>342</td>
<td>8%</td>
</tr>
<tr>
<td>CustomerNumber</td>
<td>2579</td>
<td>61%</td>
<td>1129</td>
<td>27%</td>
<td>496</td>
<td>12%</td>
</tr>
<tr>
<td>Password</td>
<td>2579</td>
<td>61%</td>
<td>1129</td>
<td>27%</td>
<td>496</td>
<td>12%</td>
</tr>
<tr>
<td>ExternalName</td>
<td>2579</td>
<td>61%</td>
<td>1129</td>
<td>27%</td>
<td>496</td>
<td>12%</td>
</tr>
<tr>
<td>ExternalPassword</td>
<td>2579</td>
<td>61%</td>
<td>1129</td>
<td>27%</td>
<td>496</td>
<td>12%</td>
</tr>
</tbody>
</table>
(shown in Table 1) showed us that for the property DeviceNumber: 48% of the fuzzed values were responded by the service. 44% of the fuzzed values were rejected by the WAF. From the total fuzzed values, 8% got a request error response in HTML. The results for the other properties were different: 61% were responded by the service, 27% was rejected by WAF and 12% caused a request error. The other requests were handled by the service without any warning, but the results showed us that improvements were needed. Based on the fuzzing heuristic “AllBadStrings”, long strings, newly added properties, non-Latin characters have been generated as an input to the system. These values did not cause any crash or could any vulnerable. Moreover, digit-only fields like customer identification number has also accepted letters. If letters are submitted to the service where digits are expected, the system raised a wrong user credentials error. The data for the non-Latin characters were stored as question marks in the database and the very long strings were stored in a shortened form in the database according to the maximum length of the database cell, hinting at an effective input validation mechanism.

Summarizing the results, we identified improvements for the web services that result from potential weaknesses in the input validation of input type, lengths and value handling of the service.

VI. CONCLUSIONS AND OUTLOOK

In this paper, we presented the experiences from introducing automated security testing to a case study, a mobile banking app and its backend, from Kuveyt Türk Bank. In contrast to security testing that was previously done mainly manual, we introduced automated security testing based on existing functional test cases using Fuzzino. The execution of tests revealed no directly exploitable vulnerabilities. However, some anomalous behavior has been detected that hint at opportunities where further hardening could be done.

Further improvements could be achieved by increasing the number of targets where automated fuzzing is applied to, the number of heuristics used for data generation and providing more type information to increase the efficiency of fuzzing by exploiting this information for tailored fuzz test data. Last but not least, behavioral fuzzing could be applied to find weaknesses in the multi-factor authentication procedures.

This study was useful in terms of security fuzz testing of authentication functions of the mobile banking application to make a kind of security check and to see any availability of vulnerable points in the system. Particularly, the heuristic “AllBadStrings” of Fuzzino has been used to expose the security breaches in one of the methods of authentication service. For further work, the other methods and other properties can be fuzzed such as generated OTP code, session keys, online verification method properties and many more. The other future work possibility could be to use other fuzzing heuristics in Fuzzino like SQL injection, buffer overflow, DoS (Denial of Service), and cross-site scripting vulnerabilities. These heuristics can be definitely applied to internet banking application as well. Finally, a comparison can be made between the results of security testing of internet banking and mobile banking application.

Acknowledgements: This work has been partially funded by the ITEA-3 project TESTOMAT under grant no. 16032. TESTOMAT: the Next Level of Test Automation, project website www.testomatproject.eu.

REFERENCES


[31] V. Garousi, A. Akın, and Ş. Şentürk, "Transitioning from manual to automated software regression testing: Experience from the banking domain" presented at the 25.th APSEC, 2018, pp. 591-597