SORA - Finding Workflow Violation Attacks in REST APIs

Rodrigo Rato
rodrigorato@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

Abstract

The importance of web applications is unquestionable, but as their popularity grew, so did the complexity of the attacks they are vulnerable to. To find these vulnerabilities, there are currently two schools of thought, either one employs specialists in the area, or penetration testers, or one makes use of automatic tools. This work aims to help this second paradigm grow in importance, as it brings many advantages over the first, such as a more systematic and consistent analysis. We will study vulnerabilities in the web paradigm, and in particular Workflow Violation Attacks, which are a class of vulnerabilities that is quite important but has been left out throughout time. We provide an in-depth study about these vulnerabilities and techniques used to find them, and then design and implement a solution, capable of finding said vulnerabilities in a setting where the vulnerable system is a server exposing a REST API. We also devise a set of test applications to be released for the community to test their tools against vulnerable programs.

Keywords: Web Security, REST API Security, Workflow Violation Attacks, Model-Based Security Testing, Black-box Testing and Continuous Security Testing

1. Introduction

Nowadays, the importance of the internet is incomparable, most services we might think of are available through it, which is very convenient. However, convenience comes at a price. The more the internet is important, the more the services are victims to increasingly complex attacks. This need for fixing security problems quickly has lead to security being a bolted-on concern instead of a main one, which is a very bad practice.

When we study these vulnerabilities, in the web paradigm, we can divide them into two different categories [4]. Either they pertain to the injection category (run-of-the-mill SQL injections and countless others types), or they pertain to the state violation category. We argue that the latter category is at least as important as the first, however, it is not nearly as popular [11]. Due to this, in this work we study state violation vulnerabilities, concerned with an application’s state, where this application is an HTTP server exposing a REST API. We will present a solution for finding these problems, while also raising awareness to them.

Consider a web application for a shop in this model, a HTTP server exposing a REST API, while disregarding authentication and authorization mechanisms for simplicity. Let this application consist of multiple endpoints, all of them are regular operations for a shop, where each of them takes the application to the next state, as seen in the automaton in figure 1.

If we look at the transition marked with 1 in this very same automaton, it is skipping a step in the application’s defined workflow. We call this a state bypass, which is an instance of a workflow violation.

In most applications, these are very subtle instances of vulnerabilities, which might lead to big problems. However, as we mentioned previously, there is not a lot of work in finding these problems.

This reason, coupled with the desire to have a tool that performs reproducible security analysis, unlike what is produced by a pentester, is what steered our work in this direction.

2. Goals and Contributions

The goal of our work is to develop a solution capable of finding workflow violation vulnerabilities, where the system under test (SUT) is a HTTP server ex-
posing a REST API.

Our solution is a black-box testing tool, which by the nature of this testing method, and the virtue of our implementation, is compatible with every system, disregarding the particularities of the technologies used within that system.

Our tool is comprised of two different parts, a modeling language called SORA-ASL, which has been built as an addition to the OpenAPI specification language, and our tool SORA.

In order to evaluate our solution, we have also devised a set of test applications. We will released this set of applications for the community to test their systems, as to the best of our knowledge, such a set of test applications does not exist.

Throughout all of our work, the main contributions are to:

1. Provide an in-depth study about Workflow Violation vulnerabilities;
2. Propose and design a solution for this problem, backed by related work in this area and comprised of:
   i. A system/API modeling language, SORA-ASL;
   ii. A tool that implements the concepts in SORA-ASL, called SORA;
3. Provide a set of test applications for Workflow Violations;
4. Define a set of evaluation metrics for both SORA and SORA-ASL and evaluate our tools accordingly;

3. Background
3.1. WPSE

WPSE by Calzavara et al. [3] is a tool that runs in browsers to enforce compliance with confidentiality, integrity and intended protocol flow goals.

To enforce these properties, models (or descriptions) of protocol runs are fed into the tool, which makes use of monitoring techniques to verify if protocol runs that are happening in the browser follow these models.

We share the same goal of verifying compliance with the intended protocol flow as the authors of this tool, as this problem is similar to the example of a workflow violation attack that we introduced earlier, where an application goes through its possible states in a different order than what is intended.

To solve this problem, the tool uses descriptions of the protocols’ intended flows, defined by means of HTTP exchanges observed by the browser, with a well defined syntactic structure and order of messages. The tool uses this descriptions to internalize the protocol as a finite state machine (FSM), similar to the example we have shown in our introductory section. This tool then monitors the browser’s execution of these protocols, to enforce the intended flow.

This is quite similar to the problem we are trying to solve, but this approach has a couple of problems. In order to monitor an application like so, we need to know the intended protocol flows. Writing such a model is very time consuming and error prone. There is also the problem associated with a possible performance loss when monitoring the application. We will take the idea of representing the system as a FSM, which seems promising, but will try to improve on the monitoring aspect, by having our system perform black-box testing on the SUT.

3.2. Swaddler

Swaddler by Cova et al. is an anomaly-based tool to detect workflow violation attacks against web applications [4]. It analyzes the internal state of a web application, learning how to match the applications’ execution points to its state. This knowledge is then applied to identify attacks that bring an application to an anomalous state. This tool also makes use of monitoring techniques, but unlike the first it has the same goal as our tool.

As the authors point out, workflow violation attacks are very difficult to detect without any knowledge about the target application: it would be difficult to analyze an application if we had access to was its traffic, like HTTP requests and responses. To solve this problem, the authors take to a runtime monitoring based approach, during both of the tool’s life cycle phases: the learning phase and the detection phase.

The learning phase is where Swaddler, by monitoring the application with code instrumentation while attack-free executions are being run, effectively learns. In this phase, Swaddler generates mappings of variables to their values.

The detection phase is very similar to the learning phase. With the data saved from the previous phase, the tool is now able to judge if the application will transition to an anomalous state after the event is applied. If the analyzer does find that the application will be in an incorrect state, it halts the application.

This approach makes use of both monitoring techniques, is very application-driven (white-box), and has a large performance setback. These are immediately some drawbacks we can refer to. However, in spite of these problems, this work is quite relevant, as Cova et al. were the first to distinguish the different classes of vulnerabilities we talked about in our introduction.
4. Solution
Our solution targets HTTP server applications that expose a REST API. We follow a black-box approach, thus eliminating the monitoring overhead that other approaches had and allowing our system to interoperate with many different technologies.

SUTs to be tested with our tools must be at least documented in OpenAPI [10]. This is not much to ask, as OpenAPI is already the de facto standard when it comes to representing and documenting REST APIs and many systems use it. By choosing OpenAPI as the base for our solution we are also allowing our system to be used with legacy implementations.

We represent the SUT as a FSM, where we will fetch information from the SUT’s OpenAPI specification to try and define its FSM. The input alphabet for the FSM is easy to define, the set of endpoints and parameter types defined in OpenAPI for the SUT are enough. However, we are still missing a lot of data to define the remaining terms of the FSM.

For the remaining elements, we see two different approaches to obtain them, either have the programmer write a model of the SUT or derive a model of the SUT from attack-free executions. We do not present these techniques as mutually exclusive, rather as complementary to each other.

To use the first technique, we will have to define a language fitting for this purpose, that shall include the notions of state for the SUT, as well as which input triggers the system to transition between states. This technique proves very effective if we want to find vulnerabilities during the development stage of the SUT, as opposed to finding problems in runtime like what the monitoring approaches do.

When it comes to the second approach, it has been successful for other works [4, 6]. However such works also helped us understand that identifying and applying attack-free executions, something that such systems depended on, to the SUT is not always trivial. An option that we can think of would be to use the SUT’s test suite, should it have one, as attack free execution, but we would still have to filter out tests that exercise bad features on the SUT. We believe that this approach alone does not help us meet our goals, as it has many constraints, so we opted to follow the first approach.

In any case, once we have a representation of the SUT in the form of a finite state machine, we will test the system in order to check if there is a transition that is possible in its implementation, and impossible in the finite state machine that represents it. Such a transition indicates that the system is vulnerable to a Workflow Violation Attack.

A technique to find workflow bypasses, for example, would be to traverse through the SUT’s states in a correct way (i.e. as defined in its FSM), and at each state try to branch out to find, possible unwanted, states. As values for each parameter in this state traversal, and in particular session variables we can use previously seen values, or values directly provided by the programmer in our tool’s configuration.

In order to accomplish a level of analysis like this, we will have to rely the concepts we are talking about. The OpenAPI specification shall be an input for our tool. As for the remaining information we need to represent the SUT’s FSM, we will define an API specification language on top of OpenAPI where this is possible, which we will call SORA-ASL. This will make it easy and practical for programmers to use our tools, as they should already be familiarized with OpenAPI well enough to understand our extensions to that language. Using the concepts available in SORA-ASL, our tool SORA will be able to perform analysis/testing on the SUT.

4.1. SORA API Specification Language (SORA-ASL)
The first component of our implementation is the API specification language (or system modeling language) we have created and named the SORA API Specification Language, or SORA-ASL. In this language, it is possible to encode test cases to find different vulnerabilities, however before we show you how to encode this information to guarantee such properties, we will explain the language in itself.

As the starting point for SORA-ASL, we chose the OpenAPI specification. As we said previously, many systems are already modeled (and documented) using this language, and we aim to use this to our advantage, bridging the gap between what programmers already document/model about their systems and what they need to specify in order to use our tool. Thus, an OpenAPI specification of any system is inherently a SORA-ASL specification. However, SORA-ASL can be used to model more aspects of the SUT than what was previously possible using just OpenAPI.

We will not formally introduce OpenAPI in this document. Should the reader want to fully understand this specification language, please refer to the OpenAPI specification [10]. We will also abstract from the fine details of SORA-ASL, however they are available in the full dissertation document that accompanies this document.

SORA-ASL introduces the ability to model concepts pertaining to the state of the SUT to OpenAPI, so that we can define all the elements needed to represent the system as a finite state machine.

To design these new concepts we are introducing, we had to keep in mind that this was all meant to
be written by a programmer, and that many other tools that forced programmers to formally define their systems had limited success because of this [2, 1]. We took this in consideration when defining SORA-ASL and avoided the need to have programmers formally define the rest of the FSM that represents the SUT. Essentially, when our goal is to detect transitions that are possible in a SUT’s state diagram of a simplified UML diagram of SORA’s internal components. All of these components are flexible enough to be extended with any features one might desire, as well as has a list of assertions, which are Python expressions that operate on the values in the request and response objects for the API call. Using the plays we can build tests, responsible for interacting with the SUT by means of the API calls performed by the plays, and asserting if the SUT is in a correct state. The tests also have a requisite plays field, so that a programmer can define when should that test be applied. A playbook is simply a collection of tests to be evaluated. The final extension is to the paths definition of the existing OpenAPI paths. In SORA-ASL, the programmer can now specify what return codes are yielded by the SUT when a given return code is returned by the SUT. This information, when paired with the tests that have requisite plays, will let us apply fuzzing techniques to the tests and generate new test cases.

4.2. The SORA tool
In this section we will briefly reiterate the goals for our tool, SORA, as well as present the main decisions we took while implementing it. We will start by describing the architecture used to build the tool, while also describing the main logic behind each of the tool’s components. Once all the components and their logic is presented, we will describe how SORA proceeds to test a SUT given its specification in SORA-ASL, as well as how does SORA automatically generate more tests for the SUT.

4.2.1 Architecture
SORA’s design tries to capture all the concepts available to a programmer in SORA-ASL.

The SORA tool

<table>
<thead>
<tr>
<th>The SORA tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigManager</td>
</tr>
<tr>
<td>Api</td>
</tr>
<tr>
<td>DefinitionManager</td>
</tr>
<tr>
<td>InstanceManager</td>
</tr>
<tr>
<td>Playbook</td>
</tr>
<tr>
<td>Test</td>
</tr>
<tr>
<td>Play</td>
</tr>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>Endpoint</td>
</tr>
<tr>
<td>Assertion</td>
</tr>
</tbody>
</table>

In figure 2 we present a UML diagram of a simplified view of SORA’s internal components. All of these components are flexible enough to be extended with any features one might desire, as well as...
to be reutilized in other projects. We also took advantage of existing tools designed to work with OpenAPI specifications, namely the Swagger toolkit, to avoid reimplementing their functionality, however this is not represented in this diagram.

In spite of not being represented in this diagram, SORA leverages an external tool, swagger-codegen from the Swagger toolkit. This tool generates server and client libraries for a system given its OpenAPI specification, which we leverage to verify if the underlying OpenAPI specification for the SORA-ASL document of the SUT is a valid one, and to generate a client library to interact with the SUT. SORA then extends this library to implement features needed to test the SUT.

The SORA component, shown in the UML diagram above, and the ConfigManager component are responsible for housekeeping tasks, such as interacting with the user and parsing the SORA-ASL specification (and interacting with swagger-codegen). When these tasks are done and every concept in the SORA-ASL specification has been internalized, we can start doing more interesting tasks, such as testing the SUT.

If you pay close attention to the UML diagram above, it bears (intentionally) a lot of resemblances with the concepts in SORA-ASL. So in SORA, there are also the concepts of assertion in a play, plays in a test and tests in a playbook. Having all this data, the Playbook component is capable of running all the tests it knows, as defined by the programmer using SORA, and evaluate all the assertions, thus checking which tests pass and fail.

SORA runs these tests according to the SUT’s playbook, exactly as specified in SORA-ASL. However, we implemented an algorithm that attempts to use fuzzing techniques upon the SUT’s original playbook, so that more test cases are generated.

4.2.2 Playbooks and Automatic Test Generation

We have described how the playbook a user writes is actually used and tested. This section, on the other hand, will focus on explaining how the original playbook (and other data) the user has written can be transformed in order to generate more test cases.

If you recall correctly, we have previously mentioned that tests in any given playbook have requisite plays. These are regular plays that serve the purpose of checking if that test can be executed in the current context, or state. Having these requisite plays would not be useful in a normal execution of the playbook’s tests in the order they are defined, as a programmer would write these tests relying on the SUT’s state after the execution of the previous test. However, this is not the rationale behind our tests.

A test should be prepared and have the necessary requisite plays in order to check if the current state for the SUT is suitable for the test. In other words, it is the responsibility of the test to assess if it is relevant (or even possible) for it to be used at any point in time.

Having the tests and playbooks written in this way allows us to change the state of the SUT before running any test once again, thus creating new test cases from having the existing ones run in a new context. In order to achieve this, we will have indeed to change the SUT’s state before running a test. The way we do this is by using some information the user has given us. The endpoints defined in the paths section of the SORA-ASL document allow the programmer to write when clauses that are assertions that specify when does the system yield a given response code. Such clauses can be interpreted as conditions for the edge cases of the system, meaning that the values on the edge of these conditions can be used to achieve our goal of triggering the system to be in different states. We refer to these values, the assertions’ edge values, as their features. A feature is an association between a variable and its possible edge values.

Following this, it is important for us to extract all the features from each assertion. The full algorithm to achieve this is, once again, depicted in the dissertation document that accompanies this article.

Simply put, our algorithm tries to extract all possible features from an assertion, using the edge cases for every possible value.

If an assertion has the association of a constant to a variable, assertion_\text{a} \leftarrow (x \leftarrow 3)$ for example, we will extract the edge cases for that variable, $\text{features} (\text{assertion}_\text{a}) = \{< x, \{2, 3, 4\} \}$ so that we can associate those values with that variable. Using this technique we can extract the values for every feature that has a direct comparison or attribution of a constant to a variable. On the other hand, if a feature is composed of multiple other features we will simply apply this procedure recursively until we obtain all the values for all features. Should multiple variables be associated with different, multiple sets of values, we should simply apply the union operation to all the sets.

Having all these features, we can now interact with the SUT and use these values to trigger it to change between states, as we have a direct association between the variables we want to change and the values we want to use. However, we cannot simply perform requests to the server using these values as parameters, as that would not make any sense and would probably not be valid for any of the paths defined in the SUT’s SORA-ASL document. To make sure we use the features’ values
in a relevant way, we employ a technique similar to fuzzing on the instances and parameters defined for each endpoint in the paths section of the SORA-ASL document.

Our technique relies on finding suitable instances, from the set of all instances known to SORA, where we can mangle those instances using the values from known features. After we have obtained all the features and their values, as well as all possible instances we can use for every possible endpoint, we will re-instantiate all the instances with all the possible feature values (a cartesian product of all possible values), and call every endpoint with a valid configuration achieved from doing this. For each of these calls, we run the user’s playbook once again, as the system is in a new state, and should any test fail, we have found a vulnerability.

An algorithm for finding authorization problems, along the lines of the ones we just explained, is also featured in SORA. This algorithm was developed as a necessary step to perform our analysis, however it has proved useful to find authorization problems, so we enhanced it in order to apply it to the analysis of the SUT. This algorithm, as well as all the previous ones, are fully described in the full dissertation document.

5. Results

To the best of our knowledge, due to the lack of works in this area, no other system that aims to find Workflow Violation Attacks in REST APIs exists. The systems we studied target web applications as a whole, so we cannot really use the same applications others used for evaluation purposes, not even for comparison, as a web application does not necessarily expose a REST API.

Due to this, we have developed a set of nineteen test applications, these applications are vulnerable to different kinds of attacks, while some have no intentional vulnerability at all. From our point of view this set of applications is a major contribution when it comes to finding this type of vulnerabilities, as we have not been able to find a set of applications for testing. We will use the applications we have developed to evaluate both the system we have developed, SORA, as well as our API Specification Language, SORA-ASL.

To evaluate the success of SORA-ASL we will have to evaluate the added complexity of using our language, as opposed to just using OpenAPI or no API modelling language at all. Our goal is for our language to be simple to use, as not to drive programmers away, which is quite subjective. Due to this, after we have presented our test applications, we will explain how our language can be used to express properties as to find the vulnerabilities introduced in said applications, while hopefully showing that it is quite simple to use.

As for our tool, SORA, we can assess it using the following metrics:

(i) Number of tests generated per definition in SORA-ASL;

(ii) Average number of problems found in a generated test case versus problems found while executing the user’s tests;

(iii) Time it takes to run our generated test suite versus the time it takes to run the user’s test suite;

As a way to evaluate both our tool SORA and SORA-ASL, we will explain how can we encode the different vulnerabilities found in our test applications in SORA-ASL, and how will SORA find these vulnerabilities. But firstly, we will explain the test applications we have developed.

5.1. Test Applications

As we have mentioned previously, due to the lack of applications to test our system with, we have developed a set of fully functioning web servers, exposing REST APIs. For all of these applications, which are described in detail in the dissertation document, we followed a development process consisting in the following phases:

(1) An OpenAPI specification for the application was written;

(2) A server application was generated using swagger-codegen, targetting the Java language and using the Spring Boot framework;

(3) The full functionality of the application was implemented in the server application generated previously;

(4) A vulnerability, or set of vulnerabilities, was purposely added to the application;

(5) The original OpenAPI specification was augmented to be a full SORA-ASL specification, with definitions for tests that should catch the introduced vulnerabilities;

All of the applications expose a very similar API, as all of them were incrementally built from a previous application. All of them have a valid OpenAPI specification document, which was then expanded by defining tests that should find those application’s vulnerabilities, thus making the document a fully functional SORA-ASL specification.

The set of test applications consists of 19 applications in total, where 11 of them are simple applications with different authentication mechanisms (HTTP basic auth, bearer token and others). The remaining 8 applications are the interesting ones, being vulnerable to workflow violations, and allowing for proper testing.
5.2. Encoding Different Properties and Vulnerabilities in SORA-ASL

Having described all of the test applications we have developed, we will now exemplify how can one express properties in SORA-ASL in order to find the vulnerabilities described for each of our test applications.

Workflow violations are the type of problems we are more interested in finding. All of our test applications designed to test these problems have a defined workflow, and some of them are vulnerable to some kind of attack that lets a wrongdoer take advantage of that vulnerability. In this section we will cover the basic properties, related to states and state bypasses, that one can express in SORA-ASL, and how will that contribute to the discovery of these problems, be it directly by running tests or indirectly by being found by generated test cases in SORA.

The first type of vulnerability we will cover here are the invalid state transitions. The expected flow for the application in question is that users can add pets, but a user cannot add a pet with the same identifier as another user has added previously, however the application lets users add pets with equal identifiers. This example is quite simple and silly, however, the idea used to catch this problem is the same we would use to catch more complex ones. When writing the SORA-ASL definition for the application, we should write the properties we are checking the SUT for as tests, as we show in listing 1, which is a set of tests to find the problem we just mentioned. However, SORA is also capable of generating new tests, which will change the SUT’s state, so it would not be necessary to write the full test to catch this problem, as by just having a test that adds and checks for a pet, SORA would be capable of generating new test cases that find this problem.

In the following code listing the full tests’ specifications has been omitted for brevity, but the essential parts of the tests, such as some assertions has been kept. What we are trying to do with this particular test is encode the flows that must not be present in the SUT as SORA-ASL tests. This test is called check_if_replaced, and it performs a very simple operation, it first adds pet pet1_a as user u1, then it adds a pet pet1_b as user u2, but where pet1_a.id == pet1_b.id. The plays that are doing this are the ones that are omitted, as the play named retrieve_good_pet1 is where the interesting part of the analysis takes place.

This play is fetching the original pet as user u1 and the assertions on this play are trying to verify that the owner for this pet is still the original one. Should any of those assertions fail, the play fails, as well as the test.

```json
- name: "check_if_replaced"
  plays:
    - name: "add_pet1_u1"
    - name: "add_bad_pet1_u2"
    # - content omitted -
    - name: "retrieve_good_pet1"
  endpoint:
    path: "/pet/{petId}"
    method: "get"
  auth:
    key: "header_u1"
  parameters:
    - name: "petId"
      in: "path"
      value: 1
  response:
    code: 200
  assertions:
    - is instance (resp, Pet)
    - eq (resp["id"], 1)
    - eq (resp["owner"], "u1")
```

Listing 1: Finding known invalid state transitions in SORA-ASL

This technique of encoding the flow that must not be present in SORA-ASL is the same we should use to find the problems in most other applications. However, it would also be valid to not use this technique, as we could just encode the valid flows and SORA would be able to fuzz them to reach the states we have reached with our tests. On the other hand, if we are trying to find known problems, we can encode that information in order to give SORA more data to work with. The great advantage of our tool’s automatic testing is the number of test cases generated per definition in SORA-ASL, as by just adding an instance, endpoint, or other definition, the number of test cases grows a lot to feature that definition. This helps us find more and more problems, but if the problem is known, as it is in our test applications, it is always best to encode that information so that SORA will be able to reach even more states, which might potentially be unwanted.

5.3. Evaluating SORA

In order to evaluate our tool, we will be using the metrics we stated at the beginning of this section. The results of evaluating our tool are also useful to evaluate the algorithms we use and have developed. However, for each of our metrics, we will give
practical examples related to the test applications we have developed, with actual numbers extracted from evaluating our tool with said applications.

The first metric we will explore is the number of tests generated per definition in SORA-ASL. We have measured the number of endpoints, instances and keys defined in the SORA-ASL specification for each of our test applications, as well as the total number of tests generated for each of these applications. Our data for this metric is presented in table 1. We can clearly see that the number of generated test cases grows quite nicely when new definitions are added to the SORA-ASL specification, however there is something quite odd in these numbers. If our test generation follows our algorithm, how come that the number of test cases generated is not simply endpoints \times instances \times keys? This is because our algorithm also takes in account which keys work for each endpoint, thus allowing SORA to apply a deduplication operation to delete redundant test cases. Unlike what one might think, it is actually advantageous for us to generate such a large number of tests with small changes to the input, as the problem here would be to generate new test cases. We don’t consider time to be a problem, as modern techniques to distribute the work-load help us diminish the time it takes to run these tests quite a lot. Our tests can be ran in parallel, for example, using different instances of the SUT.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Endpoints</th>
<th>Instances</th>
<th>Keys</th>
<th>Generated Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth-bypass (all)</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>state-bypass (1 and 2)</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>state-bypass (3 to 6)</td>
<td>5</td>
<td>21</td>
<td>6</td>
<td>439</td>
</tr>
<tr>
<td>state-bypass (7 and 8)</td>
<td>6</td>
<td>24</td>
<td>7</td>
<td>515</td>
</tr>
</tbody>
</table>

Table 1: Test cases automatically generated for our test applications

As for the second metric we mentioned, which is the number of problems found in automatic test cases versus problems found in the users’ test cases, we have compiled a table presenting this data for our applications, similar to the previous table. This data can be seen in table 2.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Problems found in Users’ tests</th>
<th>Problems found in automatic test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>state-bypass-1</td>
<td>0</td>
<td>0 (previously 3)</td>
</tr>
<tr>
<td>state-bypass-2</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>state-bypass-3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>state-bypass-4,5,6</td>
<td>1</td>
<td>432</td>
</tr>
<tr>
<td>state-bypass-7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>state-bypass-8</td>
<td>1</td>
<td>507</td>
</tr>
</tbody>
</table>

Table 2: Problems found in the users’ test cases versus problems found by SORA’s automatic tests

With this data we can see that, in spite of the users’ tests finding little to no problems, our automatic test cases catch quite a lot of problems. Part of this is due to the written tests only targeting one vulnerability, or the applications only being vulnerable to that particular vulnerability, however, using the data provided by the user (all the definitions in SORA-ASL), SORA is capable of generating a large number of test suites to catch a problem. There is also a small curiosity presented in this data, the application state-bypass-1 was not intended to be vulnerable, but had an authorization problem during development that was caught by our tool. This unintended vulnerability was promptly patched, so that the test applications respect their specification.

The last metric we will present is the time it takes to run the users’ test suites versus our generated test suites. Potentially, having this metric as a way to compare our system with others (had they existed) or even to test our applications, would not mean a lot, as the time it takes to make an API call can vary from SUT to SUT, and even depend on the network that they are one. However, for our testing, all our SUTs are developed using the same technologies, are all in the same local network, and tested using the same system specifications (hardware and software). Both the server running the SUTs, and the client performing the requests using SORA are the same machine running GNU/Linux 5.3.7, with an Intel Core i7-6700HQ CPU (@ 2.60 GHz) and 8 GB of DDR4-2133 MHz RAM.

The values presented in table 3 reflect this metric. This table is also useful should we want to take a look at how long does it take our system to perform a test in average. Looking at this data, it is also clear that the automatic tests take more time to run when the applications are not vulnerable, as more and more test cases are generated and they must be fully executed to find if the application is vulnerable or not. On the other hand when an application is vulnerable, the vulnerability gets immediately reported, thus ending that particular test’s execution, and given that SORA generates quite a lot of tests for any given vulnerability (see table 2), this is reflected in the execution times for a test.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>User Tests (seconds)</th>
<th>Automatic Tests (seconds)</th>
<th>Generated Test Cases</th>
<th>Vulnerable (yes or no)</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth-bypass</td>
<td>∼0.28</td>
<td>∼0.71</td>
<td>33</td>
<td>Yes</td>
</tr>
<tr>
<td>state-bypass-1</td>
<td>0.35</td>
<td>2.76</td>
<td>33</td>
<td>No</td>
</tr>
<tr>
<td>state-bypass-2</td>
<td>0.35</td>
<td>2.50</td>
<td>439</td>
<td>Yes</td>
</tr>
<tr>
<td>state-bypass-3</td>
<td>0.61</td>
<td>99.94</td>
<td>439</td>
<td>No</td>
</tr>
<tr>
<td>state-bypass-4</td>
<td>0.67</td>
<td>61.99</td>
<td>439</td>
<td>Yes</td>
</tr>
<tr>
<td>state-bypass-5</td>
<td>0.48</td>
<td>62.60</td>
<td>439</td>
<td>Yes</td>
</tr>
<tr>
<td>state-bypass-6</td>
<td>0.81</td>
<td>62.71</td>
<td>515</td>
<td>Yes</td>
</tr>
<tr>
<td>state-bypass-7</td>
<td>0.81</td>
<td>115.37</td>
<td>515</td>
<td>No</td>
</tr>
<tr>
<td>state-bypass-8</td>
<td>0.78</td>
<td>103.58</td>
<td>515</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3: Execution times for user tests and automatic tests
6. Conclusions
We have introduced the current paradigm for finding security vulnerabilities and confirmed that, despite its importance and the advantages it brings, methods to find vulnerabilities automatically, based on formal models, are not very popular nowadays.

We have also studied the web paradigm, where we divided vulnerabilities into two different classes, where the first are Input Validation Attacks and the second are Workflow Violation Attacks. The first class has been studied exhaustively, and automated analysis methods to find these problems have made it into different tools such as Burp Suite [14] and OWASP ZAP [12]. The latter class, however, has had limited studies done. Automated methods to find this latter class of problems have been created, but to the best of our knowledge, there exists no system that targets REST APIs, they all target web applications and the web platform in general.

Having studied all of this, we proposed and implemented a new approach to find Workflow Violation Attacks on systems that expose a REST API. Our solution works in a black-box setting, so that it integrates well with the development process of the target system, avoids both the overhead and complications brought by dynamic analysis and monitoring and is language agnostic, problems which we have also witnessed in other works we studied.

Our tool is comprised of two different parts, a modeling language called SORA-ASL, which is based on the OpenAPI specification language, and a tool that implements the concepts in SORA-ASL to verify the specification and fuzz it to generate new test cases, called SORA.

Finally, we evaluated our solution using relevant metrics that we discussed.

References


