ISSN: 2454-9940

INTERNATIONAL JOURNAL OF APPLIED SCIENCE ENGINEERING AND MANAGEMENT

E-Mail :
editor.ijasem@gmail.com
editor@ijasem.org

www.ijasem.org

INTERNATIONAL JOURNAL OF APPLIED SCIENCE ENGINEERING AND MANAGEMENT

ISSN 2454-9940

www.ijasem.org

Vol 19, Issue 2, 2025

Assessing The Safety Of REST Apis Used By Cloud Services

[1]Mrs. G. Haritha Rani, [2]Tadepalli Pavan Kumar,

[1]Assistant Professor, Department of MCA, Rajamahendri Institute of Engineering & Technology. Bhoopalapatnam, Near Pidimgoyyi,Rajahmundry,E.G.Dist.A.P. 533107.

[2]Student, Department of MCA, Rajamahendri Institute of Engineering & Technology Bhoopalapatnam, Near Pidimgoyyi,Rajahmundry,E.G.Dist.A.P. 533107.

Abstract

The REST API is the programming language of choice for accessing most contemporary online and cloud applications. This article delves into the topic of service compromise and how an attacker may take advantage of security holes in a service's REST API. In order to capture the desired qualities of REST APIs and services, we provide four security principles. Afterwards, we demonstrate how to include active property checks into a stateful REST API fuzzer, allowing it to automatically test and identify rule violations. We go over several efficient and modular ways to create such checks. We address the security implications of the new issues discovered in various Azure and Office 365 cloud services that have been put to production using these checks. These issues have been resolved. Polishchuk, Marina Microsoft Research will extensively test the API's underlying cloud service in an effort to identify service failures that will be reported as "500 Internal Server Errors" by a test client. Its scope is limited to the identification of unhandled exceptions, but it appears promising and reports numerous new issues detected. This article presents four security guidelines for REST APIs and services, which aim to encapsulate their best features. • The rule of use after free. When you remove a resource, it should no longer be available. Generating tests; Ensuring security; Using cloud and web services; REST APIs

INTRODUCTION

The popularity of cloud computing is skyrocketing. Cloud platform providers such as Amazon Web Services [2] and Microsoft Azure [13] have deployed thousands of new cloud services in the past few years. Their customers are "digitally transforming" their businesses by modernizing their processes and collecting and analyzing all kinds of new data. Today, REST APIs are the primary means by which cloud services are accessible programmatically [9]. REST APIs provide a unified method to create, monitor, manage, and remove cloud resources. They are built on top of the ubiquitous HTTP/S protocol. Using an interface-description language like Swagger (now called OpenAPI), developers of cloud services may describe their REST APIs and provide example client code [25]. Using the REST API, a cloud service may be accessed according to the details laid forth in a Swagger specification. This includes the types of queries that the service can process, the possible answers, and the format of those responses. To what extent are all those APIs secure? As of right now, there is no clear answer to this issue. There is a lack of mature tools that can automatically verify the security and reliability of cloud services using their REST APIs. To detect flaws, certain tools for testing REST APIs collect live API traffic, then process, fuzz, and replay it [4], [21], [6], [26], [3]. In order to evaluate services that are deployed behind REST APIs more thoroughly, stateful REST API fuzzing [5] was recently suggested. This method, when used to a REST API specified using Swagger, produces series of requests rather than individual ones. Microsoft Research Resource-leak rule was the primary location of this author's work. If a resource creation fails, it shouldn't be available and shouldn't "leak" into the backend service state in any way. Rule based on resource hierarchy. No other parent resource should be able to access a child resource's parent resource. The regulation pertaining to user-namespaces. You can't have resources from one user namespace available to resources from another. As we'll see in the section below, cloud resources can be compromised in an elevation-of-privilege attack, information can be stolen from other users in an information disclosure attack, or the backend service can be rendered inoperable due to a denial-of-service attack if these rules are violated. We demonstrate the extension of a stateful REST API fuzzer to test and identify such rule violations. We establish an active property checker for every rule that (a) finds rule breaches and (b) creates new API calls to test them. Basically, each checker is designed to actively try breaking its own rule, in addition to actively checking

for any rule violation. We go over some modular implementation details for these checks to make sure they don't conflict with one another. We also go over how to quickly build each checker by removing likely-redundant tests wherever feasible, because each checker adds additional tests to an already-large state space exploration. Beyond the "500 Internal Server Errors" that baseline stateful REST API fuzzing may discover, these checks are designed to uncover security rule violations. We discovered additional issues in many production Azure and Office 365 cloud services by using these checks. By identifying additional kinds of errors at a little extra testing cost, security checkers boost the usefulness of REST API fuzzing. Here are some important points that this study brings up:

We provide guidelines that characterize REST API security features. • In order to test and identify rule infractions, we create and deploy active checkers. • We provide comprehensive experimental findings that assess the efficacy and performance of these active checkers on three live cloud services. We examine the security implications of the new issues that were detected in various operational Azure and Office 365 cloud services using these checkers. How the remainder of the article is structured is as follows. We review the history of stateful REST API fuzzing in Section II. We provide active checkers to test and identify breaches of the criteria that we establish in Section III, which encapsulate desired aspects of secure REST APIs. Section IV details the outcomes of our experiments with active checkers running on real-world cloud services. We address the security implications of newly discovered flaws by these checkers in Section V. We wrap up the paper in Section VII after discussing relevant work in Section VI. II. Fusing Stateful Rest APIs Before providing security property checks that may be applied as extensions of this basic method, we review the notion of stateful REST API fuzzing [5] in this section. Section III follows. We think that REST APIs make cloud services accessible. Requests are messages sent by a client software to a service, while replies are messages received back. Protocols like HTTP/S are used to transmit such messages. A unique HTTP status code, ranging from 2xx to 5xx, is assigned to each response. One example of a specification language for REST APIs is Swagger [25], which is also called OpenAPI. What kinds of queries can a service process, what kinds of answers may be expected, and what formats those responses should take are all detailed in a Swagger specification, which is part of the REST API documentation. For the purposes of this article, a REST API is defined as a relatively small collection of queries. The request body, resource path, authentication token, and request type are the four components that make up each request tuple. These five values—PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), and PATCH (update)—are the RESTful request types that may be used. A cloud resource and its parent hierarchy may be identified by its resource path, which is a string. The regular expression (/resourceType/resourceName/)+ is usually used to match non-empty sequences of cloud resources, where resourceType is the kind of resource and resourceName is the particular name of that type. The request usually attempts to create, access, or delete the particular resource mentioned in the path that is last. Additional parameters and their values, whether mandatory or optional, may be included in the request body b to ensure the proper execution of the request. As an example, the following is a multi-line request to acquire the attributes of a single Azure DNS zone [14]: access token for user authentication GET

```
https://management.azure.com/
subscriptions/{subscriptionId}/
resourceGroups/{resourceGroupName}/
providers/Microsoft.Network/
dnsZones/{zoneName}
?api-version=2018-03-01 { }
```

The request is a GET request with an empty body at the end, and it needs three resource names—a subscriptionID, a resourceGroupName, and a zoneName—in its path. The PUT and POST methods of the REST API are used to create new resources, while the DELETE method is used to delete existing ones. To generate a new resource of type T, a producer must be a request whose execution does just that. An identifier, or "id," is a representation of a freshly formed resource. One may hear the term "dynamic object" used to describe resources because of the way they are produced on the fly. A consumer for the resource type T is a request that includes a resource name of type T in either its route or content. The resource name of type T, commonly called the dynamic object type, will be used sometimes. This GET request uses three resources—subscriptions, resourceGroups, and dnsZones—in the Azure DNS zone example but doesn't create any new resources. Within the request bodies or resource routes of individual requests, users have the option to define a tiny limited range of values that should be randomly selected; these are termed fuzzable values. In the body of a request, a user may, for example, indicate that a particular integer number may be either 0, 10,

1000000, or -10. The term for this collection of values is a fuzzing dictionary. A rendering of a request that contains fuzzable values indicates that each fuzzable value has been mapped to a single concrete value chosen from its fuzzing dictionary. Therefore, there are nk alternative renderings for a request with n fuzzable values, each of which may take k possible values. If the matching request, when executed, delivers a proper response (specified in the following paragraph), then the rendering is considered legitimate. The onus for determining which values to fuzz and which fuzzing dictionaries to use is on the user. A directed graph is used to describe the state space of a service, where nodes stand for service states and edges connect them. The execution of a single request r from a given state s of the service results in a successor state s, represented as s r → s. Any answer other than a 2xx indicates that the request r is invalid, a 3xx or 4xx indicates that the request is legitimate, and a 5xx response indicates that the request is bugged. It is possible to explore the service's state space from a starting point when no resources are available by sending a series of queries. When this kind of investigation seeks to uncover service states, it called being stateful.

It must be queried repeatedly in order to be reached: In order to execute more requests and attain deeper service states, resources may be utilized in later requests in the same sequence that were produced by earlier requests in the series. Several search techniques, such as a systematic breadth-first search or a random search, may be used to explore state spaces [5]. Given the unbounded nature of request sequence length, the potentially unlimited sets of renderings, and the blackbox nature of the service under test, state spaces may be enormous—if not infinite. Luckily, intriguing problems may be discovered by only partially exploring the state space. For the sake of this discussion, an error is considered to have occurred when a request sequence results in an HTTP status code of 500. Instead of taking the chance of a live event with unknown effects, it is advisable to correct these issues that cause "500 Internal Server Errors" and other unhandled exceptions. These exceptions are caused by unusual input request sequences. We will sometimes talk about test cases, which are executions of request sequences, and tests, which are executions of individual requests, in the following. Additionally, we will refer to the generic state-space exploration algorithm discussed in this part as the primary mechanism that drives stateful REST API verification. Section III: Rest API Security Checks Here we outline and create active security rule checkers for REST APIs. To begin, four guidelines for the security of REST APIs are introduced in Section III-A. To test and identify security rule breaches, we detail how to create active checkers in Section III-B. There is a singular emphasis on a certain kind of security rule violation by each active checker. In Section III-C, we go over the several ways in which each checker may be integrated with the others and with the primary driver of stateful REST API fuzzing in a modular fashion. We provide a novel search technique for scalable property checker test creation in Section III-D. To prevent the user from receiving several reports of the same problem, we detail how to bundle together checker violations in Section III-E. Rule No. A. – Security In order to capture the desired qualities of REST APIs and services, we provide four security principles. We provide an example for each rule and talk about the security implications of it. All four guidelines are based on actual issues with previously released cloud services that were discovered via manual penetration testing or by analyzing the causes of events that were apparent to customers. Later in Section V, we will provide examples of new, previously undiscovered problems that we discovered as rule violations in production Azure and Office 365 services that were already deployed. The law of use following free consumption. When you remove a resource, it should no longer be available. This means that all further operations (such as reads, updates, or deletes) on the same resource must fail after a successful DELETE action.

For instance, after deleting the account associated with identifier user-id1 via a DELETE request to URI /users/user-id1, all further attempts to utilize user-id1 must fail and produce a "404 Not Found" HTTP status code. When an API may still access a removed resource, it is a use-after-free violation. Never again shall this occur. This is an obvious flaw that might compromise the service's backend and allow users to evade their resource limitations. A regulation about the loss of resources. If a resource creation failed, it shouldn't be available and shouldn't "leak" any related resources in the backend service state. What this means is that each subsequent action on a resource must likewise fail with a 4xx response if the execution of a PUT or POST request to create that resource fails (for whatever reason). Additionally, the user should not see any noticeable side effects in the backend service state as a result of successfully creating that resource type. To illustrate, a resource that was unable to be established cannot be used to meet the user's service quotas, and the user must be allowed to reuse the name of the resource. For instance, in order to generate the URI /users/user-id1 with a faulty PUT request, it is necessary to get a 4xx

answer. This URI must also be inaccessible for any future requests to read, edit, or delete. When an uncreated resource "leaks" some influence on the backend service state, even if it wasn't properly generated, a resource-leak violation has occurred. For example, a remove request cannot remove the resource even if it is listed in a later GET request, or efforts to recreate the resource result in "409 Conflict" answers. The capacity of that resource type (e.g., if resource quota limitations are reached and no new resources can be added) and the performance of the service (e.g., owing to unnecessary huge database tables) might be negatively affected by such breaches, hence they must never happen. Resource-hierarchy rule. No other parent resource should be able to access a child resource's parent resource. What this means is that when a new parent resource is used in place of an existing one, the child resource must not be accessible in any way—read, update, or delete—even though it was successfully created from the parent resource and identified as such in the service resource paths. Using the resource-hierarchy rule as an example, if you create users user-id1 and user-id2 and assign report report-id1 to user user-id1, then add report report-id1 to user user-id2, and then issue POST requests to URIs /users/user-id1/reports/report-id1 to create users user-id1 and user-id2, respectively, and then add report report-id1 to user user-id1, then subsequent requests to URI /users/user-id2/reports/report-id1 must fail. To violate the resource-hierarchy, one must ensure that no sub-resource that was formed from one parent resource may be accessed from another parent resource without a parent-child relationship. In cases where such infractions are feasible, an adversary may be able to provide an illicit parent object identity.

```
Inputs: seq, global_cache, reqCollection
# Retrieve the object types consumed by the last request and
# locally store the most recent object id of the last object type
n = seq.length
req_obj_types = CONSUMES(seq[n])
# Only the id of the last object is kept, since this is the
# object actually deleted.
target_obj_type = req_obj_types[−1]
target_obj_id = global_cache[target_obj_type]
# Use the latest value of the deleted object and execute
# any request that type−checks.
for req in reqCollection:
    # Only consider requests that typecheck.
    if target_obj_type not in CONSUMES(req)
        continue
    # Restore id of deleted object.
    global_cache[target_obj_type] = target_obj_id
    # Execute request on deleted object.
    EXECUTE(req)
    assert ''HTTP status code is 4xx''
    if mode != 'exhaustive':
            break
```

for example, user-id3, and then take control of an illegal child object, such report-id1, by reading or writing to it. There should never be any instances of resource hierarchy violations since they are obvious defects that might cause harm. Policy regarding user-namespaces. You can't have resources from one user namespace available to resources from another. With respect to REST APIs, we take into account user namespaces that are established by the user token that is used to engage with the API (for instance, OAUTH token-based authentication [18]). To illustrate the point, after creating the URI /users/user-id1 with the token token-of-user-id1, it is imperative that the resource user-id1 cannot be accessed with the token token-of-user-id2 of any other user. When a resource that was generated in one user's namespace may be accessed from another user's namespace, it is called a user namespace violation. An attacker might potentially get unauthorized access to another user's resources by executing REST API calls with an unauthorized authentication token. This could happen if such a violation were to occur. Part B: Active Verifiers For the regulations outlined in Section III-A, we have active checkers in place. In stateful REST API fuzzing, an active checker keeps an eye on the primary driver's exploration of state space and proposes additional tests to make sure certain rules aren't broken. As a result, a proactive checker expands the search area by running additional tests that aim to break certain rules. A passive checker, on the other hand, does not run any additional checks but instead watches the search that the primary driver is doing. Using a modular design grounded on two ideas, we develop dynamic checkers: 1) The state space exploration of a stateful REST API is unaffected by checkers, which are separate from the core driver of the fuzzing process. Second, each checker works independently of the others; they create tests by looking at the primary driver's requests, but not the ones that other checks have run.

Every time the main driver finishes running a new test case, we run all the checkers to enforce the first principle. To ensure that the checkers do not interfere with one another while working on separate test cases, we prioritize their application order according to their semantics, thereby enforcing the second principle (more on this later in this section). Following this, we outline the specifics of each checker's implementation and provide improvements to curb the growth of state spaces. Utilization verification tool. In Figure 1, the use-after free rule checker's implementation is presented using syntax similar to Python. Following the execution of a DELETE request by the main driver (refer to Figure 4), the algorithm is invoked and receives three inputs:

a sequence of requests, or seq of requests, representing the most recent test case executed by the main driver; the global cache of dynamic objects, or global_cache, for all available API requests; and the most recent object types and ids for all dynamic objects, or reqCollection, for all dynamic objects. To begin, on line 5, we acquire a list of all the kinds of dynamic objects that were used by the previous request. Then, we create a temporary variable called target_obj_id to keep the id of the last object type. We choose the final type in req_object_types as the actual type of the deleted object, even if the last request may be consuming several object types. (A remove request on the URI /users/userId1/reports/reportId1 would remove just reports, even though it would combine two object types—users and reports.) Following this basic setup, starting on line 12, the for-loop iteratively processes all requests in reqCollection, excluding those that do not consume the target object type (line 14). The method EXECUTE (line 19) uses the recovered target object id from the global cache of dynamic objects (line 17) to execute request req once it finds a request, req, that consumes the target object type. Because the EXECUTE method utilizes object ids accessible in global cache to execute requests, the target object id is restored in the global cache many times. In the event that any of these requests are granted, a use-after-free violation will be triggered on line 20 (see to Section III-A).

Contribution beyond stateful REST API fuzzing. The checkers enhance the primary driver of baseline stateful REST API fuzzing in two ways: first, by running more tests, they increase the size of the state space; and second, by looking for replies other than 5xx and potentially flagging unusual 2xx responses as faults that violate the rules. So, it's evident that they improve the main driver's bug-finding skills; using them together, the main driver can discover flaws that it couldn't uncover on its own. Active property checking vs passive monitoring. In our previous discussion, we established that the checkers would augment the primary driver's search area with extra test cases designed to trigger and identify certain rule violations. It is very unlikely that rule breaches could be detected by passive runtime monitoring of these rules in conjunction with the primary driver, meaning that those additional tests would not be executed. Specifically, passive monitoring alone is unlikely to discover use-after-free and resource-leak rule violations. This is because the primary driver's default state space exploration probably won't try to re-use deleted resources or resources after a failure, respectively. Because the basic main driver doesn't try to replace object IDs or authentication tokens, passive monitoring would also miss resource hierarchy and user-namespace rule breaches. That is to say, in comparison to non-checker tests, the extra test cases produced by the checkers are not superfluous; rather, they are essential for discovering rule violations. The checkers work in tandem with one another. Our four defined checkers are mutually supportive; that is, by definition, no two checks may provide identical new tests due to the fact that their preconditions are mutually incompatible. To begin, request sequences that conclude with a DELETE request activate just one checker: the use-after free checker. Additionally, in the event that the most recent request has returned an incorrect HTTP status code, the resource-leak checker is the only checker that is enabled. Last but not least, request sequences that do not conclude with a DELETE request have the resource-ownership checker enabled as the sole other checker. Fourthly, the user-namespace checker obviously adds another orthogonal dimension to the state space as it conducted tests using an attacker token that was distinct from the authentication token used by the main driver and all other checks. D. Checkers Search Methods When fuzzing stateful REST APIs, the breadth-first search (BFS) is the primary search approach used to generate tests. The search space is defined by all conceivable request sequences. When it comes to grammar, this search technique covers all the bases. It covers every potential request rendering and every possible request sequence up to a certain length. The search, however, does not scale well with increasing sequence length as BFS usually explores a huge search space. Hence, BFS-Fast was implemented as an optimization. Unlike BFS, BFS-Fast only adds each request to a single request sequence of length n, rather than to all of them, whenever the search depth grows to a new number n+1 [5]. While BFS Fast does cover all potential ways a request might be rendered, it does not investigate all request sequences of a certain duration. While BFS-Fast outperforms BFS in terms of scalability, it does this by investigating a fraction of all potential request sequences.

The amount of infractions that the security checkers are able to actively verify is, however, limited by this. Our new search approach, BFS-Cheap, aims to overcome this constraint. Following the opposite trade-off of BFS-Fast, BFS-Cheap investigates all potential request sequences for a given sequence length, but does not cover all possible renderings. For example, here's how BFS-Cheap works with an n-sequence set (seqSet) and a collection of requests (reqCollection): To process each sequence in seqSet,

ISSN 2454-9940

www.ijasem.org

Vol 19, Issue 2, 2025

add each element in reqCollection to the end of seq, run the new sequence taking into account all potential renderings of req, and add no more than one valid and one incorrect sequence rendering to seqSet. The use-after-free, resource-hierarchy, and user-namespace checks adhere to valid renderings, but the resource-leak checker adheres to faulty renderings. For an experimental evaluation, see Section IV-B; BFS-Cheap is therefore a compromise between BFS and BFS-Fast. In order to prevent a huge seqSet, it thoroughly investigates all potential request sequences up to a certain length (similar to BFS) and adds no more than two additional renderings to each sequence (similar to BFS-Fast). As the length of a sequence rises, seqSet may still include a manageable number of sequences thanks to two additional renderings for each sequence that is being actively checked against all the security requirements outlined in Section III-A. Keep in mind that the "cheap" suffix is derived from the fact that BFS-Cheap is a less expensive variant of BFS that adds no more than one valid rendering to the BFS "frontier" setSeq for every new sequence. As a result, less resources are generated compared to when all possible interpretations of each request sequence are investigated, as in BFS. Think of a request specification that uses an enum type to describe 10 distinct versions of the same resource type. Once BFS-Cheap has successfully developed a resource of a single flavor, it will cease to manufacture any more of that flavor. On the other hand, BFS and BFS-Fast will generate 10 identical resources but with ten distinct flavors. D. Bug Collecting To set the stage for talking about actual infractions discovered by active checkers, we first explain the bucketization technique that is used to classify comparable infractions. "Bugs" are rule infractions while discussing active checkers. The request sequence that caused each issue to occur is linked with it. In light of this characteristic, we construct per-checker bug buckets according to this procedure: Compute all non-empty request sequence suffixes that trigger if a new bug is detected.

API	Total Req.	Search Strategy	Max Len.	Tests	Main	Checkers	Checker Stats			
							Use-Aft-Free	Leak	Hierarchy	NameSpace
Azure A	13	BFS	3	3255	48.1%	51.9%	11.5%	1.5%	0.1%	38.8%
		BFS-Cheap	4	4050	55.0%	45.0%	10.0%	0.8%	2.4%	31.8%
		BFS-Fast	9	4347	59.2%	40.8%	15.5%	0.2%	0.1%	25.1%
Azure B	19	BFS	5	7721	46.4%	53.6%	3.6%	0.4%	0.2%	49.4%
		BFS-Cheap	5	7979	46.2%	53.8%	3.5%	0.4%	0.2%	49.7%
		BFS-Fast	40	17416	65.3%	34.7%	0.3%	0.0%	0.1%	34.3%
O-365 C	18	BFS	3	11693	89.4%	10.6%	0.0%	1.0%	0.1%	9.5%
		BFS-Cheap	4	10982	95.9%	4.1%	0.0%	0.0%	0.1%	4.0%
		BFS-Fast	33	18120	66.9%	33.1%	0.0%	0.0%	0.1%	33.0%

sequence added once for 500 bugs.

(Tests), the proportion of tests created by the main driver (Main), all four checkers combined (Checkers), and each search method after one hour of searching in isolation. Beginning with the lowest one, the total number of requests in each API is shown in the second column. Insert the news sequence into an existing bug bucket if it has a suffix. Create a separate bug bucket for the news series if necessary. There is no need to keep the two systems apart when dealing with bug buckets; they are identical to the one used in stateful RESTAPI fuzzing[5]. This is due to the fact that failing conditions are specified independently for each rule. Due to checker complementarity, only one checker for a given sequence length will ever trigger a problem; nonetheless, the main driver and checkers are both capable of triggering the "500 Internal Server Error" issue. Only the bug bucket of the primary driver or error checker that caused it will have the news

EXPERIMENTALEVALUATION

Results of studies with three production cloud services are reported in this section. Our experimental setup and the services we provide are detailed in Section IV-A. Following that, in Section IV-B, we evaluate and contrast the three research methods outlined in Section III-D. Section IV-C details the findings, which indicate the amount of rule violations recorded by each checker on the three cloud services and the effect of different optimizations. Section A. Experimental Setup We provide the findings of an experiment that was carried out using three cloud services: Azure and Azure Bare Two, which provide management services, and O-365, which is an Office 365 messaging service. The names of these services have been anonymised so that they cannot be targeted. For these three services, the amount of requests in the RESTAPI might vary between thirteen

ISSN 2454-9940
www.ijasem.org
Vol 19, Issue 2, 2025

and nineteen. The three services we chose are typical of the cloud services we looked at in terms of size and complexity. Section V summarizes our overall experience with various additional services, and we have conducted comparable studies with around a dozen of them. There is a publicly published Swagger standard for every service we are considering [15]. Following previous work, we assemble the specification of each service to generate a test-generation language [5]. There is executable Python code for every grammar rule. All of the tests mentioned here utilized the same syntax and fuzzy dictionary for the provided service and API. The results are consistently accurate. Using an internet-connected PC and a single-threaded fuzzer, we conducted our fuzzing tests. the proper subscription to each service that grants access to its API. We didn't need any more service expertise or unique test setup. As mentioned in [5], our fuzzy logic contains a garbage collector that removes dynamic objects and other resources that are no longer required to prevent service quota limitations from being exceeded. No one can see what goes on in the background of the services we test, even though our fuzz production services are up and available to everyone with a subscription. When it gets a response, our fuzzy logic system just looks at the HTTP status code. The client sends all queries to the target services across the internet, and the services parse the answers. The experiments presented in this section are not totally controlled since we do not control the distribution of these services. Having said that, we ran these tests many times and found no significant differences. B. Analyzing Search Approaches With the use of security checkers, we have compared our new search technique, BFS-Cheap, against BFS and BFS-Fast in order to fuzz genuine services. The findings of our trials with Azure A, Azure B, and Office 365 C are shown here. Each service was tested independently with three different search tactics over the course of an hour, as shown in Table I. We reported the total number of API requests (Total Req.), the maximum sequence length (MaxLen.), the number of tests, the percentage of requests sent by the main driver (Main.) and the active checkers (Checkers.) We also reported the individual contribution of each checker for each experiment. Table I clearly demonstrates that, across all services, BFS achieves the lowest depth, BFS-Fast the highest, and BFS-Cheap, which is closer to BFS than BFS-Fast, offers a compromise between these two extremes. The overall number of tests created varies among services, influenced by how quickly each service responds. The overall number of tests grows dramatically for BFS FAST with Azure Band O-365C, but otherwise this amount stays nearly the same for any given service. With O-365C, this boost seems to

EXAMPLESOFRESTAPISECURITY VULNERABILITIES

Nearly a dozen production Azure and Office 365 cloud services, comparable in size and complexity to the three services mentioned above, have been fuzzy-zed as of this writing. Every one of these services has a couple of new problems discovered by our fuzz testing. Roughly two-thirds of these issues are "500 Internal Server Errors," and our new security checkers have identified rule breaches in one-third. All of these issues have been resolved once we notified the service owners. We stress that reliability of the rules 394 is enhanced even when security checks do not detect any defects. This license is only valid for usage at Middlesex University. This document was downloaded from IEEE Xplore on August 31, 2020, at 16:21:28 UTC. Limitations are in place. They make sure the service is reliable and secure by checking that it cannot be breached. To illustrate the security implications of these issues, this section provides instances of actual defects discovered in Azure and Office 365 services that have been deployed. So that we don't single out any one service, we mask the names and other identifying information of those services. Use-after-free violation in Azure. This use-after-free violation was discovered in Microsoft Azure service. Initiate the creation of resource R by submitting a PUT request. 2) Use a DELETE request to remove resource R. Step three: Make a new PUT request to create a specific-type child resource of the removed resource R. A "500 Internal Server Error" is the outcome of these request sequences. Because (1) it tries to re-use the deleted resource in Step 3 and (2) the result of Step 3 differs from the anticipated "404 Not Found" response, the Use-after-free checker finds this. Resource-hierarchy violation in Office365. The following issue was found by the resource-hierarchy checker in an Office 365 messaging service that allows users to compose, respond, and modify messages. First, create a new message called msg-1 by sending a POST request to /api/posts/msg-1. 2) Make a second message called msg-2 and send it using the POST method to: /api/posts/msg-2. (3) Make a reply-1 to the first message (using the POST request to /api/posts/msg-1/replies/reply-1). 4) Make changes to the reply-1 by submitting a PUT request to /api/posts/msg-2/replies/reply-1, using msg-2 as the message

ISSN 2454-9940
www.ijasem.org
Vol 19, Issue 2, 2025

identifier. Despite expecting a "404 Not Found" error, the last request in Step 4 unexpectedly gets a "200 Allowed" answer.

This infraction of the rule shows that the reply-posting API implementation does not examine the whole hierarchy when verifying the reply's rights. An attacker might potentially exploit security flaws in a system if validation checks for the hierarchy are missing. This would allow them to circumvent the parent hierarchy and access child items. Azure resource leak occurred. The same issue occurred in another Azure service due to the resource-leak checker. 1) Use a PUT request to generate a new CM resource with the given name and a specified deformity in its body. As it stands, this produces the bugged "500 Internal Server Error" message. 2) If you want a list of all CM resources, you'll get an empty list. Third, using a PUT request, create a new CM resource with the same name X as in Step 1, but in a different area (e.g., US-West instead of US-Central). The resource should have a well-formed body. Surprisingly, rather than the anticipated "200 Created," the last request in Step 3 yields a 4009 Conflict. The service has entered an inconsistent state due to this behavior, which was caused by the undesired side effects of the unsuccessful request in Step 1. The user is accurate; the CM resource X that was tried to be created in Step 1 has not been generated, as shown by the GET request in Step 2. Step 3's second PUT request, however, demonstrates that the service retains memory of the previous PUT request's unsuccessful attempt to create the CM resource X. An attacker may possibly harm the system by creating an infinite number of these "zombie" resources by repeating Step 1 with new names. This would allow them to surpass their official limit, since unsuccessful resource creations are (correctly) not tallied against the user's quota. Still, the backend service obviously remembers them (incorrectly). Eager Resource-Accounting Denial of Service Attack is another example. During our five hours of fuzzing another Azure service, we unintentionally caused a significant decline in its health. The results about its origin are summed up here. To avoid going over our cloud resource limits, our fuzzing program employs a trash collector. For example, if resource type Y has a default quota of 100, then no more than 100 of that type may be generated at any one time. To prevent the number of live resources from exceeding limits, our garbage collector deletes (via a DELETE request) resources that are no longer in use. Our fuzzing tool usually reaches quota restrictions in minutes and can't continue exploring state space without trash collection. The backend processes for this Azure service might take minutes to finish, yet the response time is lightning fast for every PUT request to create a resource of a certain type—let's call it IM—. Similarly, deleting an IM resource causes a deletion job that takes minutes to finish but also returns fast. Nevertheless, these PUT and DELETE requests for IM resources update counts towards quotas hastily, without waiting the many minutes really required to perform the jobs in their entirety. Therefore, a malicious actor might easily flood the backend service by creating and deleting a large number of IM resources rapidly without going over their limit. We unintentionally set off a Denial-of-Service attack using our fuzzing tool. Until all remove backend operations are finished, which might take a few minutes for IM resources, the consumption counts for remove requests should not be updated towards quotas. This would address the problem. This ensures that the official quota for backend tasks is still linearly limited, as requests to create new IM resources (PUT) will not be allowed until all requests to delete resources (DELETE) have been processed.

RELATED WORK

Our work extends stateful REST API fuzzing [5]. Given a Swagger specification of a REST API, this a fuzzing language is constructed from the specification and used to automatically produce request sequences that meet the standard. Unlike conventional grammar-based fuzzing, in which the user manually develops a grammar, stateful REST API fuzzing automates the generation of a fuzzing grammar [20], [22], [24]. In model-based testing, test generation algorithms serve as an inspiration for the BFS and BFS-Fast search techniques [27].

using methods described in [12], [28] to create minimum test suites that include a full finite-state machine model of the investigated system. In order to improve upon stateful REST API fuzzing, this paper does two things: first, it introduces a set of security rules for REST APIs and matching checkers to efficiently test and detect violations of these rules; and second, it introduces BFS-Cheap, a new search strategy that offers a compromise between BFS and BFS-Fast when using active checkers. Requests and answers to REST APIs go across the HTTP protocol, making them amenable to HTTP-fuzzers. Fuzzers can

capture and replay HTTP traffic, parse the contents of HTTP requests and responses (such as embedded JSON data), and then fuzz them using either pre-defined heuristics or user-defined rules. Examples of such fuzzers are Burp [7], Sulley [23], BooFuzz [6], the commercial AppSpider [4], and Qualys's WAS [21]. Recent extensions to tools that record, parse, fuzz, and replay HTTP traffic have made use of Swagger standards to guide the fuzzing of HTTP requests via REST APIs and to parse them [4, 21, 26, 3]. Their fuzzing is stateless, meaning it can only fuzze the parameter values of individual requests; nevertheless, these tools can't construct new request sequences since they don't do any global analysis of Swagger specifications. Consequently, stateless fuzzers become troublesome when active checks are added to them. To the contrary, we have developed active checks that target particular REST API rule breaches, expanding the scope of stateful REST API fuzzing. Since the majority of HTTP-fuzzers are actually extensions of more conventional web-page crawlers and scanners, they typically have a long list of properties that are specific to HTTP that they can check. For example, they can verify that responses use HTTP correctly and even look for cross-site scripting attacks or SQL injections if completely rendered web pages with HTML and Javascript code are returned. Unfortunately, the majority of REST APIs do not offer web-pages in their answers, rendering most of the previously listed testing capabilities useless. New security criteria tailored to RESTAPI use are introduced in our study, distinguishing it from HTTP-fuzzers and web scanners. A security risk exists when an attacker might potentially utilize a rule's violation to compromise a service's integrity or gain unauthorized access to sensitive data or resources. Unlike other non-"exploitable" REST API use guidelines, we do not include how to verify request idempotence in this work [9]. This means that sending the same request, such as GET or PATCH, twice would not change the result. It is surprising that there is so little information available on how to utilize REST APIs securely, considering how popular they are. The majority of the security recommendations in books on REST APIs [1] or micro-services [17] or from organizations like OWASP [19] (Open Web Application Security Project) focus on how to handle authentication tokens and API keys. The documentation for the REST API is lacking in specifics when it comes to managing resources and validating user input. The four security rules presented in this work are novel, as far as we are aware. Our checkers create new tests to trigger rule violations, in addition to monitoring API request and response sequences as in conventional runtime verification [8], [11]. We utilized the term "active checker" from [10] to describe this in Section III. We use a number of separate security checks all at once, much as in [10]. However, in contrast to [10], we do not produce additional tests via symbolic execution, constraint formation, or solution. Our fuzzing tool and its checkers can only view requests and answers from REST APIs; they have no idea how the services we test really function. Cloud services are often dispersed, complicated systems with components written in various languages. Therefore, broad symbolic-execution-based approaches may not be the best choice. However, this is something that might be addressed in future research. Penetration testing, sometimes known as pen testing, is the current gold standard for protecting cloud services. This method involves having security professionals examine the cloud service's code, design, and architecture from a security standpoint. Since pen testing requires a lot of manual effort, it is not only costly but also has limited coverage and depth. Fuzzing tools and security checkers, such as those covered in this article, may supplement pen testing by partially automating the detection of certain types of security flaws.

CONCLUSION

To capture the best features of REST APIs and services, we laid down four security criteria. We proceeded to demonstrate how active property checkers may be integrated into a stateful REST API fuzzer to automatically test and identify rule violations. Using the fuzzer and checkers outlined in this work, we have successfully fuzzed about a dozen production Azure and Office-365 cloud services. Nearly every one of these services has a couple of new problems discovered by our fuzzing. The majority of these issues are "500 Internal Server Errors," but our new security checkers have identified rule violations as accounting for around one third of the flaws. All of these issues have been resolved once we notified the service owners. It is rather evident that security vulnerabilities might arise from

disobeying the four security principles presented in this study. Our current bug "fixed/found" ratio is approximately 100%, indicating that all of the issues we detected have been thoroughly addressed by the service owners. Fixing these issues is preferable than risking a live event, which might be purposefully or accidentally caused by an attacker and could have unforeseen effects. Lastly, the fact that our fuzzing method does not disclose any false alarms and that these problems can be readily reproduced is helpful. On what scale do these findings apply? In order to discover the answer, we must inspect further characteristics and fuzz more services via their REST APIs in order to identify various types of vulnerabilities and problems. Surprisingly, there is a lack of security-related guidelines about the use of REST APIs, despite the recent expansion of these APIs for use in cloud and online services. In this regard, our study contributes four rules whose infractions are security-relevant and which are not easy to verify and fulfill.

REFERENCES

[1]. S. Allamaraju. RESTful Web Services Cookbook. O'Reilly, 2010.
[2]. Amazon. AWS. https://aws.amazon.com/.
[3]. APIFuzzer. https://github.com/KissPeter/APIFuzzer.
[4]. AppSpider. https://www.rapid7.com/products/appspider.
[5]. V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API Fuzzing. In 41st ACM/IEEE International Conference on Software Engineering (ICSE'2019), May 2019.
[6]. BooFuzz. https://github.com/jtpereyda/boofuzz.
[7]. Burp Suite. https://portswigger.net/burp.
[8]. D. Drusinsky. The Temporal Rover and the ATG Rover. In Proceedings of the 2000 SPIN Workshop, volume 1885 of Lecture Notes in Computer Science, pages 323–330. Springer-Verlag, 2000.
[9]. R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, UC Irvine, 2000.
[10]. P. Godefroid, M. Levin, and D. Molnar. Active Property Checking. In Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software), pages 207–216, Atlanta, October 2008. ACM Press.
[11]. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In Proceedings of RV'2001 (First Workshop on Runtime Verification), volume 55 of Electronic Notes in Theoretical Computer Science, Paris, July 2001.
[12]. R. L¨ ammel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In Proceedings of TestCom'2006, 2006.
[13]. Microsoft. Azure. https://azure.microsoft.com/en-us/.
[14]. Microsoft. Azure DNS Zone REST API. https://docs.microsoft.com/en us/rest/api/dns/zones/get. [15] Microsoft. Microsoft Azure Swagger Specifications. https://github.com/ Azure/azure-rest-api-specs. [16] Microsoft. Office. https://www.office.com/.
[15]. S. Newman. Building Microservices. O'Reilly, 2015. [18] OAuth. OAuth 2.0. https://oauth.net/.
[16]. OWASP (Open Web Application Security Project). https://www.owasp. org.
[17]. Peach Fuzzer. http://www.peachfuzzer.com/.
[18]. Qualys Web Application Scanning (WAS). https://www.qualys.com/ apps/web-app-scanning/.
[19]. SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automation with-spike/.
[20]. Sulley. https://github.com/OpenRCE/sulley.
[21]. M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, 2007. [25] Swagger. https://swagger.io/.
[22]. TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.
[23]. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing Approaches. Intl. Journal on Software Testing, Verification and Reliability, 22(5), 2012.
[24]. M. Yannakakis and D. Lee. Testing Finite-State Machines. In Proceed ings of the 23rd Annual ACM Symposium on the Theory of Computing, pages 476–485, 1991.