ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

Verifying the REST API Security of Cloud Services

[1] S. Sushmitha, [2] P. Arthi,

[1]Assistant Professor, Megha Institute of Engineering & Technology for Women, Ghatkesar.
[2] MCA Student, Megha Institute of Engineering & Technology for Women, Ghatkesar.

Article Info

Received: 30-04-2025 Revised: 16-06-2025 Accepted: 28-06-2025

Abstract

A REST API is the standard programming interface for accessing most contemporary online and cloud applications. An attacker might potentially compromise a service by taking advantage of security holes in its REST API, as discussed in this article. To capture the best features of REST APIs and services, we provide four security criteria. To further automate testing and detection of rule violations, we demonstrate how to add active property checks to a stateful REST API fuzzer. How to efficiently and modularly build such checks is something we cover. We describe the security consequences of the new vulnerabilities discovered in several production-ready Azure and Office 365 cloud services using these checkers. We have resolved all of these issues.

Keywords

Security, REST APIs, cloud computing, and test generation

I. INTRODUCTION

People are flocking to cloud computing. Providers of cloud platforms, such as Amazon Web Services [2] and Microsoft Azure [13], and their customers, who are "digitally transforming" their companies through process modernization and data analysis, have deployed thousands of new cloud services in recent years. These days, REST APIs are the go-to method for programmatically accessing cloud services [9]. REST APIs provide a standard method to build, 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]. What kinds of queries can a cloud service process, what kinds of replies may be expected, and how those responses should be formatted are all detailed in a Swagger specification, which covers the service's REST API. Do you know how safe all those APIs are? Even 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. The goal of several of the existing tools for testing REST APIs is to find problems in the API by capturing live traffic, processing it using fuzz and replaying it [4, 21, 6, 26, 3]. To go even further into testing services hosted behind REST APIs, stateful REST API fuzzing [5] was suggested not long ago. This method takes a Swagger specification for a REST API and uses it to automatically produce sequences of requests rather than individual ones. in order to fully test the API's underlying cloud service, we're looking for service crashes that go unhandled and show up as "500 Internal Server Errors" on a test client. The scope of the effort is limited to the detection of unhandled exceptions, however it appears promising and reports numerous new issues detected. Here, we lay down four guidelines for protecting RESTful APIs and services, which should cover all the bases. • The rule of use after free. Once erased, a resource can never be recovered. • Rule of resource leakage. An unsuccessfully generated resource must not only be inaccessible, but it must also not "leak" any unwanted effects into the backend service state. The rule of resource hierarchy. It is not

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

allowed for another parent resource to access a child resource's parent resource. Rule 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, an attacker could exploit a breach in these rules to launch an elevation-of-privilege attack, an information disclosure attack, or a denial-of-service attack, all of which could compromise cloud resources or bypass quotas. We demonstrate the process of enhancing a stateful REST API fuzzer to examine and identify rule infractions. We provide an active property checker for every rule that does two things:(1) finds rule violations and(2) produces new API calls to test them. To rephrase, rather than passively observing for rule violations, each checker actively seeks to violate its own rule. We go over several modular ways to create such checks, making sure they don't conflict with one other. We also cover how to efficiently build each checker by removing likely-redundant tests wherever feasible, because each checker adds additional tests to an already-large state space exploration. In contrast to baseline stateful REST API fuzzing, which can only identify "500 Internal Server Errors," these checks are designed to find security rule breaches. Several operational Azure and Office 365 cloud services have new issues discovered using these checks. Incorporating security checkers into REST API fuzzing enhances its usefulness by identifying a wider range of issues with less incremental testing overhead. The following are some of the contributions of this paper: • We provide rules that characterize the security features of REST APIs. • We create and deploy active checkers to examine and identify rule violations. We provide comprehensive experimental findings that assess the efficacy and efficiency of these active checkers on three live cloud services. Using these checks, we discovered additional vulnerabilities in several production-level Azure and Office 365 cloud services, and we go over the security consequences of these vulnerabilities. Below is the outline for the remainder of the article. Part II provides some context for understanding stateful REST API fuzzing. We provide active checkers to test and identify breaches of these criteria in Section III, and we also add rules that encapsulate desired aspects of safe REST APIs. Results from experiments using active checkers on live cloud services are detailed in Section IV. 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. STATEFUL REST API FUZZING

Section III introduces security property checks that may be used as expansions of this basic system, after which this section reviews the notion of stateful REST API fuzzing [5]. 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. The HTTP/S protocol is used to transmit these messages. Two, three, four, or five-digit HTTP status codes are assigned to each response. One specification language for REST APIs is Swagger [25], which is also called OpenAPI. The Swagger specification details the REST API access to a service, including the types of queries that the service may process, the possible answers, and the format of each. A REST API is defined by us as a limited collection of requests. In each request r, there is a tuple {a,t,p,b} that includes the following elements: an authentication token (a), the kind of request (t), a resource path (p), and the request content (b). Request types may take one of five possible RESTful values: PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), or PATCH (update). A cloud resource and its parent hierarchy may be identified by its resource path, which is a string. Usually, p is a non-empty string that matches the pattern (/~resourceType~/resourceName~/)+, where resourceType is the cloud resource type and resourceName is the specific name of that kind of resource. In most cases, a request will attempt to create, access, or delete the resource that is last specified in the route. In order for the request to be processed properly, the request body b could include extra parameters along with their values. As an example, the following is a multi-line request to acquire the attributes of a specific Azure DNS zone [14]:

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

The GET request has three resource names—a subscriptionID, a resourceGroupName, and a zoneName—in its route, and the body (at the end, represented by {}) is unfilled.

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

III. SECURITY CHECKERS FOR REST APIS

Here, we outline the features and functionality of active security rule checkers for REST APIs. We begin by outlining four guidelines for protecting REST APIs in Section III-A. Active checkers for testing and detecting security rule breaches are described 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. In Section III-D, we provide a novel approach to finding property checkers for scalable test creation. To prevent the user from receiving several reports of the same problem, we detail how to bundle together checker violations in Section III-E. Section A: Regulations about Security To capture the best features of REST APIs and services, we provide four security criteria. We talk about the security consequences of each rule and provide examples to back them up. Real flaws in deployed cloud services discovered by manual penetration testing or root cause analysis of customer-visible occurrences inspired all four guidelines. Later in Section V, we will provide examples of additional, previously undiscovered problems that we discovered as rule violations in the production Azure and Office 365 services that were already deployed. The law of use following free consumption. Once erased, a resource can never be recovered. Put simply, if a DELETE action is successful on a resource, then any read, update, or delete operation on the same resource will fail. In order to remove the account associated with user-id1, for instance, all further attempts to utilize user-id1 must fail and produce a "404 Not Found" HTTP status code. This is achieved by sending a remove request to the URI /users/user-id1. 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. When a resource creation fails, it shouldn't be available and shouldn't "leak" any related resources from 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). On top of that, the user shouldn't see any unintended consequences when the resource type is successfully created in the backend service state. For example, the name of the failed-to-be-created resource must be reusable by the user, and it must not be tallied in the user's resource counter towards service quotas. To illustrate, a response is required after the submission of a faulty PUT request to generate the URI /users/user-id1,a4xx. 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. Attempts to re-create this resource result in "409 Conflict" answers, or the resource may be listed by a future GET request but cannot be removed with a DELETE request. This kind of infraction is completely unacceptable since it might lead to unforeseen effects on the service's performance (for instance, because of excessively big database tables) or the capacity of the specific resource type (for example, if resource quota limitations are surpassed and no new resources can be added).

Resource-hierarchy rule. A child resource of a parent resource must not be accessible from another parent resource. In other words, if a resource `child` is successfully created from a resource `parent` and identified as such in service resource paths of the form `⟨parentType⟩/parent/⟨childType⟩/child/`, the `child` resource must not be accessible (i.e., must not be successfully read, updated or deleted) when substituting the `parent` resource by any other parent resource.

For example, after issuing `POST` requests to URIs `/users/user-id1`, `/users/user-id2`, and `/users/user-id1/reports/report-id1` to create users `user-id1`, `user-id2`, and then add report `report-id1` to user `user-id1`, subsequent requests to URI `/users/user-id2/reports/report-id1` must fail since, according to the resource-hierarchy rule, report `report-id1` belongs to user `user-id1` but not to user `user-id2`.

Access to a sub-resource that was formed from a parent resource but does not have a parent-child connection is an example of a resource-hierarchy violation. In cases where such infractions are feasible, an adversary may be able to provide an illicit parent object identity.

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

```
1  Inputs: seq, global_cache, reqCollection
2  # Retrieve the object types consumed by the last request and
3  # locally store the most recent object id of the last object type.
4  n = seq.length
5  req_obj_types = CONSUMES(seq[n])
6  # Only the id of the last object is kept, since this is the
7  # object actually deleted.
8  target_obj_type = req_obj_types[−1]
9  target_obj_id = global_cache[target_obj_type]
10 # Use the latest value of the deleted object and execute
11 # any request that type−checks.
12 for req in reqCollection:
13     # Only consider requests that typecheck.
14     if target_obj_type not in CONSUMES(req)
15         continue
16     # Restore id of deleted object.
17     global_cache[target_obj_type] = target_obj_id
18     # Execute request on deleted object.
19     EXECUTE(req)
20     assert ''HTTP status code is 4xx''
21     if mode != 'exhaustive':
22             break
```

Fig. 1: Use-after-free checker.

(for instance, user-id3), and then acquire (read) or commandeer (write) an illegal child object (for instance, report-id1). Instances of resource hierarchy violations are obvious defects that pose a threat and should never occur. Rule for user-namespaces. You can't have resources from one user namespace available to resources from another. While discussing REST APIs, we take into account user namespaces that are specified by the user token that is used to access the API (for example, OAUTH token-based authentication [18]). For instance, user-id1 resource cannot be accessed using token-of-user-id2 of another user after a POST request to build URI /users/user-id1 with token-of-user-id1 has been sent. 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 Rules outlined in Section III-A are enforced by means of active checkers. 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. In this way, an active checker increases the size of the search area by running additional tests that aim to break certain criteria. By contrast, a passive checker does not run any additional checks but instead watches the primary driver's search. Based on two concepts, we create active checkers using a modular architecture: 1) The state space exploration of stateful REST API fuzzing is unaffected by checkers since they are separate from the primary driver. 2) Tests are generated by separate checkers that are autonomous from one another and examine just the requests made by the primary driver.

```
1  Inputs: seq, global_cache, reqCollection
2  # Retrieve the object types produced by the whole sequence and by
3  # the last request separately to perform type checking later on.
4  seq_obj_types = PRODUCES(seq)
5  target_obj_types = PRODUCES(seq[−1])
6  for target_obj_type in target_obj_types:
7      for guessed_value in GUESS(target_obj_type):
8          global_cache[target_obj_type] = guessed_value
9          for req in reqCollection:
10             # Skip consumers that don't consume the target type.
11             if CONSUMES(req) != target_obj_type:
12                 continue
13             # Skip requests that don't typecheck.
14             if CONSUMES(req) − seq_obj_types:
15                 continue
16             # Execute the request accessing the ''guessed'' object id.
17             EXECUTE(req)
18             assert ''HTTP status code in 4xx class''
19             if mode != 'exhaustive':
20                 break
```

Fig. 2: Resource-leak checker.

All of the checkers are executed once the main driver completes running a new test case, in order to enforce the first principle. As for the second principle, we make sure that checkers don't interfere with one other and work on separate test cases by ordering them according to their semantics (we'll get into this further later on). Following this, we outline the specifics of each checker's implementation and provide improvements to curb the growth of state spaces. Utilization verification tool. Figure 1 shows the use-afterfree rule checker's implementation in a notation 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 take the final type in req_object_types as the actual type of the deleted object, even if the last request may be consuming more than one object type. The DELETE request at the URI /users/userId1/reports/reportId1 consumes two sorts of objects: reports and users, but it only deletes report objects. The for-loop iterates over all requests accessible in reqCollection and skips those that don't consume the target object type after this initial setup at line 14. In order for the EXECUTE function (line

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

19) to carry out the execution of request req, the target object id is restored in the global cache of dynamic objects (line 17) after a request consuming the target object type is located. The reason the global cache keeps restoring the target object id is because the EXECUTE function utilizes the object ids in global_cache to execute requests. A use-after-free violation will be triggered if any of these requests are successful (see to Section III-A).

```
1  Inputs: seq, global_cache
2  # Record the object types consumed by the last request
3  # as well as those of all predecessor requests.
4  n = seq.length
5  last_request = seq[n]
6  target_obj_types = CONSUMES(seq[n])
7  predecessor_obj_types = CONSUMES(seq[:n])
8  # Retrieve the most recent id of each child object consumed
9  # only by the last request. These are the objects whose
10 # hierarchy we will try to violate.
11 local_cache = {}
12 for obj_type in target_obj_types − predecessor_obj_types:
13   local_cache[obj_type] = global_cache[obj_type]
14 # Render sequence up to before the last request
15 EXECUTE(seq, n−1)
16 # Restore old children object ids that do NOT belong to
17 # the current parent ids and must NOT be accessible from those.
18 for obj_type in local_cache:
19     global_cache[obj_type] = local_cache[obj_type]
20 EXECUTE(last_request)
21 assert ''HTTP status code is 4xx''
```

Fig. 3: Resource-hierarchy checker.

Lastly, on line 21, the inner loop (optionally) ends when one request for each target object type is detected, limiting the amount of subsequent tests created for each request sequence. If the value exhaustive option is not specified for the mode variable, this option is utilized. In Section IV, we provide comprehensive experimental data on the effects of this optimization. The tool for checking for resource leaks. You can see the description of the resource-leak rule checker in Figure 2. Just like the use-after-free checker, this method requires three parameters. The primary driver, whose most recent request resulted in an incorrect HTTP status code in the response (refer to Figure 4), is the target of this checker. The procedure starts by determining the kinds of dynamic objects created by the whole series (seq_obj_types) and the most recent request (target_obj_types) (lines 4 and 5). Three layered for loops implement the algorithm's core logic. In the first loop (line 6), all object types that were generated by the previous request are iterated over. Line 7 of the second loop iterates over all the object ids that were "guessed" for the current object type that returned an incorrect HTTP status code. You may provide an object type to the GUESS function, and it will return a list of probable object ids that fit that type but were unsuccessfully constructed. For example, if the API response indicates that creating a dynamic object with the ID "objx1" and the type "x" fails, the checker will try to run any request that uses the type "x" and indicate that it fails when using the ID "objx1". To prevent an explosion in the number of further tests, the total number of estimated values per object id is restricted to a parameter value that the user provides. On line 8, the global cache of correctly constructed dynamic objects is momentarily updated with an object-id value that is guesswork. Then, on line 9, the inner loop iterates over all of the requests in reqCollection until it finds one that consumes the supplied target object type and is executable (based on the object types created by the current sequence). The "guessed" item ids that were previously stored in the global cache are used to perform these queries (line 17). In this approach, the algorithm endeavors as

```
1  Inputs: seq, global_cache, reqCollection
2  # Execute the checkers after the main driver.
3  n = seq.length
4  if seq[n].http_type == ''DELETE'':
5          UseAfterFreeChecker(seq, global_cache, reqCollection)
6  else:
7  if seq[n].http_response == ''4xx'':
8          ResourceLeakChecker(seq, global_cache, reqCollection)
9  else:
10         ResourceHierarchyChecker(seq, global_cache)
11         UserNamespaceChecker(seq, global_cache)
```

Fig. 4: Checkers dispatcher.

Contribution beyond stateful REST API fuzzing. Checkers enhance basic 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, they may catch unexpected 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 uncover flaws that it couldn't detect on its own. Active property checking vs passive monitoring. The checkers we define, as said before, provide more test cases to the main driver's search area with the goal of triggering and detecting specific rule violations. However, without actually running those additional tests, passive runtime monitoring of these rules alongside the primary driver is unlikely to be able to identify rule breaches. Because the primary driver's default state space exploration probably wouldn't try to re-use deleted resources or resources after a failure, passive monitoring alone would likely miss use-after-free and resource-leak rule violations, respectively. Because the basic main driver doesn't try to replace object identifiers or authentication tokens, passive monitoring would also miss resource hierarchy and user-namespace rule breaches. To

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

rephrase, the extra test cases produced by the checkers are not superfluous in comparison to non-checker tests; rather, they are essential for discovering rule violations. The checkers work in tandem with one another. All four of our checkers work well with one another; in fact, due to the fact that their respective preconditions are inherently incompatible, no two of them can ever provide identical new tests. First, request sequences that conclude with a DELETE request activate the use-afterfree checker. No other checker does this. As a second point, if the most recent request's HTTP status code is incorrect, just the resource-leak checker will be engaged. Thirdly, request sequences that do not conclude with a DELETE request have the resource-ownership checker engaged as the lone checker. Finally, the user-namespace checker clearly 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. Methods for Finding Checkers Stateful REST API fuzzing [5] relies on a breadth-first search (BFS) in the search space defined by all potential request sequences as its primary search method for test creation. 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. Each request is only added to one request sequence of length n in BFS-Fast, as opposed to all of them in BFS, whenever the search depth grows to a new number n+1 [5]. Full grammar coverage is only provided by BFSFast in regard to all conceivable renderings of individual requests; it does not investigate all request sequences of a certain length. A subset of all potential request sequences is explored by BFS-Fast, which allows it to scale better than BFS. 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. For a particular sequence length, BFS-Cheap investigates all potential request sequences, but not with all conceivable renderings. This is in contrast to BFS-Fast, which fully covers all possible request renderings at every stage. In particular, the following is how BFS-Cheap functions when given an n-sequence set (seqSet) and a collection of requests (reqCollection): Add all the potential renderings of each req to the end of each seq, run the new sequence while evaluating the possible renderings of req, and add no more than one valid and one incorrect sequence rendering to seqSet for each seq ∈ seqSet. The use-after-free, resourcehierarchy, and user-namespace checking all rely on proper renderings, but the resource-leak checker relies on faulty renderings. Therefore, BFS-Cheap is a compromise between BFS and BFS-Fast; for an experimental assessment, see Section IV-B. To prevent a huge seqSet (like BFS-Fast), it investigates all potential request sequences up to a certain sequence length (like BFS) and adds no more than two additional renderings to each sequence. By introducing two additional renderings for each sequence, we can actively verify all the security requirements outlined in Section III-A, all while keeping the number of sequences in seqSet manageable even as the length of the sequence rises. It should be noted that the suffix "cheap" is derived from the fact that BFS-Cheap is a less expensive variant of BFS in which the BFS"frontier" setSeq only receives one correct rendering for each news sequence. This results in less resource development compared to BFS, which explores all viable renderings of each request sequence. Consider a request definition that specifies 10 distinct types of resources, each described by an enum type. After one resource of a certain flavor has been successfully developed, BFS-Cheap will cease producing more of that flavor. However, BFS and BFS-Fast will generate 10 identical resources with ten distinct favors. Substantiation of Bugs Our definition of the bucketization technique used to group related violations precedes our discussion of genuine violation instances detected using active checkers. We define "bugs" as rule breaches in the context of 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: Each time a new issue is discovered, calculate all nonempty suffixes of the request sequence that causes

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%

TABLE I: Comparison of BFS, BFS-Fast and BFS-Cheap. Shows the maximum sequence length (Max Len.), the number of requests sent (Tests), the percentage of tests generated by the main driver (Main) and by all four checkers combined (Checkers) and individually, with each search strategy after 1 hour of search. The second column shows the total number of requests in each API.

the insect, beginning with the tiniest one. Include the new sequence in an existing bug bucket if it has a suffix that has already been logged. If it isn't possible, make a fresh bug bucket for the novel sequence. We keep distinct bug buckets for each checker as the failure circumstances are defined individually for each rule. This bug bucketization method is identical to the one in stateful REST API fuzzing [5]. Except for "500 Internal Server Error" flaws, which may be caused by both the main driver and checkers, each defect will only be triggered by one checker for a certain sequence length due to checker complementarity. The bug bucket of the primary driver or checker that triggered the new sequence will only be updated once for 500 bugs.

IV. EXPERIMENTAL EVALUATION

Here we detail the outcomes of our trials using three real-world cloud services. Section IV-A details these services and our experimental setup. Section IV-B then compares the three search algorithms outlined in Section III-D. After that, we exhibit the results (Section IV-C) that illustrate how many rule violations each checker found on the three cloud services and how different optimizations affected those findings. (A) Experimental Environment The following are the outcomes of our trials conducted with three anonymous cloud services: O-365 C is a communications service for Office365 [16], whereas Azure A and Azure B are two management services for Azure [13]. From thirteen to nineteen queries per service, it is the range of the three services' REST APIs. These three services were chosen because they are typical of the cloud services we examined in terms of scale and complexity. So far, we have conducted comparable trials with around twelve other production services; Section V summarizes our overall experience with these additional providers. There is a publicly-available Swagger specification for every service we are considering [15]. Following previous work, we create a test-generation language by compiling the specification of each service [5]. There is executable Python code for every grammar rule. All the tests presented here utilized the same syntax and fuzzing dictionaries for a specific service and API. Renders are not produced at random. To conduct our fuzzing studies, we used a single-threaded fuzzer on an internet-connected PC. Each service API was accessible thanks to a valid subscription. There was no need for any additional service expertise or unique test setup. To prevent going over our service cap, our fuzzer incorporates a garbage-collector, similar to the one in [5], which removes unused resources (dynamic objects). Even though we test production services that are live and available to subscribers, we can't see what's happening behind the scenes of the services we test. The only thing our fuzzer looks at in response data are the HTTP status codes. We send all client-side queries across the internet to the target services, and when we get their answers, we parse them. The experiments presented in this section are not entirely controlled since we do not have control over the distribution of these services. The findings did not differ much, however, and we repeated the trials many times. B. Analyzing Rival Search Techniques For the purpose of fuzzing actual services with security checkers, we now compare our new search approach, BFS-Cheap, against BFS and BFS-Fast. Here we show the outcomes of tests conducted with three different Office 365 services: Azure A, Azure B, and O-365 C. With a fixed budget of one hour each experiment, Table I displays individual tests with the three search techniques on each service. There are a lot of metrics that are reported for each experiment. These include the total number of API requests, the maximum sequence length, the number

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

of tests, the percentage of requests sent by the main driver and active checkers, and the individual contribution of each checker. Based on Table I, it is evident that BFS achieves the lowest depth for all services, BFS-Fast reaches the highest depth, and BFS-Cheap offers a compromise between the two, being closer to BFS than BFS-Fast. As a result of differences in response times, the overall number of tests produced differs across providers. The overall number of tests grows significantly for BFSFAST with Azure B and O-365 C, while for all other services, this number stays rather constant. This growth seems to be true for O-365 C.

API	Total Req.	Mode	Statistics		Bug Buckets				
			Tests	Checkers	Main	Use-Aft-Free	Leak	Hierarchy	NameSpace
Azure A	13	optimized	4050	45.0%	4	3	0	0	0
		exhaustive	2174	54.5%	4	3	0	0	0
Azure B	19	optimized	7979	46.2%	0	0	1	0	0
		exhaustive	9031	63.9%	0	0	1	0	0
O-365 C	18	optimized	10982	4.1%	1	0	0	1	0
		exhaustive	11724	11.4%	0	0	0	1	0

TABLE II: Comparison of modes optimized and exhaustive for two Azure and one Office-365 services. Shows the number of requests sent in 1 hour (Tests) with BFS-Cheap, the percentage of tests generated by all four checkers combined (Checkers), and the number of bug buckets found by the main driver and each of the four checkers. Optimized finds all the bugs found by exhaustive but its main driver explores more states faster given a fixed test budget (1 hour).

to be because BFS-FAST generates much less unsuccessful requests for these two services than BFS and BFS-Cheap. Requests that do not succeed are returned to our fuzzer, the client, with longer wait times. It is well knowledge that services may slow down future requests by delaying replies to rejected ones. When it comes to Azure B, BFS-Fast runs more tests than BFS or BFS-Cheap. This is due to the fact that BFS-Fast's request sequences are more in-depth, but they include numerous DELETE requests, which are quicker to perform (their replies are returned practically quickly). While BFSCheap falls somewhere in the middle, BFS has the greatest overall percentage of checker tests (Checkers) and BFS-FAST has the lowest. According to Section III-D, the reason for inventing BFS-Cheap was to address the fact that BFS-Fast produces more tests than any other method, but it prunes its search area and activates checkers less often. O-365 C stands out, however, with a 33% increase in BFS-generated tests. A greater number of successful requests (refer to the preceding paragraph) caused more checker tests, which seems to be the cause of this surge. We can see that the amount of tests generated by each checker differs between services from the information in Table I. This figure is calculated by taking into account the depth of the object hierarchy for the resource hierarchy checker, the number of unsuccessful resource creation requests for the use-after-free checker, and the number of DELETE requests performed for the use-after-free checker. In contrast, the majority of tests created by the checker are from the user-namespace checker, and it is activated more often and regularly. Next, we'll talk about how the three search algorithms yielded roughly identical bug counts for all three services.

V. EXAMPLES OF REST API SECURITY VULNERABILITIES

Almost a dozen operational Azure and Office 365 cloud services, comparable in size and complexity to the three used before, have been fuzzed as of this writing. Every one of these services has a couple of new vulnerabilities discovered by our fuzzing efforts. Our new security checkers have identified rule violations as accounting for about one third of these issues, while "500 Internal Server Errors" accounts for around two thirds. All of these issues have been resolved once we notified the service owners. We stress that security testers boost confidence in the service's overall dependability and security even if they don't find any vulnerabilities; this is because they make sure the rules they verify cannot be broken. This section discusses the security significance of real-world defects detected in deployed Azure and Office 365 services and provides instances of such problems. We ensure that no specific service is targeted by anonymizing the names and crucial information of such services. Use-after-

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

free violation in Azure. We discovered the following use-after-free violation in an Azure service. 1) Use a PUT request to create a new resource called R. A DELETE request should be used to delete resource R. 3) Make a new PUT request to create a specific-type child resource of the removed resource R. An error message stating "500 Internal Server Error" is produced by this series of queries. This is caught by the Use-after-free checking because (1) the removed resource is attempted to be used again in Step 3, and (2) the result from Step 3 is not the anticipated "404 Not Found" answer. Resource-hierarchy violation in Office365. According to the resource-hierarchy checker, there is a flaw in an Office 365 messaging service that allows users to publish, respond, and modify messages. 1) Make a single message called msg-1 by sending a POST request to /api/posts/msg-1. 2) Make second message msg-2 (using POST /api/posts/msg-2). thirdly, using the POST request to /api/posts/msg-1/replies/reply-1, create a reply-1 to the first message. 4) Use msg-2 as the message identifier and edit reply-1 using a PUT request (with the format /api/posts/msg-2/replies/reply-1). 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 instance experiencing a resource leak. A different Azure service had the same issue due to the resource-leak checker. 1) Make a brand-new CM resource with the name X and a specific deformity (using a PUT request). As it stands, this produces the bugged "500 Internal Server Error" message. 2) There is no data supplied when you ask for a list of all CM resources. 3) Substitute a different area (e.g., US-West for US-Central) and a PUT request into Step 1 to create a new resource of type CM with the same name X. The last request in Step 3 actually returns a "409 Conflict" rather than the predicted "200 Created." This is somewhat unexpected. The service has entered an inconsistent state due to this behavior, which was caused by the unwanted sideeffects of the unsuccessful request in Step 1. Step 2's GET request confirms the user's suspicions: the CM resource X, which was supposed to be generated in Step 1, is still not there. Step 3's second PUT request, however, demonstrates that the service retains memory of the first PUT request's unsuccessful attempt to create the CM resource X. An attacker may theoretically abuse this flaw to their heart's content by creating an infinite supply of these "zombie" resources by repeating Step 1 with various names. This would allow them to surpass their official limit, since unsuccessful resource creations are (correctly) not tallied against the user's quota. However, it is evident that they are remembered (incorrectly) by some backend service. Additional Illustration: An Anxious Denial-of-Service Attack on Resources. We inadvertently caused another Azure service's health to drastically decline after five hours of fuzzing. What follows is a synopsis of the research on what caused it. To ensure that the amount of cloud resources used during fuzzing does not go beyond limits, our program incorporates a trash collector. For example, if the default quota for a resource type Y is 100, then no more than 100 of those resources may be generated at any one time. Through the usage of a DELETE request, our garbage collector ensures that the number of living resources never exceeds quotas. Our fuzzing tool usually reaches quota restrictions in minutes and can't continue exploring state space without trash collection. A PUT request to create a resource of a certain type—let's call it "IM"—in this particular Azure service produces a response rapidly, but in reality, it activates additional operations that take minutes to finish in the service backend. In the same vein, deleting an IM resource (using the DELETE command) yields the same result in a matter of minutes. While these PUT and DELETE requests indeed update IM resource counts towards quotas, they do so much too promptly and without waiting the many minutes really required to do the operations. Consequently, a malicious actor might swiftly generate and destroy several IM resources without going over their allotted limit, causing an overwhelming amount of backend operations and effectively flooding the backend service. We unintentionally set off a Denial-of-Service attack using our fuzzing tool. Fixing this issue may be as simple as waiting a few minutes after all remove backend operations have finished, in the case of instant messaging resources, before updating use counts towards quotas for remove requests. This ensures that the quantity of backend tasks is still limited by the official limitation, since pending DELETE requests will prevent any further IM resource-creation PUT requests from being processed.

VI. RELATED WORK

Our approach expands upon fuzzing for stateful REST APIs [5]. To automatically produce sequences of queries that fulfill a Swagger specification of a REST API, the specification is first turned into a fuzzing language. In contrast to the more

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

conventional grammar-based fuzzing methods, where the user develops a grammar by hand, stateful REST API fuzzing automates the construction of a fuzzing grammar [20], [22], [24]. The BFS and BFS-Fast search algorithms take their cues from the model-based testing test generation methods [27], For creating minimum test cases that encompass a whole finite-state machine model of the system being tested, see [12], [28]. A new search strategy, BFS-Cheap, provides a middle ground between BFS and BFS-Fast when using active checkers. This paper also introduces a set of security rules for REST APIs and corresponding checkers to efficiently test and detect violations of these rules. The paper expands upon stateful REST API fuzzing in two ways. Using an HTTP-fuzzer to test REST APIs is possible since both requests and answers to REST APIs are sent via the HTTP protocol. 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]. In order to better understand HTTP requests made over REST APIs and direct their fuzzing, many tools that collect, parse, fuzz, and replay HTTP traffic have since been updated to use Swagger specifications [4, 21, 26, 3]. Unfortunately, these tools are limited to fuzzing the parameter values of individual requests and do not do any global analysis of Swagger specifications. As a result, they cannot construct new sequences of requests. This is because their fuzzing is stateless. Thus, it is not a good idea to add active checks to stateless fuzzers. Our approach, on the other hand, adds active checks that target specific REST API rule breaches to stateful REST API fuzzing. Many HTTP-fuzzers have their roots in older web-page crawlers and scanners, so they can check for a wide variety of HTTP-specific properties. For example, they can ensure that responses use proper HTTP-usage and even detect SQL-injections or cross-site scripting attacks when entire web pages with HTML and Javascript code are returned. On the other hand, web-pages are not often returned by REST APIs, rendering most of the previously described testing capabilities useless. Our study presents new security criteria that are tailored to RESTAPI use, in contrast to HTTP-fuzzers and web scanners. Because an adversary may utilize a rule's infraction to compromise a service's health or steal sensitive data or resources, these regulations are considered security-related. Request idempotence, in which sending the same GET or PATCH request again has no impact on the result, is one example of a rule in REST API use that is not "exploitable" when broken, although we don't cover it in this article. Considering how common REST APIs are, it's surprising that there isn't much information on how to use them securely. Managing authentication tokens and API keys is a common theme in security guidelines from organizations such as OWASP [19] (Open Web Application Security Project) or books on REST APIs [1] or micro-services [17]. When it comes to managing resources and validating inputs, the REST API does not provide any explicit instructions. The four security rules presented in this work are novel, as far as we are aware. In Section III, we utilized the term "active checker" from [10] to indicate that our checkers create new tests with the express purpose of detecting rule violations, rather than just monitoring API request and response sequences as in conventional runtime verification [8], [11]. We employ numerous independent security checkers concurrently, much as in [10]. Our approach to creating new tests differs from that of [10] in that it does not include symbolic execution, constraint derivation, 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. It would be beneficial to delve more into this possibility in future study, since cloud services are often intricate distributed systems with components written in various languages. Consequently, generic symbolic-execution-based techniques may appear difficult. Penetration testing, sometimes known as pen testing, is the primary method now used to guarantee the security of cloud services. This involves security specialists reviewing the architecture, design, and code of cloud services from a security standpoint. Pen testing is costly, has limited reach, and requires a lot of human effort. In addition to pen testing, fuzzy logic and security checkers (such as the ones covered in this article) may help automate the process of finding certain types of security vulnerabilities.

VII. CONCLUSION

In order to capture the desired qualities of REST APIs and services, we established four security criteria. Afterwards, we demonstrated how to include active property checkers into a stateful REST API fuzzer in order to automatically test for 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

ISSN:0976-0172

Journal of Bioscience And Technology
www.jbstonline.com

cloud services. Every one of these services has a couple of new vulnerabilities discovered by our fuzzing efforts. Our new security checkers have identified rule violations as accounting for about one third of these issues, while "500 Internal Server Errors" accounts for around two thirds. The service owners were notified of all the issues, and they have all been resolved. It is rather evident that security vulnerabilities might arise from disobeying the four security principles presented in this study. The service owners have all taken the issues we detected seriously; as a result, our current bug "fixed/found" ratio is very close to 100%. In addition, fixing these problems is preferable than taking the chance of a live occurrence, which might be caused by an attacker or accidentally, and the results of which are uncertain. Lastly, the fact that these errors can be reproduced with relative ease and that our fuzzing method does not produce any false positives is a plus. On what scale do these findings apply? The only way to find out is to fuzz additional services using their REST APIs and examine more attributes to find 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. Contributing four rules whose infractions are security-relevant and nontrivial to verify and fulfill, our work takes a start in that direction.

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/enus/rest/api/dns/zones/get.

[15] Microsoft. Microsoft Azure Swagger Specifications. https://github.com/ Azure/azure-rest-api-specs.