TRAFFIC RULES VIOLATION DETECTION SYSTEM

[1]A.DILIP REDDY, [2]A.MASTAN BABU, [3] B.MURALI KARTHIK, [4]D.VARUN, Mrs. [5]Palle Swetha,

[1,2,3,4] U.G.Scholor, Department of ECE, Sri Indu College Of Engineering & Technology, Ibrahimpatnam, Hyderabad.

[5]Assistant Professor, Department of ECE, Sri Indu College Of Engineering & Technology, Ibrahimpatnam, Hyderabad.

Abstract—

These days, REST APIs are the norm for programmatic access to most online and cloud applications. This article delves into the topic of service compromise via REST API vulnerabilities and how an attacker may potentially exploit them. We provide four security guidelines that include the best features of REST APIs and services. 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.
Topics: REST APIs, cloud computing, security, and test generation

INTRODUCTION

The world of cloud computing is booming. Providers of cloud platforms, such as Amazon Web Services [2] and Microsoft Azure [13], and their customers, who are "digitally transforming" their businesses 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]. Built on top of the widely-used HTTP/S protocol, REST APIs provide a standard method to create, monitor, manage, and remove resources in the cloud. Using an interface-description language like Swagger (now called OpenAPI), developers of cloud services may describe their REST APIs and provide sample client code [25]. Any information about a cloud service's REST API, such as the types of queries the it can process, the possible answers, and the format of those responses, may be found in a Swagger specification. Is it safe to use all those APIs? 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. As a means of discovering flaws, certain tools for testing REST APIs record live API traffic, which is then processed, fuzzed, and replayed [4], [21], [6], [26], [3]. If you want to test the services running behind REST APIs more thoroughly, you might look at stateful REST API fuzzing [5]. In order to find unhandled exceptions (service crashes) that a test client can detect as "500 Internal Server Errors," this approach automatically generates sequences of requests instead of single requests, based on a Swagger specification of a REST API. The goal is to exercise the cloud service deployed behind that API thoroughly. Although there are several new issues reported and the work seems promising, it only covers the detection of unhandled exceptions. In this article, we provide four security guidelines that include the best features of RESTful APIs and services. Consider the use-after-free rule. After deletion, a resource can no longer be accessed. • Rule of resource leakage. A failed resource creation must not only be inaccessible, but it must also not "leak" any side effects into the backend service state. Regulation pertaining to the hierarchy of resources. There can be no way for one parent resource to access another parent resource's kid resource. The regulation pertaining to user-namespaces. It is essential that resources generated in one user namespace cannot be accessed from another. As we'll see in the section below, an attacker could exploit a breach in these rules to do things like compromise the backend service state and make it stop working (Denial-of-Service attack), steal information from other users (Information-Disclosure attack), or even bypass quotas and hijack cloud resources (Elevation-of-Privilege attack). We demonstrate the extension of a stateful REST API fuzzer to test and identify such rule violations. 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. So, in addition to looking for rule violations, each checker also attempts 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 go over how to quickly build each individual checker, by removing likely-redundant tests wherever feasible, since each checker creates additional tests, on top of

an already-large state space exploration. By design, these checks may uncover security rule violations that baseline stateful REST API fuzzing misses (beyond the "500 Internal Server Errors"). Several operational Azure and Office 365 cloud services have new issues discovered using these checks. With security checkers, REST API fuzzing may identify additional kinds of problems with a little incremental testing cost, which boosts its value. The following contributions are made by this paper: We provide guidelines that outline the characteristics of REST APIs' security. • In order to test and identify rule infractions, we create and deploy active checkers. Using three live cloud services, we provide comprehensive experimental findings that assess the efficacy and functionality of these active checkers. We uncovered additional issues in numerous operational Azure and Office-365 cloud services using these checkers, and we outline the security implications of these bugs. Below is the outline for the remainder of the article. Section II provides a review of the history of stateful REST API fuzzing. Section III lays out the principles that should be in place to ensure that REST APIs are safe, and it also provides active checks to ensure that these rules are being followed. Experimental findings using active checkers on production cloud services are presented 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

Here we review what stateful REST API fuzzing is and how it works [5], and then in Section III we provide security property checks that build upon this foundational method. 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. One unique HTTP status code, ranging from 2xx to 5xx, is always appended to each response. The REST API definition language Swagger [25], once known as OpenAPI, is one example. 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 limited collection of requests. A tuple _a, t, p, b_ represents each request r.

- a is an *authentication token*,
- t is the *request type*,
- p is a *resource path*, and
- b is the *request body*.

Any one of the five possible options for request type t in REST is valid: PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), or PATCH (update). One way to identify a cloud resource and its parent hierarchy is by looking at its resource path, which is a string. The regular expression is usually met by p, which is a (non-empty) sequence.

```
(/⟨resourceType⟩/⟨resourceName⟩/)+
```

the kind of a cloud resource is indicated by resourceType, and the name of that resource is given by resourceName. In most cases, the request attempts to create, access, or delete the particular resource that is listed last in the route. In order for the request to be processed correctly, the request body b might include extra parameters and the values they can take. An example request to acquire the attributes of a particular Azure DNS zone is shown below [14] (on many lines):

```
⟨ 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 blank. In general, new resources are created by REST API requests of the PUT or POST type, while old resources are deleted by requests of the DELETE type. A producer for the resource type T is a request whose execution results in the creation of a new resource of that kind. The shorthand for a freshly formed resource is its identification. We sometimes refer to resources as dynamic objects because of the way they are formed. A consumer for the resource type T is a request that includes a resource name of type T in either its route or content. From time to time, we shall talk about the dynamic object type by its resource name, which is type T. No new resources are created by the GET request illustrated in the Azure DNS zone example; instead, three resources of the types subscriptions, resourceGroups, and dnsZones are consumed. Users may define a limited finite collection of specified values—called fuzzable values—to be randomly picked within resource routes or request bodies of individual requests. In the body of a request, a user may indicate that an integer value may be anything from 0 to 10,000,000. A fuzzing dictionary is a collection of values like this. The rendering of a request indicates the mapping of each fuzzable value to a single concrete value taken from its fuzzing dictionary, given that the request contains fuzzable values. Therefore, there are nk alternative renderings for a request with n fuzzable values, each of which may take k possible values. A valid rendering is one that, when executed, produces a legitimate answer, as stated in the next paragraph. Users must provide the values they want to fuzz and the fuzzing dictionaries that go along with them. 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_, indicated as s r→ s_. If a request r in state s prompts a 2xx answer, it is legitimate; if it prompts a 3xx or 4xx response, it is invalid; and if it prompts a 5xx response, it is a problem. 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. Stateful exploration occurs when it tries to access service states that can only be reached by sending a series of requests; for example, in order to exercise more requests and get to deeper service states, later requests in the sequence may consume resources produced by earlier requests. Several search techniques, such as a systematic breadth-first search or a random search, may be used to explore state space [5]. Due to the fact that request sequence length is not limited, sets of alternative renderings may be extremely vast, and the service under test is seen as a blackbox, state spaces can be huge—if not infinite. The good news is that it could be enough to only partially explore the state space to find intriguing issues. The receipt of an HTTP status code of 500 after the execution of a request sequence is considered an error in our environment. 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 refer to executions of request sequences as test cases in the following, and executions of single requests as tests. Here we will refer to the generic state-space exploration technique as the core driver of stateful REST API fuzzing again.

SECURITY CHECKERS FOR REST APIS

Active checks for REST API security rules are defined and described in this section. We begin with the four REST API security rules that are introduced in Section III-A. To test and identify security rule breaches, we detail how to create active checkers in Section III-B. A single kind of security rule violation is the primary focus of each active checker. Section III-C delves into the topic of modular combination, specifically looking at how each checker may be used in conjunction with the primary driver of stateful REST API fuzzing and with each other. We provide a novel search technique for scalable property checker test creation in Section III-D. Section III-E explains how to classify checker breaches so that the user isn't notified of the same issue more than once. Rule No. A. – Security Our four-rule security framework records the best features of REST APIs and services. We talk about the security consequences of each rule and provide examples to back them up. A combination of manual penetration testing and root cause analysis of customer-visible problems led to the discovery of actual defects in deployed cloud services, which served as the basis for all four criteria. In Section V, we will show you some examples of new, undiscovered problems that we discovered in Azure and Office 365 production services as rule violations. The law of use following free consumption. Once erased, a resource can never be recovered. Put simply, any further operations (such as reading, updating, or deleting) on a resource that has been successfully deleted will always fail. If you want to remove the account associated with user-id1, for instance, you may do that by sending a remove request to the URI /users/user-id1. After that, all further attempts to use that ID must fail and return a "404 Not Found" HTTP status code. When an API still has access to a removed resource, it is a use-after-free violation. Seriously, this can't take place. There is an obvious flaw that might corrupt the service backend state and allow users to evade resource limits. A regulation about the loss of resources. An unsuccessfully produced resource must not

only not be available, but also not "leak" any related resources in the backend service state. It means 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 a new resource fails (for whatever reason). In addition, the user should not experience any unintended consequences related to the successful generation of that resource type in the backend service state. When a resource fails to be generated, it shouldn't be counted against the user's service quotas, and the user should be free to reuse the name of the resource. For instance, in order to generate the URI /users/user-id1 with a faulty PUT request, a 4xx answer is required. It is required that any future attempts to access, modify, or delete this URI also fail. The creation of an unsuccessful resource that "leaks" into the backend service state is considered a resource-leak violation. Example: a later GET request lists the resource, but a DELETE request fails to remove it. Attempts to recreate the resource also fail, and a "409 Conflict" answer is returned. The capacity for that resource type (e.g., if resource quota restrictions 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. There can be no way for one parent resource to access another parent resource's kid resource. To rephrase, when a resource is successfully created from another resource and marked as such in service resource paths like _parentType_/parent/_childType_/child/, the child resource cannot be read, updated, or deleted when the parent resource is replaced with another resource. For instance, if you create users user-id1 and user-id2 using POST requests to URIs /users/user-id1, /users/user-id2, and /users/user-id1/reports/report-id1, and then add report report-id1 to user user-id1, then you can't access report-id2 from URI /users/user-id2/reports/report-id1. This is because the resource-hierarchy rule states that report report-id1 belongs to user user-id1 but not to user-id2. The absence of a parent-child link between two resources, even when both originated from the same parent, is a resource-hierarchy violation. If these kinds of breaches are feasible, an attacker may be able to provide an invalid parent object ID.

```
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
```

Fig. 1: **Use-after-free checker.**

(for instance, user-id3), hijack (for instance, report-id1), or steal (for instance, read) an illegal child object. Violating the resource hierarchy is an obvious problem that might cause harm and should never occur. Rule about user-namespaces. You can't have resources from one user namespace available to resources from another. When discussing REST APIs, we take into account user namespaces that are determined by the user token that is used to access the API (for example, OAUTH token-based authentication [18]). If you want to construct a URI /users/user-id1 with a certain token, like token-of-user-id1, but another user wants to access the same resource with a different token, like token-of-user-id2, you can't do it. When a resource that was generated in one user's namespace is accessible from another user's namespace, it is called a user namespace violation. If this kind of breach were to ever happen, the perpetrator might potentially get unauthorized access to another user's resources by making REST API queries using a stolen authentication token. Part B: Active Verifiers The regulations outlined in Section III-A are enforced by means of active checkers. An active checker proposes additional tests to verify that certain criteria are not broken while the primary driver of

stateful REST API fuzzing explores state space. As a result, a proactive checker expands the search area by running additional tests that aim to break certain rules. By contrast, a passive checker does not run any additional checks but instead watches the primary driver search. We use a modular design based on two ideas to create active checkers: 1) The state space exploration of stateful REST API fuzzing is unaffected by checkers since they are separate from the primary driver. 2) Each checker operates independently of the others and creates tests by examining the primary driver's requests, except those that are processed by other checks.

```
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.**

We make sure the first principle is followed by executing all the checkers as soon as the main driver finishes running a new test case. 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). What follows is a presentation of the implementation specifics of each checker along with optimizations to control the growth of state spaces. Utilization verification tool. In Figure 1, you can see the use-after-free rule checker's implementation in notation similar to Python. Figure 4 shows that the main driver executes a DELETE request. The algorithm takes three inputs: a sequence of requests, which is the most recent test case; the global cache of dynamic objects, which contains the most recent object types and ids for all dynamic objects created so far; and the request collection, which is the set of all available API requests.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 most recent type in req_object_types as the one that was really removed, even if the last request could have consumed many types of objects. (The URI /users/userId1/reports/reportId1 consumes two object types—users and reports—but only reports are deleted by a DELETE request.) The for-loop iterates over all requests accessible in reqCollection and skips those that do not consume the target object type (line 14) after this initial setup (line 12). 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 frequently. 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.**

In order to control the amount of extra tests that are created for each request sequence, the inner loop may end after one request for each target object type is detected (line 21), which is optional. In the absence of an exhaustive value for the mode variable, this option is invoked. In Section IV, we provide comprehensive experimental findings on the effects of this optimization. Detector of resource abuse. Figure 2 shows the resource-leak rule checker. Just like the use-after-free checker, this method requires three inputs. Request sequences run by the main driver whose most recent request resulted in an improper HTTP status code in the response are subject to this checker's operation (see to Figure 4). At the outset, the method determines the types of dynamic objects generated by the whole series (seq_obj_types) and the most recent request (target_obj_types) (lines 4 and 5). Encapsulated inside three levels of for loops is the algorithm's core logic. All of the object types returned by the previous request are iterated over in the first loop (line 6). In the second loop, which starts on line 7, the object ids that were "guessed" for the current object type that returned an incorrect HTTP status code are iterated over. 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 object type "x" failed, the checker will try to run any request that uses the "x" object type and will assert that it fails when using the "objx1" object ID. To prevent an explosion in the number of extra tests, each object ID may only have a maximum of a user-provided parameter value of guessed values. 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 through all of the requests in the request collection until it finds one that consumes the supplied target object type and is executable (based on the object types created by the current sequence). On line 17, the queries are carried out with the help of the "guessed" object identifiers that were previously stored in the global cache. The algorithm's goal is to

```
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.**

cause a violation of resource leakage (refer to Section III-A) or state that no violation of resource leakage happens for the specified request sequence (line 18). Lastly, on line 19, the inner loop ends (optionally) when one request for each estimated item is discovered, limiting the amount of subsequent tests created for each input sequence. The optimization is assessed in Section IV. Hierarchy-checker for resources. Figure 3 shows the rundown of how the resource-hierarchy rule checker was implemented. Two arguments are sent into the algorithm: the current global cache of dynamic objects (global_cache) and a series of requests (seq), which represents the most recent test case run by the primary driver. Line 6 indicates the object types eaten by the most recent request in the current series, and line 7 indicates the object types consumed by all requests in the sequence prior to the previous request. This information is recorded by the algorithm as target_obj_types. Following that, on lines 12 and 13, the identifiers of the objects that were used just by the last request are kept locally. By running requests that attempt to access them using incorrect parent objects, the checker will try to breach the hierarchy of these child objects. With that in mind, the present sequence is run until the final request (but not including it) at line 15. The final step is to restore the old child object ids (lines 18 and 19) and then use them in conjunction with the new parent object ids (line 20) to perform the last request. You can't use these parent object identifiers with the restored child object identifiers. In this approach, the program either claims that the requested sequence does not violate the resource hierarchy (line 21) or attempts to induce a resource-hierarchy violation (see Section III-A). Namespace and user verification tool. We don't present this checker in detail because of space limitations. To summarize, this checker tries to use a different authentication token to re-execute the legitimate last request of each test case that the main driver has performed. If this works, a user namespace violation (refer to Section III-A) is reported, and an attacker with a different authentication token could potentially hijack the objects used in the last request. C. Bringing All Checkers Together Here are the steps to execute the four checkers that were described before. The code in Figure 4 is called whenever the stateful REST API fuzzer enters a new state, as defined in Section II. This code implements the state-specific checkers in response to the most recent request. We will now go over some key features of these checkers and how they work together.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. Because of this, it is evident that they enhance the main driver's bug-finding skills. They are able to detect faults that the main driver alone would miss. Active property checking vs passive monitoring. In order to trigger and identify particular rule violations, our defined checks add new test cases to the search area that the primary driver explores, as previously said. On the other hand, without running those additional tests, passive runtime monitoring of these rules in tandem with 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 not discover use-after-free and resource-leak rule violations, respectively. Passive monitoring would also miss resource-hierarchy and user-namespace rule breaches as the baseline main driver doesn't try to swap object IDs or authentication tokens, respectively. 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. We describe four checkers that work well together; due to the fact that their preconditions are mutually incompatible, no two checkers can ever provide identical new tests. To begin, request sequences that conclude with a DELETE request activate no other checkers beyond the use-afterfree checker. Second, if the most recent request's HTTP status code is invalid, just the resource-leak checker will be engaged. Thirdly, request sequences that do not conclude with a DELETE request have the resource-ownership checker triggered as the sole other checker. Finally, the user-namespace checker obviously adds another orthogonal dimension to the state space as it ran tests with an attacker token that was distinct from the authentication token used by the main driver and all the other checks. D. Methods for Finding Checkers In stateful REST API fuzzing, a breadth-first search (BFS) is used to generate tests. This search method encompasses all potential request sequences in the search space. With this search approach, you can be sure that every conceivable grammar rendering of a single request, as well as every possible request sequence up to a certain length, will be covered. But, when the length of the sequence grows, the search does not scale effectively since BFS usually explores a huge search space. This led to the development of BFS-Fast, 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, when the search depth grows to a new number n + 1. This is in contrast to BFS [5]. While BFSFast does provide comprehensive grammar coverage for individual requests, 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. This, however, reduces the amount of infractions that the security auditors can actively verify. We provide BFS-Cheap, a novel search approach, to address this constraint. For a particular sequence length, BFS-Cheap investigates all potential request sequences, but not with all possible renderings. This is in contrast to BFS-

Fast, which covers all possible request renderings at every stage. In particular, BFS-Cheap handles the following inputs: a collection of requests (reqCollection) and a set of sequences (seqSet) of length n: Add all the requests from the reqCollection to the end of each sequence seq in seqSet, run the new sequence taking all the potential renderings of req into consideration, and add no more than one valid and one invalid rendering to seqSet for each sequence. The resource-leak checker uses faulty renderings, but the use-after-free, resourcehierarchy, and user-namespace checks use valid ones.For an experimental assessment, see Section IV-B; BFS-Cheap is therefore a compromise between BFS and BFS-Fast.Like BFS, it searches for all potential request sequences up to a certain length, but unlike BFS-Fast, it adds no more than two additional renderings to each sequence to prevent it from becoming a gigantic seqSet. With two additional renderings for each sequence examined, all the security requirements specified in Section III-A may be actively checked, and the number of sequences in seqSet remains manageable even as the length of the sequence rises. It should be noted that the "cheap" suffix is derived from the fact that BFS-Cheap is a less expensive variant of BFS. In this variant, the BFS "frontier" setSeq only receives one correct rendering for each new sequence. As a result, less resources are generated compared to when all possible interpretations of each request sequence are investigated, as in BFS. As an example, consider a request description that specifies 10 distinct kinds of a single resource type using an enum type. After producing a resource of a single flavor, BFS-Cheap will cease further resource creation. When compared, BFS and BFS-Fast will produce ten identical resources with ten distinct flavors. Substantiation of Bugs A definition of the bucketization scheme used to group similar violations is provided before we go into examples of actual violations found with active checkers. We call rule infractions "bugs" when discussing active checkers. There is a unique request sequence for each bug that was executed to trigger it. We generate per-checker bug buckets using the following approach, given this property: You should calculate every nonempty prefixes of the request sequence that occurs once a new problem is discovered.

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.

BFS, BFS-Fast, and BFS-Cheap are compared in Table I. Displayed are the following metrics after one hour of searching: maximum sequence length (Max Len.), total requests (Tests), percentage of tests produced by the main driver (Main), and by all four checkers combined (Checkers). The total number of requests for each API is shown in the second column. The problem is listed from smallest to largest. Put the new sequence into an existing bug bucket if the suffix is already there. If not, create a new bug bucket for the updated sequence. We have distinct, perchecker bug buckets since the failure circumstances are set individually for each rule, but otherwise, this bug bucketization technique is identical to the one in stateful REST API fuzzing [5]. Due to checker complementarity, only one checker may trigger a problem at a time; nevertheless, the main driver and checkers can both trigger the "500 Internal Server Error" issue. The new sequence will only be put to the bug bucket of the primary driver or checker that triggered it once for 500 bugs.

IV. EXPERIMENTAL EVALUATION

Here we detail the outcomes of our trials using three real-world cloud services. In Section IV-A, we've detailed these services and our experimental setup. Section IV-B then compares the three search algorithms outlined in Section III-D. Section IV-C details the findings, which include the total number of rule violations recorded by each checker across all three cloud services and the effects of different optimizations. I. Experimental Environment To protect the privacy of the three cloud services we tested, we have anonymised their names: Azure A and Azure B are two management services provided by Azure [13], whereas O-365 C is a communications service offered by Office365 [16]. The REST API of these three services receives between thirteen and nineteen queries. Among the cloud services we examined, those three stood out due to their size and complexity, so we decided to focus on them. Section V summarizes our overall experience with the various production services that we have tested so far, which

number in the dozen or so. There is a publicly published Swagger specification for every service that is being considered [15]. Following previous work [5], we construct a test-generation language by combining the specifications of each service. Python code that can be executed is used to encode each grammar. Each of the tests described here utilized the same syntax and fuzzing dictionary for a specific API and service. The production of these representations is not haphazard. Our fuzzing tests were conducted using a single-threaded fuzzer on an internet-connected PC. Each service API was accessed via a genuine subscription. There was no need for any additional service expertise or unique test setup. To prevent going over our service quota, our fuzzer contains a garbage-collector, much as in [5]. This garbage-collector gets rid of resources (dynamic objects) that are no longer required. We can only test production services that are live and available to subscribers; however, we do not have access to the inner workings of the services we test. When it gets a response, our fuzzer just looks at the HTTP status code. Requests from the client side are sent to the target services over the internet, and their answers are processed upon receipt. The trials described here are not completely controlled as we have no say over the rollout of these services. But we ran the same tests many times and got the same findings each time. B. Analyzing Rival Search Techniques While using security checkers to fuzz actual services, we now evaluate our new search approach, BFS-Cheap, in comparison to BFS and BFS-Fast. Using Azure A, Azure B, and O-365 C as our descriptors, we provide the outcomes of our trials with three different Azure services. According to Table I, we ran separate tests using each of the three search techniques on each service for a certain amount of time. Our reports for each experiment include the following metrics: total API requests (Total Req.), maximum sequence length (Max Len.), number of tests, percentage of requests sent by main driver (Main.) and active checkers (Checkers.), and individual checker contributions (Checkers.). According to Table I, out of all the services, BFS reaches the lowest depth, BFS-Fast the highest, and BFS-Cheap, which is closer to BFS than BFS-Fast, offers a compromise between the two extremes. Based on how quickly each service responds, the total number of tests created varies across all of them. With the exception of BFSFAST with Azure B and O-365 C, where the overall number of tests increases substantially, this number stays relatively constant for any particular service. If this growth is true for O-365 C, then

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. Our fuzzer, as the client, receives these unsuccessful requests with longer wait times. It is well-known that services may throttle future requests—that is, attempt to slow them down—by delaying answers to unsuccessful requests. When it comes to Azure B, BFS-Fast goes through with 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 a large number of DELETE requests, which are quicker to run (their replies are returned practically quickly). While BFS and BFS-FAST have the largest and lowest overall percentage of checker tests (Checkers), respectively, BFSCheap falls somewhere in the middle. As mentioned in Section III-D, the reason for inventing BFS-Cheap was to address the fact that BFS-Fast produces more tests than its competitors, but that it prunes its search area and engages checkers less often. One notable exception is the 33% increase in BFS-generated tests for O-365 C. The increased number of checker tests seems to be a direct result of the higher volume of successful requests, as mentioned in the preceding paragraph. According to the data in Table I, the amount of tests generated by each checker differs among services. This figure is dependent on the use-after-free checker's number of DELETE requests, the resource-leak checker's number of unsuccessful resource-creation requests, and the resource-hierarchy checker's depth of the object hierarchy. The user-namespace checker, on the other hand, is the most often activated and accounts for the bulk of the tests created by the checker. What follows is a discussion of how the number of bugs discovered for all three services is almost same across all three search methodologies. C. Evaluating Alternative Checker Methods After introducing the optimized and exhaustive modes in Section III, we will now compare their performance. For each of the four checkers and the main driver from Section II, Table II displays the total number of requests

issued during an hour of fuzzing with BFS-Cheap in the Tests column. You can see in the chart that the primary driver and each of the testers uncovered a certain number of distinct defects, or "bug buckets," in an hour of searching. Both the optimal and exhaustive modes, which were previously mentioned, have their results given. The amount of tests differs among services and checker settings, as we can see. Nevertheless, it is predictable that the exhaustive mode always generates a greater proportion of tests from the checkers. The primary driver is free to explore more states at once in the optimized mode since the checkers generate fewer tests for each visited state. For all three services, the optimized mode still detects all the unique issues (bug buckets) discovered by the exhaustive mode, even though there are less checker tests per visited state. When comparing the optimal and exhaustive search modes for the O-365 C service, the primary driver discovers an additional problem within one hour of searching in the optimized mode. Table II shows an intriguing inversion that shows how useful the optimized checkers mode is even more. When running on Azure A, we see that the optimized mode generates almost twice as many tests as the exhaustive mode (4050 vs 2174). Intuitively, this seems backwards. Our research led us to the conclusion that the exhaustive mode of the user-namespace checker generates tests with much longer response times for Azure A. In fact, when run in exhaustive mode, this particular checker performs more tests than when run in optimized mode; nonetheless, the total test throughput is negatively impacted by the presence of costly activities, namely those with high latency. All all, we discovered and reported seven distinct problems to the service developers while conducting tests with these three offerings; the primary driver alone detected 4,500 issues, while each of the checkers—with the exception of the user-namespace checker—found three. Several intriguing bugs discovered by the checkers proposed in this research will be covered in the part that follows.

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. We found a few new issues in each of these services by fuzzing in virtually every instance. Our new security checkers have identified rule violations in around one third of the issues, whereas "500 Internal Server Errors" account for almost two-thirds of the defects. The service owners were notified of these issues, and they have all been resolved. Even if security testers don't identify any problems, they boost trust that the rules they verify cannot be broken, leading to increased confidence in the overall dependability and security of the service. What follows is a discussion of the security implications of real-world instances of vulnerabilities discovered in Azure and Office 365 services that have been implemented. 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. We discovered the following use-after-free violation in an Azure service. Initiate the creation of resource R by submitting a PUT request. A DELETE request should be used to delete resource R. Third, using a separate PUT request, make a copy of the deleted resource R that is of a certain kind. An error message stating "500 Internal Server Error" is produced by this series of queries.Because (1) it tries to re-use the deleted resource in Step 3 and (2) the result from 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 component that allows users to compose messages, respond to them, and modify them. 1) Make a first message called msg-1 using the POST request to /api/posts/msg-1. 2) Make a new message called msg-2 and send it using the POST request to the address /api/posts/msg-2. thirdly, using the POST request to /api/posts/msg-1/replies/reply-1, create a reply-1 to the initial message. Fourth, use the message identification msg-2 in a PUT request to edit reply-1 with the following parameters: /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. Potential security flaws include missing validation checks for hierarchy. An attacker might use them to access child items by avoiding the parent hierarchy. Azure resource leak occurred. This issue occurred in another Azure service when the resource-leak checker was used. 1) With a PUT request, create a new resource with the name X and type CM. The resource should have a certain malformed body. An issue in and of itself, this causes a "500 Internal Server Error" to be returned. 2) If you want a list of all CM resources, you'll get an empty list. Thirdly, 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. An unexpected "409 Conflict" rather than the anticipated "200 Created" is returned by the last request in Step 3. Because of the unanticipated consequences of the unsuccessful request in Step 1, the service has achieved an inconsistent state, as shown by this behavior. That the user's perception is accurate is shown by the GET request in Step 2; the CM resource with the ID X that was tried to be created in Step 1 has not been generated. Nevertheless, the service's ability to recall the unsuccessful attempt to create the CM resource X in

the first PUT request in Step 1 is shown by the second PUT request in Step 3. The attacker could theoretically construct an infinite number of these "zombie" resources by repeatedly running Step 1 with new names. This would allow them to surpass their official limit, since unsuccessful resource creations are (correctly) not counted against the user's quota. However, it is evident that they are remembered (incorrectly) by some component of the backend service. Another Case in Point: An Exuberant Denial-of-Service Attack on Resources. We inadvertently caused another Azure service's health to drastically decline after five hours of fuzzing. Here, we compile the results on the source of the problem. To avoid going overboard on cloud resource limitations, our fuzzing program employs a trash collector. The number of active resources can never exceed quotas since our garbage collector deletes (using a DELETE request) resources that are no longer needed. For example, if a default quota for a resource type Y is 100, then no more than 100 of that type may be generated at any one time. Our fuzzing tool usually reaches its quota restrictions in minutes and can't continue exploring state space without trash collection. Any PUT request to create a resource of a certain type—let's call it IM—in this Azure service really activates additional processes that take minutes to finish in the service backend, but it delivers a response fast. The same holds true for IM resources; a DELETE request for one provides the desired result in a flash, but it also initiates deletion processes that may take several minutes to finish. Nevertheless, these PUT and DELETE requests for IM resources update quota counts aggressively and too rapidly, without waiting for the many minutes really required to complete the job. Because of this, an attacker might build and remove IM resources rapidly without going over their limit, which would trigger a deluge of backend jobs and essentially overload the backend server. Our fuzzing tool unintentionally set off this Denial-of-Service attack. In order to address this issue, it is recommended to wait a few minutes after all delete backend operations have finished, especially for IM resources, before updating use counts towards quotas for DELETE requests. So long as previous DELETE requests are completely processed before future IM resource-creation PUT requests can be processed, the official quota will continue to linearly limit the quantity of backend jobs.

VI. RELATED WORK

Extending stateful REST API fuzzing is what we do here [5]. A REST API's Swagger specification may be used to automatically construct request sequences that adhere to the standard by compiling the specification into a fuzzing language. Instead of the user having to manually build a language like in classic grammar-based fuzzing [20], [22], [24], stateful REST API fuzzing automates the development of a fuzzing grammar. In model-based testing, test generation techniques are used to create minimum test suites that include a complete finite-state machine model of the system being tested. These algorithms serve as an inspiration for the BFS and BFS-Fast search methods [27], [12], [28]. This paper adds to the stateful REST API fuzzing framework by doing two things: (i) creating a set of security rules for REST APIs and matching checkers to efficiently test and detect when these rules are violated, and (ii) creating a new search strategy called BFS-Cheap that provides a compromise between BFS and BFS-Fast when active checkers are used. You can use HTTP-fuzzers to test REST APIs since all of their requests and replies go over 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 interpret HTTP requests sent via REST APIs and direct their fuzzing, many tools that record, analyze, and replay HTTP traffic have since been updated to make use of Swagger standards [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 novel request sequences. This is because their fuzzing is stateless. So, it's not a good idea to add active checks to stateless fuzzers. Our approach, on the other hand, adds active checks that target particular REST API rule breaches to 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 entire web pages with HTML and Javascript code are returned. But most REST APIs don't provide web pages in their answers, so those checking skills aren't useful for them. Our study presents new security criteria that are tailored to REST API use, in contrast to HTTP-fuzzers and web scanners. These regulations are relevant to security because an adversary might potentially utilize their infractions to compromise a service's integrity or get sensitive data or resources without authorization. On the other hand, we don't go into detail on how to verify other REST API use criteria [9] in this work. For example, we don't cover request idempotence, which means that sending the same request many times won't change the result. Surprisingly, there is a lack of documentation on how to utilize REST APIs securely, despite their popularity. Organizations such as OWASP [19] (Open Web Application Security

Project) and publications on REST APIs [1] or micro-services [17] devote the majority of their security recommendations to topics related to API key and authentication token management. There is a lack of specific instructions on how to manage resources and validate inputs using the REST API. As far as anybody can tell, this study is the first to propose four new security rules. Section III utilized the phrase "active checker" from [10] to indicate that our checkers create new tests to detect rule violations, rather than just monitoring API request and response sequences as in conventional runtime verification [8], [11]. We employ several separate security checkers all at once, much as in [10]. However, we do not create additional tests by symbolic execution, constraint formation, or solution, as was done in [10]. 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. Pen testing, which involves security professionals reviewing the architecture, design, and code of cloud services from a security standpoint, is the major approach used today to assure the security of cloud services. Pen testing is costly, has limited reach, and requires a lot of human effort. 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.

VII. CONCLUSION

To ensure that REST APIs and services retain their desired qualities, we implemented 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. Up to this point, we have used the fuzzer and checkers outlined in this article to fuzze around twelve production Azure and Office-365 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. We notified the service owners of all the issues, and they have all been resolved. Vulnerabilities in security may be easily identified when the four criteria presented in this article are violated. The service owners have all taken the issues we detected seriously; as a result, our bug "fixed/found" ratio is almost 100% at the moment. Also, it's better to repair these problems now than to risk a real catastrophe, which may be caused by an attacker or happen accidentally, and the results would be unpredictable. Lastly, the fact that our fuzz testing method does not produce any false positives and that these errors are repeatable is helpful. On what scale do these findings apply? This can only be discovered by thoroughly testing additional services via their REST APIs and inspecting a wider range of attributes for various types of faults and security issues. Unexpectedly, there is a lack of security-related guidelines about the use of REST APIs, despite the recent proliferation of these APIs for use in cloud and online services. In this regard, our study contributes four rules whose infractions are important to security and which are not easy to verify and resolve.

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 RESTAPI Fuzzing. In *41st ACM/IEEE International Conference on SoftwareEngineering (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 *Proceedingsof the 2000 SPIN Workshop*, volume 1885 of *Lecture Notes in ComputerScience*, pages 323–330. Springer-Verlag, 2000.
[9] R. T. Fielding. Architectural Styles and the Design of Network-basedSoftware 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 Conferenceon Embedded Software)*, pages 207–216, Atlanta, October 2008. ACMPress.

[11] K. Havelund and G. Rosu. Monitoring Java Programs with JavaPathExplorer. In *Proceedings of RV'2001 (First Workshop on RuntimeVerification)*, volume 55 of *Electronic Notes in Theoretical ComputerScience*, Paris, July 2001.
[12] R. L¨ammel and W. Schulte. Controllable Combinatorial Coverage inGrammar-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.
[16] Microsoft. Office. https://www.office.com/.
[17] S. Newman. *Building Microservices*. O'Reilly, 2015.
[18] OAuth. OAuth 2.0. https://oauth.net/.
[19] OWASP (Open Web Application Security Project). https://www.owasp.org.
[20] Peach Fuzzer. http://www.peachfuzzer.com/.
[21] Qualys Web Application Scanning (WAS). https://www.qualys.com/apps/web-app-scanning/.
[22] SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automationwith-spike/.
[23] Sulley. https://github.com/OpenRCE/sulley.
[24] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force VulnerabilityDiscovery*. Addison-Wesley, 2007.
[25] Swagger. https://swagger.io/.
[26] TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.
[27] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-BasedTesting Approaches. *Intl. Journal on Software Testing, Verification andReliability*, 22(5), 2012.
[28] M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedingsof the 23rd Annual ACM Symposium on the Theory of Computing*,pages 476–485, 1991.