ISSN: 2321-2152

IJMECE

International Journal of modern electronics and communication engineering

E-Mail
editor.ijmece@gmail.com
editor@ijmece.com

www.ijmece.com

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

Verifying the REST API Security of Cloud Services

[1] B. Keerthi Reddy, [2] V. Naveen Kumar, [3] K. Nikhil, [4] M. Gopi Priya, [5] P. Nikith, [6] Mr. Dandu Srinivas, [7]Dr. C. Sasikala

[1,2,3,4,5]UG Scholar, Dept. of CS, Narsimha Reddy Engineering College, Maisammaguda, Kompally, Secunderabad, India.

[6] Assistant Professor,Dept.of CSE, Narsimha Reddy Engineering College, Maisammaguda, Kompally, Secunderabad, India.

[7] Professor, Dept. Of EEE, Narsimha Reddy Engineering College, Maisammaguda, Kompally, Secunderabad, India.

Abstract—

These days, REST APIs are the norm for programmatic access to most online and cloud applications. In this article, we will look at how a malicious actor may take over a service by taking advantage of security holes in its REST API. In order to capture the desired qualities of REST APIs and services, we provide four security principles.

To further automate testing and detection of rule violations, we demonstrate how to add active property checks to a stateful REST API fuzzer. We go over several efficient and modular ways to create such checks. 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

I. INTRODUCTION

A new era of cloud computing is dawning. Cloud platform providers such as Amazon Web Services [2] and Microsoft Azure [13] have deployed thousands of new cloud services in the past few years. Their customers are "digitally transforming" their businesses by modernizing their processes and collecting and analyzing all kinds of new data. The majority of cloud services may now be accessed programmatically using REST APIs [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. Swagger, formerly known as OpenAPI, is an interface-description language that cloud service developers may use to describe their REST APIs and provide sample client code [25]. A Swagger specification describes how to contact a cloud service using its REST API, including what queries the service can accept, what replies may be received, and the response format.

How secure are all those APIs? Today, this question is still largely open. Tools for automatically evaluating cloud services via their REST APIs and assessing whether these services are dependable and safe are still in their infancy. Some tools available for testing REST APIs collect live API communication, and then analyze, fuzz, and replay the data with the hope of detecting flaws [4], [21], [6], [26], [3]. Recently, stateful REST API fuzzing [5] was suggested to precisely test more deeply services exposed behind REST APIs. Given a Swagger specification of a REST API, this technique automates IN order to comprehensively exercise the cloud service deployed behind that API, with the purpose of identifying unhandled exceptions (service failures) that may be identified by a test client as “500 Internal Server Errors”. While that effort appears promising and reports numerous new issues detected, its scope is constrained to the detection of unhandled exceptions.

In this work, we offer four security principles that capture desirable aspects of REST APIs and services.

• Use-after-free rule. A resource that has been deleted must no longer be accessible.

• Resource-leak rule. In order to avoid any potential consequences in the backend service state, a resource that could not be generated successfully should not be available.

• The rule of resource hierarchy. No two parent resources can have access to the same child resource. The rule pertaining to user-namespaces. Nobody else should be able to access resources made in one user's namespace.

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 lead to the hijacking of cloud resources or the bypassing of quotas. Here we demonstrate how to add support for testing and detecting rule violations in a stateful REST API fuzzer. We establish an active property checker for every rule that does two things: (1) finds rule violations and (2) generates new API requests to test them. Thus, in addition to looking for rule violations, each checker also tries to break its own rule.

To ensure that these checkers do not conflict with one another, we go over possible modular implementations. We also go over ways to efficiently implement each individual checker, by removing likely-redundant tests whenever possible, because each checker creates new tests, on top of an already-large state space exploration.

By design, these checkers can uncover security rule violations that baseline stateful REST API fuzzing misses (beyond the "500 Internal Server Errors"). Several production Azure and Office 365 cloud services had new bugs discovered using these checkers. By identifying more kinds of errors at a small incremental testing cost, security checkers raise the utility of REST API fuzzing.

Here are some important points that this paper brings up:

Here, we lay out the groundwork for active checkers by defining rules that describe the security properties of REST APIs. We then create and deploy these checkers to test and detect rule violations. Finally, we provide comprehensive experimental results that evaluate the effectiveness and performance of these active checkers on three production cloud services. We discovered new vulnerabilities in various production Azure and Office 365 cloud services using these scanners, and we talk about the security implications of these vulnerabilities.

The rest of the paper is organized as follows. In Section II, we recall background information on stateful REST API fuzzing. In Section III, we introduce rules that capture desirable properties of secure REST APIs and present active checkers to test and detect violations of these rules. In Section IV, we present experimental results with active checkers on production cloud services. In Section V, we discuss new bugs found by these checkers and their security implications.

In Section VI, we discuss related work, and we conclude the paper in Section VII.

II. STATEFUL REST API FUZZING

Section III introduces security property checks that may be used as expansions of this fundamental approach; in this section, we review the notion of stateful REST API fuzzing [5].

When it comes to cloud services, we think REST APIs are the way to go.

Requests are messages sent by a client software to a service, while replies are messages received back. This kind of communication is carried out using the Hypertext Transfer Protocol, Secure. There is a unique HTTP status code (2xx, 3xx, 4xx, or 5xx) assigned to each answer.

One example of a specification language for REST APIs is Swagger [25], which is also called OpenAPI. The Swagger standard details the REST API access to a service, including the types of queries that the service may process, the possible answers, and the format for each.

Using a limited number of queries, we characterize a REST API. Every request r is a tuple that contains the following elements: an authentication token (a), the kind of request (t), a resource path (p), and the request content (b).

There are five possible options for request type t in REST: PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), and PATCH (update). One way to identify a cloud resource and its parent hierarchy is by looking at its resource path, which is a string. For most cloud resources, p is a non-empty sequence that matches the regular expression (/_resourceType_/_resourceName_/)+, where resourceType is the type of the resource and resourceName is the name of the individual resource of that kind. In most cases, the request attempts to create, access, or delete the particular resource that is listed last in the route. Additional parameters and their values may be included in the request body b. These parameters can be necessary or optional, depending on the request.You need to include three resource names—a subscriptionID, a resourceGroupName, and a zoneName—in the route of this GET request, and the body (at the end, represented by { }) is blank.

While DELETE requests remove resources from an API, PUT and POST requests usually add new ones. A producer for the resource type T is a request whose execution results in the creation of a new resource of that kind. An identifier, or "id," is a representation of a freshly formed resource. We sometimes refer to resources as dynamic objects because of the way they are formed. An example of a consumer request

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

would be one that includes a resource name of type T either in 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. Fuzzing dictionaries include such sets of values. 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. With n fuzzable values that may take on k potential values each, the number of possible renderings is nk. When the matching request is executed and provides a valid response (specified in the next paragraph), we say that the rendering is legitimate. The fuzzing dictionaries and values that users choose to fuzz are entirely up to them.

A directed graph with nodes representing service states and edges representing transitions between them is what we call a service's state space. Executing a single request r from a given state s of the service results in a successor state s_, which is represented by the expression 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.

The state space of the service that can be reached from an initial state when no resources exist may be explored by performing sequences of requests. When this kind of investigation tries to access service states that can only be reached by sending out a series of queries, it is being stateful:

The resources used in later requests in the same sequence might be generated by previous ones, allowing for more requests to be executed and deeper service states to be reached.

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. When we obtain a 500 HTTP status code after running a request sequence, we consider it a bug. It is safer to address these issues rather to risk a live event with unknown effects. Unhandled exceptions, such as "500 Internal Server Errors," are caused by unexpected input request sequences and may corrupt the service state and severely harm its health.

We will sometimes refer to executions of request sequences as test cases in the following, and executions of single requests as tests. The primary mechanism behind stateful REST API fuzzing is the basic state-space exploration technique discussed in this section.

III. 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. We lay out the steps to build active checkers for testing and finding security rule breaches in Section III-B. There is just one kind of security rule violation that each active checker targets. 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 test creation with property checkers 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
In order to capture the desired qualities of REST APIs and services, we provide four security principles. We provide an example for each rule and talk about the security implications of it. 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. Later in Section V, we will provide examples of new, previously undiscovered problems that we discovered as rule violations in production Azure and Office 365 services that were already deployed. The rule of use after free. After deletion, a resource can no longer be accessed. That is to say, any further operations (such as read, update, or delete) on a resource that has been successfully deleted will always fail.
If you want to remove the account associated with

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

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 a removed resource is still available via the API, it is considered a use-after-free violation. Seriously, this can't take place. The service's backend state might be corrupted and resource quotas could be bypassed due to this obvious fault. 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. That is to say, if a 4xx error occurs as a result of an unsuccessful PUT or POST request to create a new resource, then all further operations on that resource will likewise fail. In addition, the user should not experience any unintended consequences related to the successful generation of that resource type in the backend service state. To illustrate, a resource that was unable to be established cannot be used to meet the user's service quotas, and the user must be allowed to reuse the name of the resource. For instance, in order to generate the URI /users/user-id1 with a faulty PUT request, a 4xx answer is required. It is required that any future attempts to access, modify, or delete this URI also fail. When a resource that was not correctly generated yet "leaks" some side-effect in the backend service state, it 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. Never let this happen since it might have unforeseen effects on the service's performance (because of excessively big database tables, for example) or the capacity of that resource type (because additional resources can't be added because resource quota limitations have been surpassed). Resource-hierarchy rule. No two parent resources may have access to the same child resource. Put simply, when a resource is successfully created from another resource and marked as such in service resource paths, the child resource must not be accessible when the parent resource is replaced with another resource. This means that the child resource cannot be read, updated, or deleted. 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, according to the resource-hierarchy rule, 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. When this kind of breach is feasible, an adversary may be able to provide an illegal parent object identity.

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

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

(for example, user-id3), and then take control of an illegal child object (for instance, report-id1) by reading or writing to it. It is imperative that no bugs involving resource hierarchy ever occur; doing so could put users in harm's way. Policy regarding user-namespaces. You can't have resources from one user namespace available to resources from another. The user token used to interact with the API (e.g., OAUTH token-based authentication [18]) defines the user namespaces in the context of REST APIs. For example, after submitting a POST request to build URI /users/user-id1 using token token-of-user-id1, resource user-id1 must not be accessible using another token token-of-user-id2 of another user. A user namespace violation occurs when a resource created within the namespace of one user is accessible from within the namespace of another user. If such a violation ever occurs, an attacker might be able to execute REST API requests using an unauthorized authentication token, and perform unauthorized operations on resources belonging to another (victim) user.

B. Active Checkers

We implement active checks for the rules provided in Section III-A. An active checker monitors the state space exploration performed by the main driver of

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

stateful REST API fuzzing and suggests new tests to assert that specific rules are not violated. Thus, an active checker augments the search space by executing new tests targeted at violating specific rules. In contrast, a passive checker monitors the search performed by the main driver without executing new tests.
We design active checkers following a modular design based on two principles:
1) Checkers are independent from the main driver of stateful REST API fuzzing and do not affect its state space exploration.
2) Checkers are independent from each other and generate tests by analyzing the requests executed by the main driver, excluding those executed by other To make sure that the first principle is followed, we make sure that all the checks are executed once 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). The next sections discuss the implementation of each checker and provide improvements to reduce the expansion of state spaces.
Checker for use after free. In Figure 1, you can see the use-after-free rule checker's implementation in notation similar to Python. After the main driver executes a DELETE request (see Figure 4), the algorithm is called and takes three inputs: a sequence of requests (seq of requests), the global cache of dynamic objects (global_cache), and the set of all available API requests (reqCollection). The global cache contains the most recent object types and ids for the dynamic objects created so far. checkers.

```
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.**

Line 5 retrieves the kinds of the dynamic objects consumed by the last request, and a temporary variable named target_obj_id is created to record 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. For instance, a DELETE request to the following URI: /users/userId1/reports/reportId1 consumes two sorts of objects: reports and users, but it only deletes reports. Following this initial setup, the reqCollection is iterated over by the for-loop (line 12), which skips requests that do not consume the target object type (line 14).
The method EXECUTE (line 19) uses the recovered target object id from the global cache of dynamic objects (line 17) to execute request req once it finds a request, req, that consumes the target object type. In order to execute a request, the EXECUTE function checks global cache for object identifiers, therefore the target object id is restored there several times. 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 depicts the

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

resource-leak rule tester. The three parameters used by the method are identical to those of the use-after-free checker. This checker processes request sequences carried out by the primary driver, whose most recent request resulted in a response with an incorrect HTTP status code (see to Figure 4 for reference). The first thing the algorithm does is find out what kinds of dynamic objects were created by the previous request (target_obj_types) and the full series (seq_obj_types) (lines 4 and 5). Three layered for loops implement the algorithm's core logic. Line 6 of the first loop contains an iteration of all object types that were generated by the previous request. On line 7, there is a second loop that iteratively checks the object ids that were "guessed" for the kind of item 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.
As an example, the checker will try to run any request that uses the object type "x" and state it fails when using the object id "objx1" if the creation of a dynamic object with object id "objx1" and object type "x" fails via the API (based on the answer obtained). In order 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.
Line 8 temporarily adds an object-id value that was guessted to the global cache of dynamic objects that were appropriately created.
Based on the object types generated by the current sequence, the inner loop (line 9) iteratively searches through the requests in reqCollection for those that are executable and consume the specified target object type. On line 17, the "guessed" object ids that were previously stored in the global cache are used to perform these queries. In this manner, the algorithm attempts 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, reqCollectior
9  else:
10         ResourceHierarchyChecker(seq, global_cache)
11         UserNamespaceChecker(seq, global_cache)
```

Fig. 4: **Checkers dispatcher.**

either claims that the provided request sequence does not include a resource-leak violation (line 18) or triggers one (see Section III-A).
When one request for each estimated item is identified, the inner loop (optionally) finishes (line 19), which limits the number of subsequent tests created for each input sequence. Section IV assesses this optimization.
Checker for resource hierarchy. Figure 3 shows the resource-hierarchy rule checker in action. A sequence of requests (seq) representing the most recent test case run by the primary driver and the current global cache of dynamic objects (global_cache) are the two inputs that the algorithm takes into account. 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. Lastly, on lines 18 and 19, the old child object ids are restored. Then, on line 20, the final request is completed using both the new parent object ids and the old child object ids. The restored child object ids do not belong to these parent object ids.
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. Space limitations prevent us from providing a comprehensive description of this checker. 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 is successful, an attacker using a different authentication token might potentially take control of the objects used in the previous request, leading to the reporting of a user namespace violation (see to Section III-A for further information).
C. Making All Checkers Into One
Here is how the four checkers mentioned before are put into action. The code in Figure 4 is called whenever the stateful REST API fuzzer enters a new state, as stated in Section II. This code implements the state-specific checks 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. By doing more tests and checking for replies other than 5xx, the checkers expand the state space and may identify unexpected 2xx responses as rule-violation

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

errors. This extends the core driver of baseline stateful REST API fuzzing. So, it's evident that they improve the main driver's bug-finding skills. They can detect flaws that the main driver couldn't identify on its own.

Active property checking vs passive monitoring. As said before, the checkers we create provide more test cases to the search area the primary driver uses to trigger and identify particular rule violations. But without actually running those extra tests, it's quite unlikely that passive runtime monitoring of these rules alongside the primary driver will pick up on rule breaches. Passive monitoring alone is unlikely to catch use-after-free and resource-leak rule breaches since the primary driver's default state space exploration probably won't try to re-use deleted resources or resources after a failure, 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. Our four defined checkers are mutually supportive; that is, due to the fact that their respective preconditions are incompatible, no two of them can ever provide new tests that are identical to one another. 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. Furthermore, for request sequences that do not conclude with a DELETE request and have valid renderings, the resource-ownership checker is the only checker that is triggered. 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. Checkers Search Methods

For stateful REST API fuzzing, the breadth-first search (BFS) is the primary search technique for test creation in the search space specified by all conceivable request sequences [5]. When it comes to grammar, this search method has you covered. It covers every conceivable request and every potential request sequence up to a certain length. Nevertheless, BFS's search performance degrades with increasing sequence length due to the often huge search area it explores. Hence, BFS-Fast was implemented as an optimization. In contrast to BFS, which adds each request to every request sequence of length n, BFS-Fast only adds each request to one request sequence of length n whenever the search depth grows to a new value n + 1 [5]. Full grammar coverage is only provided by BFSFast for individual requests and not for request sequences of a certain length; this means that not all conceivable requests may be explored. A subset of all potential request sequences is explored by BFS-Fast, which allows it to scale better than BFS.

Sadly, this reduces the amount of infractions that the security auditors can actively verify. We provide a novel search method, BFS-Cheap, to overcome this restriction.

In contrast to BFS-Fast, which aims to cover all potential request renderings at every stage, BFS-Cheap follows the opposite trade-off and instead investigates all possible request sequences for a certain sequence length, but not with all conceivable renderings.

In particular, BFS-Cheap handles the following inputs: a collection of requests (reqCollection) and a set of sequences (seqSet) of length n:

For every sequence seq in seqSet, add every request from reqCollection to the end of seq, run the new sequence taking into account all potential renderings of req, and add no more than one valid and one incorrect rendering to seqSet.

The use-after-free, resource-hierarchy, and user-namespace checking all rely on correct renderings, but the resource-leak checker relies on faulty renderings.

For an experimental assessment, see Section IV-B; BFS-Cheap is therefore a compromise between BFS and BFS-Fast.

To prevent a huge sequenceSet, it investigates all potential request sequences up to a certain length (similar to BFS) and adds no more than two additional renderings to each sequence (similar to BFS-Fast). 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. Keep in mind that the "cheap" suffix is derived from the fact that BFS-Cheap is a cost-effective variant of BFS that adds no more than one acceptable rendering to the BFS "frontier" setSeq for every new sequence. As a result, less resources are generated compared to when all possible interpretations of each request sequence are investigated, as in BFS. 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. On the other hand, BFS and BFS-Fast will generate 10 identical

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

resources but with ten distinct flavors.
Section E. Bug Hole Filling
We describe the bucketization method that is used to put together comparable infractions before we talk about real-world instances of violations identified using active checkers. "Bugs" are rule infractions while discussing active checkers. You can trace each problem back to its specific request sequence. We generate per-checker bug buckets using the following approach, given this property: When a new issue is discovered, calculate all nonempty prefixes in the request sequence that causes

API	Total Req.	Search Strategy	Max Len.	Tests	Main	Checkers	Checker Stats			
							Use-Aft-Free	Leak	Hlerarchy	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.

beginning with the tiniest insect. Put the new sequence into an existing bug bucket if the suffix is already there. In any case, you'll need to create a fresh 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]. 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.
One instance of the new sequence will be added to the bug bucket of the primary driver or checker that triggered it, for 500 bugs.

IV. EXPERIMENTAL EVALUATION

Here we detail the outcomes of our trials using three real-world cloud services. Section IV-A details our experimental setup and these services. 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.

(A) Experimental Environment

Experiments were conducted using three anonymous cloud services, and we provide the results: 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]. All three of these services have REST APIs that get 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]. We follow previous work [5] by assembling the specifications of each service into a language for test creation. Python code that can be executed is used to encode each grammar.

All the tests presented here utilized the same syntax and fuzzing dictionaries for a specific service and API. The produced representations are not random. A single-threaded fuzzer, an internet-connected PC, and a current service membership that grants access to each service API were used to conduct our fuzzing studies. There was no need for any additional service expertise or unique test setup. In the same way as in [5], our fuzzer incorporates a garbage-collector that gets rid of dynamic objects that are no longer in use so that we don't go over our service quota.

While 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. When it gets a response, our fuzzer just looks at the HTTP status code. The client side initiates all requests by sending them over the internet to the specified services, and then parses the results. 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. Examining Different Search Methods

Now we'll take a look at how BFS-Cheap, our new search technique, stacks up against BFS and BFS-Fast when it comes to fuzzing actual services using security checkers. Using Azure A, Azure B, and O-365 C as our descriptors, we provide the outcomes of our trials with three different Azure services.

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

Table I displays the results of separate tests conducted on each service using the three search algorithms, with a time budget of one hour each trial. We document the following metrics for every experiment: 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 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. Based on how quickly each service responds, the total number of tests created varies across all of them. The overall number of tests grows dramatically for BFSFAST with Azure B and O-365 C, while for all other services, this number stays relatively consistent. 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 handles these two services with a much lower failure rate 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).
As before, BFSCheap falls somewhere in the middle, with BFS having the greatest overall percentage of checker tests (Checkers) and BFS-FAST having the lowest. 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. More successful requests (refer to the preceding paragraph) likely caused more checker tests, which is why there was such a surge.
We can see that the amount of tests generated by each checker differs between services from the information in Table I. How many use-after-free requests were conducted, how many resource creation requests failed, and how deep the object hierarchy is for the resource hierarchy checker all contribute to this amount. 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.
Next, we'll talk about how the three search algorithms yielded roughly identical bug counts for all three services.

C. Evaluating Alternative Checker Methods

Following our discussion of the two modes in Section III, we will now compare their performance. The Tests column in Table II displays the total number of requests issued during an hour of fuzzing using BFS-Cheap. The proportion of requests produced by the main driver of Section II or any of the four checks is also shown. 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. The outcomes for both the optimal and exhausting modes are shown.
The amount of tests differs among services and checker settings, as we can see. As anticipated, nevertheless, the exhaustive mode consistently produces a larger proportion of tests created by the checkers. Optimal mode allows the primary driver to explore more states quicker by reducing the number of tests produced per visited state by the checkers. Even though there are fewer checker tests per visited state in optimized mode, all three services still detect all the unique issues (bug buckets) identified by exhaustive mode. In addition, the primary driver discovers an additional issue using the optimized mode for the O-365 C service after an hour of searching compared to the exhaustive mode. The significance of the optimized checkers mode is further shown by an intriguing inversion that is shown in Table II. Our observations in Azure A reveal that the optimized mode generates almost twice the number of tests as the exhaustive option (4050 vs 2174). This seems backwards at first glance. 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. While this particular checker does more tests in exhaustive mode than in optimized mode, the total test throughput is slower due to the presence of costly operations (i.e., high latency).
We discovered and submitted a total of seven distinct

ISSN 2321-2152
www.ijmece.com
Vol13,Issue 1, 2025

issues to the creators of these three services throughout our testing. The main driver detected 4,500 bugs, and each of the checks (except the user-namespace checker) found three flaws. Several intriguing bugs discovered by the checkers proposed in this research will be covered in the part that follows.

V. EXAMPLES OF REST API SECURITYVULNERABILITIES

So far, we have successfully fuzzed three production Azure and Office 365 cloud services that are comparable in size and complexity to the ones mentioned above. Nearly every one of these services has a couple of new problems discovered by our fuzzing. Our new security checkers have found rule violations, accounting for about one third of the issues, while "500 Internal Server Errors" accounts for around two thirds. The service owners were notified of these issues, and they have all been resolved.

We stress that security testers boost trust in the service's overall dependability and security even if they don't uncover any problems; this is because they are more certain that the rules they verify cannot be broken.

The security significance of these issues is discussed in this section, which includes instances of actual problems discovered in Azure and Office 365 services that have been deployed. We take precautions to protect the privacy of those services by masking their names and other identifying information.

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 analyzer in an Office 365 messaging service that allows users to compose messages, react to them, and modify them.

1) Make a first message called msg-1 using the POST request to /api/posts/msg-1.

Make a second message called msg-2 and send it using the POST method to the address /api/posts/msg-2.

3) Make a reply-1 to the first message (using the POST request /api/posts/msg-1/replies/reply-1).

4) Use msg-2 as the message identification and edit reply-1 using a PUT request (with the syntax PUT /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.

If the validity tests for the hierarchy are missing, it might be possible for an attacker to circumvent the parent hierarchy and access child items. This could lead to security issues.

Azure instance experiencing a resource leak. A different Azure service had the same issue due to the resource-leak checker.

1) With a PUT request, create a new resource with the name X and type CM. The resource should have a certain malformed body. A "500 Internal Server Error" is the result, and that's a problem in and of itself.

2) If you want a list of all CM resources, you'll get an empty list.

Third, using a PUT request, create a new CM resource with the same name X as in Step 1, but in a different area (e.g., US-West instead of US-Central).

An unexpected "409 Conflict" rather than the anticipated "200 Created" is returned by the last request in Step 3. 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. 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.

Step 3's second PUT request, however, demonstrates that the service retains memory of the previous PUT request's unsuccessful attempt to create the CM resource X. Because these unsuccessful resource creations are (correctly) not counted against the user's resource quota, an attacker could theoretically build an unlimited number of these "zombie" resources by repeating Step 1 using numerous different names. This would allow them to surpass their official quota. But it's obvious that some part of the backend service is remembering them (wrongly).

Another Case in Point: A DoS Attack on Resource Accounting using Anger.

During our five hours of fuzzing another Azure service, we unintentionally caused a significant decline in its health. What follows is a synopsis of the research about its origin.

For our fuzzing program to stay under its cloud resource restrictions, we implemented a trash collector. Our garbage collector ensures that the number of active resources never exceeds limits by eliminating (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 moment. If we didn't have trash collection, our fuzzing tool would usually hit its quota restrictions in minutes and stop exploring state space.

Requests to generate resources of a certain type—let's call it "IM"—through this Azure service trigger additional processes that take minutes to perform in the service backend, but they deliver a response promptly. Similarly, deleting an IM resource just takes a few minutes and provides a response just as fast. 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. Therefore, an attacker may create-then-delete IM resources rapidly without going over their limit, while simultaneously initiating a deluge of backend operations and ultimately overwhelming the backend service. We unintentionally set off a Denial-of-Service attack using our fuzzing tool.

To address this security hole, we should update the use counts for DELETE requests to their quotas only when all delete backend operations are finished, which is usually a few minutes later for IM resources. By blocking further IM resource-creation PUT requests until previous DELETE requests are entirely finished, the quantity of backend jobs is still linearly limited by the official limitation.

VI. RELATED WORK

An extension of stateful REST API fuzzing is our work [5]. To automatically produce sequences of requests that meet the specification, a fuzzing language is used in conjunction with a Swagger specification of a REST API. 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. The model-based testing [27] finite-state-machine model of the system being tested is the basis for the BFS and BFS-Fast search techniques. In order to improve upon stateful REST API fuzzing, this paper does two things: first, it introduces a set of security rules for REST APIs and matching checkers that can efficiently test and detect compliance with these rules; and second, it introduces BFS-Cheap, a new search strategy that provides a compromise between BFS and BFS-Fast when employing active checkers.
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]. Recent extensions to tools that record, parse, fuzz, and replay HTTP traffic have made use of Swagger standards to assist the fuzzing of HTTP requests via REST APIs [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.
Due to their origins as extensions of more conventional web-page crawlers and scanners, most HTTP-fuzzers are able to check a wide variety of properties specific to HTTP. For example, they can ensure that responses use proper HTTP-usage and even detect cross-site scripting attacks or SQL injections when 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.
Authentication token and API key management is a

common theme in security recommendations from groups like OWASP [19] (Open Web Application Security Project) and publications on REST APIs [1] and micro-services [17]. There is a lack of specific instructions on how to manage resources and validate inputs using the REST API. 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 describe our checkers. Unlike conventional runtime verification, which consists of only monitoring API request and response sequences, our checkers also create new tests to identify rule violations. Our approach is based on using numerous independent security checkers at the same time, as described in [10]. However, we do not create additional tests by symbolic execution, constraint formation, or solution, as was done in [10]. Since our fuzzing tool and its checkers only see REST API calls and answers, they are unable to discern the inner workings of the services that we test. 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, time-consuming, and has restricted coverage because of how much manual effort is required. 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 identify and protect RESTful APIs and services, we laid down four guidelines. To further automate testing and detection of rule violations, we demonstrated how to add active property checks to a stateful REST API fuzzer. Using the fuzzer and checkers outlined in this work, we have successfully fuzzed about a dozen 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. Our current bug "fixed/found" ratio is virtually 100%, indicating that all of the service owners have taken the issues we detected seriously. 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? The only way to find out is to examine additional attributes and run more REST API-based bug and vulnerability scans on more services. Considering the current surge in REST APIs for online and cloud services, it is surprising that there is surprisingly little advise about the security-related use of REST APIs. 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. InProceedings 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 Sy