Checking Security Properties of Cloud Services REST APIs

Vaggelis Atlidakis[*] *Columbia University* *Patrice Godefroid* *Microsoft Research* *Marina Polishchuk* *Microsoft Research*

Abstract

Most modern cloud and web services are programmatically accessed through REST APIs. This paper discusses how an attacker might compromise a service by exploiting vulnerabilities in its REST API. We introduce four security rules that capture desirable properties of REST APIs and services. We then show how a stateful REST API fuzzer can be extended with active property checkers that automatically test and detect violations of these rules. We discuss how to implement such checkers efficiently and in a modular way. Thanks to these checkers, we found new bugs in several deployed production Azure and Office-365 cloud services, and we discuss their security implications.

1 Introduction

Cloud computing is exploding. Over the last few years, thousands of new cloud services have been deployed by cloud platform providers, like Amazon Web Services [2] and Microsoft Azure [13], and by their customers who are "digitally transforming" their businesses by modernizing their processes while collecting and analyzing all kinds of new data.

Today, most cloud services are programmatically accessed through REST APIs [9]. REST APIs are implemented on top of the ubiquitous HTTP/S protocol, and offer a uniform way to create (PUT/POST), monitor (GET), manage (PUT/POST/PATCH) and delete (DELETE) cloud resources. Cloud service developers can document their REST APIs and generate sample client code by describing their APIs using an interface-description language such as Swagger (recently renamed OpenAPI) [23]. A Swagger specification describes how to access a cloud service through its REST API, including what requests the service can handle, what responses may be received, and the response format.

How secure are all those APIs? Today, this question is still largely open. Tools for automatically testing cloud services via their REST APIs and checking whether these services are reliable and secure are still in their infancy. Some tools available for testing REST APIs capture live API traffic, and then parse, fuzz and replay the traffic with the hope of finding bugs [4, 19, 6, 24, 3]. Recently, *stateful REST API fuzzing* [5] was proposed to specifically test more deeply services deployed behind REST APIs. Given a Swagger specification of a REST API, [5] shows how to automatically generate *sequences of requests*, instead of single requests, in order to thoroughly exercise the cloud service deployed behind that API, with the goal of finding unhandled exceptions (service crashes) that can be detected by a test client as "`500` Internal Server Errors". While [5] looks promising and reports many new bugs found, its scope is restricted to the detection of unhandled exceptions.

In this paper, we introduce four security rules that capture desirable properties of REST APIs and services.

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

Resource-leak rule. A resource that was not created successfully must not be accessible and not "leak" any side-effect in the backend service state.

Resource-hierarchy rule. A child resource of a parent resource must not be accessible from another parent resource.

User-namespace rule. A resource created in a user namespace must not be accessible from another user namespace.

Violations of such rules might allow an attacker to hijack cloud resources or bypass quotas (*Elevation-of-Privilege* attack), or to steal information from other users (*Information-Disclosure* attack), or to corrupt the backend service state so that it no longer operates properly (*Denial-of-Service* attack), as will be discussed later.

We then show how a stateful REST API fuzzer can be extended to test and detect violations of such rules. For each rule, we define an active property checker which (1) generates new API requests to test rule violations and (2) detects any rule violation. In other words, each checker actively tries to break its rule in addition to monitoring for any rule violation. We discuss how to implement such checkers in a modular way, so that checkers do not interfere with each other. Since each checker generates new

[*]The work of this author was mostly done while visiting Microsoft Research.

tests in addition to an already-large state space exploration, we also discuss how to implement each individual checker *efficiently*, by eliminating likely-redundant tests whenever possible.

By construction, these checkers can find security rule violations beyond the "`500` Internal Server Errors" that can be detected by baseline stateful REST API fuzzing. Thanks to these checkers, we found new bugs in several deployed production Azure and Office-365 cloud services. The use of security checkers increases the value of REST API fuzzing by detecting more types of bugs at a modest incremental testing cost.

This paper makes the following contributions:

- We introduce rules that describe security properties of REST APIs.
- We design and implement active checkers to test and detect violations of these rules.
- We present experimental results evaluating the performance and effectiveness of these active checkers on several production cloud services.
- With these checkers, we found new bugs in several deployed production Azure and Office-365 cloud services, and we discuss their security implications.

The rest of the paper is organized as follows. In Section 2 we recall background information on stateful REST API fuzzing. In Section 3 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 4 we present experimental results with active checkers on production cloud services. In Section 5 we discuss new bugs found by these checkers and their security implications. In Section 6 we discuss related work, and conclude the paper in Section 7.

2 Stateful REST API Fuzzing

In this section, we recall the definition of stateful REST API fuzzing from [5], before introducing in Section 3 security property checkers that can be implemented as extensions of this basic scheme.

We consider cloud services accessible through REST APIs. A client program can send messages, called *requests*, to a service and receive messages back, called *responses*. Such messages are sent over the HTTP/S protocol. Each response is associated with a single HTTP status code which is either in the `2xx`, `3xx`, `4xx` or `5xx` ranges.

Swagger [23], also known as OpenAPI, is an example of specification language to define REST APIs. A Swagger specification describes how to access a service through its REST API, including what requests the service can handle, what responses may be received, and the response format.

We define a REST API as a finite set of requests. Each request r is a tuple of the form $\langle a,t,p,b\rangle$ where

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

A request type t is any of the following five REST-allowed values: PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), PATCH (update). The resource path p is a string identifying a cloud resource and its parent hierarchy. Typically, p is a (non-empty) sequence matching the regular expression

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

where `resourceType` denotes the type of a cloud resource and `resourceName` is the specific name of the resource of that type. The last resource named in the path is typically the specific resource that the request tries to create, access or delete. The request body b may include additional parameters and their values that may be required or optional for the request to be executed successfully.

For instance, here is a request to get the properties of a specific Azure DNS zone [14] (shown on multiple lines):

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

This request is of type GET, its path requires three resource names, namely a `subscriptionID`, a `resourceGroupName` and a `zoneName`, and its body (at the end) denoted by { } is empty.

REST API requests of type PUT or POST typically create new resources, while DELETE requests destroy existing resources. A request whose execution creates a new resource of type T is called a *producer* for that resource type T. A newly created resource is represented by its *identifier*, or *id* for short. Because resources are dynamically created, we will sometimes call them *dynamic objects*. A request which requires a resource name of type T in its path or its body is called a *consumer* for that resource type T. We will sometimes refer to the resource name of type T as the *dynamic object type*. In the Azure DNS zone example above, the GET request shown consumes three resources of type `subscriptions`, `resourceGroups` and `dnsZones` respectively, but does not produce any new resource.

Inside resource paths or request bodies of individual requests, the user is allowed to specify that some specific values, called *fuzzable values*, are to be chosen randomly among a (small finite) set of specific values. For instance, a user might specify that a given integer value in the body of a request may be, say, either `0`, `10`, `1000000`, or `-10`. Such a set of values is called a *fuzzing dictionary*. Given a request with fuzzable values, a *rendering* of that request denotes a mapping of each fuzzable value to a single concrete value selected in its fuzzing dictionary. Thus, a request with n fuzzable values which can each take k possible values results in n^k possible renderings. A rendering is called *valid* if the execution of the corresponding request returns a valid response (defined in the next paragraph). Users are responsible for identifying values they want to fuzz and their associated fuzzing dictionaries.

We define the *state space* of a service as a directed graph where nodes represent service states and edges are transitions between these. Given a state s of the service, executing a single request r leads to a successor state s': this execution is denoted by $s \xrightarrow{r} s'$. The execution of a request r in a state s is either *valid* if it triggers a `2xx` response, *invalid* if it triggers a `3xx` or `4xx` response, or a `bug` if it triggers a `5xx` response.

Given an initial state where no resources exist, the state space of the service reachable from that initial state can be explored by executing *sequences of requests*. Such an exploration is *stateful* when it attempts to explore service states that are reachable only using sequences of multiple requests: earlier requests in a sequence may produce resources that are consumed in subsequent requests in that sequence in order to exercise more requests and reach deeper service states.

State-space exploration can be performed using various search strategies, e.g., a systematic breadth-first search or a random search [5]. State spaces can be large, even infinite, because the length of request sequences is not bounded, because the sets of possible renderings can be very large, and because the service under test is viewed as a blackbox. Fortunately, a partial state-space exploration may be sufficient to reveal interesting bugs. In our context, a bug is defined as a `500` HTTP status code being received as a response after executing a request sequence. Such "`500` Internal Server Errors" are unhandled exceptions triggered by unexpected input request sequences, which may corrupt the service state and severely damage the service health: it is safer to fix such bugs rather than risk a live incident with unknown consequences.

In what follows, we will sometimes use the terms *test cases* to refer to executions of request sequences while *tests* refer to executions of single requests. We will also call the general state-space exploration algorithm of this section the *main driver* of stateful REST API fuzzing.

3 Security Checkers for REST APIs

In this section, we define and describe active checkers for security rules of REST APIs. First, in Section 3.1, we introduce four REST API security rules. In Section 3.2, we describe how to implement active checkers for testing and detecting security rule violations. Each active checker focuses on a single type of security rule violation. In Section 3.3, we discuss how each checker can be combined in a modular way with the other checkers and with the main driver of stateful REST API fuzzing. In Section 3.4, we propose a new search strategy for scalable test generation with security checkers. In Section 3.5, we describe how to group together checker violations in order to avoid reporting the same bug multiple times to the user.

3.1 Security Rules

We introduce four security rules that capture desirable properties of REST APIs and services. We illustrate each rule with an example and discuss its security implications. All four rules are inspired by past real bugs in deployed cloud services, which were found either by manual pen testing or root cause analysis of customer-visible incidents. Examples of new previously-unknown bugs we found as rule violations in deployed production Azure and Office-365 services are presented later in Section 5.

Use-after-free rule. A resource that has been deleted must no longer be accessible. In other words, after a successful DELETE operation on any resource, any subsequent operation – like read, update or delete – on that resource must fail.

For example, after issuing a `DELETE` request to URI `/users/user-id1` in order to delete the account with identifier `user-id1`, all subsequent attempts to use `user-id1` must fail and thus return a "`404` Not Found" HTTP status code in their response.

A use-after-free violation occurs when a resource that has been deleted still remains accessible through the API. This must never happen. It is a clear bug, which might lead to bypassing resource quotas and corrupting the service backend state.

Resource-leak rule. A resource that was not created successfully must not be accessible, and must not "leak" any associated resources in the backend service state. In other words, if the execution of a `PUT` or `POST` request to create a new resource fails (for any reason), any operation on that resource must also fail with a `4xx` response. Furthermore, no side-effects associated with successful creation of that resource type must occur in the backend service state and be visible to the user. That is, a failed-to-be-created resource must not be counted in the users resource counter towards service quotas, and the name of the failed-to-be-created resource must be reusable by the user.

For example, after issuing a malformed `PUT` request to create URI `/users/user-id1`, a `4xx` response must be received. Any subsequent request to access (read, update or delete) this URI must also fail.

A resource-leak violation occurs when a resource that was not successfully created nevertheless "leaks" some side-effect in the backend service state. For instance, the resource may be listed by a subsequent `GET` request, yet it cannot be deleted with a `DELETE` request, or subsequent attempts to re-create this resource return "`409` Conflict" responses. Such violations must never happen, as they may have unintended consequences on the capacity for that resource type (e.g., if resource quota limits are reached and no new resources can be created) and on the performance of the service (e.g., due to unnecessarily large database tables).

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

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

A resource-hierarchy violation occurs when a sub-resource originally created from a parent resource is accessible from a different parent resource with no parent-child relationship. When such violations are possible, an attacker might be able to provide an unauthorized parent object identifier (e.g., `user-id3`), and then steal (read) or hijack (write) an unauthorized child object (e.g., `report-id1`). Resource-hierarchy violations are clear bugs, are potentially dangerous, and must never happen.

User-namespace rule. A resource created in a user namespace must not be accessible from another user namespace. In the context of REST APIs, we consider user namespaces defined by the user token used to interact with the API (e.g., OAUTH token-based authentication [16]).

For example, after issuing a `POST` request to create 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.

3.2 Active Checkers

We implement active checkers for the rules defined in Section 3.1. An *active checker* monitors the state space exploration performed by the main driver of 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. Hence, a passive checker does not augment the state space explored by stateful REST API fuzzing.

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 checkers.

We enforce the first principle by running all checkers at select execution points after the main driver has finished executing a test case. We enforce the second principle by prioritizing the order of applying checkers based on their semantics so that they operate on different test cases and do not interfere with each other (more on this later in this section). In what follows, we present the implementation details of each checker as well as optimizations to limit state-space explosion.

Use-after-free checker. The implementation of the use-after-free rule checker is described in Figure 1 in python-like notation. The algorithm takes three inputs: a sequence of requests, denoted `seq`, which is the latest test case executed by the main driver; the global cache of dynamic objects, denoted `global_cache`, which contains the most recent object types and ids for the dynamic objects created so far; and the request collection, denoted

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

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

```
 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[n])
 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.**

`reqCollection`, which is the set of all available API requests.

First, the types of the dynamic objects consumed by the last request are retrieved (line 5) and the id of the last object type, denoted `target_obj_type`, is stored in a temporary variable, denoted `target_obj_id`. Although the last request may be consuming more than one object type, we consider the last type in `req_object_types` as the actual type of the deleted object. [1] After this initial setup, the for-loop (line 12) iterates over all requests available in `reqCollection` and skips those that do not consume the target object type (line 14). Once a request, `req`, that consumes the target object type is found, the target object id is restored in the global cache of dynamic objects (line 17) and is therefore used by function EXECUTE (line 19) which executes request `req`. Note that the target object id is repeatedly restored in the global cache because the function EXECUTE uses object ids available in `global_cache` when executing a request. If any of these requests succeeds, line 20 will trigger a use-after-free violation (see Section 3.1).

Finally, in order to limit the number of additional test cases generated for each request sequence, the inner loop (optionally) terminates when one request for each target object type is found (line 21). This option is used if the variable `mode` is not set to value `exhaustive`. We present detailed experimental results regarding the impact of this optimization in Section 4.

[1]This is better understood with an example: imagine a DELETE request on the URI `/users/userId1/reports/reportId1` which consumes two object types (users and reports) but it only deletes report objects.

Resource-leak checker. The implementation of the resource-leak rule checker is described in Figure 2 in python-like notation. The algorithm takes the same three inputs as the use-after-free checker. However, one key difference between this checker and all others is that the resource-leak checker operates on request sequences whose last request execution by the main driver has led to an invalid HTTP status code in the response. The reason for selecting only such sequences is that when an invalid status code is returned and the last request was attempting to create one or several new resources (i.e., the last request is a resource producer), the requested dynamic objects must not be created in the backend state; otherwise, a leak occurs: (some of these) dynamic objects may have been created in the backend state yet the user may not have access to these through the API.

Initially, the algorithm identifies the dynamic object types produced by the whole sequence, denoted `seq_obj_types`, and produced by the last request, denoted `target_obj_types` (lines 4 and 5).

The main logic of the algorithm is implemented in three nested for loops. The first loop (line 6) iterates over all object types produced by the last request. The second loop (line 7) iterates over object ids "guessed" for the current object type for which an invalid HTTP status code was received. The function GUESS take as argument an object type and returns a set of possible object ids matching this type and which were not created successfully. For instance, if the creation of a dynamic object with object type "x" and object id "objx1" fails through the API (according to the response received), the checker will attempt to execute any request that consumes the object type "x" and assert it fails when using the object id "objx1". Note that the total number of

```
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 first predecessor that
15 # produces any of the target types. This propagates new
16 # values for all predecessor object ids.
17 n_predecessors = 0
18 for req in seq:
19   n_predecessors += 1
20   if PRODUCES(req) ∩ (target_obj_types−predecessors_obj_types):
21     break
22 EXECUTE(seq, n_predecessors)
23 # Restore old children object ids that do NOT belong to
24 # the current parent ids and must NOT be accessible on
25 # top of the latest predecessors' rendering.
26 for obj_type in local_cache:
27     global_cache[obj_type] = local_cache[obj_type]
28 EXECUTE(last_request)
29 assert ''HTTP status code is 4xx''
```

Fig. 3: **Resource-hierarchy checker.**

```
Inputs: seq, global_cache
# Because this checker is applied last, we need to re−render the
# current sequence in order to propagate proper object ids.
EXECUTE(seq)
# Retrieve the object types consumed by the whole sequence and
# locally store the most recent object ids of those objects.
target_obj_types = CONSUMES(seq)
local_cache = {}
for obj_type in target_obj_types:
    local_cache[obj_type] = global_cache[obj_type]
for i, req in enumerate(seq):
    # If not in exhaustive mode, render only the last request.
    if mode != 'exhaustive' and i != seq.length:
        continue
    # Skip requests that are not consumers.
    if not CONSUMES(req):
        continue
    # Reset global cache of object ids and use an alternate (attacker)
    # token to execute the sequence up to before the last request.
    global_cache.reset()
    EXECUTE(seq − req, use_attacker_token)
    # Restore the object ids belonging to the benign user and try to hijack
    # them by executing the last request using an attacker token which is
    # not authorized for those object ids.
    for obj_type in local_cache:
        global_cache[obj_type] = local_cache[obj_type]
    EXECUTE(req, use_attacker_token)
    assert ''HTTP status code is 4xx''
```

Fig. 4: **User-namespace checker.**

guessed values per object id is limited to a user-provided parameter value in order to avoid an explosion in the number of additional test cases.

In line 8, the guessed object-id values are temporarily added to the global cache of properly-created dynamic objects. These guessed values will be, later on, used in an attempt to access dynamic objects whose creation had failed according to the HTTP response received over the API. Specifically, the inner loop (line 9) iterates over all requests in `reqCollection` to find requests that are executable (given the object types produced by the current sequence) and that consume the given target object type. These requests are executed (line 17) using the "guessed" object ids that are previously registered to the global cache. This way, the algorithm tries to trigger a resource-leak violation (see Section 3.1) or assert that no such violation occurs for the given request sequence (line 18).

Finally, in order to limit the number of additional test cases generated for each input sequence the inner loop (optionally) terminates when one request for each target object type is found (line 19). We present detailed experimental results regarding the impact of this optimization in Section 4.

Resource-hierarchy checker. The implementation of the resource-hierarchy rule checker is described in Figure 3 in python-like notation. The algorithm takes two inputs: a sequence of requests, denoted `seq`, which is the latest test case executed by the main driver and the global cache of dynamic objects, denoted `global_cache`, which contains the most recent object types and ids for the dynamic objects created so far.

First, the algorithm records the object types produced by the last request of the current sequence denoted `target_obj_types` (line 6), and the object types produced by all other requests of the sequence before the last request, denoted `predecessor_obj_types` (line 7).

Afterwards, the ids of the objects consumed only by the last request are stored locally (lines 12 and 13). These are the child objects whose hierarchy the checker will try to violate by executing requests that try to access them using invalid parent objects. To this end, in line 22, the current sequence is executed up to (and not including) the first request that produces any of the children target types (the appropriate number of predecessor requests, denoted `n_predecessors`, is calculated in the for-loop of line 18.) Finally, the old child object ids are restored (lines 26 and 27) and the last request is executed using the old child object ids on top of new parent object ids generated by the latest predecessor requests (line 28). These parent object ids are not proper parent objects of the restored child object ids. This way, the algorithm tries to trigger a resource-hierarchy violation (see Section 3.1) or assert that no such violation occurs for the given request sequence (line 29).

```
Inputs: seq, global_cache, reqCollection
# Apply the checker after the main driver.
n = seq.length
if seq[n].http_type == ''DELETE'':
        UseAfterFreeChecker(seq, global_cache, reqCollection)
else:
        ResourceLeakChecker(seq, global_cache, reqCollection)
        ResourceHierarchyChecker(seq, global_cache)
UserNamespaceChecker(seq, global_cache)
```

Fig. 5: **Checkers dispatcher.**

User-namespace checker. The implementation of the user namespace rule checker is described in Figure 4 in python-like notation. The algorithm takes three inputs: a sequence of requests, denoted `seq`, which is the latest test case executed by the main driver; the global cache of dynamic objects, denoted `global_cache`, which contains the most recent object types and ids for the dynamic objects created so far; and an attacker token (an alternate token representing the attacker's identity, which must not have access to the same resources as the token used by the main driver), denoted `attacker_token`.

First, since the user namespace checker is applied last, the input sequence is re-rendered (line 4) to prevent interference with previously applied checkers, and, in particular, ensure that consistent object ids exist in `global_cache`. Then, the algorithm records the object types produced by the last request of the current sequence, denoted `target_obj_types` (line 7). The object ids of the dynamic objects produced by the current sequence are then stored locally (lines 9 and 10). Afterwards, the outer loop (line 11) iterates over the current sequence until the first request, denoted `req`, which consumes some object type is found (lines 16 and 17).

Note the usual optimization to limit the size of the additional test cases generated for each input sequence (line 14). Once a request `req` that consumes some object type is found, the global cache of object ids is reset (line 20) and the current sequence is executed up to before `req` using the `attacker_token` (line 21). This constructs an attacker namespace containing predecessor dynamic objects. Afterwards, the object ids belonging to the victim user are restored (lines 25 and 26) and `req` is executed using the attacker's identity (line 27).

To hijack the objects belonging to a victim user, `req` is then executed with the attacker's identity after restoring the victim's object ids. If the request succeeds, a user namespace violation (see Section 3.1) has been caught for the current sequence (line 28).

3.3 Combining All Checkers

The four checkers defined in the previous section are executed as follows. Whenever the stateful REST API fuzzer reaches a new state (as defined in Section 2), its main driver calls the code shown in Figure 5. Depending on the type of the last request executed, this code activates the checkers that are applicable to the current state.

We now discuss important properties of these checkers, compared to the main driver of Section 2 and also compared to each other.

Contribution beyond stateful REST API fuzzing. The checkers *extend* the main driver of baseline stateful REST API fuzzing in two ways: (1) they extend the state space by executing additional tests and (2) they check for responses other than `5xx` and can flag unexpected `2xx` responses as rule-violation bugs. Thus, they clearly *increase* the bug-finding capabilities of the main driver: they can find bugs that the main driver alone would not find.

Active property checking versus passive monitoring. As discussed earlier, the checkers we define extend the search space explored by the main driver with additional test cases aimed at triggering and detecting specific rule violations. In contrast, passive runtime monitoring of these rules in conjunction with the main driver, i.e., without executing those new tests, would likely be unable to detect rule violations. For example, use-after-free and resource-leak rule violations would likely not be detected with passive monitoring alone because the default state space exploration performed by the main driver would likely not attempt to re-use deleted resources or resources after a failure, respectively. Similarly, resource-hierarchy and user-namespace rule violations would not be detected by passive monitoring either because the baseline main driver does not attempt to substitute object identifiers or authentication tokens, respectively. In other words, the additional test cases generated by the checkers are *necessary* to find rule violations and are *not redundant* with respect to non-checker tests.

Complementarity among the checkers. The four checkers we define complement each other: no two checkers will ever generate the same new test cases, by construction, because their preconditions are all mutually exclusive. First, the use-after-free checker is the only checker activated by request sequences that end in a DELETE request. Second, the resource-leak checker is the only checker activated when the last request executed returned an invalid HTTP status code. Third, the resource-ownership checker is the only checker activated on request sequences with valid renderings that do not end in a DELETE request. Fourth and last, the user-namespace checker executed tests using an attacker token different from the authentication token used by the main driver and all other checkers, so it clearly extends the state space in another, orthogonal dimension.

3.4 Search Strategies for Checkers

The main search strategy used for test generation in stateful REST API fuzzing [5] is a breadth-first search (BFS) in the search space defined by all possible request sequences. This search strategy provides full grammar coverage both with respect to all possible renderings of each individual request and with respect to all possible request sequences up to a given sequence length. However, since the search space explored by BFS is typically enormous, the search does not scale well as the sequence length increases. Therefore, an optimization called BFS-Fast was introduced. With BFS-Fast, whenever the search depth increases to a new value $n+1$, each request is appended to at most one request sequence of length n, instead of to all of them as in BFS [5]. BFS-Fast provides full grammar coverage only with respect to all possible renderings of individual requests but does not explore all request sequences of a given sequence length.

Although BFS-Fast scales better compared to BFS, it does so by exploring only a subset of all possible request sequences. Unfortunately, this limits the number of violations the security checkers can actively check. To alleviate this limitation, we introduce a new search strategy, called **BFS-Cheap**.

BFS-Cheap follows the inverse trade-off of BFS-Fast: it sacrifices full coverage of all possible request renderings at every state but explores all possible request sequences for a given sequence length, albeit not with all possible renderings. Specifically, given a set of sequences of length n, called `seqSet`, and a set of requests, called `reqCollection`, BFS-Cheap operates as follows:

> For each sequence `seq` $\in$ `seqSet`, append each `req` $\in$ `reqCollection` to the end of `seq`, execute the new sequence while considering the possible renderings of `req`, and add to `seqSet` at most one valid (if any) and one invalid (if any) sequence rendering.

Valid renderings are used by the use-after-free, resource-hierarchy, and user-namespace checkers, while invalid renderings are used by the resource-leak checker.

BFS-Cheap thus provides a middle-ground between BFS and BFS-Fast (see Section 4.2 for an experimental evaluation). It explores all possible request sequences up to a given sequence length (like BFS) and adds at most two new renderings for each sequence in order to avoid an enormous `seqSet` (like BFS-Fast). Two new renderings per sequence explored allow for active checking of all the security rules defined in Section 3.1 while maintaining a tractable number of sequences in `seqSet` as the sequence length increases.

Note that the suffix "cheap" comes from the fact BFS-Cheap is a cheaper version of BFS where at most one valid rendering is added to the BFS "frontier" `setSeq` for each new sequence. This leads to the creation of fewer resources than those created when all valid renderings of each request sequence are explored, as in BFS. For instance, imagine a request definition with an enum type describing ten different flavours of the same resource type. BFS-Cheap will stop creating resources once one resource of one flavour is successfully created. In contrast, BFS and BFS-Fast, will create ten resources of the same type with ten different flavours.

3.5 Bug Bucketization

Before discussing examples of real violations found with active checkers, we define the bucketization scheme used to group together similar violations. In the context of active checkers, we define "bugs" as rule violations. Each bug is associated with the request sequence that was executed to trigger it. Given this property, we use the following procedure to create *per-checker* bug buckets:

> Whenever a new bug is found, compute all non-empty suffixes of the request sequence which triggers the bug, starting with the smallest one. If a suffix appears in a previously-recorded bug bucket, add the new sequence to that existing bug bucket. Otherwise, create a new bug bucket for the new sequence.

This bug bucketization scheme is the same as in [5], except that, since failure conditions are defined differently for each rule, we maintain separate, *per-checker* bug buckets. Each bug will always be reported by one checker for a specific sequence length (because of checker complementarity), except for "`500` Internal Server Error" bugs which may be triggered by both the main driver and checkers. For `500` bugs, the new sequence will be added only once to the bug bucket of the main driver or checker that triggered it first.

4 Experimental Evaluation

In this section, we report results of experiments with three production cloud services. These services and our experimental setup are described in Section 4.1. Then, we compare in Section 4.2 the three search strategies described in Section 3.4. Next, we present results showing the number of rule violations reported by each checker on the three cloud services as well as the impact of various optimizations (Section 4.3).

4.1 Experimental Setup

We report results of experiments performed while testing three services. Two of these services are from Azure and one is from Office-365, and will be referred to as **Azure**

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 1: **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.

A, **Azure B** and **O-365 C**. The number of requests in the REST API of each of these three services ranges from 13 to 19 requests. We selected those three services because their size and complexity are representative among the cloud services we analyzed. So far, we have performed similar experiments with about a dozen production services, and our general experience with those other services is summarized in Section 5.

Every service we consider has a publicly-available Swagger specification [2]. For each service, we compile its specification to produce a test-generation grammar as described in [5]. Each grammar is encoded as executable python code. For a given service and API, the same grammar and fuzzing dictionaries were used across all the experiments reported in this section.

We ran our fuzzing experiments using a single-threaded fuzzer running on a PC connected to the internet and a valid service subscription that allows access to each service API. No other special test setup or service knowledge was required. As in [5], our fuzzer includes a garbage-collector that deletes no-longer-used resources (dynamic objects) in order to avoid exceeding service quota limits.

Note that we view the target services we fuzz as complete black boxes. We fuzz production services already deployed and accessible to anyone (with a valid subscription), but we have no visibility as to what happens inside the backend of the services we test. Our fuzzer can only observe the HTTP status codes of the responses it receives. All client-side requests are sent over the internet to the target services, and responses are parsed when they are received. No advance notice was given to the owners and operators of those services. Because we do not control the deployment of those services, the experiments reported in this section are not fully controlled, and we cannot fully guarantee their reproducibility. However, we did perform these experiments several times and, among these, the overall results did not vary significantly.

[2]Available from https://github.com/Azure/azure-rest-api-specs

4.2 Comparing Search Strategies

We now compare our new search strategy, BFS-Cheap, with BFS and BFS-Fast when using security checkers to fuzz real services. We present results of experiments with two Azure and one Office-365 services, denoted by **Azure A**, **Azure B**, and **O-365 C** respectively.

Table 1 shows individual experiments with the three search strategies on each service, over a fixed time budget of one hour per experiment. For each experiment, we report the total number of requests in the API (Total Req.), the maximum sequence length generated (Max Len.), the total number of requests sent (Tests), the percentage of the requests sent by the main driver (Main) and the active checkers (Checkers) as well as the individual contribution of each checker.

Table 1 clearly shows that, for all services, BFS reaches the smallest depth, BFS-Fast reaches the largest depth, and BFS-Cheap provides a trade-off between these two extremes, while being closer to BFS than BFS-Fast. The total number of tests generated varies across services, depending on the speed of the responses received from each service. For any given service, this number remains roughly similar except for BFS-FAST with **Azure B** and **O-365 C** where the total number of tests increases significantly. For **O-365 C**, this increase seems to be due to a significantly lower number of failed requests generated by BFS-FAST for these two services compared to BFS and BFS-Cheap. Such failed requests are sent back to the client (our fuzzer) with larger time delays. Delaying responses to failed requests is a well-known mechanism used by services to *throttle* future re-

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 2: **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).

quests, i.e., to try to slow them down. For **Azure B**, BFS-Fast executes more tests because its request sequences are deeper but include many DELETE requests which are faster to execute (their responses are received almost instantly): BFS-Fast executes about 9 times more DELETE requests than BFS or BFS-Cheap.

The total percentage of checker tests (Checkers) is the highest for BFS and the lowest for BFS-FAST, while BFS-Cheap is again in between. Indeed, while BFS-Fast generates the largest number of tests, its search space is pruned and activates checkers less often, as discussed in Section 3.4 – this is the precise motivation for introducing BFS-Cheap in that section. An exception is the 33% spike in checker-generated tests by BFS-FAST for **O-365 C**. This spike seems to be due to a larger number of (fast) successful requests (see the previous paragraph), which in turn triggered more checker tests.

From the individual checker statistics in Table 1, we observe that the number of tests they each generate varies from service to service. This number depends on the number of `DELETE` requests executed for the use-after-free checker, the number of failed resource-creation requests for the resource-leak checker, and the depth of the object hierarchy for the resource-hierarchy checker. In contrast, the user-namespace checker is triggered more consistently more often and contributes the largest percentage of checker-generated tests.

For all three services, the number of bugs found is nearly the same for all three search strategies and is discussed next.

4.3 Comparing Checker Optimizations

We now compare the performance of the two modes *optimized* and *exhaustive* discussed in Section 3.

Table 2 shows how many requests were sent in one hour of fuzzing with BFS-Cheap in the *Tests* column, and what percentage of those requests were generated by either the main driver of Section 2 or by any of the four checkers. The table also shows how many unique bugs (bug buckets) were found in one hour of search by the main driver and by each of the checkers. Results are presented for both the *optimized* and the *exhaustive* modes previously discussed.

We observe that the number of tests varies for different services and checker modes. However, the percentage of tests generated by the checkers is always higher with the *exhaustive* mode, as expected. Since in the *optimized* mode the checkers produce fewer tests per visited state, the main driver is allowed to explore more states faster. Yet, despite the lower number of checker tests per visited state, for all three services considered, the *optimized* mode finds all the unique bugs (bug buckets) found by the *exhaustive* mode. Moreover, for the **O-365 C** service, the main driver finds one more bug with the *optimized* mode compared to the *exhaustive* mode within one hour of search.

Table 2 reveals an interesting inversion that further demonstrates the value of the *optimized* checkers mode. In **Azure A**, we observe that the *optimized* mode produces almost twice as many tests than than the *exhaustive* mode (4050 versus 2174). At first sight, this is counter-intuitive. After a deeper investigation, we discovered that some of the tests produced by the *exhaustive* mode of the user-namespace checker have significantly larger response times for service **Azure A**. Indeed, this specific checker in *exhaustive* mode executes additional tests compared to the *optimized* mode, but containing expensive operations (i.e., high latency) that slow down the overall test throughput.

During the course of all experiments with these three services, we found and reported a total of 7 unique bugs to the developers of those services, including 4 `500` bugs found by the main driver and 3 bugs found by each of the checkers except the user-namespace checker. In the next section, we discuss several interesting bugs found thanks to the checkers introduced in this paper.

5 Examples of REST API Security Vulnerabilities

At the time of this writing, we have fuzzed nearly a dozen production Azure and Office-365 cloud services of size and complexity similar to the three services used in the previous section. In almost all cases, our fuzzing was able to find about a handful of new bugs in each of these services. About two thirds of those bugs are "`500` Internal Server Errors", and about one third are rule violations reported by our new security checkers. We reported these bugs to the service owners, and all have been fixed or are in the process of being fixed.

We emphasize that, even when the security checkers do not find any bugs, they increase confidence that the rules they check cannot be violated and therefore they increase confidence in the overall service reliability and security.

This section presents examples of real bugs found in deployed Azure and Office-365 services and discuss their security relevance. We anonymize the name of those services and key details not to target any specific service.

Use-after-free violation in Azure. In an Azure service, we found the following use-after-free violation.

1. Create a new resource R (with a PUT request).
2. Delete resource R (with a DELETE request).
3. Create a new child resource of the deleted resource R and of a specific type (with another PUT request).

This sequence of requests results in a "`500` Internal Server Error". The Use-after-free checker catches this as (1) it attempts to re-use in Step 3 the deleted resource in Step 2 and (2) the response of Step 3 is different from the expected "`404` Not Found" response.

Resource-hierarchy violation in Office365. In an Office365 messaging service where users can post messages and then reply and edit these, the resource-hierarchy checker detected the following bug.

1. Create a first message `msg-1` (with a request POST /api/posts/msg-1).
2. Create a second message `msg-2` (with a request POST /api/posts/msg-2).
3. Create a reply `reply-1` to the first message (with a request POST /api/posts/msg-1/replies/reply-1).
4. Edit the reply `reply-1` with a PUT request using `msg-2` as message identifier (with a request PUT /api/posts/msg-2/replies/reply-1).

Surprisingly, the last request in Step 4 returns a "`200` Allowed" response while it must have returned a "`404` Not Found" response. This rule violation shows that the above resource hierarchy is not being respected by the API that posts a reply. Specifically, the API implementation is assuming that knowing a valid reply ID implies access to the entire hierarchy. Missing hierarchy validation checks are potential security vulnerabilities: an attacker might be able to exploit them to access child objects by bypassing the parent hierarchy. To fix this bug, the service must check that the reply identifier matches the message identifier with which it was created before perfoming any data update.

Resource-leak violation in Azure. In another Azure service, the resource-leak checker triggered the following bug.

1. Create a new resource of type CM and of name X with a specific malformed body (with a PUT request). This returns a "`500` Internal Server Error", which is already a bug.
2. Get a list of all resources of type CM: the returned list is empty.
3. Create a new resource of type CM with the same name X as in Step 1 with a well-formed body but in a different region (e.g., US-West versus US-Central) with a PUT request.

Unexpectedly, the last request in Step 3 returns a response "`409` Conflict" instead of an expected "`200` Created". This behavior means that the service has reached an inconsistent state: the failed request in Step 1 has left unintended side-effects on the service state. Indeed, the GET request in Step 2 shows that the user view is correct: the CM resource named X attempted to be created in Step 1 has not been created. However, the second PUT request in Step 3 proves that the service still remembers the failed creation of the CM resource named X attempted in the first PUT request of Step 1.

This is clearly a bug and is potentially dangerous: an attacker could create a very large number of such "zombie" resources by repeating Step 1 using many different names, and exceed his/her official quota since such failed resource creations are (correctly) not counted towards the user' resource quota. Yet, they are clearly remembered (incorrectly) somewhere in the backend service.

Note that the resource-leak checker executed Step 3 directly after Step 1 to find this bug. The GET request in Step 2 was performed afterwards by us when analyzing the bug, and was added to the above bug description in order to better explain the issue found and its possible security implications.

Other Example: Eager Resource-Accounting DoS Attack. During the course of our experiments, we found another type of cloud-service security vulnerability by accident. Specifically, we started fuzzing another Azure service, planning to fuzz it overnight. However, after fuzzing that service for about five hours, we were contacted by the Azure team owning that service: they had detected the unusual traffic created by our fuzzing tests and asked us to stop those tests immediately. Indeed, they told us our experiments had unintentionally caused serious health issues to this service. We now summarize the security and reliability vulnerability that was determined to be the root cause of the incident which we accidentally triggered.

Our fuzzing tool uses a garbage collector not to exceed quotas for the cloud resources created during fuzzing. For instance, if a default quota for a resource type Y is 100, at most 100 resources of that type can be created at any time, and our garbage collector makes sure that the number of live resources never exceeds quotas by deleting (using a DELETE request) resources that are no longer used. Without garbage collection, our fuzzing tool would typically reach quota limits in minutes, and all subsequent resource-creation requests would fail with "quota-exceeded" errors, which would prevent exploring further the state space of that service.

In the case of this specific Azure service, it turns out that any PUT request to create a resource of a specific type, let us call it IM, returns a response quickly (nearly instantaneously) but actually triggers other tasks that take minutes to complete in the service backend. Similarly, a DELETE request for a resource of that same type IM also returns quickly but also trigger delete tasks that also take minutes to complete.

The health issue we triggered in this service by accident is due to the way internal service counters tracking resource usage are updated when creating and deleting resources of type IM in the service. Specifically, PUT and DELETE requests that create and delete resources of type IM update counters towards quotas *eagerly*, too quickly, without waiting for the several minutes actually needed to fully complete these actions. As a result, an attacker could abuse such an eager resource-accounting scheme and create-then-delete quickly many resources of type IM without exceeding his/her quota, while triggering a very large number of backend tasks, orders-of-magnitude more than the official quota, hence literally flooding the backend service with many such tasks. Such a Denial-of-Service attack is the incident our fuzzing tool and its garbage collector unintentionally triggered in that service that night.

A fix to this vulnerability is to update usage counters towards quotas for DELETE requests *only when all delete backend operations have been completed*, i.e., minutes later in the case of IM resources. This way, the amount of backend tasks is still linearly bounded by the official quota, since subsequent IM resource-creation PUT requests will be blocked until preceding DELETE requests have been fully completed.

6 Other Related Work

Our work builds upon the *stateful REST API fuzzing* approach recently introduced in [5]. Given a Swagger specification of a REST API, this specification is compiled into a fuzzing grammar, which is then used to automatically generate sequences of requests that satisfy the specification. This approach automates the generation of a fuzzing grammar compared to traditional *grammar-based fuzzing* [18, 20, 22] where the user manually writes a grammar. The BFS and BFS-Fast search strategies are inspired by test generation algorithms used in *model-based testing* [25, 12, 26] for generating minimal test suites that cover an entire finite-state-machine model of a system under test. This paper extends prior work [5] (i) by introducing a set of security rules for REST APIs and corresponding checkers for efficiently testing and detecting violations of these rules; and (ii) by introducing a new search strategy, BFS-Cheap, which offers a middle-ground between BFS and BFS-Fast when using active checkers.

Since REST API requests and responses are transmitted over the HTTP protocol, HTTP-fuzzers can be used to fuzz REST APIs. Fuzzers like Burp [7], Sulley [21], BooFuzz [6], or the commercial AppSpider [4] and Qualys's WAS [19], can capture/replay HTTP traffic, parse HTTP requests/responses and their contents (like embedded JSON data), and then fuzz those using either pre-defined heuristics [4, 19] or user-defined rules [21, 6]. Some tools to capture, parse, fuzz, and replay HTTP traffic have recently been extended to leverage Swagger specifications in order to parse HTTP requests over REST APIs and guide their fuzzing [4, 19, 24, 3]. However, these tools do not perform any global analysis of Swagger specifications and therefore cannot generate new sequences of requests: their fuzzing is *stateless*, i.e., restricted to fuzzing parameter values of individual requests. Therefore, adding active checkers to stateless fuzzers is problematic. In contrast, our work extends *stateful* REST API fuzzing with active checkers targeting specific REST API rule violations.

Because most HTTP-fuzzers were born as extensions of traditional web-page crawlers and scanners, they often support a long list of HTTP-focused properties they can check, such as checking for proper HTTP-usage in responses and even checking for cross-site-scripting attacks or SQL-injections if whole web-pages (with HTML and Javascript code) are returned as part of the responses. However, for most REST APIs, responses do not include web-pages, and most of this checking capability is irrelevant.

Compared to HTTP-fuzzers and web scanners, our paper introduces new security rules that are *targeted specifically* at REST API usage. These rules are security-related because their violations might be exploited by a malicious attacker to harm the health of a service or steal unauthorized information or resources. In contrast, we do not discuss in this paper how to check other REST API usage rules [9], such as request idempotence (i.e., repeating identical requests like GET or PATCH have no further effect on the outcome), which are not "exploitable" when violated.

Given the widespread use of REST APIs, there is surprisingly very little guidance provided on secure REST API usage. Most of the security guidance from organisations like OWASP [17] (Open Web Application Security Project) or from books on REST APIs [1] or microservices [15] is about managing authentication tokens and API keys. No detailed guidance is provided regarding REST API input validation and resource management. To the best of our knowledge, the four security rules introduced in this paper are new.

In Section 3, we used the term *active* checker to denote that our checkers do not simply monitor sequences of API requests and their responses as in traditional runtime verification [8, 11], but also generate new tests specifically aimed at triggering rule violations as in [10]. Like in [10], we use multiple independent security checkers simultaneously. But unlike [10], we do not use symbolic execution, constraint generation and solving in order to generate new tests. Indeed, the inner workings of the services we test are invisible to our fuzzing tool and its checkers, which only see REST API requests and responses. Since cloud services are usually complex distributed systems whose components are written in different languages, general symbolic-execution-based approaches seem problematic, but it would be worth exploring this option further in future work.

In practice, the main technique used today to ensure the security of cloud services is *penetration testing*, or *pen testing* for short, which means security experts review the architecture, design and code of cloud services from a security perspective. Since pen testing is labor intensive, it is expensive and limited in scope and depth. Fuzzing tools and security checkers like those discussed in this paper can partly automate the discovery of specific classes of security vulnerabilities, and are complementary to pen testing.

7 Conclusion

We introduced four security rules that capture desirable properties of REST APIs and services. We then showed how a stateful REST API fuzzer can be extended with active property checkers that automatically test and detect violations of these rules. So far, we have fuzzed nearly a dozen production Azure and Office-365 cloud services using the fuzzer and checkers described in this paper. In almost all cases, our fuzzing was able to find about a handful of new bugs in each of these services. About two thirds of those bugs are "`500` Internal Server Errors", and about one third are rule violations reported by our new security checkers. We reported these bugs to the service owners, and all have been fixed or are in the process of being fixed. Although still preliminary, our results are encouraging.

How security-critical are the bugs we found? This is still unclear. What is clear is that these bugs have all been taken seriously by the service owners we reported them to: our current bug "fixed/found" ratio is nearly 100% so far. Indeed, it is safer to fix these bugs rather than risk a live incident – provoked intentionally by an attacker or triggered by accident – with unknown consequences. Moreover, these bugs are easy to reproduce, and our fuzzing approach does not report false alarms.

How general are these results? To find out, we need to fuzz more services through their REST APIs and check more properties to detect different kinds of bugs and security vulnerabilities. Given the recent explosion of REST APIs for cloud and web services, there is surprisingly little guidance about REST API usage from a security point of view. Our paper makes a step in that direction by contributing four rules whose violations are security-relevant and which are non-trivial to check and satisfy.

References

[1] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

[2] Amazon. AWS. `https://aws.amazon.com/`.

[3] APIFuzzer. `https://github.com/KissPeter/APIFuzzer`.

[4] AppSpider. `https://www.rapid7.com/products/appspider`.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*, May 2019.

[6] BooFuzz. `https://github.com/jtpereyda/boofuzz`.

[7] Burp Suite. `https://portswigger.net/burp`.

[8] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 2000 SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer-Verlag, 2000.

[9] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, UC Irvine, 2000.

[10] P. Godefroid, M. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software)*, pages 207–216, Atlanta, October 2008. ACM Press.

[11] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of RV'2001 (First Workshop on Runtime Verification)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, July 2001.

[12] R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of TestCom'2006*, 2006.

[13] Microsoft. Azure. `https://azure.microsoft.com/en-us/`.

[14] Microsoft. Azure DNS Zone REST API. `https://docs.microsoft.com/en-us/rest/api/dns/zones/get`.

[15] S. Newman. *Building Microservices*. O'Reilly, 2015.

[16] OAuth. OAuth 2.0. `https://oauth.net/`.

[17] OWASP (Open Web Application Security Project). `https://www.owasp.org`.

[18] Peach Fuzzer. `http://www.peachfuzzer.com/`.

[19] Qualys Web Application Scanning (WAS). `https://www.qualys.com/apps/web-app-scanning/`.

[20] SPIKE Fuzzer. `http://resources.infosecinstitute.com/fuzzer-automation-with-spike/`.

[21] Sulley. `https://github.com/OpenRCE/sulley`.

[22] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[23] Swagger. `https://swagger.io/`.

[24] TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`.

[25] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing Approaches. *Intl. Journal on Software Testing, Verification and Reliability*, 22(5), 2012.

[26] M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 476–485, 1991.