CHECKING SECURITY PROPERTIES OF CLOUD SERVICE REST APIS

[1]M. SHIVANI, [2]D.SAI KRISHNA

[1]MCA Student, [2]Assistant Professor

DEPARTMENT OF MCA

SREE CHAITANYA COLLEGE OF ENGINEERING, KARIMNAGAR

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 in a modular and efficient way. Using these checkers, we found new bugs in several deployed production Azure and Office365 cloud services, and we discussed their security implications. All these bugs have been fixed.

I. 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 and Microsoft Azure , 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 . 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) . 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 . Recently, stateful REST API fuzzing was proposed to specifically test more deeply services deployed behind REST APIs. Given a Swagger specification of a REST API, this approach automatically generates sequences of requests, instead of single requests.

II. LITERATURE SURVEY

1) Model driven security for web services

AUTHORS: MM Alam et al.

Model driven architecture is an approach to increase the quality of complex software systems based on creating high level system models that represent systems at different abstract levels and automatically generating system architectures from the models. We show how this paradigm can be applied to what we call model driven security for Web services. In our approach, a designer builds an interface model for the Web services along with security requirements using the object constraint language (OCL) and role based access control (RBAC) and then generates from these specifications a complete configured security infrastructure in the form of Extended Access Control Markup Language (XACML) policy files. Our approach can be used to improve

productivity during the development of secure Web services and quality of resulting systems.

2) Run-time generation, transformation, and verification of access control models for self-protection

AUTHORS: Chen, Bihuan; Peng, Xin; Yu, Yijun; Nuseibeh, Bashar and Zhao, Wenyun (2014).

A self-adaptive system uses runtime models to adapt its ar-chitecture to the changing requirements and contexts. How-ever, there is no one-to-one mapping between the require-ments in the problem space and the architectural elements in the solution space. Instead, one refined requirement may crosscut multiple architectural elements, and its realization in volves complex behavioral or structural interactions manifested as architectural design decisions. In this paper we pro-pose to combine two kinds of self-adaptations: requirements-driven self-adaptation, which captures requirements as goal models to reason about the best plan within the problem space, and architecture-based self-adaptation, which cap-tures architectural design decisions as decision trees to search for the best design for the desired requirements within the contextualized solution space. Following these adaptations, component-based architecture models are reconfigured using incremental and generative model transformations. Com-pared with requirements-driven or architecture-based approaches, the case study using an online shopping bench-mark shows promise that our approach can further improve the effectiveness of adaptation (e.g. system throughput in this case study) and offer more adaptation flexibility

3. Towards development of secure systems using umlsec.

AUTHORS: Jan J¨urjens

We show how UML (the industry standard in object-oriented modelling) can be used to express security requirements during system development. Using the extension mechanisms provided by UML, we incorporate standard concepts from formal methods regarding multi-level secure systems and security protocols. These definitions evaluate diagrams of various kinds and indicate possible vulnerabilities.On the theoretical side, this work exemplifies use of the extension mechanisms of UML and of a (simplified) formal semantics for it. A more practical aim is to enable developers (that may not be security specialists) to make use of established knowledge on security engineering through the means of a widely used notation.

III. SYSTEM ANALYSIS

EXISTING SYSTEM:

Scanning of Swagger-based Representational State Transfer (REST) APIs - In addition to scanning Simple Object Access Protocol (SOAP) web services, Qualys WAS leverages the Swagger specification for testing REST APIs. Users need to only ensure the Swagger version 2.0 file (JSON format) is visible to the scanning service, and the APIs will automatically be tested for common application security flaws. - Enhanced API Scanning with Postman Support - Postman is a widely-used tool for functional testing of REST APIs. A Postman Collection is a file that can be exported from the tool that clubs together related requests (API endpoints) and shares them with other users. These collections are exported in JSON format. With the release of Postman Collection support in Qualys WAS, customers have the option to configure their API scans using the Postman Collection for their API.

DISADVANTAGES OF EXISTING SYSTEM:

- SOAP APIs are largely based and use only HTTP and XML.
- On other hand Soap API requires more resources and bandwidth as it needs to convert the data in XML which increases its payload and results in the large sized file.
- On other hand SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern.

PROPOSED SYSTEM:

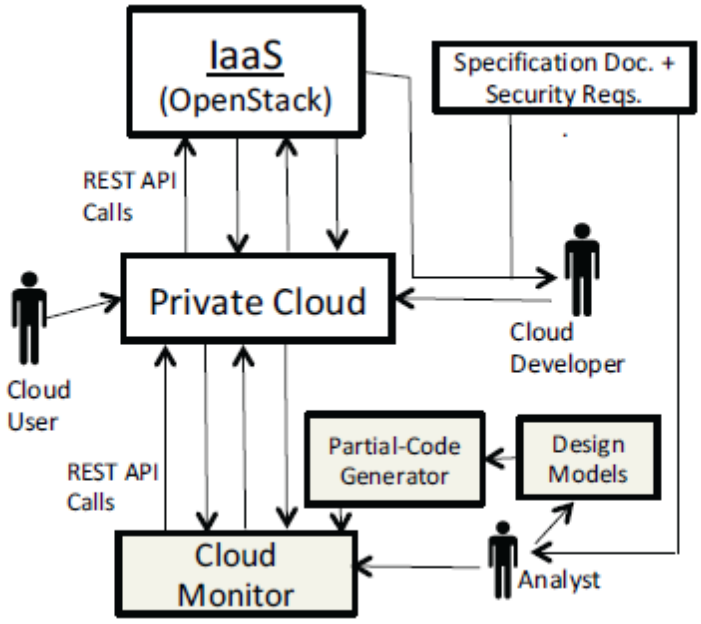
REST APIs are implemented on top of the ubiquitous HTTP/S protocol, and offer a uniform way to create (PUT/POST), monitor (GET), manage (PUT/POST/PATCH) and delete (DELETE) cloud resources. Cloud service developers can document their REST APIs and generate sample client code by describing their APIs using an interface-description language such as Swagger (recently renamed OpenAPI) [25]. 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

ADVANTAGES OF PROPOSED SYSTEM:

- REST APIs are usually simple to build and adapt.
- With the initial URI, the client does not require routing information.
- 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.
- Clients can have a generic 'listener' interface for notifications.

-->The approach is implemented as a semi-automatic code generation tool in Django, a Python web framework.

SYSTEM DESIGN



IV. IMPLEMENTATION

MODULES:

- user
- cloud
- admin
- REST API.

User

It defines the access rights of the cloud users. A volume can be created, if the it has not exceeded its quota of the permitted

volumes and a user Authorization is an important security concern in cloud computing environments. a POST request from the authorized user on the volumes resource would create a new volume. a DELETE request on the volume resource by an authorized user would delete the volume . if the user of the service is authorized to do so, and the volume is not attached to any instance .It aims at regulating an access of the users to system resources.

Cloud

.The cloud monitors contain contracts used to automatically verify the implementation . A cloud developer uses IaaS to develop a private cloud for her/his organization that would be used by different cloud users within the organization. In some cases, this private cloud may be implemented by a group of developers working collaboratively on different machines. We use Django web framework to implement cloud monitor and OpenStack to validate our implementation.

Admin

the cloud administrator using Keystone and users or user groups are assigned the roles in these projects. It defines the access rights of the cloud users in the project. A volume can be created, if the project has not exceeded its quota of the permitted volumes and a user is authorized to create a volume in the project. Similarly, a volume can be deleted, if the user of the service is authorized to do so, and the volume is not attached to any instance, i.e., its status is not in-use.

REST API:

REST is an acronym for **RE**presentational **S**tate **T**ransfer. It is an architectural style for **distributed hypermedia systems**.By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resource through representations; self-descriptive messages; and, hypermedia as the engine of application state.The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting. REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

V. 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 all these bugs to the service owners, and all have been fixed. Indeed, violations of the four security rules introduced in this paper are clearly potential security vulnerabilities. The bugs we found have all been taken seriously by the respective service owners: our current bug “fixed/found” ratio is nearly 100%. Moreover, 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. Finally, it helps that these bugs are easily reproducible and that our fuzzing approach reports no 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 nontrivial 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’2019), May 2019.

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

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

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

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

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

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

[12] R. Lammel and W. Schulte. Controllable Combinatorial Coverage in ¨ Grammar-Based Testing. In Proceedings of TestCom’2006, 2006.

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

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

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