Dynamic Application Security Testing - Fuzzing: Brute-Force API Vulnerability Scanning

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Justin Gou

Spring, 2022

Advisor

Rosanne Vrugtman PhD, Department of Computer Science
Daniel G. Graham PhD, Department of Computer Science.

Technical Report

Abstract

Robinhood Markets, Inc. exposes a large number of API endpoints to the public that are not regularly scanned for potential web vulnerabilities. Large amounts of API endpoints creates a large attack surface for malicious attackers to target. Without regular vulnerability scanning on these publicly accessible endpoints, any vulnerability could pose a large risk on the Robinhood infrastructure.

To address this problem, a solution was proposed to use fuzzing, a method of brute-force dynamic analysis, to automatically test all API endpoints. DAST is implemented as a web API and designed so that developers can easily add custom tests and endpoints to be automatically and regularly scanned.

DAST was enhanced with various web vulnerability scanning capabilities, including scanning for HTTP smuggling, server-side request forgery, authentication bypass, etc. Developers can easily add endpoints and custom scans as needed. The project was not ready for production deployment at the end of term due to difficulties with deployment and complications with load testing, as brute forcing will cause an unnecessary amount of noise. With deployment, DAST will be integrated with Robinhood's internal communication system to report findings.

With this research, the DAST system is now able to perform automated testing for common web vulnerabilities across hundreds of API endpoints, significantly reducing risk and potential for an external breach.

1. Introduction

"A chain is only as strong as its weakest link". This idiom is true in many situations, where if the weakest part of a system falls, the entire system will crumble. The same can be said about any given computer system in the sense of cybersecurity; if any part of the system is vulnerable to an attack, the entire system is vulnerable, regardless of how strong the rest of the system is. In this situation, the Robinhood app, a mobile application designed to allow everyday people to participate in financial investments in stocks, options, cryptocurrency, etc., is run with web API endpoints, all of which are public accessible and not regularly tested for vulnerabilities.

Any publicly accessible domains become easy targets for attackers, as it requires no extra effort to access the system. To counteract this, regular testing should be performed on these endpoints to scan for common web vulnerabilities. At Robinhood, I implemented such testing features through extension of an internal Dynamic Application Security Testing (DAST) system. Through this research, developers could add specified endpoints to the system and the system would automatically scan for vulnerabilities, which would then be reported back to proper entities via internal messaging system.

2. Background

Any webpage is vulnerable to a variety of web application vulnerabilities. The Open Web Application Security Project (OWASP) keeps an annually updated list called the OWASP Top Ten, which highlights the top ten most common web vulnerabilities of any given year [1]. Some of these include SQL injection, server-side request forgery (SSRF), broken authentication, etc. To highlight the importance of protecting systems against these attacks, in 2019, Capital One was victim to a SSRF attack, causing a breach of more than 100 million customers' personal data [2].

This could have easily been prevented if the vulnerability were caught internally before the attacker was able to it.

A Dynamic Application Security Testing (DAST) system is a common black-box technique for information security teams to test code through execution as a form of dynamic analysis. The DAST system at Robinhood was implemented as a Django web API and was designed with the idea that general software developers could add endpoints to the system to be automatically scanned on a regular basis.

Web application APIs must be defined in a way that can be processed by the DAST system. To do so, APIs can be essentially serialized using the OpenAPI schema format, sometimes known as Swagger specifications. These schemas are in the form of JavaScript Object Notation (JSON) and provide information such as the endpoint, the HTTP method, the request parameters, etc. These schemas can easily define hundreds of API endpoints to be processed and scanned by DAST.

To perform dynamic analysis, one common technique is known as fuzzing, which is the idea to essentially brute-force attempt inputs until the system fails to handle the input. In a typical example, a tool like American Fuzzy Lop (AFL) is used to automatically run a program thousands, if not millions, of times to attempt to achieve unexpected behavior. Very minimal research has been done on fuzzing web APIs, however, the same idea follows, where various inputs will be passed into the data or query sections of an HTTP request to attempt to cause unexpected behavior.

3. Related Works

Various DAST systems are commercially available, including software by Acunetix, Fortify WebInspect, Tenable.io, etc. [3]. In all cases, these external technologies have a price associated with them, which Robinhood was not looking to allocate money for. While some of these scanners provide features that match the specifications of my research, such as Fortify WebInspect's ability to process OpenAPI schemas, the reason for avoiding these commercial scanners is simply due to the cost and lack of customizability for the infrastructure of Robinhood. With an internally developed system, security engineers have full control over how endpoints are tested. The system is not significantly difficult to implement, thus, the choice to develop an internal system was clear.

One common open-source web fuzzing tool is called Fuzz Faster U Fool (FFUF), which is supported by Offensive Security, a renowned cybersecurity organization [4]. I personally researched this tool as an option to perform web API fuzzing, but eventually decided against using it due to the difficulty of integrating such a tool with the architecture of DAST. It would also be difficult to adapt the tool to generalized API endpoints, as it was designed to fuzz specific endpoints with known information about the endpoint, which is not fully defined through OpenAPI specifications.

4. Project Design

DAST had already been in development when my research began. The original state of the system had only included features for synthetic monitoring, which is a technique to simulate API calls to ensure the APIs are returning the expected results. While this is useful to ensuring proper functionality of the APIs, this feature does not scan for vulnerabilities and does not perform automated tests across large amounts of endpoints, as it is expected that developers manually create the test cases.

4.1 System Architecture

DAST is implemented as a Django web API, where visiting different URLs will perform scans. In the backend, the registry consists of Endpoint and Synthetic objects. Synthetic objects contain information needed to make a synthetic API call, including a validation condition. Endpoint objects store endpoint information, such as the URL, HTTP method, query/body parameters, etc.

The app is built using Bazel, which is an open-source automated build and test tool developed by Google [5]. DAST is built as a Python binary with dependencies drawn from the monorepo workspace. For my project, I had to create new build targets to support the software I would use for the API fuzzing.

4.2 Adding Fuzzing

The original infrastructure of DAST did not easily support API fuzzing, as I needed some way to ingest OpenAPI schemas and automatically run synthetics on those endpoints. To do this, I needed some external software. I explored a variety of different fuzzers, include BooFuzz, APIFuzzer, ChopChop [6, 7, 8]. After thorough analysis of these tools, the tools were either not suited for the architecture of the DAST system, showed little community support, or simply did not provide the features for proper fuzzing of API endpoints.

After further exploration and discussion with other security engineers, I decided to use a tool called Nuclei, an open-source template-based vulnerability scanner maintained by Project Discovery [9]. This tool is deployed as a Go binary, which is suitable for the DAST architecture, as Bazel supports the use of Go binaries. The template-based nature of the tool allows for incredible flexibility in determine how and what scans would be run on which endpoints. It supports multiple forms of validation, including HTTP status code, HTTP body regex matching, and content length checks, that can be easily used to check for various web vulnerabilities.

The templates take the form of a YAML file and are run using a command-line interface. To integrate this with the DAST architecture, which is in Python, the templates were created with the YAML PyPi package written into tempfiles to avoid unnecessary clutter. The YAML files were then run by Nuclei using a subprocess call in Python. This was a difficult decision to make because there was no easy way to run the command-line interface that wouldn't introduce vulnerabilities. Using a subprocess call may introduce vulnerabilities, as if an attacker were able to modify the command to be run, they could achieve arbitrary code execution, which would be a significant breach in the system. Fortunately, after further analysis, because the tool was an internal tool that would not be accessible by external attackers, the Application Security team concluded that using subprocess would be safe for a minimum viable product (MVP).

With a way to run Nuclei through Python, I next needed to add a way to automatically fuzz ingested endpoints through the DAST system. The first step to this was to ingest the OpenAPI schemas, which defined the endpoints of a given API. To do this, I essentially needed a way to parse the JSON file, which I could easily do with the JSON PyPi package. Unfortunately, it was not going to be this simple due to the nature of the OpenAPI schemas. I was able to extract most of the information I needed to define the endpoints using the JSON PyPi package, however, some endpoints had complex body parameters that were defined in additional fields in a recursive manner. I needed access to these parameters in order to properly simulate API calls, as some endpoints have required parameters. In order to resolve these recursive parameters, I used an open-source Python package called Prance, which had a ResolvingParser object that did exactly what I needed [10].

With properly resolved parameters, I was able to populate the DAST registry with the Endpoint objects necessary to perform scanning. Next, I needed a way for the DAST API to automatically perform scans on all the endpoints created. To do this, I had to re-architecture parts of the DAST, specifically, the Synthetic registrant class. The idea was to have the API generate a set of Synthetic objects when a scan was invoked. The original Synthetic class did not support automatically generated Synthetics, as the scans were implemented by searching for the explicit definition of the scan code to run, which in the case of generated Synthetics, did not exist. To do this, I used a Polymorphic model, where Synthetics could be GeneratedSynthetics, which hold an endpoint and a SyntheticGenerator, which was simply a function that generated and invoked the YAML Nuclei templates. With this reorganization, I was able to add a new endpoint to the DAST API, called "/fuzz", which would automatically run all fuzzing tests on the specified endpoint. Upon deployment, the DAST endpoint can be invoked by automated scans to regularly perform fuzzing on all generated Endpoint objects.

4.3 Nuclei Templating

The idea of a template is rather simple: users can create templates to trigger HTTP requests in a certain pattern and perform validation on the returned results to determine if the request performed as expected. The implementation of such template becomes more complicated than expected, as Nuclei runs off very specific formatting and refuses to run templates if any slight formatting is incorrect. To list a few, I had to modify the way YAML PyPi package handled blocked strings (multi-line strings), outputted quotes, and indentation size. This was incredibly tedious to deal with because if Nuclei noticed the formatting was incorrect, it simply would not run the template with no error message as to why.

The general structure of the template is as follows: general information, HTTP requests (raw or parameterized), matchers.

The general information included information such as the author, severity of the vulnerability, the template name, etc. This information was populated with general information to fulfill the template requirement.

The HTTP requests for the fuzzing templates I created were all created as raw HTTP requests. This means I constructed the HTTP request with the proper parameters, including HTTP method, URL path, authentication token, body/query parameters, etc., depending on the vulnerability I was testing for. I had originally attempted to use the built-in parameterized HTTP requests, where I could specify the information using YAML fields instead of putting in the raw HTTP request string, however, this was significantly restricting on the types of requests I could make and did not work for many of the vulnerabilities I was testing for, so I figured it was easier to simply created the raw HTTP requests.

After specifying the requests, the last structure of the templates was simply the matchers. Nuclei defines a variety of different types of matchers to determine if the HTTP request made returned the expected result. In most cases, the matchers I used were simply matching for the HTTP status code, making sure the request was made properly and successfully returned, or failed to return, information. Sometimes determining the matchers was difficult, as it was difficult to have the matcher work as intended for all generalized endpoints. There were situations where I had the matcher working for some of the endpoints but not others simply because some of the requests made returned different results. I got around this issue by making the matchers as generalized as possible to satisfy any miscellaneous cases.

Using this structure, I was able to write Python code to generate proper Nuclei templates in YAML format to perform vulnerability scans on the API endpoints.

4.4 Web Vulnerabilities

For my research, I implemented six different website application vulnerability scans on top of a simple health check: authentication verification, basic authentication bypass, HTTP method brute-force, HTTP CL-TE smuggling, server-side request forgery, and X-Forwarded-For authentication bypass.

The health check was the first scan I implemented. This health check simply verified that the endpoint was active and accurate to the OpenAPI definition. This will ensure that the OpenAPI JSON files are properly maintained with the latest information, as outdated information could cause problems since the OpenAPI JSON files are used elsewhere to define the endpoints. This check was implemented by simply checking that a request made to the URL path did not return an invalid HTTP status code, namely 500, 503, etc.

The next vulnerability I checked for was an authentication verification scan. Robinhood is a large financial technology company and all the endpoints must be secured with authentication in to avoid information leaks, as all information stored is incredibly critical to customers and would cause significant financial troubles if leaked. Because of this, it is beneficial to verify that all API endpoints must be accessed with a valid JSON Web Token (JWT), also known as an authentication token. This token was passed into DAST with an AWS encrypted secret and propagated into the HTTP request under the Bearer HTTP request field. This token must be fully encrypted until use due to the sensitivity of the token, as access to this token could cause an Account Takeover (ATO). When the request is made with the proper authentication token, a valid HTTP response code is expected, i.e. 2xx. This is enough information to define a Nuclei template to check that a valid authentication returns a valid HTTP response.

Another form of authentication, besides using a JWT, is using Basic authentication. Unfortunately, basic authentication includes a weak encryption technique, which if the request were intercepted, could easily be decoded and used to cause an ATO. This check is easy to perform, as allowing basic authentication requires a special signature in the HTTP response that can be checked using HTTP body text matchers.

In the essence of fuzzing, another scan I performed was verifying that the only valid HTTP method for any given URL is the one specified in the OpenAPI JSON file, i.e. if an endpoint is said to accept GET requests, it does not accept POST requests. The reason for such a check is because if any other HTTP methods are open, it could be exploitable in unpredictable ways, as there is no information as to why the method is accepted. To implement this fuzzing, I had to use a payload, which is the idea of substituting in values from a provided wordlist. I provided a list of all possible HTTP methods and ensured that the only method that returned a valid HTTP response code was the one specified in the OpenAPI definition.

One common web vulnerability is known as HTTP smuggling, or sometimes known as HTTP desync. This occurs when the backend server handling the request disagrees with the requests made by the frontend. One way this could happen is if the user specified Content-Length and Transfer-Encoding fields in the raw HTTP request. In this type of HTTP smuggling, the frontend server uses the Content-Length header to determine the boundaries of the HTTP request, while the backend server uses the Transfer-Encoding header [11]. If these do not align, it will cause a desync between the servers, and sometimes will trigger an unexpected request. If an attacker is able to cause the backend server to make additional requests, they could easily obtain information they do not have access to, compromising the integrity of the system. This scan was easy to implement with Nuclei, as I was able to specify multiple raw HTTP requests, each with a Content-Length and Transfer-Encoding header that would cause a desync if the server was vulnerable to such an attack. I could determine if the attack was successful in determining if the HTTP response was from the additional HTTP request that should not have been made.

Another vulnerability scan I implemented was a server-side request forgery attack. This occurs when the attacker causes the server to return information that should not be directly accessible to the attacker (forging a server-side request, hence the name). In particular, I implemented the same SSRF attack that was used in the Capital One breach in 2019, where AWS provided an insecure way of obtaining metadata for provided EC2 instances, known as IMDS. The metadata could be obtained by simply making a request to an unprotected URL, 169.254.169.254. Such metadata could then be used to obtain reverse shells and remote access to the servers. This was rather simple to implement, as I simply needed to make a few requests to try to cause the endpoint to visit the metadata-obtaining URL, such as passing it in as a query parameter or in the body. In the response, I simply looked for an authentication token, as that is what the metadata is expected to return.

Finally, I looked for another way to bypass authentication using the X-Forwarded-For HTTP header. This header works because it specifies where the HTTP request originated from. In some cases, specifying the origin of the request as the server itself will bypass authentication, since the server believes it was a safe request made by itself. This was easy to implement with Nuclei, as I could specify the X-Forwarded-For header in the raw HTTP request as 127.0.0.1, which simply means the server itself (the localhost).

5. Results

From this experience, I extended Robinhood Markets' in-house DAST system to include an automated fuzzing feature for API endpoints to regularly scan for potential web vulnerabilities. The system will enable developers to perform automated scans on their APIs to ensure the security of the applications. DAST is currently maintained by the application security team at Robinhood, whom I worked closely with to develop this research.

The DAST project at Robinhood is still under development, but is planned to be deployed in the near future for software developers to use. With my contributions, regular vulnerability scanning of publicly accessible API endpoints will be automated, drastically reducing the risk of a successful cyberattack.

6. Conclusion

Through this project, I was able to add an automatic fuzzing feature to Robinhood's internal DAST system, which would ingest OpenAPI schemas and run scans on those endpoints. The scanning searches for authentication verification and bypass techniques, HTTP smuggling vulnerabilities, SSRF attack potential, etc. The scanning was implemented with a template-based vulnerability scanner, Nuclei, and integrated with the Django/Bazel framework DAST was originally built off. In the future, when DAST is deployed, software engineers can simply add the OpenAPI schema to DAST and DAST will automatically, regularly run scans on those endpoints, reporting any results to the engineers.

7. Future Work

When I left Robinhood, DAST was not ready for deployment, as it was having build issues in production. The application security team will continue developing DAST and put my work into production.

Further work in API fuzzing could include searching for a wider variety of vulnerabilities. As an intern, I was unfamiliar with the vulnerabilities and it was difficult to find a way to test for them, hence restricting my ability to provide a large range of test cases. For example, some other vulnerabilities to look for include command/SQL injection, broken object level authorization, cross-site request forgery, etc.

8. Evaluation

My program at UVA has prepared me well for this project. In CS 3240, Advanced Software Development, I learned to use the Python web framework, Django, which I worked directly with at Robinhood. It was incredibly important to understand how the Model-View-Controller (MVC) architecture worked, as it allowed me to easily make changes to the models to adapt to my updated generated synthetics. CS 3240 also prepared me in using Git/Github, the distributed version control system used at Robinhood.

As my role at Robinhood was cybersecurity focused, CS 3701, Introduction to Cybersecurity, also prepared me for this. I was able to gain enough knowledge to understand how some of the vulnerabilities worked, such as SSRF. While it was nowhere near comprehensive, I was much more prepared to work in such a field through the knowledge from CS 3701.

References

[1] OWASP Contributors. 2021. Owasp Top Ten. (September 2021). Retrieved October 27, 2021 from https://owasp.org/www-project-top-ten/

[2] Rob Wright. 2019. Capital one hack highlights SSRF concerns for AWS. (August 2019). Retrieved October 27, 2021 from https://searchsecurity.techtarget.com/news/252467901/Capital-One-hack-highlights-SSRF-concerns-for-AWS

[3] David Balaban. 2021. Top DAST tools 2021: Dynamic Application Security Testing. (October 2021). Retrieved October 27, 2021 from https://www.serverwatch.com/reviews/dast-dynamic-application-security-testing/

[4] FFUF. 2021. Fast web fuzzer written in go. (May 2021). Retrieved October 27, 2021 from https://github.com/ffuf/ffuf

[5] Google. 2021.(2021). Retrieved October 27, 2021 from https://bazel.build/

[6] Joshua Pereyda. 2021. Boofuzz: A fork and successor of the Sulley Fuzzing Framework. (September 2021). Retrieved October 27, 2021 from https://github.com/jtpereyda/boofuzz

[7] Peter Kiss. 2021. APIFuzzer: Fuzz test your application using your openapi or swagger API definition without coding. (October 2021). Retrieved October 27, 2021 from https://github.com/KissPeter/APIFuzzer/

[8] Michelin. 2021. Chopchop: Chopchop is a CLI to help developers scanning endpoints and identifying exposition of sensitive services/files/folders. (January 2021). Retrieved October 27, 2021 from https://github.com/michelin/ChopChop

[9] Project Discovery. 2021. nuclei: Fast and customizable vulnerability scanner based on simple YAML based DSL. (October 2021). Retrieved October 27, 2021 from https://github.com/projectdiscovery/nuclei

[10] Ronny Pfannschmidt. 2021. Prance: Resolving Swagger/openapi 2.0 and 3.0 parser. (September 2021). Retrieved October 27, 2021 from https://github.com/RonnyPfannschmidt/prance

[11] Port Swigger. 2021. What is HTTP request smuggling? (2021). Retrieved October 27, 2021 from https://portswigger.net/web-security/request-smuggling