Go Fuzz Yourself:

How to Find More Vulnerabilities in APIs Through Fuzzing

Summary
This white paper demystifies API fuzzing, why it should be incorporated into every API penetration test, and what free, open source fuzzers are available and how to use them.

KNIGHTINK

Author Information
Alissa Valentina Knight
Partner
Knight Ink
1980 Festival Plaza Drive
Suite 300
Las Vegas, NV 89135
ak@knightinkmedia.com

detectify

Publication Information
This white paper is sponsored by Detectify.

Initial Date of Publication:
August 2021
Revision: 1.0

TABLE OF CONTENTS

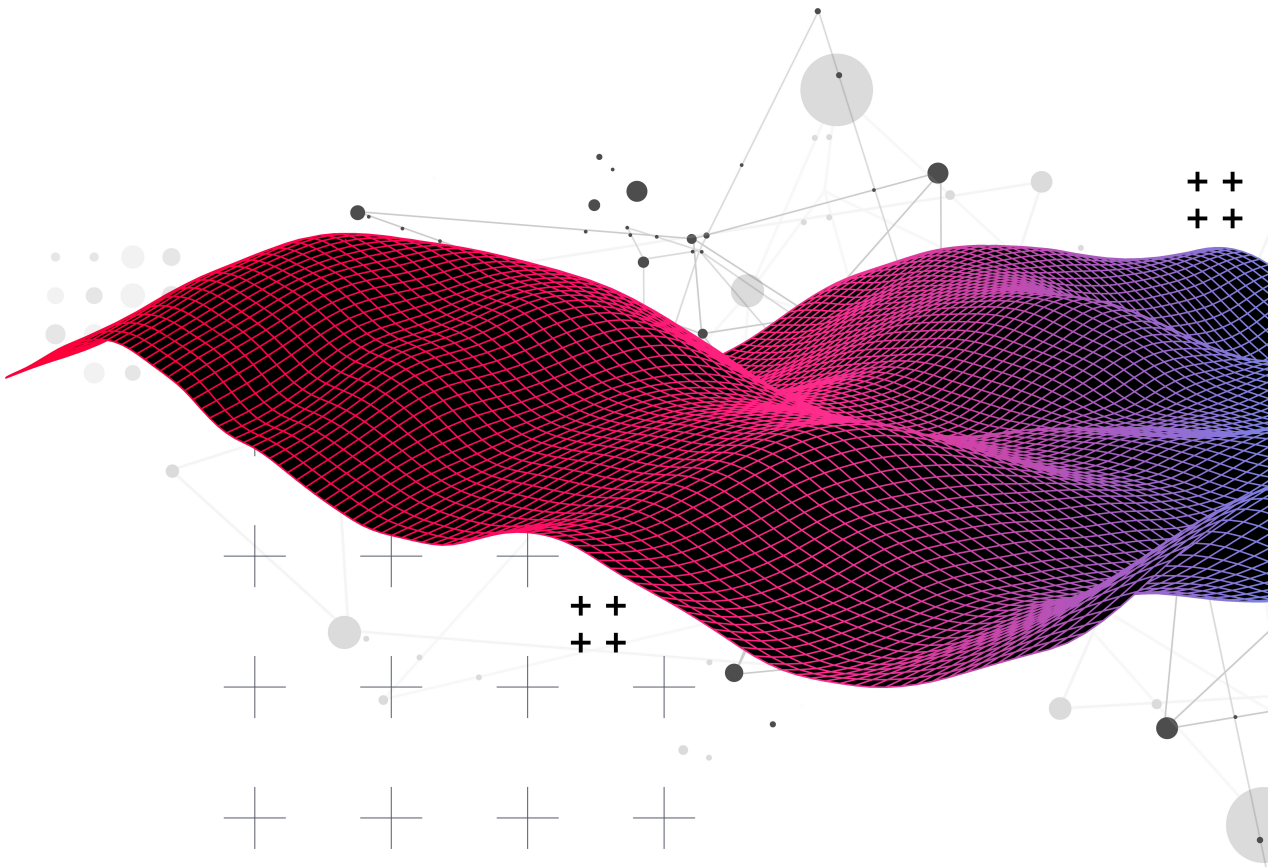


TABLE OF CONTENTS

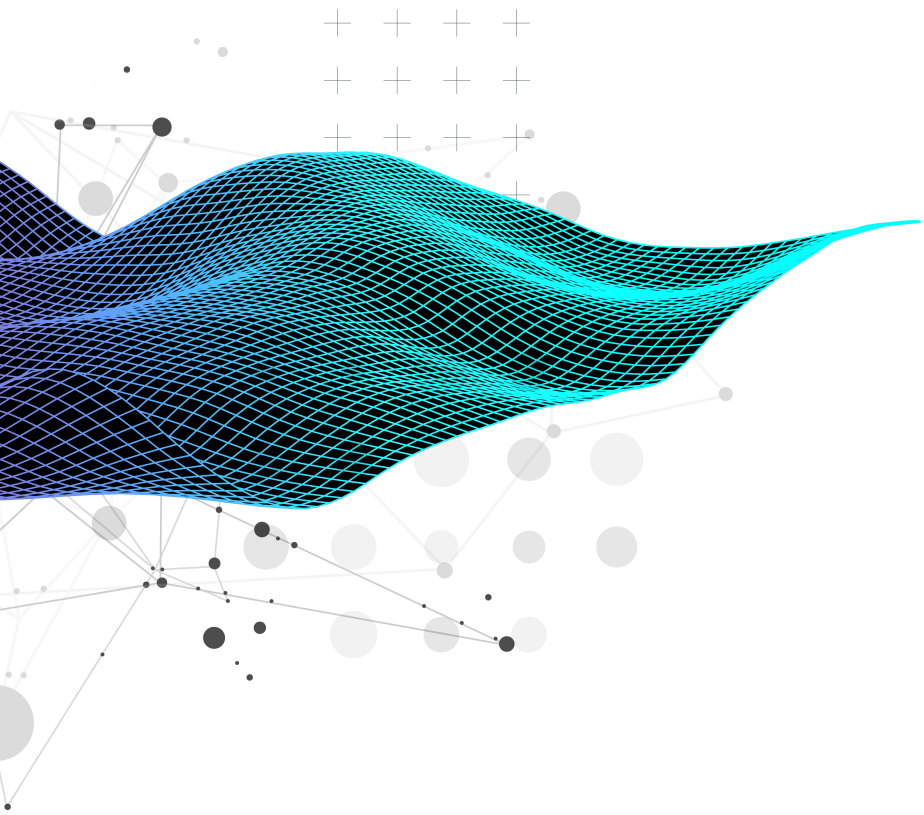


TABLE OF CONTENTS

KEY TAKEAWAYS

- Omitting fuzzing from your penetration testing of APIs leaves vulnerabilities undetected that other tactics and techniques won't find.
- Fuzz testing of APIs is setting the value of API operation parameters to a value that the developer didn't expect to receive.
- Because APIs process untrusted inputs, fuzzing is fundamental to the penetration testing process, often finding vulnerabilities missed by static program analysis and manual code inspection.
- Content discovery is the identification of unlinked files and folders in a web application. Specifically, when targeting APIs, it is an effective tool when attempting to discover undocumented API endpoints.
- API documentation is different from API specifications, such as OpenAPI (formerly Swagger). API documentation provides a reference manual for both developers and non-developers in how the API behaves and what the expected inputs and responses are in plain English.
- API specification files such as OpenAPI is a format for describing an API, defining all its available endpoints (such as /R4/patients) and operations (such as PUT), supported authentication methods, contact information, and more. Currently at version 3, OpenAPI files are clear text and written in specific languages, such as YAML or JSON.
- HTTP verbs (methods) specifies a specific action for the API endpoint to perform, such as GET, POST (the most commonly used verbs), and for REST APIs, PUT, PATCH, and DELETE. Each verb corresponds to a particular action and expected response code, such as 200 (OK). Meaning, the action was performed successfully. 401 (Unauthorized) or 404 (Not Found), means the file or resource doesn't exist. Other response codes exist, but these are the most common.
- Content discovery in APIs, specifically to the end of discovering API endpoints requires patience. It wasn't until about two hours into the scanning effort and attempting different base URLs before I was able to get something other than a HTTP 403 or 404 error code response from the EHR systems I was targeting in my vulnerability research campaign. When I finally did, it was a HTTP 200 and HTTP 415 response to a POST to an endpoint that I sent to Burp and continued to manipulate within Burp Repeater.

INTRODUCTION

Application programming interfaces (APIs) is essentially a way to describe the protocols and tools used for two systems to talk to each other — akin to a Rosetta Stone between applications that allows them to talk that couldn't previously understand each other without APIs.

A false sense of security is created when a Chief Information Security Officer (CISO) thinks her APIs are secure simply because regular penetration testing is performed. This false sense of security can lead to an expensive data breach and loss to customer and shareholder confidence.

An organization can think they're secure because penetration testing and vulnerability assessments are being performed of their APIs, which can include static and dynamic code analysis and testing against the OWASP API Security Top 10. But when the penetration tester doesn't include fuzzing in her testing, this can leave a number of critical vulnerabilities undiscovered.

This white paper provides a deep-dive into what API fuzzing is, what free, open source fuzzing tools exist, how to use them, and more importantly, how to interpret their results.

Consequently, this paper aims to inform CISOs and other cybersecurity leaders on the criticality of ensuring that penetration testers performing testing of the APIs understand the profound importance of incorporating fuzzing into their tactics and techniques to ensure more thorough and accurate results.

Audience

This paper was written for red team members who want to learn how to properly perform comprehensive penetration testing of their APIs by incorporating fuzzing into their tactics and techniques used in testing.

Additionally, CISOs and other cybersecurity leaders can use this white paper to be better informed on what their expectations should be of those performing penetration tests of the APIs they are responsible for securing and that the results of fuzzing are a fundamental part of the final report delivered to them from the penetration tester(s) performing the work.

PROBLEM STATEMENT

Penetration testing of APIs performed today is profoundly inadequate if fuzzing isn't performed as one of the tactics employed against the endpoint. Many penetration testers don't include fuzzing in their repertoire of tactics and techniques either because:

1. They haven't been exposed to fuzzing;
2. Don't know how to use fuzzing tools; or
3. Don't know how to properly interpret the results.

This creates a false sense of security with the organization thinking all possible vulnerabilities were found for remediation, leaving potentially vulnerable APIs exposed to the internet.


Because APIs process untrusted inputs, fuzzing is fundamental to the penetration testing process, often finding vulnerabilities missed by static program analysis and manual code inspection.

When Penetration Testing Doesn't Include Fuzzing

Vulnerability Scanners
Approaching penetration testing of web applications can be done using manual testing of vulnerabilities in the OWASP Security Top 10, such as SQL injection, cross-site scripting (XSS), and more; and automated scanning using tools, but these same scanners can't be used to find vulnerabilities in APIs due to the nature of how different APIs work from traditional web apps.

There are two types of approaches to identifying vulnerabilities in applications, static application security tools (SAST) and dynamic application security tools (DAST). These scanners approach vulnerability identification differently.

SAST performs what's also referred to as "static code analysis" where the source code is checked line-by-line for vulnerabilities, often responsible for a lot of false positives.

SAST lacks context in security, meaning it doesn't understand how an API fulfills its contract or uses the data coming into it and is blind to third-party libraries and frameworks where source code is typically not available. DAST was designed to send Web 1.0 style page parameter requests during the running state of an application. DAST can not invoke the API as it's unaware of how to form the requests expected by the API endpoints. (Williams, J., 2015)

API Fuzzers
There are different taxonomies of fuzzers that leverage different techniques towards the same goal. While different people opine on what these categories of fuzzers are, it's generally accepted that Microsoft's taxonomy makes the most sense. These include:

- Knowledge of Input;
- Knowledge of target application structure; and
- Method of generating new input.

Knowledge of Input
This category of fuzzers are also referred to as "smart fuzzers" as they have a knowledge of the application's expected input and format. For example, the encoding expected such as Base64 or file-type, such as PE headers.

Knowledge of Target Application Structure
Before I decompose the different taxonomies of fuzzers, it's important to note that in the broader hemisphere of testing is black box and white box testing. In lay terms, white box testing is simply where the tester or the tool has the requisite information needed of the target that obviates the need to figure it out (a la guessing) while black box testing is the antithesis of that, ultimately requiring the tester or the tool to figure it out on their own by hunting for the answers to those questions.

In the application of fuzzers, the smartest fuzzers will have full (white box) or partial (gray box) information on the target API.

API FUZZERS

API FUZZERS

Method of Generating New Input

Fuzzers are capable of generating the test input data from scratch randomly or semi-randomly or by taking pre-existing properly structured stimulus by mutating it to generate different permutations for stimulus to evaluate how the target application responds.

This paper demonstrates the efficacy of two different types of fuzzers, Kiterunner and RESTler and decomposes the idiosyncratic distinctions between them through the presentation of their empirical output run against public APIs.

Fuzzing should be used because it's an effective way to find security vulnerabilities in applications.

According to Microsoft, thousands of vulnerabilities have been discovered as a result of fuzzing. Fuzzing has become so critical to the software development life cycle (SDLC) that Microsoft requires it as part of its Microsoft Security Development Lifecycle for every untrusted interface.

Content Discovery

Content discovery is the automated identification of hidden/unlinked resources that aren't visible or able to be spidered in a target web application. While content discovery tools date back several decades, this class of tools have continued to evolve from Web 1.0 to modern applications, such as APIs to become more context-aware.

Content discovery is performed differently across different tools, such as Burp suite, which uses name guessing, web crawling, and extrapolation from naming conventions. Traditional content discovery tools simply look through a wordfile to find files or folders that might exist on the target web server. Newer content discovery tools have evolved to enable targeting of modern web applications such as APIs. Many content discovery tools are now specifically made for RESTful APIs using openAPI (formerly Swagger) files in order to search for files and folders of interest typically placed there with the different API frameworks.

Routes and Endpoints

In traditional web applications, filenames and folders are mapped to actual files and folders that exist on the web server, e.g. https://www.server.com/index.php, which actually maps to a file on the web server called index.php. However, in modern web applications, such as APIs, you'll have a server https://www.server.com/api/r4 which specifies a route. On the server is a routes file that points to a snippit of code instead of an actual file. In the latter example URI, /r4 is the API endpoint.

THE OPENAPI SPECIFICATION

The OpenAPI Specification

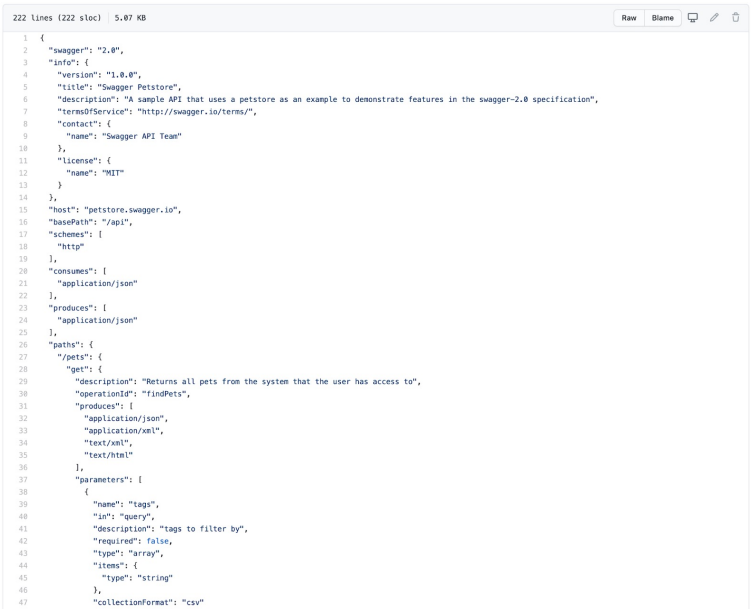
Originally called Swagger specification, OpenAPI specification is a format for describing an API, defining all of its available endpoints (such as /R4/patients) and operations (such as PUT), supported authentication methods, contact information, and more. Currently at version 3, OpenAPI files are clear text and written in specific languages, such as YAML or JSON.

An OpenAPI spec can produce documentation for an API so that anyone, even non-developers can understand what the API is for, how to access it, what the expected inputs and outputs are, and more in plain english.

In summary, the OpenAPI interface-description language for REST APIs has largely become the most popular specification for describing how to access a cloud service through its REST API that details what requests it can handle, what responses may be received, and the response format (Atlidakas, V, Godefrroid, P, Polishchuk, M, 2019).

The OpenAPI specification should come first and can generate the documentation for the API, but it can go further than that, including generating a mock server, the stubs of the API for publishing, the SDK for the different stacks for creating the API consumers from a single source of truth.

Figure 1. Sample OpenAPI File



```
222 lines (222 sloc)  5.07 KB                                                Raw  Blame
{
  "swagger": "2.0",
  "info": {
    "version": "1.0.0",
    "title": "Swagger Petstore",
    "description": "A sample API that uses a petstore as an example to demonstrate features in the swagger-2.0 specification",
    "termsOfService": "http://swagger.io/terms/",
    "contact": {
      "name": "Swagger API Team"
    },
    "license": {
      "name": "MIT"
    }
  },
  "host": "petstore.swagger.io",
  "basePath": "/api",
  "schemes": [
    "http"
  ],
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json"
  ],
  "paths": {
    "/pets": {
      "get": {
        "description": "Returns all pets from the system that the user has access to",
        "operationId": "findPets",
        "produces": [
          "application/json",
          "application/xml",
          "text/xml",
          "text/html"
        ],
        "parameters": [
          {
            "name": "tags",
            "in": "query",
            "description": "tags to filter by",
            "required": false,
            "type": "array",
            "items": {
              "type": "string"
            },
            "collectionFormat": "csv"
```

Source: Knight Ink

HTTP Methods, Headers, and Values

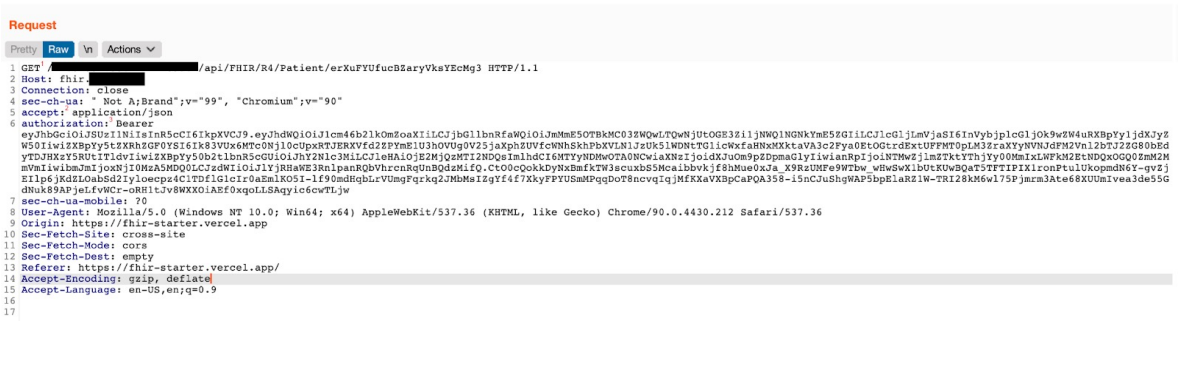
HTTP headers are options that can be passed along from an HTTP client to an HTTP server (web server) or visa-versa from an HTTP server to a client.

These values can contain a number of different options. The different header types include request headers, response headers, representation headers, and payload headers. Header options can include values, such as Authorization, which contains authentication credentials; or a cookie header, which can contain a stored HTTP cookie that the server uses to identify an already authenticated user/session. See Figure 2 below for a sample.

HTTP verbs (methods) specify a specific action for the API endpoint to perform, such as GET, POST (the most commonly used verbs), and for REST APIs, PUT, PATCH, and DELETE. Each verb corresponds to a particular action and expected response code, such as 200 (OK), meaning the action was performed successfully; 401 (Unauthorized), or 404 (Not Found), meaning the file or resources doesn't exist. Other response codes exist, but these are the most common.

HTTP parameters are the variable part of a resource that determine the type of action you want to take. These are passed as options to the endpoint in order to elicit a specific response, such as header parameters, query parameters, and so-on.

Figure 2. Sample OpenAPI File



Source: Knight Ink

Fuzzing
WITH KITERUNNER

CONTENT DISCOVERY WITH KITERUNNER

Overview

In this section, I present a real-life use case of Kiterunner employed against public APIs. The empirical data presented in this paper underscores the importance of running content discovery tools against APIs in order to find unlinked files and folders accessible via the API. Additionally, this paper provides prescriptive step-by-step use cases of how to use Kiterunner against API endpoints and the results that specific stimulus generated by Kiterunner provides.

Kiterunner is a tool for performing forced browsing, checking for accessibility to unlinked files or folders that aren't directly referenced in the web application but available to adversaries. Kiterunner doesn't simply guess for random permutations of different files and folders. Its creator, Shubham Shah, codified over 67,000 different openAPI files he gathered across the internet that Kiterunner uses for its checks that can be typically found in APIs generated by off-the-shelf frameworks like Flask, Rails, and Express.

Kiterunner's braintrust is built around how APIs are built by using these openAPI files instead of just random brute forcing.

Content discovery is the use of an automated tool targeted at web servers that typically uses a flat textfile of folders and filenames that it cycles through in order to find files and folders in the "web root" of the server.

Installing Kiterunner (Mac)

```
$ brew install go
$ sudo apt install golang-go
```

Linux

```
# build the binary
make build


# symlink your binary
ln -s $(pwd)/dist/kr /usr/local/bin/kr


# compile the wordlist
# kr kb compile <input.json> <output.kite>
kr kb compile routes.json routes.kite


# scan away
kr scan hosts.txt -w routes.kite -x 20 -j 100 --ignore-length=1053
```

Running with Kites

1. Download large and small wordlists (.kite files) from the Kiterunner Github page

   The JSON datasets can be found below:
   - routes-large.json (118MB compressed, 2.6GB decompressed)
   - routes-small.json (14MB compressed, 228MB decompressed)

Alternatively, it is possible to download the compile .kite files from the links below:

- routes-large.kite (40MB compressed, 183M decompressed)
- routes-small.kite (2MB compressed, 35MB decompressed)

2.
```
kr scan https://<ip>/ -w routes-large.kite
```

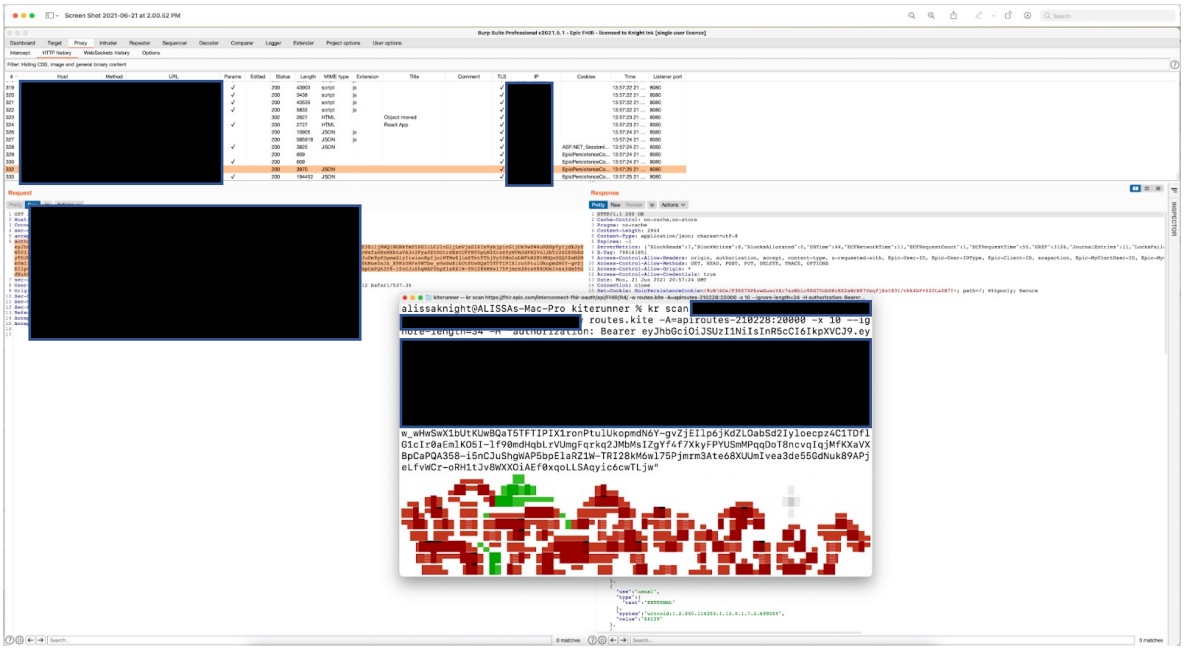
Tokens: Modifying Your Header

Hopefully, the API you're targeting has implemented bearer tokens and is using tokens for authentication and scopes for authorization. If the API has implemented a Bearer token, it could be a random string of characters or a JSON Web Token (JWT), which encodes the access token into an access token and ships in the form of a JSON object.

If the API has implemented tokens, you'll need to authenticate with the API using the Chromium browser within Burp Suite for example, then copy and paste the token you receive into your command line for Kiterunner in order to send authenticated requests to the API. Otherwise, not doing so will result in a bunch of HTTP 403 (FORBIDDEN) error messages rendering your Kiterunner scan useless.

In order to pass fields to the header of your requests within Kiterunner, pass the -H flag to Kiterunner and specify your header fields, in our case, the Bearer token between quotes. (Figure 3).

Figure 3: Virtual application path based routing illustration



Source: Knight Ink

Kiterunner in the Weeds

Depth Scanning

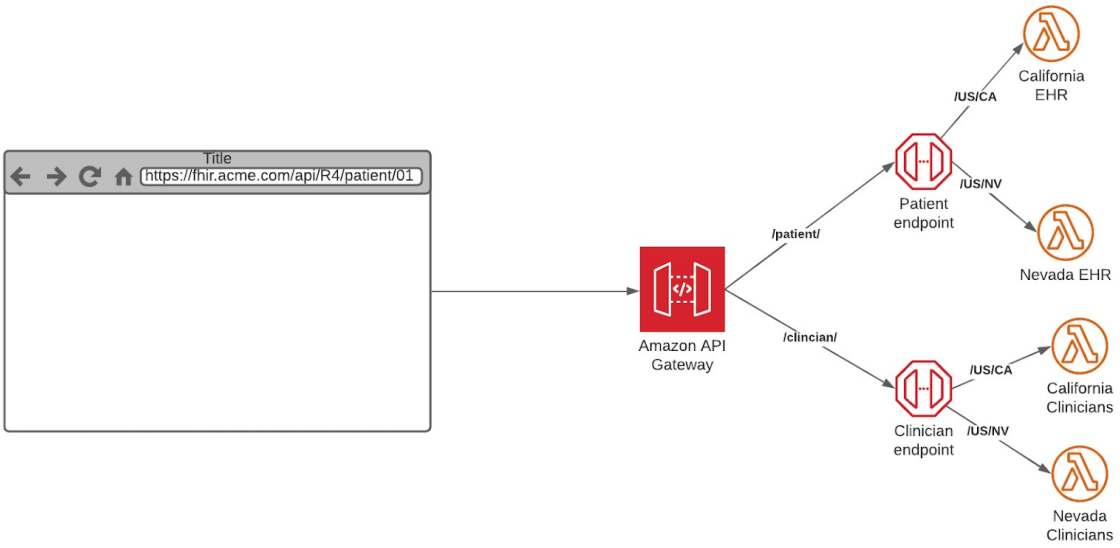
A particularly powerful feature of Kiterunner is its ability to define how many levels deep within the folder structure a scan should be performed of a path, especially useful in path-based routing (Figure 4).

Path-based routing is simply being able to route traffic to backend servers based on paths defined in the API request. For example, server.com/images could be routed to a backend images server farm or server.com/patients could be routed to servers designed specifically for holding patient EHRs.

By default, when depth isn't specified, Kiterunner will scan a target at 1 level deep, meaning if you specified a target of server.com, Kiterunner would scan server.com/api.

At level 2, it would scan server.com/api/R4 and at level 3, it would scan server.com/api/R4/patient.

Figure 4: Virtual application path-based routing illustration



Source: Knight Ink

Assetnote Wordlists

Wordlists are flat files of common paths found in web applications and are used by many of the popular content discovery tools, such as Dirbuster.

However, Assetnote culls together and updates their wordlists from Swagger files collected from across the internet on the 28th of each month giving you the richest dataset of wordlists specifically designed for APIs than any other tool.

When wordlists are compiled from .txt files into .kite files, they are cached in ~/.cache/kiterunner/wordlists. An example of one of the wordlists from the most recent list of generated wordlists on Activelist.io can be found in Figure 5 below.

The wordlists from Activelist.io are quite large. If you don't want to use the entire wordlist, you have an option you can pass Kiterunner when starting the tool that will tell it to only use the first X number of lines that you specify and then gracefully stop.

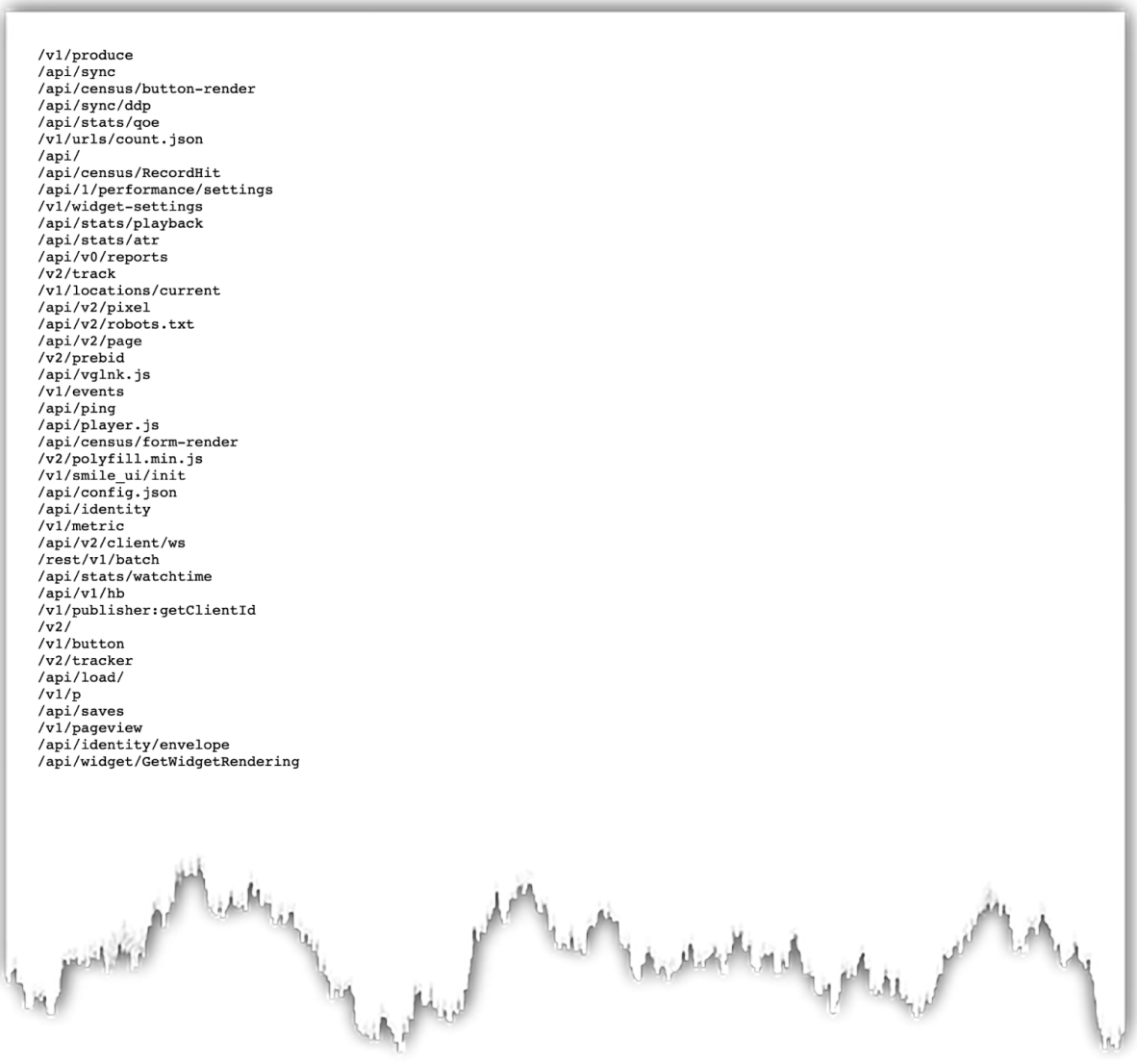
For example, as shown in the documentation, if you want to only use the first 20,000 lines of the API routes wordlist:

this will use the first 20000 lines in the api routes wordlist

```
$ kr scan <target> -A=apiroutes-210228:20000 -x 10 --ignore-length=34
```

However, unless you know for sure you don't want to use the latter portion of a wordlist, this can cause you to miss findings so use with caution knowing you may miss something.

Figure 5: Recent entries from Kiterunner's generated wordlists



Source: Knight Ink

Replaying Requests

Kiterunner's output doesn't provide much utility beyond seeing the HTTP response code for the HTTP requests sent by Kiterunner. You'll need to dig further into the actual request to better understand what was sent and what was received back from the API.

This requires use of the replay option in Kiterunner. Replay will replay a specific API request sent from the previous or currently running session.

```
$ kr kb replay -w <wordlist> "GET    400 [   311,
18,   7] https://acme.com/interconnect-fhir-
oauth/api/FHIR/R4/v1/book
0cf6832e5e4c0aa6b0f5e513fd0c85708446cda0
"
```

Send to Burp Suite

Kiterunner has the ability to send all of its requests that you replay through a proxy server for further manipulation instead of using Kiterunner's replay option at the command line.

By sending everything through Burp's Proxy, you have the ability to interdict every API request and analyze the stimulus and response in intercept and forward it on or replay it with different permutations of the original request (Figure 6).

Figure 6: Using Kiterunner replay to dig further into an HTTP response from the FHIR API server

Source: Knight Ink

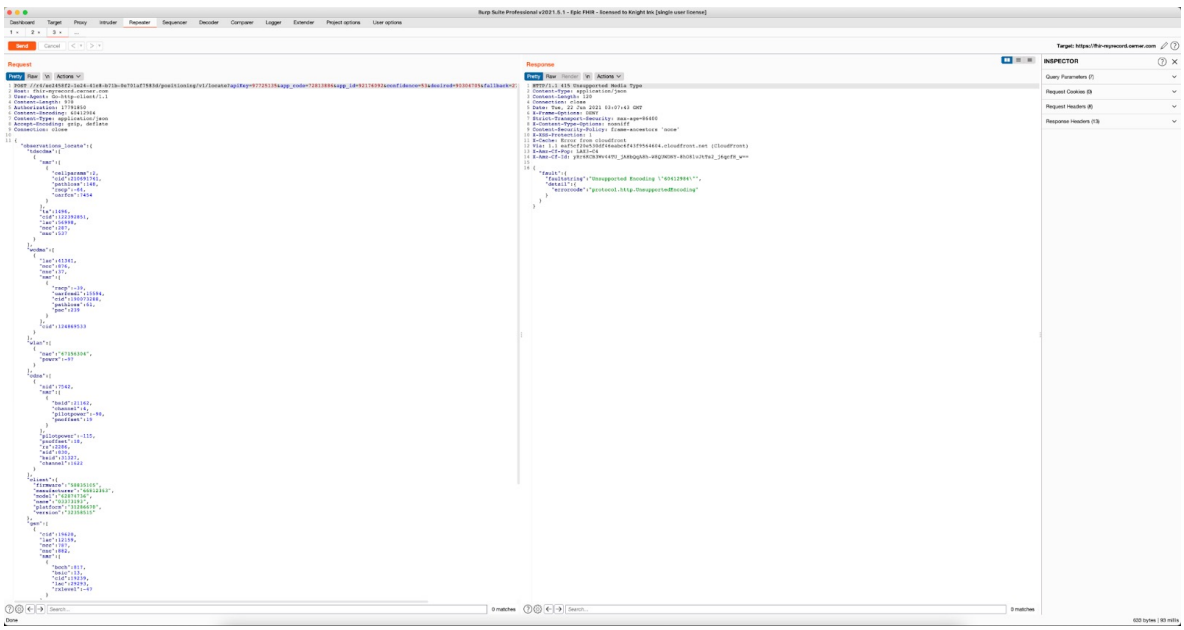
To do this, you'll need to enable the proxy within Burp, disable intercept, then add the --proxy option to Kiterunner when running it.

This will force Kiterunner to send all of its requests and the responses will then be captured within Burp.

Steps to send Kiterunner through Burp's proxy:

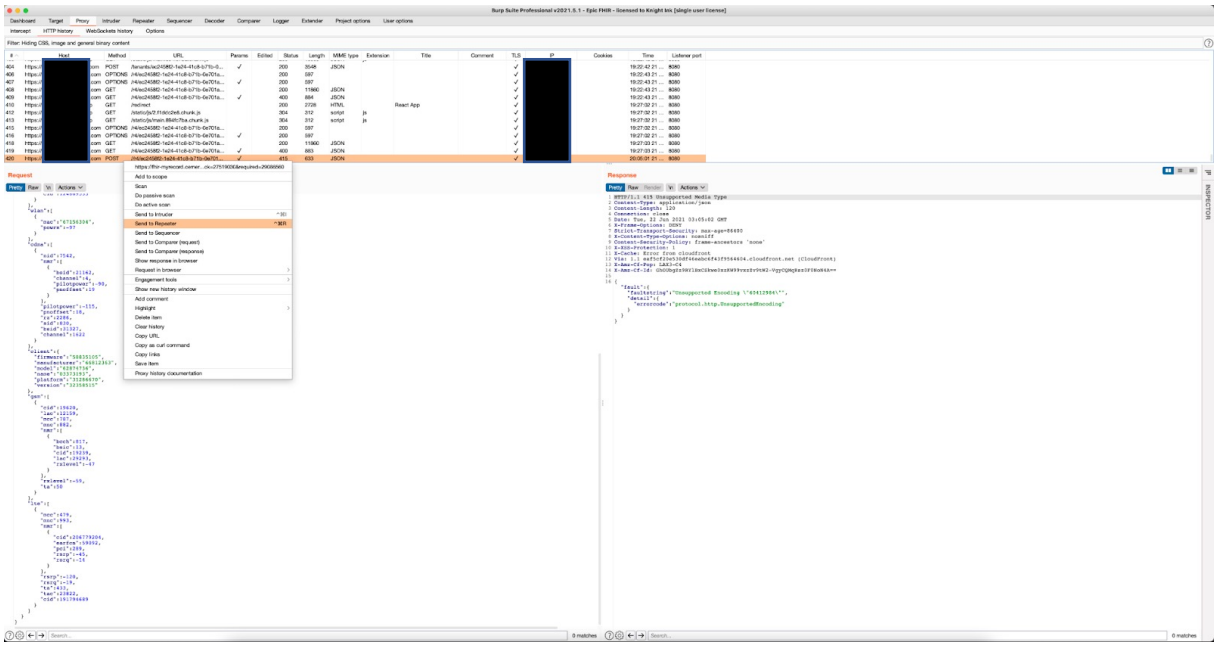
1. Find IP/PORT Burp is listening on
2. Turn off intercept
3. Append --proxy=http://127.0.0.1:8080 to Kiterunner
4. View request in Burp's HTTP history window

Figure 7: Analyzing Kiterunner replays within Burp Suite



Source: Knight Ink

Figure 8. Sending a HTTP 415 response code to Repeater in Burp for further tampering



Source: Knight Ink

WRESTLING WITH RESTler

WRESTLING WITH RESTler

RESTler, developed at Microsoft, is the world's first stateful API fuzzer that automatically generates tests and automatic execution by first reading the OpenAPI specification in order to automatically find vulnerabilities in the API.

A high-level picture for the first-time usage of RESTler can be found in Figure 9 below.

Building RESTler

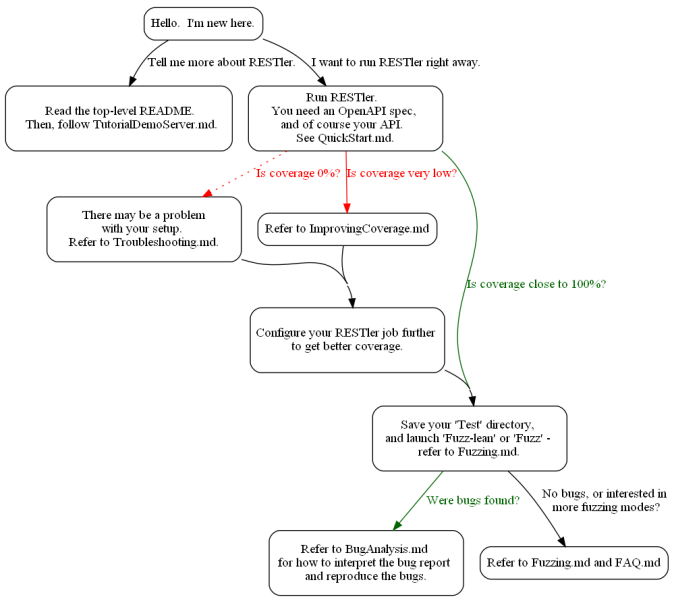
The following section is adapted from the build instructions available on the Github repository for RESTler.

1. The prerequisites for installing RESTler are Python 3.8.2 and .Net 5.0. Currently, RESTler supports Windows and Linux on 64-bit machines but has experimental support for macOS.
2. First, create the directory where you'll be building RESTler
3. Switch to the repo root directory and run the following Python script:

```
$ python ./build-restler.py --dest_dir
<full_path_to_directory_you_created_in_st
ep_2>
```

NOTE: If you receive a nuget error NU1403 when building, a quick workaround is to clear your cache with this command: $ dotnet nuget locals all--clear

Figure 9. RESTler high-level flow for first-time usage



Source: Microsoft

The below results are typical for an API that does not require authentication and has a few GET endpoints without any input parameters.

- 29 requests (endpoint + method) were found in the Swagger spec.
- 3 requests were attempted by RESTler. The other 26 were not attempted because RESTler determined that they depend on outputs of one of the requests that were executed, and a failure prevented that output resource from being available to the other requests.
- 2 requests were successful.
- The coverage is 7% (2/29)
- There were no bugs found. Sometimes, bugs are found in the quick start phase.

In Fuzz-lean mode, RESTler executes once every endpoint+method in a compiled RESTler grammar with a default set of checkers to see if bugs can be found quickly.

In Fuzz mode, RESTler will fuzz the service under test during a longer period of time with the goal of finding more bugs and issues (resource leaks, perf degradation, backend corruptions, etc.). Warning: The Fuzz mode is the more aggressive and may create outages in the service under test if the service is poorly implemented.

```
$ restler.exe fuzz-lean --grammar_file
<RESTLer grammar.py file> --dictionary_file
<RESTler fuzzing-dictionary.json file> --
token_refresh_interval <time in seconds> --
token_refresh_command <command>

$ restler.exe fuzz --grammar_file <RESTLer
grammar.py file> --dictionary_file <RESTler
fuzzing-dictionary.json file> --
token_refresh_interval <time in seconds> --
token_refresh_command <command> --
time_budget <max number of hours
(default 1)
```

An optional settings file can also be passed to RESTler by adding the command-line option --settings <path_to_settings_file.json>. For a list of available settings, see SettingsFile.

Outputs: see the sub-directory FuzzLean or Fuzz (similar to Test)

RESTLer will generate a sub-directory Fuzz[Lean]\RestlerResults\experiment<GUID>\logs including the following files:

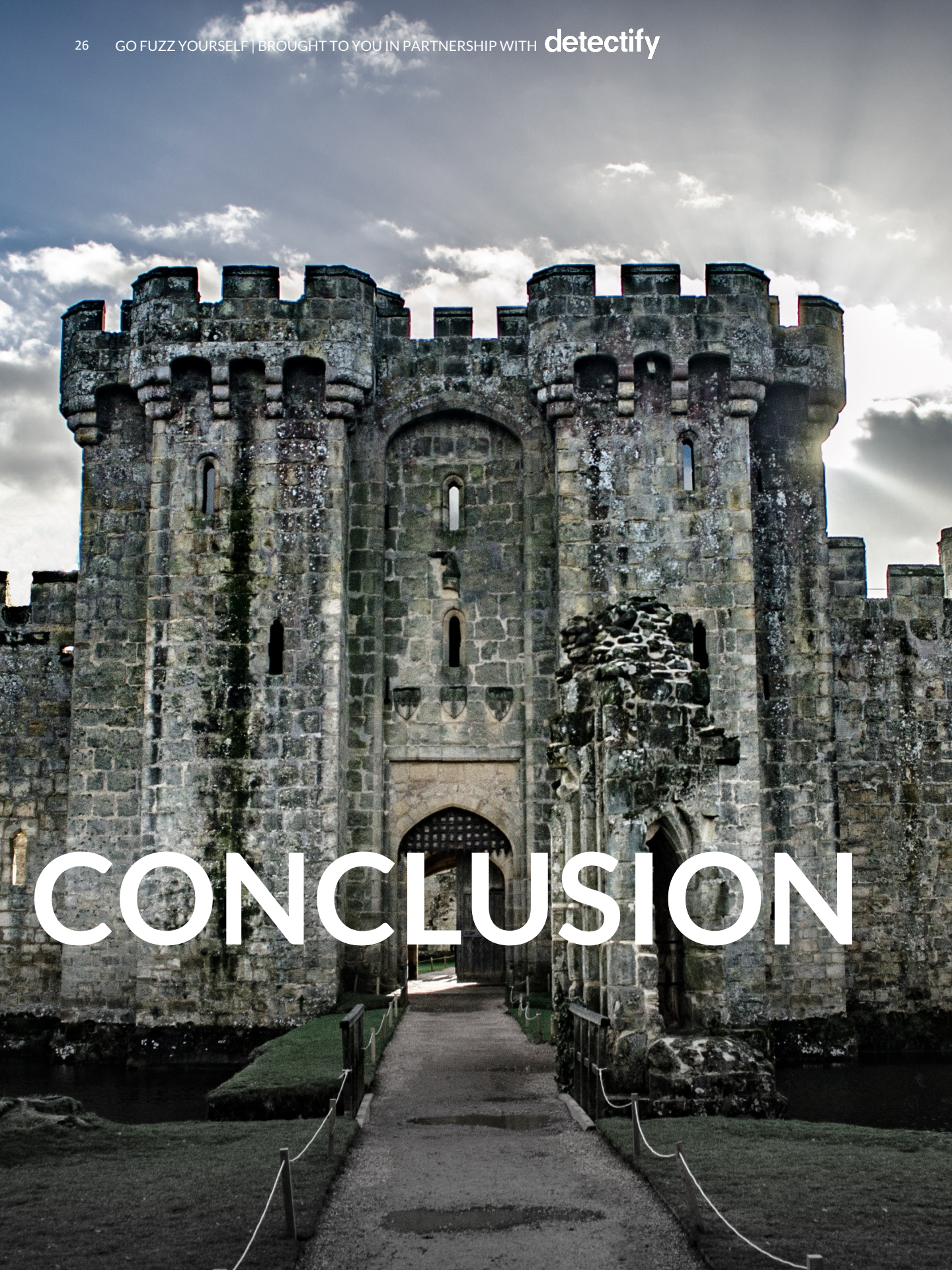
- bug_buckets.txt reports bugs found by RESTler. Those bugs are either "500 Internal Server Errors" found by the RESTler "main_driver" or property checker violations

RESTler currently detects these different types of bugs:

- "500 Internal Server Errors" and any other 5xx errors are detected by the "main_driver"
- UseAfterFreeChecker detects that a deleted resource can still being accessed after deletion
- NameSpaceRuleChecker detects that an unauthorized user can access service resources
- ResourceHierarchyChecker detects that a child resource can be accessed from a non-parent resource
- LeakageRuleChecker detects that a failed resource creation leaks data in subsequent requests
- InvalidDynamicObjectChecker detects 500 errors or unexpected success status codes when invalid dynamic objects are sent in requests
- PayloadBodyChecker detects 500 errors when fuzzing the JSON bodies of requests

RESTler will also generate a sub-directory Fuzz\ResponseBuckets including the following files:

- runSummary.json is a report on all the HTTP error response codes that were received
- errorBuckets.json includes a sample of up to 10 pairs of <request, response> for each HTTP error codes in the 4xx or 5xx ranges that were received



CONCLUSION

Conclusion

I have performed dozens of penetration tests of APIs in my career, from hacking healthcare APIs to hacking federal and state law enforcement vehicles through the manufacturer's APIs giving me remote control of the vehicles.

In every penetration test I've performed, fuzzing has always been the most profoundly important step of the tactics and techniques I use in my kill chain to find vulnerabilities that only fuzzing can find.

Manipulation of parameters, especially when dealing with hundreds in large scale deployments of API endpoints can and should only be done with an automated fuzzing tool.

If you have the distinct feeling that you may have missed vulnerabilities in your API penetration test, it's most likely because you didn't perform any fuzzing of the target endpoints.

In this paper, I demystified API fuzzing, explained the different type of API fuzzers among the more popular options and explained their usage and how to interpret their results.

I used Kiterunner, my go-to open source fuzzer for performing an API penetration test against real-world APIs and explained how to point it at Burp Suite for further replay attacks, extending Kiterunner's capabilities with those offered by Burp Suite.

It's my belief that a penetration test of an API that doesn't incorporate fuzzing in the tactics and techniques used by the penetration tester is consequently incomplete until fuzzing has been performed.

SOURCES

Assetnote. (n.d.). assetnote/kiterunner. GitHub. Retrieved June 22, 2021, from https://github.com/assetnote/kiterunner#depth-scanning

OWASP. (n.d.). Forced Browsing Software Attack | OWASP Foundation. Retrieved June 22, 2021, from https://owasp.org/www-community/attacks/Forced_browsing

Li, V. (2020, January 26). Fuzzing Web Applications - The Startup. Medium. https://medium.com/swlh/fuzzing-web-applications-e786ca4c4bb6?source=linkShare-14a8b702dc61-1624248014&_branch_match_id=link-935370843063544148

RapidAPI. (2021, April 20). What is an API Endpoint? | API Endpoint Definition | RapidAPI. The Last Call - RapidAPI Blog. https://rapidapi.com/blog/api-glossary/endpoint/

Swagger. (n.d.). About Swagger Specification | Documentation | Swagger. Swagger Specification. Retrieved June 22, 2021, from https://swagger.io/docs/specification/about/

OpenAPI. (n.d.). OAI/OpenAPI-Specification. OpenAPI Specification - GitHub. Retrieved June 22, 2021, from https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.2.md

Todd Fredrich, Pearson eCollege. (n.d.). HTTP Methods for RESTful Services. Pearson ECollege. Retrieved June 22, 2021, from https://www.restapitutorial.com/lessons/httpmethods.html

RapidAPI. (2021b, April 23). What are API Parameters? | Parameters Definition | API Glossary. The Last Call - RapidAPI Blog. https://rapidapi.com/blog/api-glossary/parameters/

Paxton-Fear, D. K. P.-F. (2021, April 14). API Recon with Kiterunner - Hacker Toolbox. YouTube. https://www.youtube.com/watch?v=hNs8fpWfcyU&t=658s

Swagger. (n.d.-b). Understanding the Differences Between API Documentation, Specifications, and Definitions. Retrieved June 22, 2021, from https://swagger.io/resources/articles/difference-between-api-documentation-specification/

Our guide to fuzzing | F-Secure. (n.d.). F-Secure. Retrieved August 12, 2021, from https://www.f-secure.com/us-en/consulting/our-thinking/15-minute-guide-to-fuzzing

Godefroid, P. (2020, March 4). A brief introduction to fuzzing and why it's an important tool for developers. Microsoft Research. https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

Williams, J. (2015, June 25). What Do You Mean My Security Tools Don't Work on APIs?!! Dark Reading. https://www.darkreading.com/application-security/what-do-you-mean-my-security-tools-don-t-work-on-apis-

Licata, S. (2020, June 19). Focus on Fuzzing: Types of Fuzzing. SAFECode. https://safecode.org/focus-on-fuzzing-types-of-fuzzing/

Web API Fuzz Testing | GitLab. (n.d.). Gitlab. Retrieved August 12, 2021, from https://docs.gitlab.com/ee/user/application_security/api_fuzzing/

GitHub - microsoft/restler-fuzzer: RESTler is the first stateful REST API fuzzing tool for automatically testing cloud services through their REST APIs and finding security and reliability bugs in these services. (n.d.). GitHub. Retrieved August 13, 2021, from https://github.com/microsoft/restler-fuzzer

https://patricegodefroid.github.io/public_psfiles/icse2019.pdf

ABOUT THE AUTHOR

Alissa Knight is a partner at Knight Ink and blends influencer marketing, content creation in writing and video production, go-to market strategies, and strategic planning for telling brand stories at scale in cybersecurity.

She achieves this through ideation to execution of content strategy, storytelling, and execution of influencer marketing strategies that take cybersecurity buyers through a brand's custom curated journey to attract and retain them as long-term partners.

Alissa is a published author, having published the first book on hacking connected cars and is working on a new series of books into hacking and securing APIs and microservices.

ABOUT KNIGHT INK

Firm Overview

Knight Ink is a content strategy, creation, and influencer marketing agency founded for category leaders and challenger brands in cybersecurity to fill current gaps in content and community management. We help vendors create and distribute their stories to the market in the form of written and visual storytelling drawn from 20+ years of experience working with global brands in cybersecurity. Knight Ink balances pragmatism with thought leadership and community management that amplifies a brand's reach, breeds customer delight and loyalty, and delivers creative experiences in written and visual content in cybersecurity.

Amid a sea of monotony, we help cybersecurity vendors unfurl, ascertain, and unfetter truly distinct positioning that drives accretive growth through amplified reach and customer loyalty using written and visual experiences.

Knight Ink delivers written and visual content through a blue ocean strategy tailored to specific brands. Whether it's a firewall, network threat analytics solutions, endpoint detection and response, or any other technology, every brand must swim out of a red sea of competition clawing at each other for market share using commoditized features. We help our clients navigate to blue ocean where the lowest price or most features don't matter.

We work with our customers to create a content strategy built around their blue ocean then perform the tactical steps necessary to execute on that strategy through the creation of written and visual content assets unique to the company and its story for the individual customer personas created in the strategy setting.

Contact Us

Web: www.knightinkmedia.com
Phone: (702) 637-8297
Address: 1980 Festival Plaza Drive, Suite 300, Las Vegas, NV 89135

ABOUT DETECTIFY

Firm Overview

Detectify offers cloud-based web application security solutions that streamline vulnerability findings in production to security defenders and application owners. Detectify collaborates with ethical hackers to source the latest security research from hacker-into-scanner in as fast as 15 minutes. It leverages fuzzing and crawling techniques and delivers reliable payload-based testing to customers. This means verified results and clearer visibility with less noise. With Detectify you will bring security up to speed and scale with development, and go to market safer.

Curious to see what Detectify will find in your websites? Head over to www.detectify.com to start a free 2-week trial!

Knight Ink
1980 Festival Plaza Drive
Suite 300
Las Vegas, NV 89135
ak@knightinkmedia.com