Statically Detecting Web Application Routes and Their Authentication and Authorization Properties

Matt Schwager
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
mschwager3@gatech.edu

ABSTRACT

Authentication, authorization, and access control bugs are plaguing modern web applications. To date, little has been done in the commercial or open source spaces to address this problem. This class of software security bugs commonly results in unintended information disclosure and elevation of privileges. Authentication and authorization procedures are often implemented as part of an application's business logic. This makes generalizing their detection a challenge.

To address this growing need, this paper introduces route-detect, a command-line application for statically detecting web application routes and their authentication and authorization properties. route-detect currently supports 6 programming languages, 16 web application frameworks, and 23 third-party authentication and authorization libraries. During its evaluation, route-detect analyzed millions of lines of code at an average rate of nearly 30 thousand lines per second, detected thousands of web application routes and their authentication and authorization properties, and found 18 unintentionally unauthorized routes in a real-world production application. It saw an average decrease in routes detected of 89.8% when evaluated against comparable regular expression text search. This allows its users to focus on route search results that matter.

route-detect enables security practitioners to quickly and accurately analyze routes for authentication and authorization concerns in web application codebases.

KEYWORDS

static analysis, static application security testing, SAST, URL routing, web application routing, authentication, authorization, access control

1 Introduction

Authentication, authorization, and access control bugs are plaguing web applications. This is evidenced by the following industry standard resources:

- **2021 OWASP Top 10** [1]
  - #1 - Broken Access Control
  - #7 - Identification and Authentication Failures (formerly Broken Authentication)
- **2019 OWASP API Top 10** [2]
  - #2 - Broken User Authentication
  - #5 - Broken Function Level Authorization
- **2022 CWE Top 25** [3]
  - #14 - CWE-287: Improper Authentication
  - #16 - CWE-862: Missing Authorization
  - #18 - CWE-306: Missing Authentication for Critical Function
- **Number of CVEs by CWE** [4]
  - #21 - CWE-284: Access Control Issues (Authorization)
  - #47 - CWE-639: Access Control Bypass Through User-Controlled Key

Authentication, authorization, and access control are repeatedly named. Web applications often specify authentication and authorizations properties based on URL routing. URL routing is the process by which web applications forward web requests to the correct application code based on the URL path of the request. For example, routes under the */admin* URL path require administrative privileges to access. Indeed, not all authentication and authorization bugs occur in route specifications, but automated tooling can begin to address the problem.

Through the use of static analysis, a tool can analyze source code for logic bugs, performance concerns, security issues, and other problematic patterns. This paper introduces route-detect, a command-line application for statically detecting web application routes and their authentication and authorization properties. route-detect's source code can be found in Appendix A1. Before proceeding, the following definitions will be helpful throughout the remainder of this paper:

- **Web application**: a software application that runs in a web browser. Notably including custom application code that handles incoming web requests.
- **URL routing**: the process by which web applications forward requests to appropriate request handlers. For

example, a request to *https://example.com/users/create* will be routed to the correct handler based on the */users/create* URL path.

- **Authentication**: a validation mechanism for determining who you are.
- **Authorization**: a validation mechanism for determining what you can access.
- **Role**: a user designation that determines what you can access. For example, the administrative and regular user roles.
- **Access control**: a security measure for determining access to resources. In the context of this paper, access control will encompass both authentication and authorization.
- **Static analysis**: analyzing code that is not executing. In other words, static source code, byte code, or machine code.
- **Security practitioner**: a computer security professional. For example, application security engineers, security researchers, penetration testers, and bug bounty researchers. This is route-detect's target audience.

The following sections describe the design and evaluation of route-detect, and considerations for future work. Additionally, there is discussion on lessons learned while developing route-detect. This includes discoveries of various routing, authentication, and authorization categories, and limitations of the tool based on different routing strategies. route-detect's initial implementation supports the following programming languages and web application frameworks:

- **Python**: Django, Flask, Sanic
- **PHP**: Laravel, Symfony, CakePHP
- **Ruby**: Rails, Grape
- **Java**: JAX-RS, Spring
- **Go**: Gorilla, Gin, Chi
- **JavaScript/TypeScript**: Express, React, Angular

2 Design

Static analysis searches code for patterns without requiring it to be running. These patterns include logic bugs, performance concerns, security issues, and other problematic pieces of code. Contrast this with dynamic analysis, which requires running code, processes, or systems to perform its evaluation. route-detect leverages static analysis to search code for web application routes and their authentication and authorization properties. This section will explore the design and implementation of route-detect.

route-detect relies on three primary technologies to provide its functionality: Semgrep [5] for performing static code analysis, D3.js [6] for visualizing routes and their access control properties, and the Python programming language for implementing application-specific behavior, providing a command-line interface, and connecting all other functionality together. The remainder of this section will describe how these technologies are used to implement route-detect.

Semgrep uses YAML-formatted configuration files, which include one or more rules, to search code for specific patterns. In route-detect's case, this means searching for routes and their access control properties. route-detect uses one configuration file for each combination of programming language and web application framework. For example, PHP and Laravel, and Java and Spring are each implemented as a configuration file. Within a configuration file, there are separate rules for authenticated routes, authorized routes, and routes that are neither authenticated nor authorized. This allows route-detect to distinguish between these various forms of routes. Configuration files managed by route-detect are then passed to Semgrep, which searches source code for the rule patterns therein. Here's an example of this functionality on the command-line (see Appendix A2 for sample output):

```
$ semgrep --config $(routes which laravel) \
      path/to/laravel/code
```

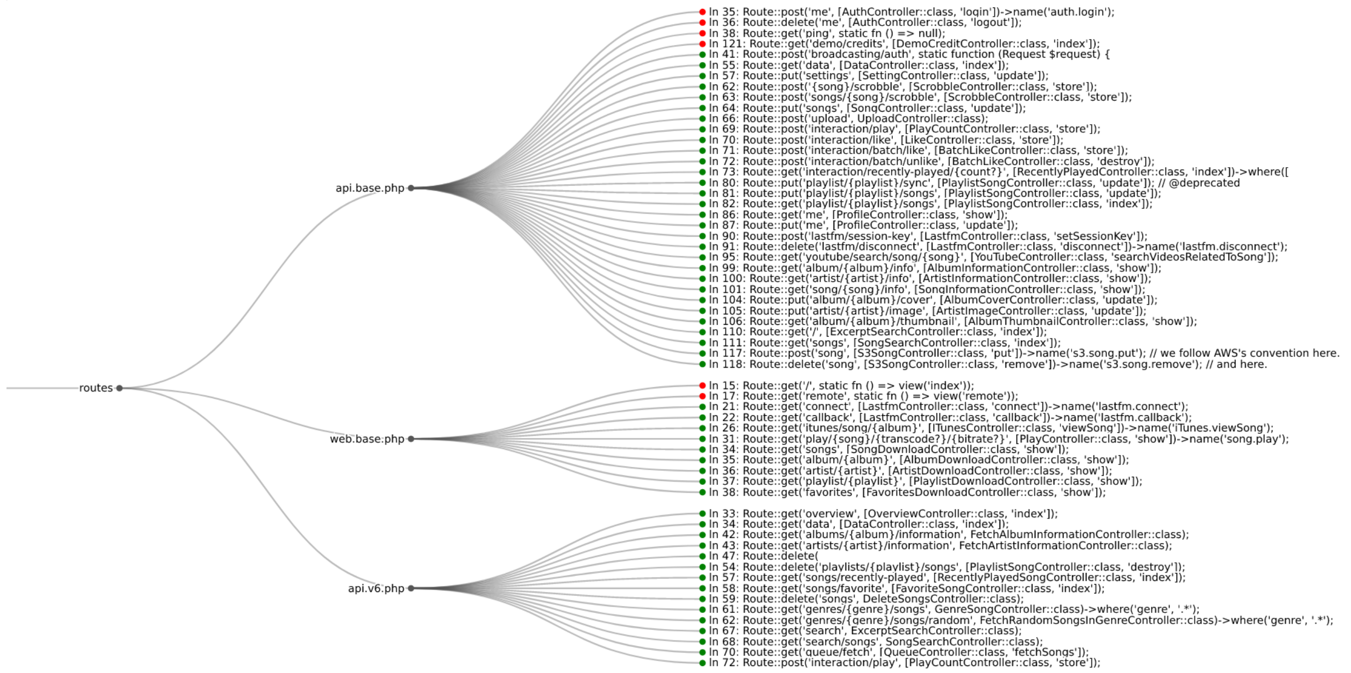
route-detect provides the *routes* command-line application. The *routes* application provides the *which* subcommand to output the filesystem location of a specified web application framework configuration file. This file location is passed to Semgrep's *--config* command-line argument via a shell subcommand to instruct Semgrep to use the specified configuration file. In this case, the user would need to know that they're analyzing a Laravel web application. If a user is unsure about what programming language or web application framework is being used, they can specify the special *all* configuration identifier. This will run all programming language and framework combinations, and report all results. The execution mode used above will output results to the command-line. To visualize results, a user must first store Semgrep output in a JSON file for later analysis:

```
$ semgrep --config $(routes which laravel) \
      --json --output laravel-routes.json \
      path/to/laravel/code
```

With the route information file available, a user can use the *viz* subcommand to visualize the results in a tree-like structure in the browser:

```
$ routes viz --browser laravel-routes.json
```

An example of this visualization can be seen in Figure 1. In this example, unauthenticated routes are represented by red nodes, and authenticated routes by green nodes. The text associated with a node is the route result's line number and line of code that correspond with the finding.



```
🔴 ln 35: Route::post('me', [AuthController::class, 'login'])->name('auth.login');
🔴 ln 36: Route::delete('me', [AuthController::class, 'logout']);
🔴 ln 38: Route::get('ping', static fn () => null);
🔴 ln 121: Route::get('demo/credits', [DemoCreditController::class, 'index']);
🟢 ln 41: Route::post('broadcasting/auth', static function (Request $request) {
🟢 ln 55: Route::get('data', [DataController::class, 'index']);
🟢 ln 57: Route::put('settings', [SettingController::class, 'update']);
🟢 ln 62: Route::post('{song}/scrobble', [ScrobbleController::class, 'store']);
🟢 ln 63: Route::post('songs/{song}/scrobble', [ScrobbleController::class, 'store']);
🟢 ln 64: Route::put('songs', [SongController::class, 'update']);
🟢 ln 66: Route::post('upload', UploadController::class);
🟢 ln 69: Route::post('interaction/play', [PlayCountController::class, 'store']);
🟢 ln 70: Route::post('interaction/like', [LikeController::class, 'store']);
🟢 ln 71: Route::post('interaction/batch/like', [BatchLikeController::class, 'store']);
🟢 ln 72: Route::post('interaction/batch/unlike', [BatchLikeController::class, 'destroy']);
🟢 ln 73: Route::get('interaction/recently-played/{count?}', [RecentlyPlayedController::class, 'index'])->where([
🟢 ln 80: Route::put('playlist/{playlist}/sync', [PlaylistSongController::class, 'update']); // @deprecated
🟢 ln 81: Route::put('playlist/{playlist}/songs', [PlaylistSongController::class, 'update']);
🟢 ln 82: Route::get('playlist/{playlist}/songs', [PlaylistSongController::class, 'index']);
🟢 ln 86: Route::get('me', [ProfileController::class, 'show']);
🟢 ln 87: Route::put('me', [ProfileController::class, 'update']);
🟢 ln 90: Route::post('lastfm/session-key', [LastfmController::class, 'setSessionKey']);
🟢 ln 91: Route::delete('lastfm/disconnect', [LastfmController::class, 'disconnect'])->name('lastfm.disconnect');
🟢 ln 95: Route::get('youtube/search/song/{song}', [YouTubeController::class, 'searchVideosRelatedToSong']);
🟢 ln 99: Route::get('album/{album}/info', [AlbumInformationController::class, 'show']);
🟢 ln 100: Route::get('artist/{artist}/info', [ArtistInformationController::class, 'show']);
🟢 ln 101: Route::get('song/{song}/info', [SongInformationController::class, 'show']);
🟢 ln 104: Route::put('album/{album}/cover', [AlbumCoverController::class, 'update']);
🟢 ln 105: Route::put('artist/{artist}/image', [ArtistImageController::class, 'update']);
🟢 ln 106: Route::get('album/{album}/thumbnail', [AlbumThumbnailController::class, 'show']);
🟢 ln 110: Route::get('/', [ExcerptSearchController::class, 'index']);
🟢 ln 111: Route::get('songs', [SongSearchController::class, 'index']);
🟢 ln 117: Route::post('song', [S3SongController::class, 'put'])->name('s3.song.put'); // we follow AWS's convention here.
🟢 ln 118: Route::delete('song', [S3SongController::class, 'remove'])->name('s3.song.remove'); // and here.

🔴 ln 15: Route::get('/', static fn () => view('index'));
🔴 ln 17: Route::get('remote', static fn () => view('remote'));
🟢 ln 21: Route::get('connect', [LastfmController::class, 'connect'])->name('lastfm.connect');
🟢 ln 22: Route::get('callback', [LastfmController::class, 'callback'])->name('lastfm.callback');
🟢 ln 26: Route::get('itunes/song/{album}', [ITunesController::class, 'viewSong'])->name('iTunes.viewSong');
🟢 ln 31: Route::get('play/{song}/{transcode?}/{bitrate?}', [PlayController::class, 'show'])->name('song.play');
🟢 ln 34: Route::get('songs', [SongDownloadController::class, 'show']);
🟢 ln 35: Route::get('album/{album}', [AlbumDownloadController::class, 'show']);
🟢 ln 36: Route::get('artist/{artist}', [ArtistDownloadController::class, 'show']);
🟢 ln 37: Route::get('playlist/{playlist}', [PlaylistDownloadController::class, 'show']);
🟢 ln 38: Route::get('favorites', [FavoritesDownloadController::class, 'show']);

🟢 ln 33: Route::get('overview', [OverviewController::class, 'index']);
🟢 ln 34: Route::get('data', [DataController::class, 'index']);
🟢 ln 42: Route::get('albums/{album}/information', FetchAlbumInformationController::class);
🟢 ln 43: Route::get('artists/{artist}/information', FetchArtistInformationController::class);
🟢 ln 47: Route::delete(
🟢 ln 54: Route::delete('playlists/{playlist}/songs', [PlaylistSongController::class, 'destroy']);
🟢 ln 57: Route::get('songs/recently-played', [RecentlyPlayedSongController::class, 'index']);
🟢 ln 58: Route::get('songs/favorite', [FavoriteSongController::class, 'index']);
🟢 ln 59: Route::delete('songs', DeleteSongsController::class);
🟢 ln 61: Route::get('genres/{genre}/songs', GenreSongController::class)->where('genre', '.*');
🟢 ln 62: Route::get('genres/{genre}/songs/random', FetchRandomSongsInGenreController::class)->where('genre', '.*');
🟢 ln 67: Route::get('search', ExcerptSearchController::class);
🟢 ln 68: Route::get('search/songs', SongSearchController::class);
🟢 ln 70: Route::get('queue/fetch', [QueueController::class, 'fetchSongs']);
🟢 ln 72: Route::post('interaction/play', [PlayCountController::class, 'store']);
```

Figure 1: Routes from the Koel music streaming server

route-detect's visualization functionality aims to aid users in quickly analyzing and diagnosing problematic routes in their codebases. In large web application codebases there can often be hundreds or thousands of routes, which is challenging to manage from the command-line. Users can leverage their knowledge of a codebase to scan for inconsistencies or otherwise problematic routes.

2.1 Route Detection

Route detection is implemented as Semgrep configuration file rules. route-detect implements a configuration file for each combination of programming language and web application framework. The following is a simplified example of a Python Flask route detection rule:

```
rules:
- id: flask-route
  pattern: |
    @$APP.route($PATH, ...)
    def $FUNC(...):
      ...
  message: Found Flask route
  languages: [python]
  severity: INFO
```

This rule searches for Python functions that use the *route* decorator. Flask uses this decorator to specify routes [7]. route-detect implements rules similar to this for each combination of programming language and web application framework. Frameworks vary greatly on their routing implementations. Semgrep was chosen to perform route-detect's static analysis for its broad, mature programming language support, and its flexibility in specifying code patterns to detect routes and access control information. route-detect supports the following programming languages and frameworks:

- **Python**: Django, Flask, Sanic
- **PHP**: Laravel, Symfony, CakePHP
- **Ruby**: Rails, Grape
- **Java**: JAX-RS, Spring
- **Go**: Gorilla, Gin, Chi
- **JavaScript/TypeScript**: Express, React, Angular

Each of these frameworks has a Semgrep configuration file managed by route-detect. The framework identifier is passed to the *routes which* subcommand, which will select the corresponding configuration file for Semgrep to run.

Rules searching for routes that are both unauthenticated and unauthorized must explicitly filter authenticated and authorized routes. Searching for all routes would not work because authenticated and authorized routes would be counted multiple times. Because of this, route-detect searches for routes that are authenticated, authorized, or neither authenticated nor authorized.

2.2 Authentication Detection

Authentication validates who you are. Common examples are form-based username-password authentication to access a website and token-based authentication [8] to access an API service.

Web application frameworks may provide first-party authentication functionality, third-party authentication integration with external libraries, or both. route-detect includes first-party authentication detection for its supported web application frameworks. Further, it supports common third-party authentication libraries when first-party authentication is not available, or when a web application framework recommends specific third-party libraries. The number of third-party authentication and authorization libraries is ever increasing, so route-detect aims to focus on supporting a few of the most popular libraries.

Detecting authenticated routes generally involves searching for route code, then modifying the rule to check for authentication logic around the route. The following is a simplified example of a PHP Laravel authentication route detect rule:

```
rules:
- id: laravel-route-authenticated
  pattern: |
    Route::post(...)->middleware("=~/^auth.*/");
  message: Found authenticated Laravel route
  languages: [php]
  severity: INFO
```

In this example, the addition of the *auth* middleware will limit results to authenticated routes. Without it, all *post* routes would be detected. In the above example, this rule is searching for first-party authentication middleware from the Laravel web application framework. In addition to first-party authentication support for all frameworks that offer it, route-detect supports the following third-party libraries:

- **Flask**: flask-login, flask-httpauth, flask-jwt-extended, Flask-JWT, flask-praetorian
- **Sanic**: sanic-jwt, sanic-jwt-extended, sanic-beskar, sanic-security, sanic-token-auth, Sanic-HTTPAuth, sanicapikey
- **Chi**: jwtauth, oauth
- **Gorilla**: go-jwt-middleware
- **Express**: passport, express-openid-connect, express-jwt
- **React**: auth0-react
- **JAX-RS**: shiro
- **Spring**: shiro

There are many different ways authentication logic can be specified for a route. Section 4.1: Routing Security Configuration Approaches will further explore these authentication configuration methods.

2.3 Authorization Detection

Authorization validates what you can access and what actions you may perform. This implies authentication is performed first, because determining what you can access requires first determining who you are. A common example of authorization is a system with administrative users and regular users. Administrative users can create other users, while regular users cannot. The system uses authorization to gate access to certain functionality, such as the ability to create users.

Web application frameworks typically specify authorization information in a similar manner as authentication information. For this reason, much of the information on authentication detection can also be applied to authorization detection. The one notable difference is route-detect's collection of role information when looking at authorized routes. While authentication is a binary state of authenticated or unauthenticated, authorization allows for arbitrary numbers of roles to be specified. For this reason, route-detect collects role information and surfaces it to the user because it may be relevant in accessing the security of an application. The following is a simplified example of a Java Spring authorization route detection rule:

```
rules:
- id: spring-route-authorized
  pattern: |
    @PostMapping(...)
    @javax.annotation.security.RolesAllowed($AUTHZ)
    $RETURNTYPE $FUNC(...) { ... }
  message: Found authorized Spring route
  languages: [java]
  severity: INFO
```

In this example, the addition of the *RolesAllowed* annotation will limit results to authorized routes. Without it, all *PostMapping* routes would be detected. Additionally, the *$AUTHZ* metavariable will gather the role information passed to *RolesAllowed* and include it in the Semgrep JSON output. This allows for further processing or investigation by route-detect users. Both first-party and third-party support for authorization functionality was less common than authentication in web application frameworks. route-detect supports the following third-party authorization libraries:

- **JAX-RS**: shiro
- **Spring**: shiro
- **Grape**: grape-cancan, doorkeeper

Similar to authentication detection, there are many different ways authorization logic can be specified on a route, and Section 4.1: Routing Security Configuration Approaches will further explore these authorization configuration methods.

2.4 Testing

route-detect employs various forms of testing to ensure the accuracy of its results, and to maintain a high quality tool for its users. This includes two forms of testing: testing Semgrep rules managed by route-detect [9], and unit testing the Python code of route-detect itself. Both forms of testing will be explored below.

Semgrep supports first-class functionality for testing custom rules. This includes ensuring that true positives and true negatives are detected as such. Tests are written by annotating real code and running Semgrep with the *--test* command-line argument. The following is an example test file for Python's Flask web application framework:

```
from flask import Flask
from flask_login import login_required

app = Flask(__name__)

# ruleid: flask-route-unauthenticated
@app.route("/")
def unauth_route():
    return render_template("test.html")

# ruleid: flask-route-authenticated
@app.route("/")
@login_required
def auth_route():
    return render_template("test.html")
```

When running Semgrep in test mode, it will ensure that the first function is detected as unauthenticated, and the second function is detected as authenticated. The *ruleid* annotation in the comments indicates to Semgrep what rule it should be searching for. Every configuration file managed by route-detect has a corresponding test file. There are currently 217 Semgrep rule tests in route-detect.

Testing is used to minimize false positives. If a false positive is encountered, the corresponding code can be included in route-detect's tests, the Semgrep rule can be improved to fix the false positive, and the test will prevent detection regressions in the future. As the corpus of tests grows, confidence in the tool increases, and rule improvements can be made without fear of breaking existing functionality. Similarly, when a false negative is encountered it can also be added to the test corpus, the corresponding Semgrep rule can then be improved, and the test will ensure that the new pattern was indeed detected.

In addition to Semgrep rule testing, route-detect uses pytest [10] to test its Python code. This includes testing functionality provided by the *routes* command-line application. The Python tests focus on the *viz* visualization subcommand because that is where the bulk of the Python code logic resides. There are currently 33 pytest tests in route-detect. Similar to Semgrep rule tests, when a bug is encountered in route-detect Python code, a test can be written to avoid regressions after a fix is applied.

Testing both route-detect's Semgrep rules and Python code helps ensure the accuracy of the tool, and the quality of the user experience when running the tool.

2.5 Visualization

Route visualization allows route-detect users to quickly inspect the routes of a codebase and their authentication and authorization properties. Security practitioners can search for bugs without needing to manually determine programming language, routing software, and route locations. Visualization provides a starting point for performing deeper evaluation. It also allows users to apply their codebase-specific knowledge to look for inconsistencies. route-detect relies on D3.js for visualization functionality.

To visualize routes with the *viz* subcommand, a user must first generate a Semgrep JSON report of a codebase's routes. The *viz* subcommand will then use the route results to generate a tidy tree visualization [11]. The filesystem path parts of the results are used as the parent nodes of the tree, and the Semgrep static analysis findings are used as the leaf nodes of the tree. For example, if there is a *flask-route* finding in *lib/api/routes.py,* then the tree will be *lib→api→routes.py→flask-route*. The text of the leaf node is the first line of code that represents the finding. See Figure 1 for a sample visualization. route-detect's visualization functionality also provides helpful quality-of-life features such as zoom, pan, and additional code context on hover. All this functionality aims to aid users in understanding the routes, and their access control properties, in web application codebases.

3 Evaluation

route-detect searches web application code for routes and their associated authentication and authorization properties. As such, validating the efficacy of this search should form the basis for evaluation. The evaluation process focused on two methods of validation: comparing against regular expressions for route detection, and manual investigation for authentication and authorization properties. Both validation methods require a corpus of test code repositories. route-detect was evaluated against 44 dependent, web application codebases.

Large, popular, open source repositories that use the target web application framework were chosen for the test corpus. These repository characteristics were chosen in an attempt to find real-world, representative examples of a target web application. Size was determined by the number of lines of code in the target programming language. Repositories with ten thousand to one hundred thousand lines of code were prioritized. Popularity was determined by the repository's number of GitHub stars. Repositories with at least one thousand GitHub stars were prioritized. In the event that size or popularity characteristics could not be fulfilled given the available open source repositories on GitHub, the requirements were relaxed to include any repositories using the target web application framework. A number of search methods were used to look for open source repositories meeting these characteristics.

Both GitHub Topics [12] and dependency graph dependents [13] were used to search for applications dependent on the target web application framework. For example, the "flask" Topic [14] can be used to search for popular Flask applications. Additionally, a reverse dependency, or dependent, search can be performed to find projects that depend on Flask. Google's site search [15] functionality can also be used. For example, querying for "site:https://github.com/google django" will search Google's open source GitHub repositories for Django usage. All of these search methods were combined with manual repository inspection to ensure the desired size and popularity characteristics. These characteristics could not always be met, but dependent applications could always be found. To avoid potential peculiarities with a single dependent repository, two or three repositories were chosen for each framework for analysis.

After the repository test corpus was acquired, the repositories were analyzed for web application routes and their access control properties. The results were then processed and grouped by programming language and web application framework.

3.1 Analyzing Dependent Applications

Regular expression text search and manual inspection were used when analyzing route-detect against web application framework dependent applications. Both regular expression text search and static analysis use application source code to perform their analyses. This makes regular expressions a useful validation mechanism. The evaluation used ripgrep [16] to perform regular expression text search.

The table in Appendix A3 compares regular expression text search to route-detect's static analysis. The comparison only involved route detection, not authentication or authorization detection. In nearly all frameworks analyzed, authentication and authorization properties were specified on separate lines of code than the route itself. Code whitespace, varying module and variable names, and code patterns spanning multiple lines make analyzing authentication and authorization information with regular expressions a challenge. On the spectrum of easy but ineffective to powerful but complex, regular expressions fall under the former. Further, because most mainstream programming languages are not regular languages, regular expressions have trouble analyzing them [17]. Therefore, a comparison was only performed against route detection. Semgrep uses semantic code analysis, not simple text search. This enables it to excel where regular expressions cannot. Whitespace can be ignored, code names can be abstracted away, and multiple lines of code can be reasonably searched. With this and other functionality, Semgrep rules were able to outperform regular expression text search.

Regular expression text search returned 73349 routes while route-detect returned 7515. This represents an average decrease in routes detected of 89.8%. The most common reasons for regular expression overcounting were common function, attribute, and configuration key names, commented out code, and missing multiline route specifications. Ruby's Rails framework highlights common function name problems. In Rails, a route request handler for an HTTP GET method uses the *get* function name. Similarly, an HTTP POST method uses *post*. Using simple text search for "get" and "post" returns many false positives. On the other hand, Semgrep rules allow specifying only functions named *get* inside a *Rails.application.routes.draw* block. This semantic understanding of Ruby code allows Semgrep to perform a more specific search. Semgrep also ignores commented out code, which removes further false positives. Finally, Semgrep's ellipsis and metavariable operators [18] allow searching over multiple lines of code, and removing concerns of specific variable, function, module, and other code names. With reliable route detection, route-detect must be further evaluated for authentication and authorization detection.

Evaluating authentication and authorization detection primarily involved detecting false positives and false negatives. Static analysis should attempt to strike a balance between false positives and false negatives [19]. This balance represents the difference between being too aggressive and not aggressive enough. In route-detect's case, correcting false positives involves noting the incorrect Semgrep result, writing a test to catch the mistake, and improving the rule to no longer catch that pattern. False negatives are more pernicious to catch. They are a *lack* of a finding, rather than an incorrect finding. One method to catch false negatives is to use a separate tool to perform the same type of analysis and note findings that it catches and route-detect doesn't. The regular expression text search mentioned above provided a separate tool against which to compare findings. Generally, the process for writing Semgrep rules was as follows:

1. Read the framework's routing documentation to get an idea for its routing implementation and create a Semgrep rule to detect basic routes
2. In a dependent repository, use regular expressions or basic text search for terms like "route", "path", "authentication", "authorization", "view", or "controller"
3. Improve the framework's Semgrep rule to cover cases found in the dependent repository
4. Run the Semgrep rule against the dependent repository
5. Manually investigate results for false positives
6. Manually investigate code near routes and their authentication and authorization properties for false negatives
7. Create a test and modify the Semgrep rule to handle false positives and false negatives
8. GOTO 2.

This process created a self-reinforcing workflow to minimize false positives and false negatives. The end result was over 13 million lines of code analyzed at an average rate of about 30 thousand lines per second, 7515 total routes detected, 1459 authenticated routes, 3139 unauthenticated routes, 999 authorized routes, 433 unauthorized routes, 1485 unknown routes, and 171 unique roles detected. The average Semgrep runtime was 10.2 seconds. This is fast for codebases with tens to hundreds of thousands of lines of code. An increase in the amount of code analyzed per unit time leads to a corresponding increase in the number of bugs detected [20]. Appendix A4 includes the full table of results.

Unknown routes are routes that were successfully detected, but could not be connected with their authentication or authorization logic via route-detect's global or interprocedural connectors. Global connectors, if detected, mean all routes in the application are either authenticated or authorized. This is a convenient mechanism for web applications to provide so that all routes are secure without needing to manually specify these security properties on each route. Interprocedural connectors attempt to connect a route with its security properties in a separate procedure or source code file. Section 4.2: Limitations will discuss details on the shortcomings of route-detect's interprocedural analysis.

In addition to being evaluated against open source, dependent, web application codebases, route-detect was run against a real-world, production codebase. It detected 18 unintentionally unauthorized web application routes. Due to privacy and security concerns, the name of this application cannot be revealed.

3.2 Related Work

Investigating web application routes and their access control properties revealed two similar technologies: a "route coverage" commercial solution by Contrast Security [21], and a Burp Suite plugin named Autorize [22] that dynamically scans applications for authentication and authorization issues. Both tools are similar to route-detect in that they broadly analyze route security, but different in that they do not perform static analysis against an application's source code.

Contrast Security's route coverage solution provides Interactive Application Security Testing (IAST). IAST is more similar to an application firewall than static code analysis. It integrates with a live, running application, associates vulnerabilities with a corresponding web request, and alerts when it detects issues. The key differences between this solution and route-detect are that route-detect does not require a running application–it can simply analyze code–and Contrast's solution is searching for real-time exploitation of vulnerabilities, not authentication and authorization information. Due to the commercial nature of Contrast's solution, I could not perform a comparison against route-detect. Like IAST, Dynamic Application Security Testing (DAST) relies on a running application to perform their security testing.

Autorize is a Burp Suite plugin for performing DAST analysis. It makes web requests to a running application to perform its security testing. Setting up a running application can be time consuming compared to simply running analyses against that application's code. Further, Autorize performs its testing through a form of differential role analysis. That is, it requires a low privilege user and a high privilege user and attempts to make judgements based on requests sent with both privilege levels. This is reasonable for an application with two privilege levels, but can become unwieldy for applications with a large number of roles. Furthermore, to perform a comprehensive analysis, each role must be compared against every other role to search for authorization discrepancies. During route-detect's evaluation of dependent web applications, in applications that used authorization, there were an average of 19 roles used. Using Autorize for these applications would be comparatively slow compared to route-detect because of the differential role comparison explosion and the requirement to send live web requests. Because of this, and time constraints during the evaluation phase, I was not able to run Autorize against the dependent web applications. IAST, DAST, and Static Application Security Testing (SAST) all have trade-offs with regards to one another.

route-detect uses SAST to perform its analyses. SAST is comparatively fast, and it can be run early in the Software Development Lifecycle (SDLC), allowing it to catch bugs earlier in development [23]. Results have shown that reporting issues earlier in the SDLC is beneficial, as developers perceive them as more important [24]. The downside of SAST is a propensity for false positives. As noted in the dependent application analysis, route-detect is not immune from false positives, but these can be quickly rectified with testing and Semgrep rule improvements.

4 Discussion

The following section discusses approaches to web application routing and its authentication and authorization implementation methods. There will also be discussion on the limitations of route-detect's efficacy based on the details of these various implementations.

Due to time constraints, claims of accuracy or inaccuracy in the following subsections are made based on manual investigation of route-detect results, not on more technical statistical analyses such as precision, recall, and F-score [25].

4.1 Routing Security Configuration Approaches

During the development and evaluation of route-detect it became apparent that specifying routes and their security properties could broadly be viewed across two different axes: *where* security properties are specified in relation to routes, and *how* routes and their security properties are specified in code. Each axis can be further split into three categories.

Where security properties are specified includes global, interprocedural, and intraprocedural locations. *How* routes and their security properties are specified includes configuration, annotation, and middleware methods. These three categories often pair across axes, but not always. Generally speaking, intraprocedurally defined routes use annotation-based security configuration, interprocedurally defined routes use configuration-based security, and middleware-based security can be intraprocedurally or interprocedurally defined. Further, web application frameworks often allow multiple configuration options along each axis. Both axes will be explored further below. See Appendix A5 for the full table of framework configuration options.

4.1.1 Security Configuration Location. Route authentication and authorization information can be specified globally, interprocedurally, or intraprocedurally.

Global configuration defines route access controls in a single location for all routes. Access controls are typically defined against a route prefix. For example, URL paths underneath the */api/v1* prefix use query parameter token authentication, and URL paths underneath the */api/v2* prefix use HTTP header token authentication. Java's Spring framework provides an example of global route security configuration [26], as seen in Appendix A6. In this example, all routes underneath the */products* path are unauthenticated, and all routes underneath the */customers* path require authentication and the *ADMIN* role.

Interprocedural configuration defines routes and their associated access controls across separate procedures or files in a many-to-many relationship. Typically, routes are specified in one or more files, then reference a view or controller in a separate file which performs request processing for that route's path. This is common in Model-View-Controller (MVC) applications [27]. In the interprocedural case, routes are logically far from their access control information, which reduces route-detect's analysis accuracy. Python's Django framework (Appendix A7) provides an example of interprocedural route security configuration [28]. In this example, the *login_required* decorator is applied to the view, which is referenced in a separate route definition file.

Intraprocedural configuration defines routes and their access controls in the same file, and generally the same procedure, or at least the same class or module. Typically, the route and its associated authentication or authorization information are defined logically close to one another. This improves route-detect's accuracy when analyzing access control information. Python's Flask (Appendix A8) framework provides an example of intraprocedural route security configuration [7]. In this example, both the *route* and *login_required* decorators are specified on the same procedure.

4.1.2 Security Configuration Method. Route authentication and authorization information can be specified by three different code patterns: configuration, annotations, and middleware.

Configuration defines routes and potentially their access control information according to a well-defined data structure, which is then interpreted by the web application framework. This route and access control data can be defined in a declarative, static file like a YAML document, as is possible with PHP's Symfony [29] framework. It can also be defined in a source code file using convention over configuration [30], as with Ruby's Rails [31] and Python's Django [28] frameworks. See Appendix A7 for an example of Django, and Appendix A9 for an example of Symfony. In the Django and Symfony examples, *urlpatterns* and *routes.yaml* are interpreted by convention by each respective framework, and subsequently used as the routing information. This configuration method is often easy to analyze for route information, but can be challenging to associate with access control information due to interprocedural association.

Annotations define routes by special programming language constructs, or otherwise denote a specific function or class as implementing a route or access control information. Examples of special language constructs are decorators in Python, annotations in Java, and attributes in PHP. Appendix A6 provides a code example of annotation-based configuration in Java's Spring framework using the *RequestMapping* annotation. Additionally, Appendix A8 provides a code example of annotation-based configuration in Python's Flask framework using the *login_required* decorator. Annotation-based configuration is generally accurately analyzed by route-detect due to its intraprocedural nature and specialized code constructs, which allow for highly specific code queries to be made for route and access control information.

Middleware provides a mechanism for specifying access control information on a set of routes. The middleware software pattern provides a pipeline of components that can perform work on a request and choose to pass the request along to the next component or exit the pipeline [32]. This pattern is well suited to the task of authentication and authorization because components can implement these access control decisions and choose to continue processing the request or not. Go's Chi and JavaScript's Express are examples of frameworks that use middleware for access control. Appendix A10 provides an example for Chi [33], and Appendix A11 provides an example for Express [34]. Chi provides the *Use* method on router objects, and Express provides the *use* method on application and router objects for specifying middleware components. Middleware-based access control can be specified intraprocedurally or interprocedurally when using subrouters defined in different procedures or source code files. These subrouters can be added to the main router under a URL path prefix. As with other access control configuration methods, intraprocedurally defined information is accurately detected by route-detect, while interprocedural information has reduced accuracy.

4.2 Limitations

As with all tools, route-detect has both strengths and weaknesses when performing its analyses. This section will discuss the limitations of route-detect, including challenges with interprocedural analysis, and implicit configuration.

4.2.1 Interprocedural Analysis. Interprocedural analysis involves analyzing code properties *across* procedure, or function, boundaries. This often also includes across source code file boundaries. In contrast, intraprocedural analysis analyzes code properties *within* a single procedure or function. Interprocedural analysis is generally slower, less accurate, and more challenging to implement than intraprocedural analysis [35]. For this reason, route-detect results were less accurate when analyzing interprocedurally defined access control information vs. intraprocedurally defined. route-detect implements a rudimentary interprocedural analysis algorithm in an attempt to join route and authentication or authorization data. This implementation yielded modest results.

To perform its interprocedural analysis, route-detect gathers route, authentication, and authorization information, then attempts to join these datasets on a piece of shared state. This shared state is often a module, class, or function name. For example, if a framework associates a route with a function to call when that route is requested, then route-detect can track the function name with a Semgrep metavariable, and later attempt to join on that name with functions that look like route handlers. This basic implementation performs poorly against frameworks whose routing is performed both interprocedurally and implicitly or dynamically. In other words, frameworks that do not make an explicit connection between route and request handler. This topic will be explored further in the following section.

Interprocedurally connecting routes with their respective route handler will generally require a fully qualified name for the handler. This involves understanding the module or namespace system for that programming language. For example, to connect a route to its controller in Ruby's Rails framework, route-detect would need to understand and re-implement Ruby's module system, and gather filesystem path information to produce fully qualified names for route handlers. Instead, route-detect attempts to simply connect a route to its controller based on class name, omitting module information, but this often produces ambiguous results as class names may not be unique. Similarly, route-detect faces issues with Python's Django framework due to Django's heavy usage of class-based views. These views often employ the mixin code pattern [36] for access control, and use class inheritance to share functionality. Tracking access control information across multiple levels of inheritance in an interprocedural fashion is a challenge for route-detect.

In sum, technologies requiring interprocedural analysis are not well supported by route-detect. It attempts to perform some basic interprocedural analysis for routes in Ruby's Rails framework, but additional improvements need to be made to improve accuracy in gathering and associating access control information.

4.2.2 Implicit Security Configuration Methods. Using implicit or dynamic features in either a framework or programming language's routing functionality makes analysis challenging. Some examples of implicit behavior include convention over configuration and Domain-Specific Languages (DSLs). Configuration by convention relies on an implicit convention for specifications rather than an explicitly defined relationship. DSLs implement a custom specification altogether. These problematic cases generally require re-implementing a framework's internal route resolution algorithm and applying it during analysis to connect the route and access control datasets. For example, Ruby's Rails framework implements a custom DSL for route specification and connects routes with application controllers by convention [31]. Access control information is specified in the controller, so to connect a route with its access control information it must be connected with its controller. Since this connection is performed by convention within the Rails framework, it is difficult for route-detect to link these two pieces of data.

Implicit or dynamic behavior makes it difficult for route-detect to connect route information with its access control information defined elsewhere. When this information is explicitly defined, route-detect can use this information to connect the two datasets.

5 Further Research

route-detect initially focused on providing security practitioners with route authentication and authorization information for common web application frameworks. Its analyses can be expanded both in breadth and in depth. Further research is needed to deepen its ability to understand route security information through anomaly detection and improved interprocedural analysis. Broader analyses can be performed with the addition of new programming languages, web application frameworks, and authentication and authorization libraries.

5.1 Anomaly Detection

Anomaly detection seeks to find outliers in a data set. From a security perspective, outliers are often interesting because they indicate a deviation from the norm. For example, code exhibiting secure-by-default behavior may provide a mechanism for opting out. Conversely, code that does not employ secure-by-default behavior may provide a way to opt-in to a security mechanism such as authentication or authorization. In either case, code that is deviating from the norm, or avoiding the by-default behavior, may warrant further investigation to ensure a secure implementation. For example, if an application requires that all procedures validate user input before further processing can occur, and one procedure omits this behavior, this could be considered anomalous, and may warrant further investigation. route-detect would benefit from this type of analysis with regards to route's authentication and authorization properties.

There are many ways to define anomalous behavior. In route-detect's case, some examples would be: if all routes in a class or module are authenticated except one, or except N for some user-configurable value; if all routes in a single source code file are authenticated except one; if all routes in a directory are authenticated except one; or, if all routes in a single source code file use the same authorization role, except one route uses a different role. Implementing anomaly detection in route-detect may aid security practitioners in detecting security bugs, or by highlighting where they should focus their efforts.

Further research into the feasibility and efficacy of anomaly detection in determining insecure web application routes is needed. If useful, this would be a welcome improvement to route-detect.

5.2 Improved Interprocedural Analysis

The limitations section describes the challenges in route-detect's interprocedural analysis. Further research is needed to improve these types of analyses. For example, improving route-detect's ability to connect a route with its authentication and authorization information in a separate procedure or source code file. Can the current implementation, which uses separate Semgrep rules for routes and authentication and authorization information, be improved to produce better results? Can other features of Semgrep like taint analysis be used instead? Should another tool be used, like GitHub's CodeQL [37]? Improving route-detect's interprocedural analysis will make the tool more accurate, and reduce false positives.

5.3 Additional Web Application Routes

The initial implementation of route-detect included broad support for various programming languages, web application frameworks, and authentication and authorization libraries. Of course, as new technologies are created, others are abandoned. Additionally, users require support for the long-tail of less common languages, frameworks, and libraries. route-detect would benefit from supporting these additional technologies. Some examples include Python's FastAPI and Pyramid frameworks, JavaScript's Vue.js framework, Java's Play Framework, Ruby's Sinatra framework, and Go's net/http package. Support for additional technologies requires additional maintenance burden, but provides broader utility for route-detect.

Security practitioners will often engage with many different technologies. For example, a security consultant may be analyzing a different codebase or organization every week or two as their engagement contract specifies. Further, a security engineer employed by a large technology company may support up to a hundred developers [38] and be engaging with different engineering teams across the organization. This constant switching means that security practitioners will often see many different technologies over time. Because of its support of common web application frameworks, and even the long-tail of technologies, security practitioners can utilize one tool for route security rather than many, disparate, inconsistent tools.

Further research is needed to support additional languages, frameworks and libraries, especially with regards to keeping analysis quality high as support for many new technologies are added.

6 Conclusion

This paper presented route-detect, a tool for statically detecting web application routes and their authentication and authorization properties. It is an improvement on both regular expression text search, and manual code analysis. During evaluation, it successfully analyzed millions of lines of code, found thousands of routes and their authentication and authorization properties, discovered 18 unintentionally unauthorized routes in a production application, and significantly reduced the time required to assess codebase routes compared to manual analysis and text search.

There is still work to be done. route-detect can be made both broader and deeper. It is a tool in the security practitioner's toolkit. It seeks to shed light on the authentication and authorization issues currently plaguing web applications. Security practitioners can use it as a precursor to deeper, more pointed route analyses, or combine it with other tools to target specific routes. The first step in addressing a problem is clarifying the issue, and route-detect aims to fill that need for web application route security.

ACKNOWLEDGMENTS

First, I would like to thank Dr. Mustaque Ahamad for his thoughtful feedback throughout the semester. His vast knowledge of academic research and deep insight into various computer security topics helped shape the research contained herein. I would also like to thank Jason Pool for his pointed feedback, and keeping me on track as work progressed.

Thank you to my group mates Nasryn El-Hinnawi, David Johnson, Zinet Kemal, Mihaela Kirkland, Tony Kirkland, Quentin Samuels, Joaquin Serna, and Joseph Shediack for providing encouragement and great suggestions on how this research and its presentation could be improved.

Finally, thank you to my family for supporting me throughout this process. Without you, I surely could not have made it this far.

REFERENCES

[1] *OWASP Top 10:2021*, https://owasp.org/Top10/.

[2] "API Security Top 10 2019." *OWASP API Security Project*, https://owasp.org/www-project-api-security/.

[3] "2022 CWE Top 25 Most Dangerous Software Weaknesses." *CWE*, https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[4] "CWE Definitions." *CWE Definitions List and Vulnerabilities for CWE Entries*, https://www.cvedetails.com/cwe-definitions.php.

[5] returntocorp. "returntocorp/semgrep: Lightweight Static Analysis for Many Languages. Find Bug Variants with Patterns That Look like Source Code." *GitHub*, https://github.com/returntocorp/semgrep.

[6] Bostock, Mike. "Data-Driven Documents." *D3.js*, https://d3js.org/.

[7] "Quickstart." *Quickstart - Flask Documentation (2.2.x)*, https://flask.palletsprojects.com/en/2.2.x/quickstart/.

[8] "What Is Token-Based Authentication?" *Okta*, https://www.okta.com/identity-101/what-is-token-based-authentication/.

[9] "Testing Rules." *Semgrep*, https://semgrep.dev/docs/writing-rules/testing-rules/.

[10] "Pytest: Helps You Write Better Programs." *Pytest*, https://docs.pytest.org/en/latest/.

[11] Bostock, Mike. "Tree, Tidy." *Observable*, 27 Nov. 2022, https://observablehq.com/@d3/tree.

[12] Frendt, Shay. "Introducing Topics." *The GitHub Blog*, 1 Feb. 2017, https://github.blog/2017-01-31-introducing-topics/.

[13] "About the Dependency Graph." *GitHub Docs*, https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.

[14] "Flask Is a Web Framework for Python, Based on the Werkzeug Toolkit." *GitHub*, https://github.com/topics/flask.

[15] "Site: Search Operator." *Google*, Google, https://developers.google.com/search/docs/monitor-debug/search-operators/all-search-site.

[16] BurntSushi. "BurntSushi/ripgrep: Ripgrep Recursively Searches Directories for a Regex Pattern While Respecting Your Gitignore." *GitHub*, https://github.com/BurntSushi/ripgrep.

[17] Woodruff, Benjamin. "Static Analysis at Scale: An Instagram Story." *Medium*, Instagram Engineering, 16 Aug. 2019, https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c.

[18] "Pattern Syntax." *Semgrep*, https://semgrep.dev/docs/writing-rules/pattern-syntax/.

[19] Distefano, Dino, et al. "Scaling Static Analyses at Facebook." *Communications of the ACM*, vol. 62, no. 8, 2019, pp. 62–70., https://doi.org/10.1145/3338112.

[20] Bessey, Al, et al. "A Few Billion Lines of Code Later." *Communications of the ACM*, vol. 53, no. 2, 2010, pp. 66–75., https://doi.org/10.1145/1646353.1646374.

[21] "Route Coverage." *Route Coverage*, https://docs.contrastsecurity.com/en/route-coverage.html.

[22] Tawily, Barak. "Portswigger/Autorize: Automatic Authorization Enforcement Detection Extension for BURP Suite Written in Jython Developed by Barak Tawily in Order to Ease Application Security People Work and Allow Them Perform an Automatic Authorization Tests." *GitHub*, https://github.com/PortSwigger/autorize.

[23] Graf, Michael. "Security Testing-SAST, Dast and IAST Explained." *Medium*, Digital Frontiers-Das Blog, 7 Mar. 2022, https://medium.com/digitalfrontiers/sast-dast-and-iast-explained-9324572a5d2b.

[24] Sadowski, Caitlin, et al. "Lessons from Building Static Analysis Tools at Google." *Communications of the ACM*, vol. 61, no. 4, 2018, pp. 58–66., https://doi.org/10.1145/3188720.

[25] Sasaki, Yutaka. "The Truth of the F-Measure." Jan. 2007.

[26] Baeldung, Written by: "Spring Security – Configuring Different Urls." *Baeldung*, 22 Sept. 2022, https://www.baeldung.com/spring-security-configuring-urls.

[27] Fowler, Martin. "Model View Controller." *P Of EAA: Model View Controller*, https://martinfowler.com/eaaCatalog/modelViewController.html.

[28] "Introduction to Class-Based Views." *Django Project*, https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/.

[29] Symfony. "Routing (Symfony Docs)." *Symfony Docs*, https://symfony.com/doc/current/routing.html.

[30] Bächle, Michael, and Paul Kirchberg. "Ruby on Rails." *IEEE Software*, vol. 24, no. 6, 2007, pp. 105–108., https://doi.org/10.1109/ms.2007.176.

[31] "Rails Routing from the Outside In." *Ruby on Rails Guides*, https://guides.rubyonrails.org/routing.html.

[32] Anderson, Rick, and Steve Smith. "ASP.NET Core Middleware." *Microsoft Learn*, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/.

[33] "Go-Chi Docs." *Chi*, https://go-chi.io/#/pages/routing.

[34] auth0. "Auth0/Express-JWT: Connect/Express Middleware That Validates a Jsonwebtoken (JWT) and Set the Req.user with the Attributes." *GitHub*, https://github.com/auth0/express-jwt.

[35] Hall, M W, Murphy, B R, and Amarasinghe, S P. "Interprocedural parallelization analysis: A case study". United States: N. p., 1995. Web.

[36] "Mix In." *wiki.c2.com*, http://wiki.c2.com/?MixIn.

[37] *CodeQL*, https://codeql.github.com/.

[38] Gupta, Manish. "Council Post: Stop Checking Boxes and Start Effectively Securing Development Pipelines." *Forbes*, Forbes Magazine, 5 May 2020, https://www.forbes.com/sites/forbestechcouncil/2020/05/05/stop-checking-boxes-and-start-effectively-securing-development-pipelines/.

APPENDIX

A1

Source code for the route-detect software can be found at: https://github.com/mschwager/route-detect.

Commit dbb2035 was used during evaluation.

A2

```
$ semgrep --config $(routes which laravel) koel/
2023-04-20 11:43:18,777 INFO routes.main Starting command which
2023-04-20 11:43:18,778 INFO routes.commands.which Printing rule path for laravel
2023-04-20 11:43:18,778 INFO routes.main Finished command which
Scanning 368 files with 3 php rules.
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 368/368 tasks 0:00:00
...
Findings:

  koel/routes/api.base.php
         routes.rules.laravel-route-authenticated
         Found authenticated Laravel route

         41┆ Route::post('broadcasting/auth', static function (Request $request) {
         42┆     $pusher = new Pusher(
         43┆     config('broadcasting.connections.pusher.key'),
         44┆     config('broadcasting.connections.pusher.secret'),
         45┆     config('broadcasting.connections.pusher.app_id'),
         46┆     [
         47┆             'cluster' => config('broadcasting.connections.pusher.options.cluster'),
         48┆             'encrypted' => true,
         49┆     ]
         50┆     );
         [hid 3 additional lines, adjust with --max-lines-per-finding]
         55┆ Route::get('data', [DataController::class, 'index']);
         ⋮┆----------------------------------------
         57┆ Route::put('settings', [SettingController::class, 'update']);
         ⋮┆----------------------------------------
         62┆ Route::post('{song}/scrobble', [ScrobbleController::class, 'store']);
         ⋮┆----------------------------------------
         63┆ Route::post('songs/{song}/scrobble', [ScrobbleController::class, 'store']);
         ⋮┆----------------------------------------
         64┆ Route::put('songs', [SongController::class, 'update']);
         ⋮┆----------------------------------------
         66┆ Route::post('upload', UploadController::class);
         ⋮┆----------------------------------------
         69┆ Route::post('interaction/play', [PlayCountController::class, 'store']);
         ⋮┆----------------------------------------
         70┆ Route::post('interaction/like', [LikeController::class, 'store']);
         ⋮┆----------------------------------------
         71┆ Route::post('interaction/batch/like', [BatchLikeController::class, 'store']);
         ⋮┆----------------------------------------
         72┆ Route::post('interaction/batch/unlike', [BatchLikeController::class, 'destroy']);
         ⋮┆----------------------------------------
         73┆ Route::get('interaction/recently-played/{count?}', [RecentlyPlayedController::class, 'index'])->where([
         74┆     'count' => '\d+',
         75┆ ]);
         ⋮┆----------------------------------------
```

```
        80┆ Route::put('playlist/{playlist}/sync', [PlaylistSongController::class, 'update']); // @deprecated
        ⋮┆----------------------------------------
        81┆ Route::put('playlist/{playlist}/songs', [PlaylistSongController::class, 'update']);
        ⋮┆----------------------------------------
        82┆ Route::get('playlist/{playlist}/songs', [PlaylistSongController::class, 'index']);
        ⋮┆----------------------------------------
        86┆ Route::get('me', [ProfileController::class, 'show']);
        ⋮┆----------------------------------------
        87┆ Route::put('me', [ProfileController::class, 'update']);
        ⋮┆----------------------------------------
        90┆ Route::post('lastfm/session-key', [LastfmController::class, 'setSessionKey']);
        ⋮┆----------------------------------------
        91┆ Route::delete('lastfm/disconnect', [LastfmController::class, 'disconnect'])->name('lastfm.disconnect');
        ⋮┆----------------------------------------
        95┆ Route::get('youtube/search/song/{song}', [YouTubeController::class, 'searchVideosRelatedToSong']);
        ⋮┆----------------------------------------
        99┆ Route::get('album/{album}/info', [AlbumInformationController::class, 'show']);
        ⋮┆----------------------------------------
        100┆ Route::get('artist/{artist}/info', [ArtistInformationController::class, 'show']);
        ⋮┆----------------------------------------
        101┆ Route::get('song/{song}/info', [SongInformationController::class, 'show']);
        ⋮┆----------------------------------------
        104┆ Route::put('album/{album}/cover', [AlbumCoverController::class, 'update']);
        ⋮┆----------------------------------------
        105┆ Route::put('artist/{artist}/image', [ArtistImageController::class, 'update']);
        ⋮┆----------------------------------------
        106┆ Route::get('album/{album}/thumbnail', [AlbumThumbnailController::class, 'show']);
        ⋮┆----------------------------------------
        110┆ Route::get('/', [ExcerptSearchController::class, 'index']);
        ⋮┆----------------------------------------
        111┆ Route::get('songs', [SongSearchController::class, 'index']);
        ⋮┆----------------------------------------
        117┆ Route::post('song', [S3SongController::class, 'put'])->name('s3.song.put');
        ⋮┆----------------------------------------
        118┆ Route::delete('song', [S3SongController::class, 'remove'])->name('s3.song.remove');
        ⋮┆----------------------------------------
        routes.rules.laravel-route-unauthenticated
        Found unauthenticated Laravel route

        35┆ Route::post('me', [AuthController::class, 'login'])->name('auth.login');
        ⋮┆----------------------------------------
        36┆ Route::delete('me', [AuthController::class, 'logout']);
        ⋮┆----------------------------------------
        38┆ Route::get('ping', static fn () => null);
        ⋮┆----------------------------------------
        121┆ Route::get('demo/credits', [DemoCreditController::class, 'index']);
...
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.
  Scan skipped: 80 files matching .semgrepignore patterns
  For a full list of skipped files, run semgrep with the --verbose flag.

Ran 3 rules on 368 files: 60 findings.
```

A3

See the evaluation script to view the ripgrep regular expression command-line arguments used for each framework.

Repository	Framework	Language	Regex route count	route-detect route count	Notes
cloudfoundry/korifi	chi	Go	626	89	Common function names
dhax/go-base	chi	Go	29	19	Common function names
go-admin-team/go-admin	gin	Go	104	101	Commented out code
gotify/server	gin	Go	43	35	route-detect misses function constant propagation
photoprism/photoprism	gin	Go	150	136	Commented out code
google/exposure-notifications-server	gorilla	Go	78	40	Common function names
portainer/portainer	gorilla	Go	219	174	Common function names
Chocobozzz/PeerTube	angular	JavaScript/TypeScript	682	155	Common attribute names
bitwarden/clients	angular	JavaScript/TypeScript	252	211	Common attribute names
ever-co/ever-demand	angular	JavaScript/TypeScript	265	235	Common attribute names
directus/directus	express	JavaScript/TypeScript	801	176	Common function names
payloadcms/payload	express	JavaScript/TypeScript	60	30	Common function names
apache/superset	react	JavaScript/TypeScript	4	26	Misses route objects
elastic/kibana	react	JavaScript/TypeScript	566	536	
mattermost/mattermost-webapp	react	JavaScript/TypeScript	39	34	route-detect misses routes without a path
DependencyTrack/dependency-track	jax-rs	Java	189	189	
eclipse/kapua	jax-rs	Java	246	246	
eclipse/kura	jax-rs	Java	53	53	
macrozheng/mall	spring	Java	287	243	Misses class-level annotations
sqshq/piggymetrics	spring	Java	17	11	Misses class-level annotations, route-detect misses interfaces
thingsboard/thingsboard	spring	Java	384	347	Misses class-level annotations
croogo/croogo	cakephp	PHP	425	17	Common function names
passbolt/passbolt_api	cakephp	PHP	1319	67	Common function

					names
BookStackApp/BookStack	laravel	PHP	267	267	
koel/koel	laravel	PHP	60	60	
monicahq/monica	laravel	PHP	186	186	
Sylius/Sylius	symfony	PHP	246	165	Common YAML keys
bolt/core	symfony	PHP	58	59	
sulu/sulu	symfony	PHP	93	57	
DefectDojo/django-DefectDojo	django-rest-framework	Python	27	75	Misses viewsets
DefectDojo/django-DefectDojo	django	Python	347	345	Commented out code
saleor/saleor	django	Python	11	11	
wagtail/wagtail	django	Python	268	258	Commented out code
apache/airflow	flask	Python	19	19	
flaskbb/flaskbb	flask	Python	0	77	Misses class-based views
getredash/redash	flask	Python	28	27	route-detect misses nested route
howie6879/owllook	sanic	Python	241	35	Common function names
jacebrowning/memegen	sanic	Python	220	33	Common function names
Mapotempo/optimizer-api	grape	Ruby	227	7	Common function names
gitlabhq/gitlabhq	grape	Ruby	18389	910	Common function names
locomotivecms/engine	grape	Ruby	373	73	Common function names
diaspora/diaspora	rails	Ruby	3710	170	Common function names
discourse/discourse	rails	Ruby	18935	764	Common function names
gitlabhq/gitlabhq	rails	Ruby	22806	747	Common function names
		Total	73349	7515	

A4

See the evaluation script for how this data was generated.

Repository	Commit hash	Framework	Language	Lines of code	Semgrep runtime	Route count	Authn route count	Unauthn route count	Authz route count	Unauthz route count	Unknown route count	Role count
cloudfoundry/korifi	f208444	chi	Go	98151	5.53	89	84	5	0	0	0	0
dhax/go-base	09eeccd	chi	Go	3509	2.14	19	2	17	0	0	0	0
go-admin-team/ go-admin	c973d68	gin	Go	17684	3.68	101	83	18	0	0	0	0
gotify/server	9d4e37a	gin	Go	12479	3.26	35	7	28	0	0	0	0
photoprism/photoprism	09c5448	gin	Go	178943	16.42	136	72	64	0	0	0	0
google/exposure-notifications-server	36ebe20	gorilla	Go	53435	2.67	40	0	40	0	0	0	0
portainer/portainer	01ea9af	gorilla	Go	70120	3.84	174	142	32	0	0	0	0
Chocobozzz/PeerTube	2b02a9b	angular	JavaScript/ TypeScript	205750	3.7	155	0	0	77	78	0	6
bitwarden/clients	07b074f	angular	JavaScript/ TypeScript	171325	4.03	211	0	0	99	112	0	12
ever-co/ever-demand	89a735d	angular	JavaScript/ TypeScript	116591	3.97	235	0	0	45	190	0	30
directus/directus	baf778f	express	JavaScript/ TypeScript	115720	4.58	176	0	176	0	0	0	0
payloadcms/payload	85dee9a	express	JavaScript/ TypeScript	45991	3.04	30	1	4	0	0	25	0
apache/superset	f3055fc	react	JavaScript/ TypeScript	181507	4.04	26	0	26	0	0	0	0
elastic/kibana	756a94c	react	JavaScript/ TypeScript	3554464	41.92	536	0	536	0	0	0	0
mattermost/mattermost-webapp	780bbe8	react	JavaScript/ TypeScript	252728	5.23	34	0	34	0	0	0	0
DependencyTrack/ dependency-track	26e4345	jax-rs	Java	75512	7.24	189	0	4	185	0	0	1
eclipse/kapua	0abf65f	jax-rs	Java	361217	12.8	246	0	246	0	0	0	0
eclipse/kura	4bd490d	jax-rs	Java	398057	11.37	53	0	2	51	0	0	5
macrozheng/mall	057f2b9	spring	Java	85114	5.75	243	1	242	0	0	0	0
sqshq/piggymetrics	6bb2cf9	spring	Java	4483	2.26	11	0	7	4	0	0	3
thingsboard/thingsboard	7.91E+06	spring	Java	309349	17.29	347	1	20	326	0	0	12
croogo/croogo	fc06486	cakephp	PHP	58438	2.22	17	1	0	0	0	16	0
passbolt/passbolt_api	62d5841	cakephp	PHP	144784	2.81	67	1	0	0	0	66	0
BookStackApp/ BookStack	fd45d28	laravel	PHP	128491	3.87	267	181	86	0	0	0	0
koel/koel	545a303	laravel	PHP	19960	2.39	60	54	6	0	0	0	0
monicahq/monica	43f4d0a	laravel	PHP	178103	3.74	186	181	5	0	0	0	0
Sylius/Sylius	db48ca6	symfony	PHP	257977	12.65	165	1	0	0	0	164	0
bolt/core	3779c32	symfony	PHP	33137	3.32	59	2	0	3	53	1	3
sulu/sulu	0d92f0a	symfony	PHP	355362	8.58	57	0	0	0	0	57	0

DefectDojo/django-DefectDojo	4dfb3a4	django-rest-framework	Python	121730	4.97	75	57	18	0	0	0	0
DefectDojo/django-DefectDojo	4dfb3a4	django	Python	121730	5.96	345	0	345	0	0	0	0
saleor/saleor	95623d1	django	Python	435019	6.36	11	0	11	0	0	0	0
wagtail/wagtail	8aa892b	django	Python	197538	6.29	258	0	258	0	0	0	0
apache/airflow	643d736	flask	Python	639597	7.76	19	17	2	0	0	0	0
flaskbb/flaskbb	bc999f1	flask	Python	22608	3.08	77	27	50	0	0	0	0
getredash/redash	5cf13af	flask	Python	38006	2.65	27	10	17	0	0	0	0
howie6879/owllook	a21eb1e	sanic	Python	5620	2.42	35	3	32	0	0	0	0
jacebrowning/memegen	09e56cc	sanic	Python	5482	2.41	33	1	32	0	0	0	0
Mapotempo/optimizer-api	5b9dc8b	grape	Ruby	39431	4.28	7	0	7	0	0	0	0
gitlabhq/gitlabhq	fb336d5	grape	Ruby	1860727	111.63	910	384	317	209	0	0	99
locomotivecms/engine	c53fe5d	grape	Ruby	28364	4.73	73	68	5	0	0	0	0
diaspora/diaspora	2fe5a7b	rails	Ruby	66030	5.49	170	29	35	0	0	106	0
discourse/discourse	c8a4081	rails	Ruby	484347	23.17	764	37	84	0	0	643	0
gitlabhq/gitlabhq	fb336d5	rails	Ruby	1860727	53.68	747	12	328	0	0	407	0
			Total	13415337	449.22	7515	1459	3139	999	433	1485	171

A5

Framework	Language	Global	Interprocedural	Intraprocedural	Configuration	Annotation	Middleware
chi	Go		x	x			x
gin	Go		x	x			x
gorilla	Go		x	x			x
angular	JavaScript/ TypeScript			x	x		
express	JavaScript/ TypeScript		x	x			x
react	JavaScript/ TypeScript			x		x	
jax-rs	Java			x		x	
spring	Java	x		x		x	
cakephp	PHP	x					x
laravel	PHP			x			x
symfony	PHP	x		x	x	x	
django	Python		x	x	x	x	x
django-rest-framework	Python		x	x	x	x	
flask	Python		x	x		x	
sanic	Python		x	x		x	
grape	Ruby			x	x		
rails	Ruby		x		x		
	Total	3	9	15	6	8	7

A6

```
// Api.java
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class ExampleResource {
    @RequestMapping(value = "/customers/id", method = RequestMethod.GET)
    public Response customer() {
        return Response.ok().build();
    }
}

// SecurityConfig.java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfiguration {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) {
        http.authorizeRequests()
            .antMatchers("/products/**")
            .permitAll()
            .and()
            .authorizeRequests()
            .antMatchers("/customers/**")
            .hasRole("ADMIN")
            .anyRequest()
            .authenticated()
            .and()
            .httpBasic();
        return http.build();
    }
}
```

A7

```
# myapp/views.py
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.http import HttpResponse
from django.views import View

class MyView(View):
    @method_decorator(login_required)
    def get(self, request):
        return HttpResponse("result")

# myapp/urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [path("about/", MyView.as_view())]
```

A8

```
# myapp.py
from flask import Flask
from flask_login import login_required

app = Flask(__name__)

@app.route("/me")
@login_required
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }
```

A9

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    public function list(): Response
    {
        // ...
    }
}


# config/routes.yaml
blog_list:
    path: /blog
    controller: App\Controller\BlogController::list
```

A10

```
// main.go
import "github.com/go-chi/chi/v5"

func main() {
    r := chi.NewRouter()

    // Public Routes
    r.Group(func(r chi.Router) {
        r.Get("/", HelloWorld)
        r.Get("/{AssetUrl}", GetAsset)
        r.Get("/manage/url/{path}", FetchAssetDetailsByURL)
        r.Get("/manage/id/{path}", FetchAssetDetailsByID)
    })

    // Private Routes
    // Require Authentication
    r.Group(func(r chi.Router) {
        r.Use(AuthMiddleware)
        r.Post("/manage", CreateAsset)
    })
}
```

A11

```
// api.js
const express = require('express')
const app = express()
const { expressjwt: jwt } = require('express-jwt')

app.use(
    jwt({
        secret: "secret",
        algorithms: ["HS256"],
    }),
);

app.get('/', (req, res) => {
  res.send('Authenticated path')
})
```