AWS Compute Blog

Building well-architected serverless applications: Implementing application workload security – part 1

by Julian Wood | on 06 JUL 2021 | in Amazon API Gateway, Amazon EC2 Container Registry, AWS Lambda, AWS Serverless Application Model, AWS WAF, AWS Well-Architected Framework, AWS Well-Architected Tool, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the introduction post for a table of contents and explanation of the example application.

Security question SEC3: How do you implement application security in your workload?

> Review and automate security practices at the application code level, and enforce security code review as part of development workflow. By implementing security at the application code level, you can protect against emerging security threats and reduce the attack surface from malicious code, including third-party dependencies.

Required practice: Review security awareness documents frequently

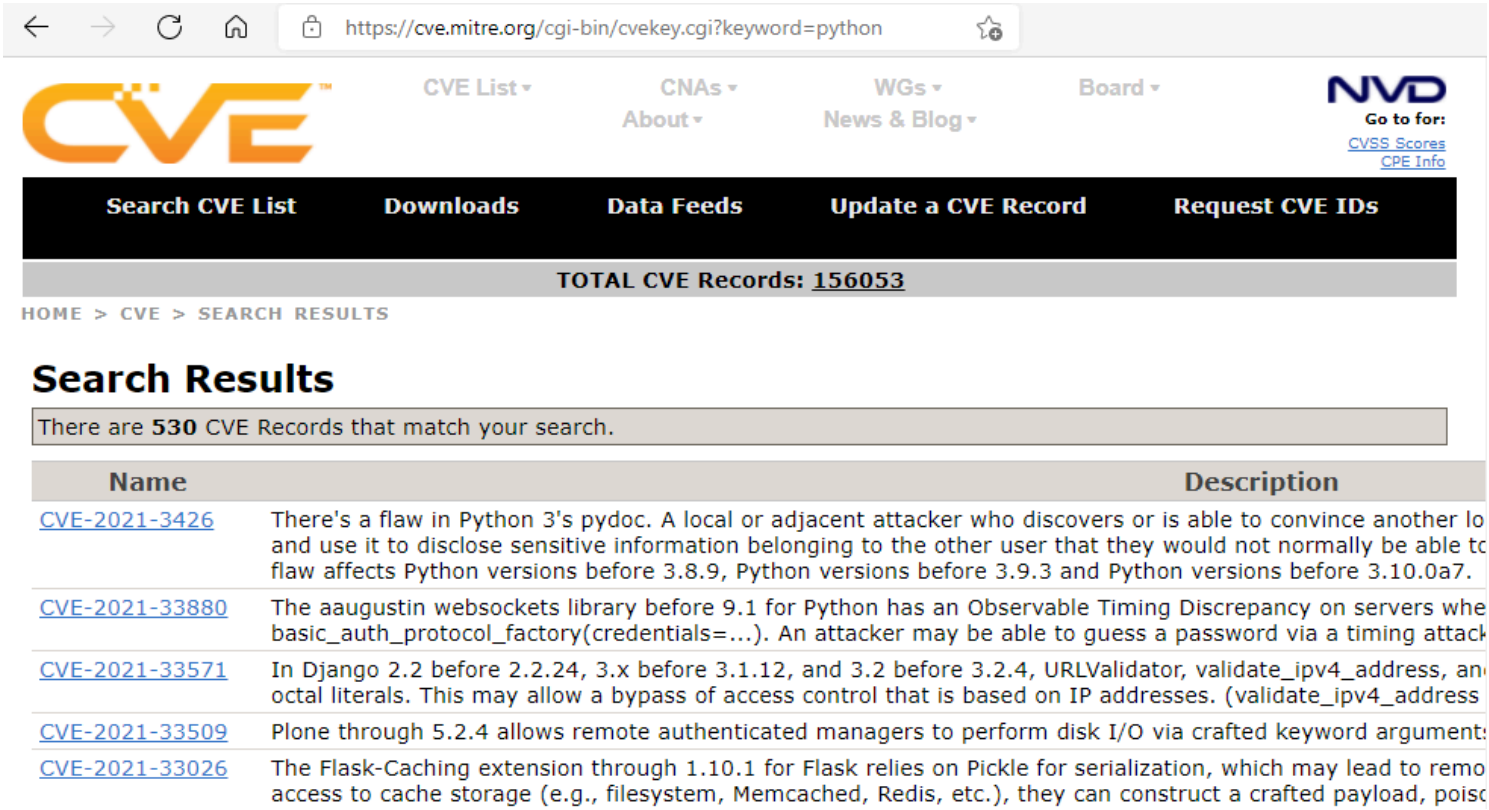
Stay up to date with both AWS and industry security best practices to understand and evolve protection of your workloads. Having a clear understanding of common threats helps you to mitigate them when developing your workloads.

The AWS Security Blog provides security-specific AWS content. The Open Web Application Security Project (OWASP) Top 10 is a guide for security practitioners to understand the most common application attacks and risks. The OWASP Top 10 Serverless Interpretation provides information specific to serverless applications.

Review and subscribe to vulnerability and security bulletins

Regularly review news feeds from multiple sources that are relevant to the technologies used in your workload. Subscribe to notification services to be informed of critical threats in near-real time.

The Common Vulnerabilities and Exposures (CVE) program identifies, defines, and catalogs publicly disclosed cybersecurity vulnerabilities. You can search the CVE list directly, for example “Python”.



CVE Python search

The US National Vulnerability Database (NVD) allows you to search by vulnerability type, severity, and impact. You can also perform advanced searching by vendor name, product name, and version numbers. GitHub also integrates with CVE, which allows for advanced searching within the CVEproject/cvelist repository.

AWS Security Bulletins are a notification system for security and privacy events related to AWS services. Subscribe to the security bulletin RSS feed to keep up to date with AWS security announcements.

The US Cybersecurity and Infrastructure Security Agency (CISA) provides alerts about current security issues, vulnerabilities, and exploits. You can receive email alerts or subscribe to the RSS feed.

AWS Partner Network (APN) member Palo Alto Networks provides the "Serverless architectures Security Top 10" list. This is a security awareness and education guide to use while designing, developing, and testing serverless applications to help minimize security risks.

Good practice: Automatically review a workload's code dependencies/libraries

Regularly reviewing application and code dependencies is a good industry security practice. This helps detect and prevent non-certified application code, and ensure that third-party application dependencies operate as intended.

Implement security mechanisms to verify application code and dependencies before using them

Combine automated and manual security code reviews to examine application code and its dependencies to ensure they operate as intended. Automated tools can help identify overly complex application code, and common security vulnerability exposures that are already cataloged.

Manual security code reviews, in addition to automated tools, help ensure that application code works as intended. Manual reviews can include business contextual information and integrations that automated tools may not capture.

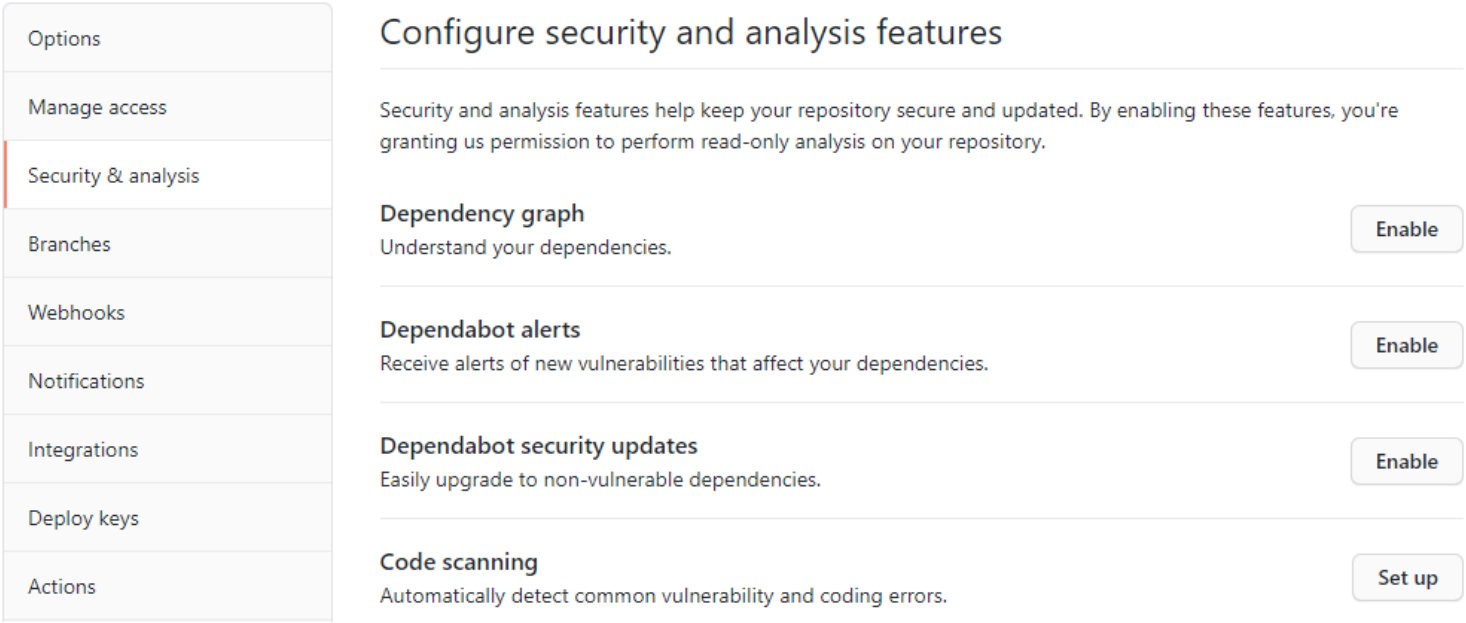
Before adding any code dependencies to your workload, take time to review and certify each dependency to ensure that you are adding secure code. Use third-party services to review your code dependencies on every commit automatically.

OWASP has a code review guide and dependency check tool that attempt to detect publicly disclosed vulnerabilities within a project's dependencies. The tool has a command line interface, a Maven plugin, an Ant task, and a Jenkins plugin.

GitHub has a number of security features for hosted repositories to inspect and manage code dependencies.

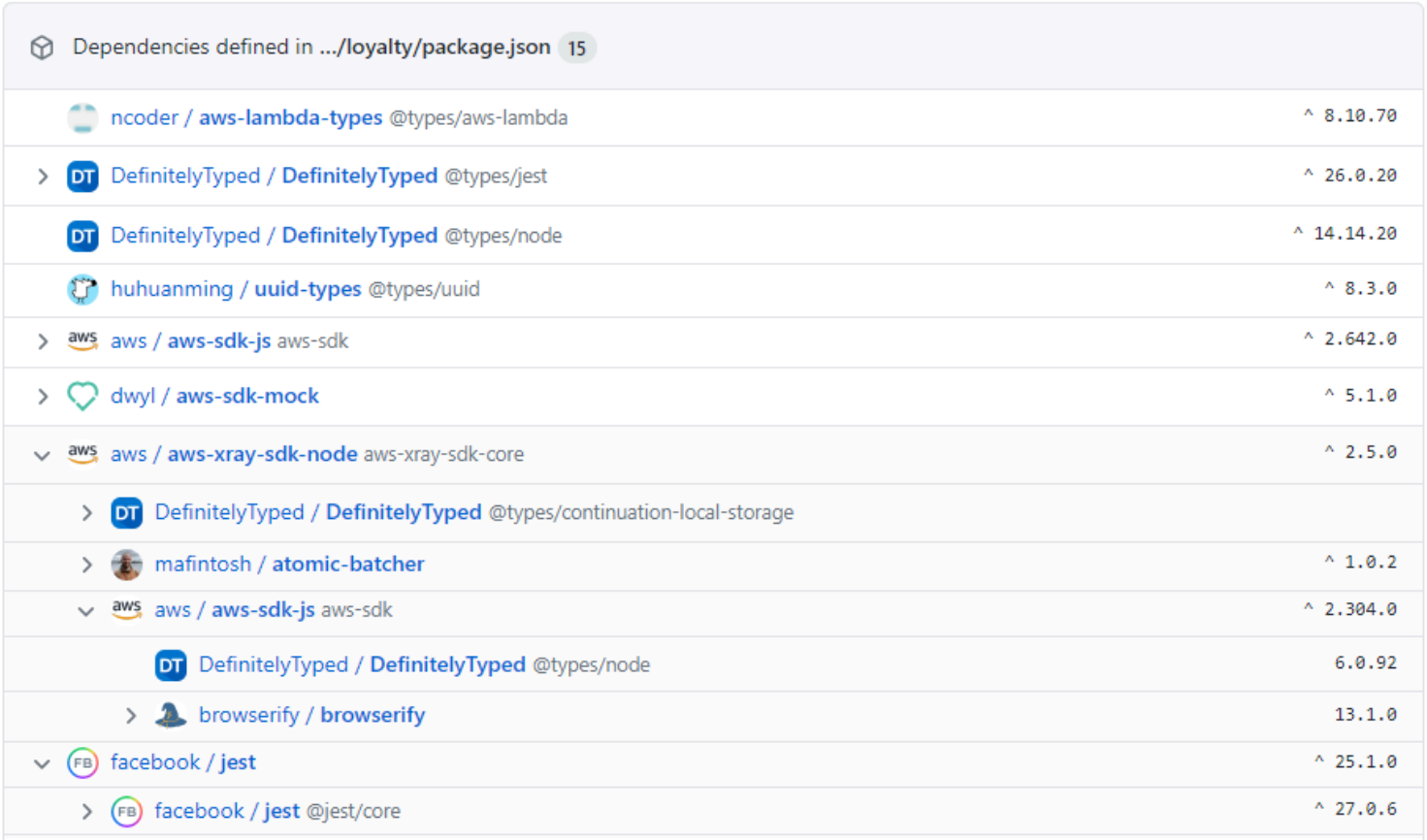
The dependency graph allows you to explore the packages that your repository depends on. Dependabot alerts show information about dependencies that are known to contain security vulnerabilities. You can choose whether to have pull requests generated automatically to update these dependencies. Code scanning alerts automatically scan code files to detect security vulnerabilities and coding errors.

You can enable these features by navigating to the **Settings** tab, and selecting **Security & analysis**.



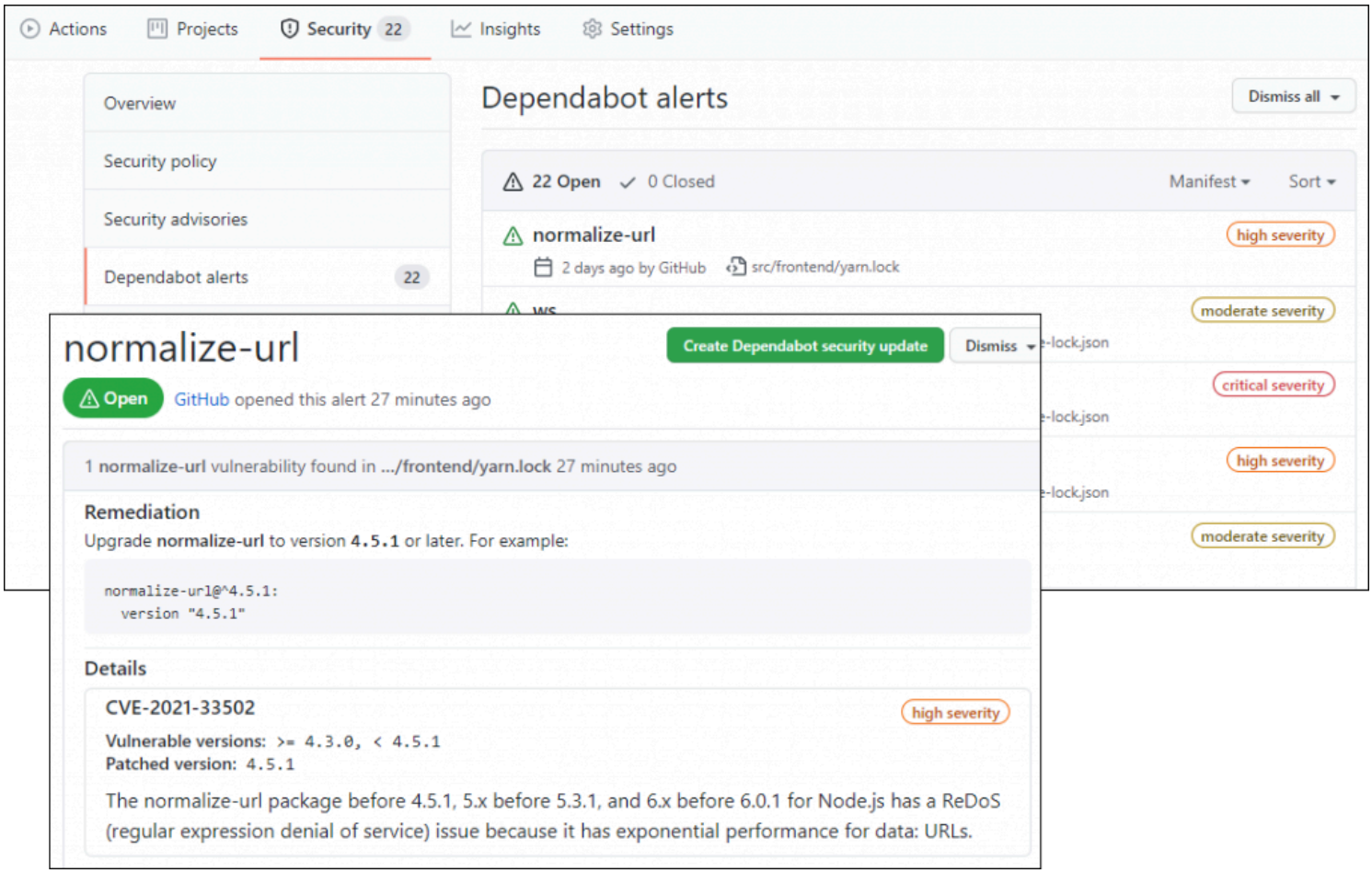
GitHub configure security and analysis features

Once Dependabot analyzes the repository, you can view the dependencies graph from the **Insights** tab. In the serverless airline example used in this series, you can view the *Loyalty* service package.json dependencies.



Serverless airline loyalty dependencies

Dependabot alerts for security vulnerabilities are visible in the **Security** tab. You can review alerts and see information about how to resolve them.



Dependabot alert

Once Dependabot alerts are enabled for a repository, you can also view the alerts when pushing code to the repository from the terminal.

```
13:22 jrwood@DeskCloud C:\..\aws-serverless-airline-booking develop ↑1 +1 git push
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.01 KiB | 346.00 KiB/s, done.
Total 6 (delta 5), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
remote:
remote: GitHub found 23 vulnerabilities on julianwood/aws-serverless-airline-booking's default branch (1 critical, 11 high
, 10 moderate, 1 low). To find out more, visit:
remote:      https://github.com/julianwood/aws-serverless-airline-booking/security/dependabot
remote:
To https://github.com/julianwood/aws-serverless-airline-booking.git
   47971a7..ecb7170  develop -> develop
```

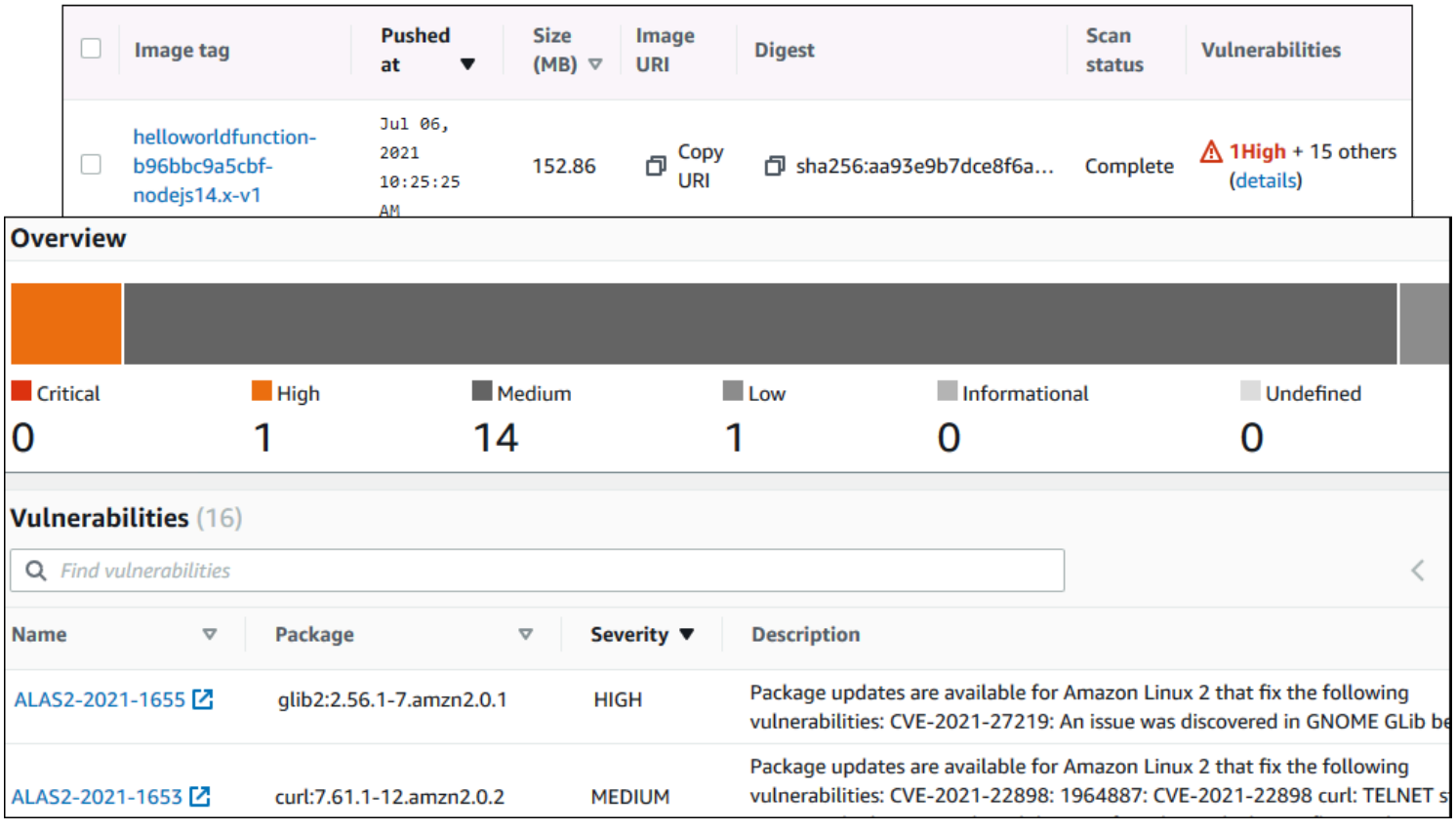
Dependabot terminal alert

If you enable security updates, Dependabot can automatically create pull requests to update dependencies.

Dependabot pull requests

AWS Partner Network (APN) member Snyk has an integration with AWS Lambda to manage the security of your function code. Snyk determines what code and dependencies are currently deployed for Node.js, Ruby, and Java projects. It tests dependencies against their vulnerability database.

If you build your functions using container images, you can use Amazon Elastic Container Registry's (ECR) image scanning feature. You can manually scan your images, or scan them on each push to your repository.



Elastic Container Registry image scanning example results

Best practice: Validate inbound events

Sanitize inbound events and validate them against a predefined schema. This helps prevent errors and increases your workload's security posture by catching malformed events or events intentionally crafted to be malicious. The OWASP Input validation cheat sheet includes guidance for providing input validation security functionality in your applications.

Validate incoming HTTP requests against a schema

Implicitly trusting data from clients could lead to malformed data being processed. Use data type validators or web application frameworks to ensure data correctness. These should include regular expressions, value range, data structure, and data normalization.

You can configure Amazon API Gateway to perform basic validation of an API request before proceeding with the integration request to add another layer of security. This ensures that the HTTP request matches the desired format. Any HTTP request that does not pass validation is rejected, returning a 400 error response to the caller.

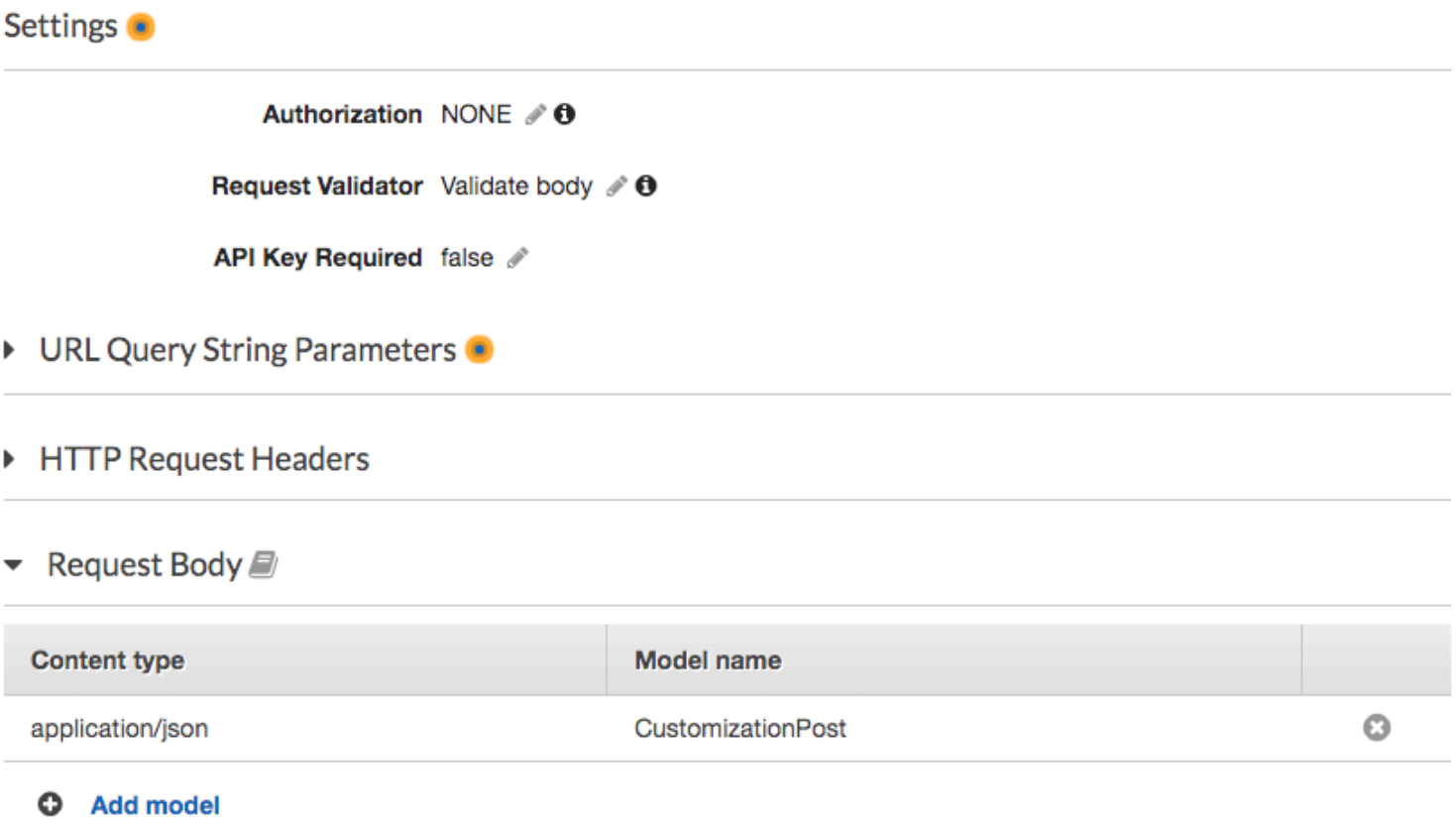
The Serverless Security Workshop has a module on API Gateway input validation based on the fictional Wild Rydes unicorn raid hailing service. The example shows a REST API endpoint where partner companies of Wild Rydes can submit unicorn customizations, such as branded capes, to advertise their company. The API endpoint should ensure that the request body follows specific patterns. These include checking the *ImageURL* is a valid URL, and the ID for *Cape* is a numeric value.

In API Gateway, a model defines the data structure of a payload, using the JSON schema draft 4. The model ensures that you receive the parameters in the format you expect. You can check them against regular expressions. The *CustomizationPost* model specifies that the *ImageURL* and *Cape* schemas should contain the following valid patterns:

JSON

```
    "imageUrl": {
      "type": "string",
      "title": "The Imageurl Schema",
      "pattern": "^https?:\/\/[-a-zA-Z0-9@:%_+.~#?&//=]+$"
    },
    "sock": {
      "type": "string",
      "title": " The Cape Schema ",
      "pattern": "^[0-9]*$"
    },
    …
```

The model is applied to the `/customizations/post` method as part of the *Method Request*. The *Request Validator* is set to **Validate body** and the *CustomizationPost* model is set for the *Request Body*.



API Gateway request validator

When testing the `POST /customizations` API with valid parameters using the following input:

```
JSON
{
   "name":"Cherry-themed unicorn",
   "imageUrl":"https://en.wikipedia.org/wiki/Cherry#/media/File:Cherry_Stella444.jpg",
   "sock": "1",
   "horn": "2",
   "glasses": "3",
   "cape": "4"
}
```

The result is a valid response:

```
{"customUnicornId":<the-id-of-the-customization>}
```

Testing validation to the POST /customizations API using invalid parameters shows the input validation process.

The *ImageUrl* is not a valid URL:

```
JSON
  {
     "name":"Cherry-themed unicorn",
```

```
    "imageUrl":"htt://en.wikipedia.org/wiki/Cherry#/media/File:Cherry_Stella444.jpg",
    "sock": "1" ,
    "horn": "2" ,
    "glasses": "3",
    "cape": "4"
}
```

The *Cape* parameter is not a number, which shows a SQL injection attempt.

JSON

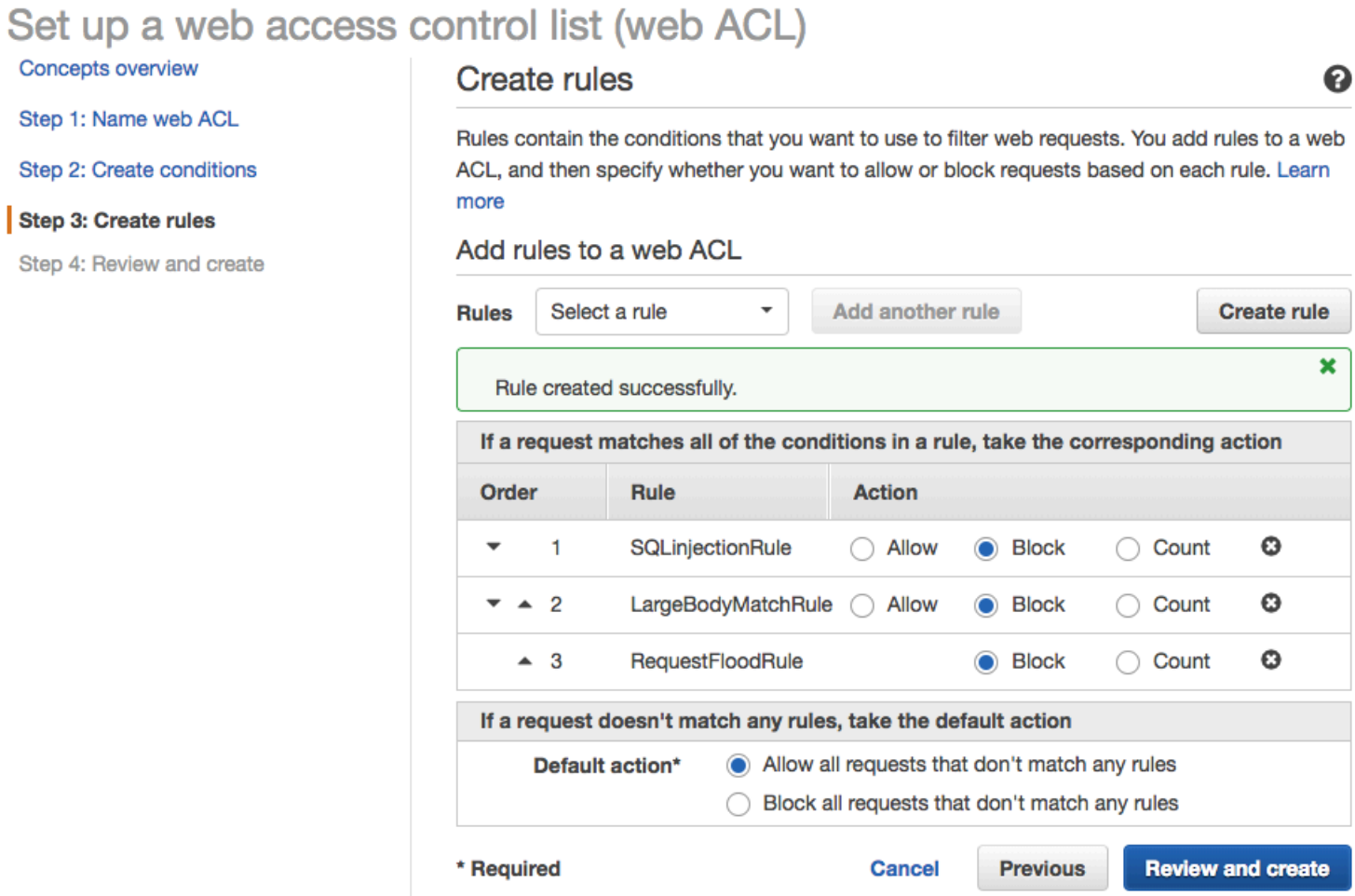
```
{
    "name":"Orange-themed unicorn",
    "imageUrl":"https://en.wikipedia.org/wiki/Orange_(fruit)#/media/File:Orange-Whole-S
    "sock": "1",
    "horn": "2",
    "glasses": "3",
    "cape":"2); INSERT INTO Cape (NAME,PRICE) VALUES ('Bad color', 10000.00"
}
```

These return a *400 Bad Request* response from API Gateway before invoking the Lambda function:

```
{"message": "Invalid request body"}
```

To gain further protection, consider adding an AWS Web Application Firewall (AWS WAF) access control list to your API endpoint. The workshop includes an AWS WAF module to explore three AWS WAF rules:

- Restrict the maximum size of request body
- SQL injection condition as part of the request URI
- Rate-based rule to prevent an overwhelming number of requests



AWS WAF ACL

AWS WAF also includes support for custom responses and request header insertion to improve the user experience and security posture of your applications.

For more API Gateway security information, see the security overview whitepaper.

Also add further input validation logic to your Lambda function code itself. For examples, see "Input Validation for Serverless".

Conclusion

Implementing application security in your workload involves reviewing and automating security practices at the application code level. By implementing code security, you can protect against emerging security threats. You can improve the security posture by checking for malicious code, including third-party dependencies.

In this post, I cover reviewing security awareness documentation such as the CVE database. I show how to use GitHub security features to inspect and manage code dependencies. I then show how to validate inbound events using API Gateway request validation.

This well-architected question continues in part 2 where I look at securely storing, auditing, and rotating secrets that are used in your application code.

For more serverless learning resources, visit Serverless Land.

TAGS: serverless, well-architected