15

DATA BREACHES AND BUG BOUNTIES

The real-world API breaches and bounties covered in this chapter should illustrate how actual hackers have exploited API vulnerabilities, how vulnerabilities can be combined, and the significance of the weaknesses you might discover.

Remember that an app's security is only as strong as the weakest link. If you're facing the best firewalled, multifactor-based, zero-trust app but the blue team hasn't dedicated resources to securing their APIs, there is a security gap equivalent to the Death Star's thermal exhaust port. Moreover, these insecure APIs and exhaust ports are often intentionally exposed to the outside universe, offering a clear pathway to compromise and destruction. Use common API weaknesses like the following to your advantage when hacking.

The Breaches

After a data breach, leak, or exposure, people often point fingers and cast blame. I like to think of them instead as costly learning opportunities. To be clear, a *data breach* refers to a confirmed instance of a criminal exploiting a system to compromise the business or steal data. A *leak* or *exposure* is the discovery of a weakness that could have led to the compromise of sensitive information, but it isn't clear whether an attacker actually did compromise the data.

When data breaches take place, attackers generally don't disclose their findings, as the ones who brag online about the details of their conquests often end up arrested. The organizations that were breached also rarely disclose what happened, either because they are too embarrassed, they're protecting themselves from ad-

ditional legal recourse, or (in the worst case) they don't know about it. For that reason, I will provide my own guess as to how these compromises took place.

Peloton

> **Data quantity:** More than three million Peloton subscribers
> **Type of data:** User IDs, locations, ages, genders, weights, and workout information

In early 2021, security researcher Jan Masters disclosed that unauthenticated API users could query the API and receive information for all other users. This data exposure is particularly interesting, as US president Joe Biden was an owner of a Peloton device at the time of the disclosure.

As a result of the API data exposure, attackers could use three different methods to obtain sensitive user data: sending a request to the */stats/workouts/details* endpoint, sending requests to the */api/user/search* feature, and making unauthenticated GraphQL requests.

The /stats/workouts/details Endpoint

This endpoint is meant to provide a user's workout details based on their ID. If a user wanted their data to be private, they could select an option that was supposed to conceal it. The privacy feature did not properly function, however, and the endpoint returned data to any consumer regardless of authorization.

By specifying user IDs in the POST request body, an attacker would receive a response that included the user's age, gender, username, workout ID, and Peloton ID, as well as a value indicating whether their profile was private:

```
POST /stats/workouts/details HTTP/1.1
Host: api.onepeloton.co.uk
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101 Firefox/84.0
Accept: application/json, text/plain, */*
--snip--
{"ids":["10001","10002","10003","10004","10005","10006",]}
```

The IDs used in the attack could be brute-forced or, better yet, gathered by using the web application, which would automatically populate user IDs.

User Search

User search features can easily fall prey to business logic flaws. A GET request to the */api/user/search/:<username>* endpoint revealed the URL that led to the user's profile picture, location, ID, profile privacy status, and social information such as their number of followers. Anyone could use this data exposure feature.

GraphQL

Several GraphQL endpoints allowed the attacker to send unauthenticated requests. A request like the following would provide a user's ID, username, and location:

```
POST /graphql HTTP/1.1
Host: gql-graphql-gateway.prod.k8s.onepeloton.com
--snip--
{"query":
"query SharedTags($currentUserID: ID!) (\n  User: user(id: "currentUserID") (\r\n__typename\n  id\r\n  location\r\n  )\r\n)'
```

By using the `REDACTED` user ID as a payload position, an unauthenticated attacker could brute-force user IDs to obtain private user data.

The Peloton breach is a demonstration of how using APIs with an adversarial mindset can result in significant findings. It also goes to show that if an organization is not protecting one of its APIs, you should treat this as a rallying call to test its other APIs for weaknesses.

USPS Informed Visibility API

> **Data quantity:** Approximately 60 million exposed USPS users
> **Type of data:** Email, username, real-time package updates, mailing address, phone number

In November 2018, *KrebsOnSecurity* broke the story that the US Postal Service (USPS) website had exposed the data of 60 million users. A USPS program called Informed Visibility made an API available to authenticated users so that consumers could have near real-time data about all mail. The only problem was that any USPS authenticated user with access to the API could query it for any USPS account details. To make things worse, the API accepted wildcard queries. This means an attacker could easily request the user data for, say, every Gmail user by using a query like this one: */api/v1/find?email=*@gmail.com*.

Besides the glaring security misconfigurations and business logic vulnerabilities, the USPS API was also vulnerable to an excessive data exposure issue. When the data for an address was requested, the API would respond with all records associated with that address. A hacker could have detected the vulnerability by searching for various physical addresses and paying attention to the results. For example, a request like the following could have displayed the records of all current and past occupants of the address:

```
POST /api/v1/container/status
Token: UserA
--snip--

{
"street": "475 L' Enfant Plaza SW",
"city": Washington DC"
}
```

An API with this sort of excessive data exposure might respond with something like this:

```
{
       "street":"475 L' Enfant Plaza SW",
       "City":"Washington DC",
       "customer": [
              {
                     "name":"Rufus Shinra",
                     "username":"novp4me",
                     "email":"rufus@shinra.com",
                     "phone":"123-456-7890",
              },
              {
                     "name":"Professor Hojo",
                     "username":"sep-father",
                     "email":"prof@hojo.com",
                     "phone":"102-202-3034",
              }
              ]
}
```

The USPS data exposure is a great example of why more organizations need API-focused security testing, whether that be through a bug bounty program or penetration testing. In fact, the Office of Inspector General of the Informed Visibility

program had conducted vulnerability assessment a month prior to the release of the *KrebsOnSecurity* article. The assessors failed to mention anything about any APIs, and in the Office of Inspector General's "Informed Visibility Vulnerability Assessment," the testers determined that "overall, the IV web application encryption and authentication were secure" (*https://www.uspsoig.gov/sites/default/files/document-library-files/2018/IT-AR-19-001.pdf*). The public report also includes a description of the vulnerability-scanning tools used in order to test the web application that provided the USPS testers with false-negative results. This means that their tools assured them that nothing was wrong when in fact there were massive problems.

If any security testing had focused on the API, the testers would have discovered glaring business logic flaws and authentication weaknesses. The USPS data exposure shows how APIs have been overlooked as a credible attack vector and how badly they need to be tested with the right tools and techniques.

T-Mobile API Breach

> **Data quantity:** More than two million T-Mobile customers
> **Type of data:** Name, phone number, email, date of birth, account number, billing ZIP code

In August 2018, T-Mobile posted an advisory to its website stating that its cybersecurity team had "discovered and shut down an unauthorized access to certain information." T-Mobile also alerted 2.3 million customers over text message that their data was exposed. By targeting one of T-Mobile's APIs, the attacker was able to obtain customer names, phone numbers, emails, dates of birth, account numbers, and billing ZIP codes.

As is often the case, T-Mobile has not publicly shared the specific details of the breach, but we can go out on a limb and make a guess. One year earlier, a YouTube user discovered and disclosed an API vulnerability that may have been similar to the vulnerability that was exploited. In a video titled "T-Mobile Info Disclosure Exploit," user "moim" demonstrated how to exploit the T-Mobile Web Services Gateway API. This earlier vulnerability allowed a consumer to access data by using a single authorization token and then adding any user's phone number to the URL. The following is an example of the data returned from the request:

```
implicitPermissions:
0:
user:
IAMEmail:
"rafae1530116@yahoo.com"
userid:
"U-eb71e893-9cf5-40db-a638-8d7f5a5d20f0"
lines:
0:
accountStatus: "A"
ban:
"958100286"
customerType: "GMP_NM_P"
givenName: "Rafael"
insi:
"310260755959157"
isLineGrantable: "true"
msison:
"19152538993"
permissionType: "inherited"
1:
accountStatus: "A"
ban:
"958100286"
customerType: "GMP_NM_P"
givenName: "Rafael"
imsi:
"310260755959157"
isLineGrantable: "false"
msisdn:
"19152538993"
permissionType: "linked"
```

As you look at the endpoint, I hope some API vulnerabilities are already coming to mind. If you can search for your own information using the `msisdn` parameter, can you use it to search for other phone numbers? Indeed, you can! This is a BOLA vulnerability. What's worse, phone numbers are very predictable and often publicly available. In the exploit video, moim takes a random T-Mobile phone number from a dox attack on Pastebin and successfully obtains that customer's information.

This attack is only a proof of concept, but it has room for improvement. If you find an issue like this during an API test, I recommend working with the provider

to obtain additional test accounts with separate phone numbers to avoid exposing actual customer data during your testing. Exploit the findings and then describe the impact a real attack could have on the client's environment, particularly if an attacker brute-forces phone numbers and breaches a significant amount of client data.

After all, if this API was the one responsible for the breach, the attacker could have easily brute-forced phone numbers to gather the 2.3 million that were leaked.

The Bounties

Not only do bug bounty programs reward hackers for finding and reporting weaknesses that criminals would have otherwise compromised, but their write-ups are also an excellent source of API hacking lessons. If you pay attention to them, you might learn new techniques to use in your own testing. You can find write-ups on bug bounty platforms such as HackerOne and Bug Crowd or from independent sources like Pentester Land, ProgrammableWeb, and APIsecurity.io.

The reports I present here represent a small sample of the bounties out there. I selected these three examples to capture the diverse range of issues bounty hunters come across and the sorts of attacks they use. As you'll see, in some instances these hackers dive deep into an API by combining exploit techniques, following numerous leads, and implementing novel web application attacks. You can learn a lot from bounty hunters.

The Price of Good API Keys

> **Bug bounty hunter:** Ace Candelario
> **Bounty:** $2,000

Candelario began his bug hunt by investigating a JavaScript source file on his target, searching it for terms such as *api*, *secret*, and *key* that might have indicated a leaked secret. Indeed, he discovered an API key being used for BambooHR human resources software. As you can see in the JavaScript, the key was base64 encoded:

```
function loadBambooHRUsers() {
var uri = 'https://api.bamboohr.co.uk/api/gateway.php/example/v1/employees/directory');
return $http.get(uri, { headers: {'Authorization': 'Basic VXNlcm5hbWU6UGFzc3dvcmQ='};
}
```

Because the code snippet includes the HR software endpoint as well, any attacker who discovered this code could try to pass this API key off as their own parameter in an API request to the endpoint. Alternatively, they could decode the base64-encoded key. In this example, you could do the following to see the encoded credentials:

```
hAPIhacker@Kali:~$ echo 'VXNlcm5hbWU6UGFzc3dvcmQ=' | base64 -d
Username:Password
```

At this point, you would likely already have a strong case for a vulnerability report. Still, you could go further. For example, you could attempt to use the credentials on the HR site to prove that you could access the target's sensitive employee data. Candelario did so and used a screen capture of the employee data as his proof of concept.

Exposed API keys like this one are an example of a broken authentication vulnerability, and you'll typically find them during API discovery. Bug bounty rewards for the discovery of these keys will depend on the severity of the attack in which they can be used.

Lessons Learned

- Dedicate time to researching your target and discovering APIs.
- Always keep an eye out for credentials, secrets, and keys; then test what you can do with your findings.

Private API Authorization Issues

> **Bug bounty hunter:** Omkar Bhagwat
> **Bounty:** $440

By performing directory enumeration, Bhagwat discovered an API and its documentation located at *academy.target.com/api/docs*. As an unauthenticated user, Omkar was able to find the API endpoints related to user and admin management. Moreover, when he sent a GET request for the */ping* endpoint, Bhagwat noticed that the API responded to him without using any authorization tokens (see *Figure 15-1*). This piqued Bhagwat's interest in the API. He decided to thoroughly test its capabilities.

Figure 15-1: An example Omkar Bhagwat provided for his bug bounty write-up that demonstrates the API responding to his /ping request with a "pong" response

While testing other endpoints, Bhagwat eventually received an API response containing the error "authorization parameters are missing." He searched the site and found that many requests used an authorization Bearer token, which was exposed.

By adding that Bearer token to a request header, Bhagwat was able to edit user accounts (see *Figure 15-2*). He could then perform administrative functions, such as deleting, editing, and creating new accounts.

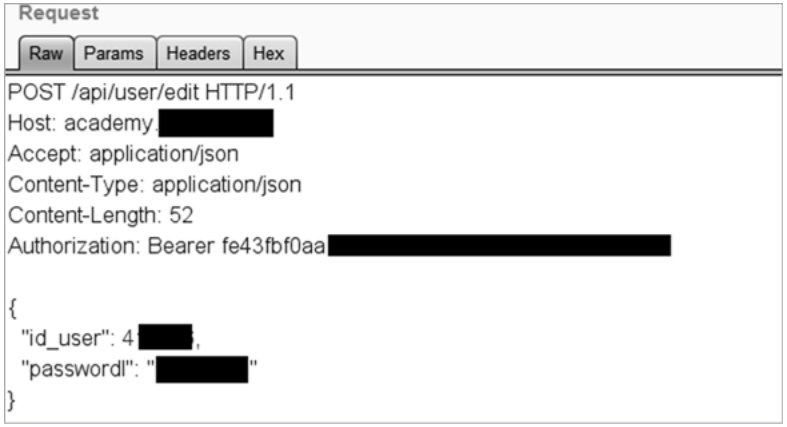

Figure 15-2: Omkar's successful API request to edit a user's account password

Several API vulnerabilities led to this exploitation. The API documentation disclosed sensitive information about how the API operated and how to manipulate user accounts. There is no business purpose to making this documentation available to the public; if it weren't available, an attacker would have likely moved on to the next target without stopping to investigate.

By thoroughly investigating the target, Bhagwat was able to discover a broken authentication vulnerability in the form of an exposed authorization Bearer token. Using the Bearer token and documentation, he then found a BFLA.

Lessons Learned

- Launch a thorough investigation of a web application when something piques your interest.
- API documentation is a gold mine of information; use it to your advantage.
- Combine your findings to discover new vulnerabilities.

Starbucks: The Breach That Never Was

> **Bug bounty hunter:** Sam Curry
> **Bounty:** $4,000

Curry is a security researcher and bug hunter. While participating in Starbucks' bug bounty program, he discovered and disclosed a vulnerability that prevented a breach of nearly 100 million personally identifiable information (PII) records belonging to Starbucks' customers. According to the Net Diligence breach calculator,

a PII data breach of this size could have cost Starbucks $100 million in regulatory fines, $225 million in crisis management costs, and $25 million in incident investigation costs. Even at a conservative estimate of $3.50 per record, a breach of that size could have resulted in a bill of around $350 million. Sam's finding was epic, to say the least.

On his blog at *https://samcurry.net*, Curry provides a play-by-play of his approach to hacking the Starbucks API. The first thing that caught his interest was the fact that the Starbucks gift card purchase process included API requests containing sensitive information to the endpoint */bff/proxy*:

```
POST /bff/proxy/orchestra/get-user HTTP/1.1
HOST: app.starbucks.com

{
"data":
"user": {
"exId": "77EFFC83-7EE9-4ECA-9849-A6A23BF1830F",
"firstName": "Sam",
"lastName": "Curry",
"email": "samwcurry@gmail.com",
"partnerNumber": null,
"birthDay": null,
"birthMonth": null,
"loyaltyProgram": null
}
}
```

As Curry explains on his blog, *bff* stands for "backend for frontend," meaning the application passes the request to another host to provide the functionality. In other words, Starbucks was using a proxy to transfer data between the external API and an internal API endpoint.

Curry attempted to probe this */bff/proxy/orchestra* endpoint but found it wouldn't transfer user input back to the internal API. However, he discovered a */bff/proxy/user:id* endpoint that did allow user input to make it beyond the proxy:

```
GET /bff/proxy/stream/v1/users/me/streamItems/..\ HTTP/1.1
Host: app.starbucks.com

{
"errors": [
```

```
{
"message": "Not Found",
"errorCode": 404
}]}
```

By using `..\` at the end of the path, Curry was attempting to traverse the current working directory and see what else he could access on the server. He continued to test for various directory traversal vulnerabilities until he sent the following:

```
GET /bff/proxy/stream/v1/me/streamItems/web\..\.\..\.\..\.\..\.\..\..\..\.\..\
```

This request resulted in a different error message:

```
"message": "Bad Request",
"errorCode": 400
```

This sudden change in an error request meant Curry was onto something. He used Burp Suite Intruder to brute-force various directories until he came across a Microsoft Graph instance using */search/v1/accounts*. Curry queried the Graph API and captured a proof of concept that demonstrated he had access to an internal customer database containing IDs, usernames, full names, emails, cities, addresses, and phone numbers.

Because he knew the syntax of the Microsoft Graph API, Curry found that he could include the query parameter `$count=true` to get a count of the number of entries, which came up to 99,356,059, just shy of 100 million.

Curry found this vulnerability by paying close attention to the API's responses and filtering results in Burp Suite, allowing him to find a unique status code of 400 among all the standard 404 errors. If the API provider hadn't disclosed this information, the response would have blended in with all the other 404 errors, and an attacker would likely have moved on to another target.

By combining the information disclosure and security misconfiguration, he was able to brute-force the internal directory structure and find the Microsoft Graph API. The additional BFLA vulnerability allowed Curry to use administrative functionality to perform user account queries.

Lessons Learned

- Pay close attention to subtle differences between API responses. Use Burp Suite Comparer or carefully compare requests and responses to identify potential weaknesses in an API.
- Investigate how the application or WAF handles fuzzing and directory traversal techniques.
- Leverage evasive techniques to bypass security controls.

An Instagram GraphQL BOLA

- **Bug bounty hunter:** Mayur Fartade
- **Bounty:** $30,000

In 2021, Fartade discovered a severe BOLA vulnerability in Instagram that allowed him to send POST requests to the GraphQL API located at */api/v1/ads/graphql/* to view the private posts, stories, and reels of other users.

The issue stemmed from a lack of authorization security controls for requests involving a user's media ID. To discover the media ID, you could use brute force or capture the ID through other means, such as social engineering or XSS. For example, Fartade used a POST request like the following:

```
POST /api/v1/ads/graphql HTTP/1.1
Host: i.instagram.com
Parameters:
doc_id=[REDACTED]&query_params={"query_params":{"access_token":"","id":"[MEDIA_ID]"}}
```

By targeting the `MEDIA_ID` parameter and providing a null value for `access_token`, Fartade was able to view the details of other users' private posts:

```
"data":{
"instagram_post_by_igid":{
"id":
"creation_time":1618732307,
"has_product_tags":false,
"has_product_mentions":false,
"instagram_media_id":
006",
"instagram_media_owner_id":"!
"instagram_actor": {
"instagram_actor_id":"!
"id":"1
```

```
},
"inline_insights_node":{
"state": null,
"metrics":null,
"error":null
},
"display_url":"https:\/\/scontent.cdninstagram.com\/VV/t51.29350-15\/
"instagram_media_type":"IMAGE",
"image":{
"height":640,
"width":360
},
"comment_count":
"like_count":
"save_count":
"ad_media": null,
"organic_instagram_media_id":"
--snip--
]
}
}
```

This BOLA allowed Fartade to make requests for information simply by specifying the media ID of a given Instagram post. Using this weakness, he was able to gain access to details such as likes, comments, and Facebook-linked pages of any user's private or archived posts.

Lessons Learned

- Make an effort to seek out GraphQL endpoints and apply the techniques covered in this book; the payout could be huge.
- When at first your attacks don't succeed, combine evasive techniques, such as by using null bytes with your attacks, and try again.
- Experiment with tokens to bypass authorization requirements.

Summary

This chapter used API breaches and bug bounty reports to demonstrate how you might be able to exploit common API vulnerabilities in real-world environments. Studying the tactics of adversaries and bug bounty hunters will help you expand your own hacking repertoire to better help secure the internet. These stories also

reveal how much low-hanging fruit is out there. By combining easy techniques, you can create an API hacking masterpiece.

Become familiar with the common API vulnerabilities, perform thorough analysis of endpoints, exploit the vulnerabilities you discover, report your findings, and bask in the glory of preventing the next great API data breach.