3

COMMON API VULNERABILITIES

Understanding common vulnerabilities will help you identify weaknesses when you're testing APIs. In this chapter, I cover most of the vulnerabilities included in the Open Web Application Security Project (OWASP) API Security Top 10 list, plus two other useful weaknesses: information disclosure and business logic flaws. I'll describe each vulnerability, its significance, and the techniques used to exploit it. In later chapters, you'll gain hands-on experience finding and exploiting many of these vulnerabilities.

OWASP API SECURITY TOP 10

OWASP is a nonprofit foundation that creates free content and tools aimed at securing web applications. Due to the increasing prevalence of API vulnerabilities, OWASP released the OWASP API Security Top 10, a list of the 10 most common API vulnerabilities, at the end of 2019. Check out the project, which was led by API security experts Inon Shkedy and Erez Yalon, at *https://owasp.org/www-project-api-security*. In Chapter 15, I will demonstrate how the vulnerabilities described in the OWASP API Security Top 10 have been exploited in major breaches and bug bounty findings. We'll also use several OWASP tools to attack APIs in Parts II and III of the book.

Information Disclosure

When an API and its supporting software share sensitive information with unprivileged users, the API has an *information disclosure* vulnerability. Information may be disclosed in API responses or public sources such as code repositories, search results, news, social media, the target's website, and public API directories.

Sensitive data can include any information that attackers can leverage to their advantage. For example, a site that is using the WordPress API may unknowingly be sharing user information with anyone who navigates to the API path */wp-json/wp/v2/users*, which returns all the WordPress usernames, or "slugs." For instance, take a look at the following request:

```
GET https://www.sitename.org/wp-json/wp/v2/users
```

It might return this data:

```
[{"id":1,"name":"Administrator", "slug":"admin"}],
{"id":2,"name":"Vincent Valentine", "slug":"Vincent"}]
```

These slugs can then be used in an attempt to log in as the disclosed users with a brute-force, credential-stuffing, or password-spraying attack. (Chapter 8 describes these attacks in detail.)

Another common information disclosure issue involves verbose messaging. Error messaging helps API consumers troubleshoot their interactions with an API and allows API providers to understand issues with their application. However, it can also reveal sensitive information about resources, users, and the API's underlying architecture (such as the version of the web server or database). For example, say you attempt to authenticate to an API and receive an error message such as "the provided user ID does not exist." Next, say you use another email and the error message changes to "incorrect password." This lets you know that you've provided a legitimate user ID for the API.

Finding user information is a great way to start gaining access to an API. The following information can also be leveraged in an attack: software packages, operating system information, system logs, and software bugs. Generally, any informa-

tion that can help us find more severe vulnerabilities or assist in exploitation can be considered an information disclosure vulnerability.

Often, you can gather the most information by interacting with an API endpoint and analyzing the response. API responses can reveal information within headers, parameters, and verbose errors. Other good sources of information are API documentation and resources gathered during reconnaissance. Chapter 6 covers many of the tools and techniques used for discovering API information disclosures.

Broken Object Level Authorization

One of the most prevalent vulnerabilities in APIs is *broken object level authorization (BOLA)*. BOLA vulnerabilities occur when an API provider allows an API consumer access to resources they are not authorized to access. If an API endpoint does not have object-level access controls, it won't perform checks to make sure users can only access their own resources. When these controls are missing, User A will be able to successfully request User B's resources.

APIs use some sort of value, such as names or numbers, to identify various objects. When we discover these object IDs, we should test to see if we can interact with the resources of other users when unauthenticated or authenticated as a different user. For instance, imagine that we are authorized to access only the user Cloud Strife. We would send an initial GET request to *https://bestgame.com/api/v3/users?id=5501* and receive the following response:

```
{
  "id": "5501",
  "first_name": "Cloud",
  "last_name": "Strife",
  "link": "https://www.bestgame.com/user/strife.buster.97",
  "name": "Cloud Strife",
  "dob": "1997-01-31",
  "username": "strife.buster.97"
}
```

This poses no problem since we are authorized to access Cloud's information. However, if we are able to access another user's information, there is a major authorization issue.

In this situation, we might check for these problems by using another identification number that is close to Cloud's ID of 5501. Say we are able to obtain information about another user by sending a request for *https://bestgame.com/api/v3/users?id=5502* and receiving the following response:

```
{
  "id": "5502",
  "first_name": "Zack",
  "last_name": "Fair",
  "link": " https://www.bestgame.com/user/shinra-number-1",
  "name": "Zack Fair",
  "dob": "2007-09-13",
```

```
  "username": "shinra-number-1"
}
```

In this case, Cloud has discovered a BOLA. Note that predictable object IDs don't necessarily indicate that you've found a BOLA. For the application to be vulnerable, it must fail to verify that a given user is only able to access their own resources.

In general, you can test for BOLAs by understanding how an API's resources are structured and attempting to access resources you shouldn't be able to access. By detecting patterns within API paths and parameters, you should be able to predict other potential resources. The bolded elements in the following API requests should catch your attention:

```
GET /api/resource/1
GET /user/account/find?user_id=15
POST /company/account/Apple/balance
POST /admin/pwreset/account/90
```

In these instances, you can probably guess other potential resources, like the following, by altering the bolded values:

```
GET /api/resource/3
GET /user/account/find?user_id=23
POST /company/account/Google/balance
POST /admin/pwreset/account/111
```

In these simple examples, you've performed an attack by merely replacing the bolded items with other numbers or words. If you can successfully access information you shouldn't be authorized to access, you have discovered a BOLA vulnerability.

In Chapter 9, I will demonstrate how you can easily fuzz parameters like *user_id=* in the URL path and sort through the results to determine if a BOLA vulnerability exists. In Chapter 10, we will focus on attacking authorization vulnerabilities like BOLA and BFLA (broken function level authorization, discussed later in this chapter). BOLA can be a low-hanging API vulnerability that you can easily discover using pattern recognition and then prodding it with a few requests. Other times, it can be quite complicated to discover due to the complexities of object IDs and the requests used to obtain another user's resources.

Broken User Authentication

Broken user authentication refers to *any* weakness within the API authentication process. These vulnerabilities typically occur when an API provider either doesn't implement an authentication protection mechanism or implements a mechanism incorrectly.

API authentication can be a complex system that includes several processes with a lot of room for failure. A couple decades ago, security expert Bruce Schneier said, "The future of digital systems is complexity, and complexity is the worst enemy of security." As we know from the six constraints of REST APIs discussed in Chapter 2, RESTful APIs are supposed to be stateless. In order to be stateless, the

provider shouldn't need to remember the consumer from one request to another. For this constraint to work, APIs often require users to undergo a registration process in order to obtain a unique token. Users can then include the token within requests to demonstrate that they're authorized to make such requests.

As a consequence, the registration process used to obtain an API token, the token handling, and the system that generates the token could all have their own sets of weaknesses. To determine if the *token generation process* is weak, for example, we could collect a sampling of tokens and analyze them for similarities. If the token generation process doesn't rely on a high level of randomness, or entropy, there is a chance we'll be able to create our own token or hijack someone else's.

Token handling could be the storage of tokens, the method of transmitting tokens across a network, the presence of hardcoded tokens, and so on. We might be able to detect hardcoded tokens in JavaScript source files or capture them as we analyze a web application. Once we've captured a token, we can use it to gain access to previously hidden endpoints or to bypass detection. If an API provider attributes an identity to a token, we would then take on the identity by hijacking the stolen token.

The other authentication processes that could have their own set of vulnerabilities include aspects of the *registration system*, such as the password reset and multifactor authentication features. For example, imagine a password reset feature requires you to provide an email address and a six-digit code to reset your password. Well, if the API allowed you to make as many requests as you wanted,

you'd only have to make one million requests in order to guess the code and reset any user's password. A four-digit code would require only 10,000 requests.

Also watch for the ability to access sensitive resources without being authenticated; API keys, tokens, and credentials used in URLs; a lack of rate-limit restrictions when authenticating; and verbose error messaging. For example, code committed to a GitHub repository could reveal a hardcoded admin API key:

```
"oauth_client":
[{"client_id": "12345-abcd",
"client_type": "admin",
"api_key": "AIzaSyDrbTFCeb5k0yPSfL2heqdF-N19XoLxdw"}]
```

Due to the stateless nature of REST APIs, a publicly exposed API key is the equivalent of discovering a username and password. By using an exposed API key, you'll assume the role associated with that key. In Chapter 6, we will use our reconnaissance skills to find exposed keys across the internet.

In Chapter 8, we will perform numerous attacks against API authentication, such as authentication bypass, brute-force attacks, credential stuffing, and a variety of attacks against tokens.

Excessive Data Exposure

Excessive data exposure is when an API endpoint responds with more information than is needed to fulfill a request. This often occurs when the provider expects the

API consumer to filter results; in other words, when a consumer requests specific information, the provider might respond with all sorts of information, assuming the consumer will then remove any data they don't need from the response. When this vulnerability is present, it can be the equivalent of asking someone for their name and having them respond with their name, date of birth, email address, phone number, and the identification of every other person they know.

For example, if an API consumer requests information for their user account and receives information about other user accounts as well, the API is exposing excessive data. Suppose I requested my own account information with the following request:

```
GET /api/v3/account?name=Cloud+Strife
```

Now say I got the following JSON in the response:

```
{
  "id": "5501",
  "first_name": "Cloud",
  "last_name": "Strife",
  "privilege": "user",
       "representative": [

        "name": "Don Corneo",
        "id": "2203"
        "email": "dcorn@gmail.com",
        "privilege": "super-admin"
```

```
        "admin": true
        "two_factor_auth": false,
        }
```

I requested a single user's account information, and the provider responded with information about the person who created my account, including the administrator's full name, the admin's ID number, and whether the admin had two-factor authentication enabled.

Excessive data exposure is one of those awesome API vulnerabilities that bypasses every security control in place to protect sensitive information and hands it all to an attacker on a silver platter simply because they used the API. All you need to do to detect excessive data exposure is test your target API endpoints and review the information sent in response.

Lack of Resources and Rate Limiting

One of the more important vulnerabilities to test for is *lack of resources and rate limiting*. Rate limiting plays an important role in the monetization and availability of APIs. Without limiting the number of requests consumers can make, an API provider's infrastructure could be overwhelmed by the requests. Too many requests without enough resources will lead to the provider's systems crashing and becoming unavailable—a *denial of service (DoS)* state.

Besides potentially DoS-ing an API, an attacker who bypasses rate limits can cause additional costs for the API provider. Many API providers monetize their APIs by

limiting requests and allowing paid customers to request more information. RapidAPI, for example, allows for 500 requests per month for free but 1,000 requests per month for paying customers. Some API providers also have infrastructure that automatically scales with the quantity of requests. In these cases, an unlimited number of requests would lead to a significant and easily preventable increase in infrastructure costs.

When testing an API that is supposed to have rate limiting, the first thing you should check is that rate limiting works, and you can do so by sending a barrage of requests to the API. If rate limiting is functioning, you should receive some sort of response informing you that you're no longer able to make additional requests, usually in the form of an HTTP 429 status code.

Once you are restricted from making additional requests, it's time to attempt to see how rate limiting is enforced. Can you bypass it by adding or removing a parameter, using a different client, or altering your IP address? Chapter 13 includes various measures for attempting to bypass rate limiting.

Broken Function Level Authorization

Broken function level authorization (BFLA) is a vulnerability where a user of one role or group is able to access the API functionality of another role or group. API providers will often have different roles for different types of accounts, such as public users, merchants, partners, administrators, and so on. A BFLA is present if you are able to use the functionality of another privilege level or group. In other words, BFLA can be a lateral move, where you use the functions of a similarly

privileged group, or it could be a privilege escalation, where you are able to use the functions of a more privileged group. Particularly interesting API functions to access include those that deal with sensitive information, resources that belong to another group, and administrative functionality such as user account management.

BFLA is similar to BOLA, except instead of an authorization problem involving accessing resources, it is an authorization problem for performing actions. For example, consider a vulnerable banking API. When a BOLA vulnerability is present in the API, you might be able to access the information of other accounts, such as payment histories, usernames, email addresses, and account numbers. If a BFLA vulnerability is present, you might be able to transfer money and actually update the account information. BOLA is about unauthorized access, whereas BFLA is about unauthorized actions.

If an API has different privilege levels or roles, it may use different endpoints to perform privileged actions. For example, a bank may use the */{user}/account/balance* endpoint for a user wishing to access their account information and the */admin/account/{user}* endpoint for an administrator wishing to access user account information. If the application does not have access controls implemented correctly, we'll be able to perform administrative actions, such as seeing a user's full account details, by simply making administrative requests.

An API won't always use administrative endpoints for administrative functionality. Instead, the functionality could be based on HTTP request methods such as GET, POST, PUT, and DELETE. If a provider doesn't restrict the HTTP methods a

consumer can use, simply making an unauthorized request with a different method could indicate a BFLA vulnerability.

When hunting for BFLA, look for any functionality you could use to your advantage, including altering user accounts, accessing user resources, and gaining access to restricted endpoints. For example, if an API gives partners the ability to add new users to the partner group but does not restrict this functionality to the specific group, any user could add themselves to any group. Moreover, if we're able to add ourselves to a group, there is a good chance we'll be able to access that group's resources.

The easiest way to discover BFLA is to find administrative API documentation and send requests as an unprivileged user that test admin functions and capabilities. *Figure 3-1* shows the public Cisco Webex Admin API documentation, which provides a handy list of actions to attempt if you were testing Cisco Webex.

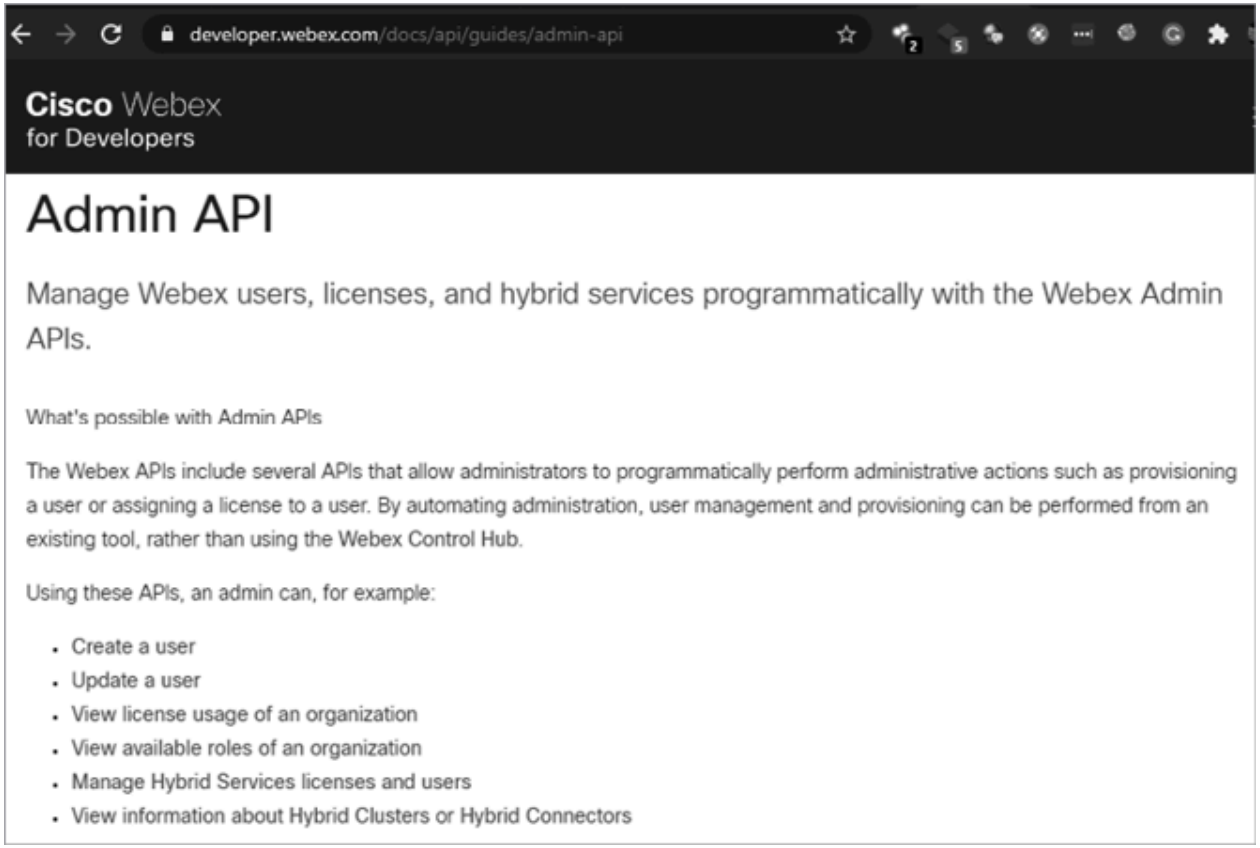

Figure 3-1: The Cisco Webex Admin API documentation

As an unprivileged user, make requests included in the admin section, such as attempting to create users, update user accounts, and so on. If access controls are in place, you'll likely receive an HTTP 401 Unauthorized or 403 Forbidden response. However, if you're able to make successful requests, you have discovered a BFLA vulnerability.

If API documentation for privileged actions is not available, you will need to discover or reverse engineer the endpoints used to perform privileged actions before

testing them; more on this in Chapter 7. Once you've found administrative endpoints, you can begin making requests.

Mass Assignment

Mass assignment occurs when an API consumer includes more parameters in their requests than the application intended and the application adds these parameters to code variables or internal objects. In this situation, a consumer may be able to edit object properties or escalate privileges.

For example, an application might have account update functionality that the user should use only to update their username, password, and address. If the consumer can include other parameters in a request related to their account, such as the account privilege level or sensitive information like account balances, and the application accepts those parameters without checking them against a whitelist of permitted actions, the consumer could take advantage of this weakness to change these values.

Imagine an API is called to create an account with parameters for `"User"` and `"Password"`:

```
{
"User": "scuttleph1sh",
"Password": "GreatPassword123"
}
```

While reading the API documentation regarding the account creation process, suppose you discover that there is an additional key, `"isAdmin"`, that consumers can use to become administrators. You could use a tool like Postman or Burp Suite to add the attribute to a request and set the value to `true`:

```
{
"User": "scuttleph1sh",
"Password": "GreatPassword123",
"isAdmin": true
}
```

If the API does not sanitize the request input, it is vulnerable to mass assignment, and you could use the updated request to create an admin account. On the back-end, the vulnerable web app will add the key/value attribute, `{"isAdmin":"true"}`, to the user object and make the user the equivalent of an administrator.

You can discover mass assignment vulnerabilities by finding interesting parameters in API documentation and then adding those parameters to a request. Look for parameters involved in user account properties, critical functions, and administrative actions. Intercepting API requests and responses could also reveal parameters worthy of testing. Additionally, you can guess parameters or fuzz them in API requests. (Chapter 9 describes the art of fuzzing.)

Security Misconfigurations

Security misconfigurations include all the mistakes developers could make within the supporting security configurations of an API. If a security misconfiguration is severe enough, it can lead to sensitive information exposure or a complete system takeover. For example, if the API's supporting security configuration reveals an unpatched vulnerability, there is a chance that an attacker could leverage a published exploit to easily "pwn" the API and its system.

Security misconfigurations are really a set of weaknesses that includes misconfigured headers, misconfigured transit encryption, the use of default accounts, the acceptance of unnecessary HTTP methods, a lack of input sanitization, and verbose error messaging.

A *lack of input sanitization* can allow attackers to upload malicious payloads to the server. APIs often play a key role in automating processes, so imagine being able to upload payloads that the server automatically processes into a format that could be remotely executed or executed by an unsuspecting end user. For example, if an upload endpoint was used to pass uploaded files to a web directory, it could allow the upload of a script. Navigating to the URL where the file is located could launch the script, resulting in direct shell access to the web server. Additionally, lack of input sanitization can lead to unexpected behavior on the part of the application. In Part III, we will fuzz API inputs in attempts to discover vulnerabilities such as security misconfigurations, improper assets management, and injection weaknesses.

API providers use *headers* to provide the consumer with instructions for handling the response and security requirements. Misconfigured headers can result in sensitive information disclosure, downgrade attacks, and cross-site scripting attacks. Many API providers will use additional services alongside their API to enhance API-related metrics or to improve security. It is fairly common for those additional services to add headers to requests for metrics and perhaps serve as some level of assurance to the consumer. For example, take the following response:

```
HTTP/ 200 OK
--snip--
X-Powered-By: VulnService 1.11
X-XSS-Protection: 0
X-Response-Time: 566.43
```

The `X-Powered-By` header reveals backend technology. Headers like this one will often advertise the exact supporting service and its version. You could use information like this to search for exploits published for that version of software.

`X-XSS-Protection` is exactly what it looks like: a header meant to prevent cross-site scripting (XSS) attacks. XSS is a common type of injection vulnerability where an attacker can insert scripts into a web page and trick end users into clicking malicious links. We will cover XSS and cross-API scripting (XAS) in Chapter 12. An `X-XSS-Protection` value of `0` indicates no protections are in place, and a value of `1` indicates that protection is turned on. This header, and others like it, clearly reveals whether a security control is in place.

The `X-Response-Time` header is middleware that provides usage metrics. In the previous example, its value represents 566.43 milliseconds. However, if the API isn't configured properly, this header can function as a side channel used to reveal existing resources. If the `X-Response-Time` header has a consistent response time for nonexistent records, for example, but increases its response time for certain other records, this could be an indication that those records exist. Here's an example:

```
HTTP/UserA 404 Not Found
--snip--
X-Response-Time: 25.5

HTTP/UserB 404 Not Found
--snip--
X-Response-Time: 25.5

HTTP/UserC 404 Not Found
--snip--
X-Response-Time: 510.00
```

In this case, UserC has a response time value that is 20 times the response time of the other resources. With this small sample size, it is hard to definitively conclude that UserC exists. However, imagine you have a sample of hundreds or thousands of requests and know the average `X-Response-Time` values for certain existing and nonexistent resources. Say, for instance, you know that a bogus account like */user/account/thisdefinitelydoesnotexist876* has an average `X-Response-Time` of 25.5 ms. You also know that your existing account */user/account/1021* receives an

`X-Response-Time` of 510.00. If you then sent requests brute-forcing all account numbers from 1000 to 2000, you could review the results and see which account numbers resulted in drastically increased response times.

Any API providing sensitive information to consumers should use Transport Layer Security (TLS) to encrypt the data. Even if the API is only provided internally, privately, or at a partner level, using TLS, the protocol that encrypts HTTPS traffic, is one of the most basic ways to ensure that API requests and responses are protected when being passed across a network. Misconfigured or missing transit encryption can cause API users to pass sensitive API information in cleartext across networks, in which case an attacker could capture the responses and requests with a man-in-the-middle (MITM) attack and read them plainly. The attacker would need to have access to the same network as the person they were attacking and then intercept the network traffic with a network protocol analyzer such as Wireshark to see the information being communicated between the consumer and the provider.

When a service uses a *default account and credentials* and the defaults are known, an attacker can use those credentials to assume the role of that account. This could allow them to gain access to sensitive information or administrative functionality, potentially leading to a compromise of the supporting systems.

Lastly, if an API provider allows *unnecessary HTTP methods*, there is an increased risk that the application won't handle these methods properly or will result in sensitive information disclosure.

You can detect several of these security misconfigurations with web application vulnerability scanners such as Nessus, Qualys, OWASP ZAP, and Nikto. These scanners will automatically check the web server version information, headers, cookies, transit encryption configuration, and parameters to see if expected security measures are missing. You can also check for these security misconfigurations manually, if you know what you are looking for, by inspecting the headers, SSL certificate, cookies, and parameters.

Injections

Injection flaws exist when a request is passed to the API's supporting infrastructure and the API provider doesn't filter the input to remove unwanted characters (a process known as *input sanitization*). As a result, the infrastructure might treat data from the request as code and run it. When this sort of flaw is present, you'll be able to conduct injection attacks such as SQL injection, NoSQL injection, and system command injection.

In each of these injection attacks, the API delivers your unsanitized payload directly to the operating system running the application or its database. As a result, if you send a payload containing SQL commands to a vulnerable API that uses a SQL database, the API will pass the commands to the database, which will process and perform the commands. The same will happen with vulnerable NoSQL databases and affected systems.

Verbose error messaging, HTTP response codes, and unexpected API behavior can all be clues that you may have discovered an injection flaw. Say, for example, you

were to send `OR 1=0--` as an address in an account registration process. The API may pass that payload directly to the backend SQL database, where the `OR 1=0` statement would fail (because 1 does not equal 0), causing some SQL error:

```
POST /api/v1/register HTTP 1.1
Host: example.com
--snip--
{
"Fname": "hAPI",
"Lname": "Hacker",
"Address": "' OR 1=0--",
}
```

An error in the backend database could show up as a response to the consumer. In this case, you might receive a response like "Error: You have an error in your SQL syntax. . . ." Any response directly from a database or the supporting system is a clear indicator that there is an injection vulnerability.

Injection vulnerabilities are often complemented by other vulnerabilities such as poor input sanitization. In the following example, you can see a code injection attack that uses an API GET request to take advantage of a weak query parameter. In this case, the weak query parameter passes any data in the query portion of the request directly to the underlying system, without sanitizing it first:

```
GET http://10.10.78.181:5000/api/v1/resources/books?show=/etc/passwd
```

The following response body shows that the API endpoint has been manipulated into displaying the host's */etc/passwd* file, revealing users on the system:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Finding injection flaws requires diligently testing API endpoints, paying attention to how the API responds, and then crafting requests that attempt to manipulate the backend systems. Like directory traversal attacks, injection attacks have been around for decades, so there are many standard security controls to protect API providers from them. I will demonstrate various methods for performing injection attacks, encoding traffic, and bypassing standard controls in Chapters 12 and 13.

Improper Assets Management

Improper assets management takes place when an organization exposes APIs that are either retired or still in development. As with any software, old API versions are more likely to contain vulnerabilities because they are no longer being

patched and upgraded. Likewise, APIs that are still being developed are typically not as secure as their production API counterparts.

Improper assets management can lead to other vulnerabilities, such as excessive data exposure, information disclosure, mass assignment, improper rate limiting, and API injection. For attackers, this means that discovering an improper assets management vulnerability is only the first step toward further exploitation of an API.

You can discover improper assets management by paying close attention to outdated API documentation, changelogs, and version history on repositories. For example, if an organization's API documentation has not been updated along with the API's endpoints, it could contain references to portions of the API that are no longer supported. Organizations often include versioning information in their endpoint names to distinguish between older and newer versions, such as */v1/*, */v2/*, */v3/*, and so on. APIs still in development often use paths such as */alpha/*, */beta/*, */test/*, */uat/*, and */demo/*. If you know that an API is now using *apiv3.org/admin* but part of the API documentation refers to *apiv1.org/admin*, you could try testing different endpoints to see if *apiv1* or *apiv2* is still active. Additionally, the organization's changelog may disclose the reasons why *v1* was updated or retired. If you have access to *v1*, you can test for those weaknesses.

Outside of using documentation, you can discover improper assets management vulnerabilities through the use of guessing, fuzzing, or brute-force requests. Watch for patterns in the API documentation or path-naming scheme, and then make requests based on your assumptions.

Business Logic Vulnerabilities

Business logic vulnerabilities (also known as *business logic flaws*, or *BLFs*) are intended features of an application that attackers can use maliciously. For example, if an API has an upload feature that doesn't validate encoded payloads, a user could upload any file as long as it was encoded. This would allow end users to upload and execute arbitrary code, including malicious payloads.

Vulnerabilities of this sort normally come about from an assumption that API consumers will follow directions, be trustworthy, or only use the API in a certain way. In those cases, the organization essentially depends on trust as a security control by expecting the consumer to act benevolently. Unfortunately, even good-natured API consumers make mistakes that could lead to a compromise of the application.

The Experian partner API leak, in early 2021, was a great example of an API trust failure. A certain Experian partner was authorized to use Experian's API to perform credit checks, but the partner added the API's credit check functionality to their web application and inadvertently exposed all partner-level requests to users. A request could be intercepted when using the partner's web application, and if it included a name and address, the Experian API would respond with the individual's credit score and credit risk factors. One of the leading causes of this business logic vulnerability was that Experian trusted the partner not to expose the API.

Another problem with trust is that credentials, such as API keys, tokens, and passwords, are constantly being stolen and leaked. When a trusted consumer's cre-

dentials are stolen, the consumer can become a wolf in sheep's clothing and wreak havoc. Without strong technical controls in place, business logic vulnerabilities can often have the most significant impact, leading to exploitation and compromise.

You can search API documentation for telltale signs of business logic vulnerabilities. Statements like the following should illuminate the lightbulb above your head:

> "Only use feature X to perform function Y."
> "Do not do X with endpoint Y."
> "Only admins should perform request X."

These statements may indicate that the API provider is trusting that you won't do any of the discouraged actions, as instructed. When you attack their API, make sure to disobey such requests to test for the presence of security controls.

Another business logic vulnerability comes about when developers assume that consumers will exclusively use a browser to interact with the web application and won't capture API requests that take place behind the scenes. All it takes to exploit this sort of weakness is to intercept requests with a tool like Burp Suite Proxy or Postman and then alter the API request before it is sent to the provider. This could allow you to capture shared API keys or use parameters that could negatively impact the security of the application.

As an example, consider a web application authentication portal that a user would normally employ to authenticate to their account. Say the web application

issued the following API request:

```
POST /api/v1/login HTTP 1.1
Host: example.com
--snip--
UserId=hapihacker&password=arealpassword!&MFA=true
```

There is a chance that we could bypass multifactor authentication by simply altering the parameter `MFA` to `false`.

Testing for business logic flaws can be challenging because each business is unique. Automated scanners will have a difficult time detecting these issues, as the flaws are part of the API's intended use. You must understand how the business and API operate and then consider how you could use these features to your advantage. Study the application's business logic with an adversarial mindset, and try breaking any assumptions that have been made.

Summary

In this chapter, I covered common API vulnerabilities. It is important to become familiar with these vulnerabilities so that you can easily recognize them, take advantage of them during an engagement, and report them back to the organization to prevent the criminals from dragging your client into the headlines.

Now that you are familiar with web applications, APIs, and their weaknesses, it is time to prepare your hacking machine and get your hands busy on the keyboard.