8

ATTACKING AUTHENTICATION

When it comes to testing authentication, you'll find that many of the flaws that have plagued web applications for decades have been ported over to APIs: bad passwords and password requirements, default credentials, verbose error messaging, and bad password reset processes.

In addition, several weaknesses are much more commonly found in APIs than traditional web apps. Broken API authentication comes in many forms. You might encounter a lack of authentication altogether, a lack of rate limiting applied to authentication attempts, the use of a single token or key for all requests, tokens created with insufficient entropy, and several JSON Web Token (JWT) configuration weaknesses.

This chapter will guide you through classic authentication attacks like brute-force attacks and password spraying, and then we'll cover API-specific token attacks, such as token forgery and JWT attacks. Generally, these attacks share the common goal of gaining unauthorized access, whether this means going from a state of no access to a state of unauthorized access, obtaining access to the resources of other users, or going from a state of limited API access to one of privileged access.

Classic Authentication Attacks

In Chapter 2, we covered the simplest form of authentication used in APIs: basic authentication. To authenticate using this method, the consumer issues a request containing a username and password. As we know, RESTful APIs do not maintain state, so if the API uses basic authentication across the API, a username and password would have to be issued with every request. Thus, providers typically use basic authentication only as part of a registration process. Then, after users have

successfully authenticated, the provider issues an API key or token. The provider then checks that the username and password match the authentication information stored. If the credentials match, the provider issues a successful response. If they don't match, the API may issue one of several responses. The provider may just send a generic response for all incorrect authentication attempts: "Incorrect username or password." This tells us the least amount of information, but sometimes providers will tilt the scales toward consumer convenience and provide us with more useful information. The provider could specifically tell us that a username does not exist. Then we will have a response we can use to help us discover and validate usernames.

Password Brute-Force Attacks

One of the more straightforward methods for gaining access to an API is performing a brute-force attack. Brute-forcing an API's authentication is not very different from any other brute-force attack, except you'll send the request to an API endpoint, the payload will often be in JSON, and the authentication values may be base64 encoded. Brute-force attacks are loud, often time-consuming, and brutish, but if an API lacks security controls to prevent brute-force attacks, we should not shy away from using this to our advantage.

One of the best ways to fine-tune your brute-force attack is to generate passwords specific to your target. To do this, you could leverage the information revealed in an excessive data exposure vulnerability, like the one you found in Lab #4, to compile a username and password list. The excess data could reveal technical details about the user's account, such as whether the user was using multifactor authentication, whether they had a default password, and whether the account has been activated. If the excess data involved information about the user, you could feed it to tools that can generate large, targeted password lists for brute-force attacks. For more information about creating targeted password lists, check out the Mentalist app (*https://github.com/sc0tfree/mentalist*) or the Common User Passwords Profiler (*https://github.com/Mebus/cupp*).

To actually perform the brute-force attack once you have a suitable wordlist, you can use tools such as Burp Suite's brute forcer or Wfuzz, introduced in Chapter 4. The following example uses Wfuzz with an old, well-known password list, *rockyou.txt*:

```
$ wfuzz -d '{"email":"a@email.com","password":"FUZZ"}' --hc 405 -H 'Content-Type: application/json' -z file,/home/hapihacker
==========================================================================
```

```
ID           Response   Lines    Word       Chars       Payload
=====================================================================
000000007:   200        0 L      1 W        225 Ch      "Password1!"
000000005:   400        0 L      34 W       474 Ch      "win"
```

The `-d` option allows you to fuzz content that is sent in the body of a POST request. The curly brackets that follow contain the POST request body. To discover the request format used in this example, I attempted to authenticate to a web application using a browser, and then I captured the authentication attempt and replicated its structure here. In this instance, the web app issues a POST request with the parameters `"email"` and `"password"`. The structure of this body will change for each API. In this example, you can see that we've specified a known email and used the `FUZZ` parameter as the password.

The `--hc` option hides responses with certain response codes. This is useful if you often receive the same status code, word length, and character count in many requests. If you know what a typical failure response looks like for your target, there is no need to see hundreds or thousands of that same response. The `–hc` option helps you filter out the responses you don't want to see.

In the tested instance, the typical failed request results in a 405 status code, but this may also differ with each API. Next, the `-H` option lets you add a header to the request. Some API providers may issue an HTTP 415 Unsupported Media Type error code if you don't include the `Content-Type:application/json` header when sending JSON data in the request body.

Once your request has been sent, you can review the results in the command line. If your `–hc` Wfuzz option has worked out, your results should be fairly easy to read. Otherwise, status codes in the 200s and 300s should be good indicators that you have successfully brute-forced credentials.

Password Reset and Multifactor Authentication Brute-Force Attacks

While you can apply brute-force techniques directly to the authentication requests, you can also use them against password reset and multifactor authentication (MFA) functionality. If a password reset process includes security questions and does not apply rate limiting to requests, we can target it in such an attack.

Like GUI web applications, APIs often use SMS recovery codes or one-time passwords (OTPs) in order to verify the identity of a user who wants to reset their

password. Additionally, a provider may deploy MFA to successful authentication attempts, so you'll have to bypass that process to gain access to the account. On the backend, an API often implements this functionality using a service that sends a four- to six-digit code to the phone number or email associated with the account. If we're not stopped by rate limiting, we should be able to brute-force these codes to gain access to the targeted account.

Begin by capturing a request for the relevant process, such as a password reset process. In the following request, you can see that the consumer includes an OTP in the request body, along with the username and new password. Thus, to reset a user's password, we'll need to guess the OTP.

```
POST /identity/api/auth/v3/check-otp HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (x11; Linux x86_64; rv: 78.0) Gecko/20100101
Accept: */*
Accept -Language: en-US, en;q=0.5
Accept-Encoding: gzip,deflate
Referer: http://192.168.195.130:8888/forgot-password
Content-Type: application/json
Origin: http://192.168.195.130:8888
Content-Length: 62
Connection: close

{
"email":"a@email.com",
"otp":"1234",
"password": "Newpassword"
}
```

In this example, we'll leverage the brute forcer payload type in Burp Suite, but you could configure and run an equivalent attack using Wfuzz with brute-force options. Once you've captured a password reset request in Burp Suite, highlight the OTP and add the attack position markers discussed in Chapter 4 to turn the value into a variable. Next, select the **Payloads** tab and set the payload type to **brute forcer** (see *Figure 8-1*).

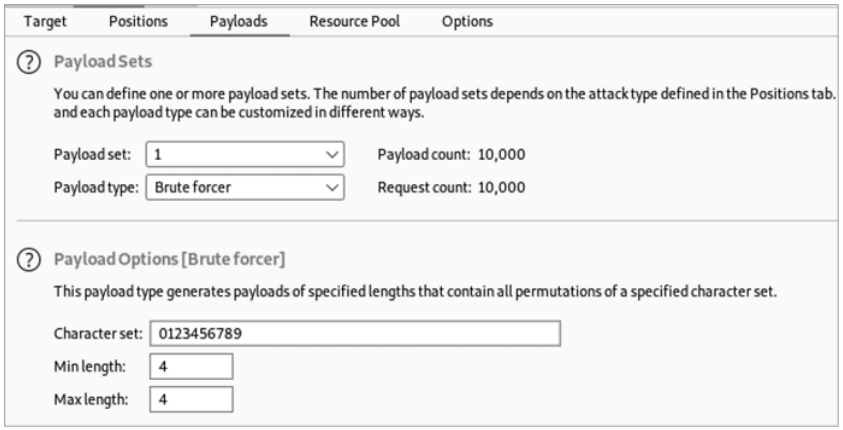

Figure 8-1: Configuring Burp Suite Intruder with the brute forcer payload type set

If you've configured your payload settings correctly, they should match those in *Figure 8-1*. In the character set field, only include numbers and characters used for the OTP. In its verbose error messaging, the API provider may indicate what values it expects. You can often test this by initiating a password reset of your own account and checking to see what the OTP consists of. For example, if the API uses a four-digit numeric code, add the numbers 0 to 9 to the character set. Then set the minimum and maximum length of the code to **4**.

Brute-forcing the password reset code is definitely worth a try. However, many web applications will both enforce rate limiting and limit the number of times you can guess the OTP. If rate limiting is holding you back, perhaps one of the evasion techniques in Chapter 13 could be of some use.

Password Spraying

Many security controls could prevent you from successfully brute-forcing an API's authentication. A technique called *password spraying* can evade many of these controls by combining a long list of users with a short list of targeted passwords. Let's say you know that an API authentication process has a lockout policy in place and will only allow 10 login attempts. You could craft a list of the nine most likely passwords (one less password than the limit) and use these to attempt to log in to many user accounts.

When you're password spraying, large and outdated wordlists like *rockyou.txt* won't work. There are way too many unlikely passwords in such a file to have any success. Instead, craft a short list of likely passwords, taking into account the con-

straints of the API provider's password policy, which you can discover during reconnaissance. Most password policies likely require a minimum character length, upper- and lowercase letters, and perhaps a number or special character.

Try mixing your password-spraying list with two types of *path of small-resistance (POS)* passwords, or passwords that are simple enough to guess but complex enough to meet basic password requirements (generally a minimum of eight characters, a symbol, upper- and lowercase letters, and a number). The first type includes obvious passwords like QWER!@#$, Password1!, and the formula *Season+Year+Symbol* (such as Winter2021!, Spring2021?, Fall2021!, and Autumn2021?). The second type includes more advanced passwords that relate directly to the target, often including a capitalized letter, a number, a detail about the organization, and a symbol. Here is a short password-spraying list I might generate if I were attacking an endpoint for Twitter employees:

```
Winter2021!
Spring2021!
QWER!@#$
Password1!
March212006!
July152006!
Twitter@2022
JPD1976!
Dorsey@2021
```

The key to password spraying is to maximize your user list. The more usernames you include, the higher your odds of gaining access. Build a user list during your reconnaissance efforts or by discovering excessive data exposure vulnerabilities.

In Burp Suite's Intruder, you can set up this attack in a similar manner to the standard brute-force attack, except you'll use both a list of users and a list of passwords. Choose the cluster bomb attack type and set the attack positions around the username and password, as shown in *Figure 8-2*.

Figure 8-2: A credential-spraying attack using Intruder

Notice that the first attack position is set to replace the username in front of *@email.com*, which you can do if you'll only be testing for users within a specific email domain.

Next, add the list of collected users as the first payload set and a short list of passwords as your second payload set. Once your payloads are configured as in *Figure 8-3*, you're ready to perform a password-spraying attack.

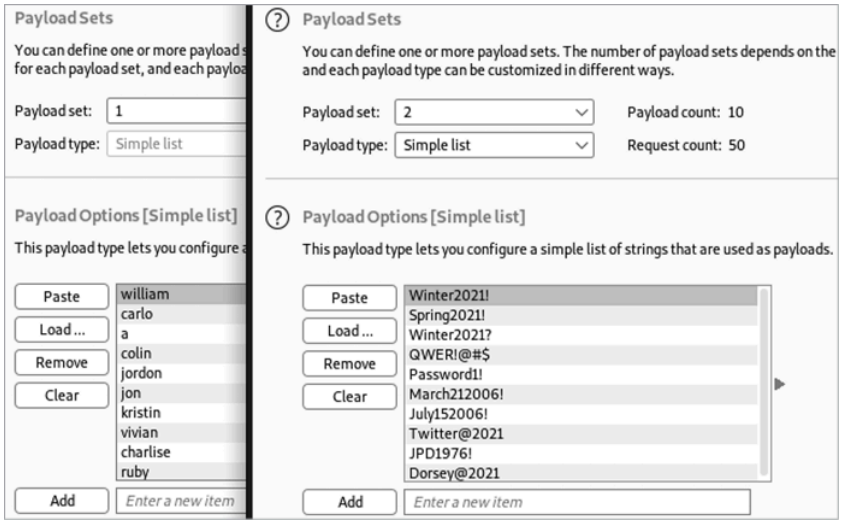

Figure 8-3: Burp Suite Intruder example payloads for a cluster bomb attack

When you're analyzing the results, it helps if you have an idea of what a standard successful login looks like. If you're unsure, search for anomalies in the lengths and response codes returned. Most web applications respond to successful login results with an HTTP status code in the 200s or 300s. In *Figure 8-4*, you can see a

successful password-spraying attempt that has two anomalous features: a status code of 200 and a response length of 682.

Request	Payload	Status	Error	Timeout	Length
5	Password1!	200			682
0		500			479
1	Winter2021!	500			479
2	Spring2021!	500			479
3	Winter2021?	500			479
4	QWER!@#$	500			479
6	March212006!	500			479
7	July152006!	500			479
8	Twitter@2021	500			479
9	JPD1976!	500			479
10	Dorsey@2021	500			479

Figure 8-4: A successful password-spraying attack using Intruder

To help spot anomalies using Intruder, you can sort the results by status code or response length.

Including Base64 Authentication in Brute-Force Attacks

Some APIs will base64-encode authentication payloads sent in an API request. There are many reasons to do this, but it's important to know that security is not one of them. You can easily bypass this minor inconvenience.

If you test an authentication attempt and notice that an API is encoding to base64, it is likely making a comparison to base64-encoded credentials on the backend. This means you should adjust your fuzzing attacks to include base64 payloads using Burp Suite Intruder, which can both encode and decode base64 values. For example, the password and email values in *Figure 8-5* are base64 encoded. You can decode them by highlighting the payload, right-clicking, and selecting **Base64-decode** (or the shortcut CTRL-SHIFT-B). This will reveal the payload so that you can see how it is formatted.

To perform, say, a password-spraying attack using base64 encoding, begin by selecting the attack positions. In this case, we'll select the base64-encoded password from the request in *Figure 8-5*. Next, add the payload set; we'll use the passwords listed in the previous section.

Now, in order to encode each password before it is sent in a request, we must use a payload-processing rule. Under the Payloads tab is an option to add such a rule. Select **Add ▸ Encoded ▸ Base64-encode** and then click **OK**. Your payload-processing window should look like *Figure 8-6*.

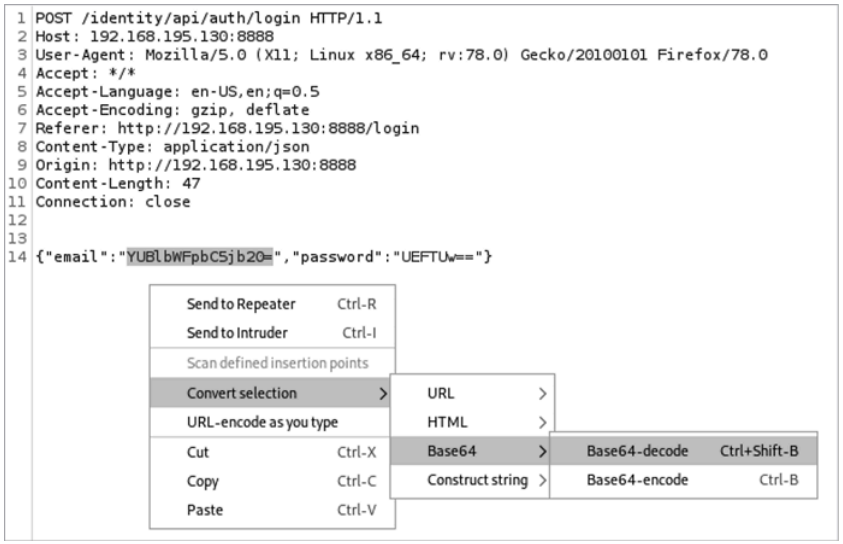

Figure 8-5: Decoding base64 using Burp Suite Intruder

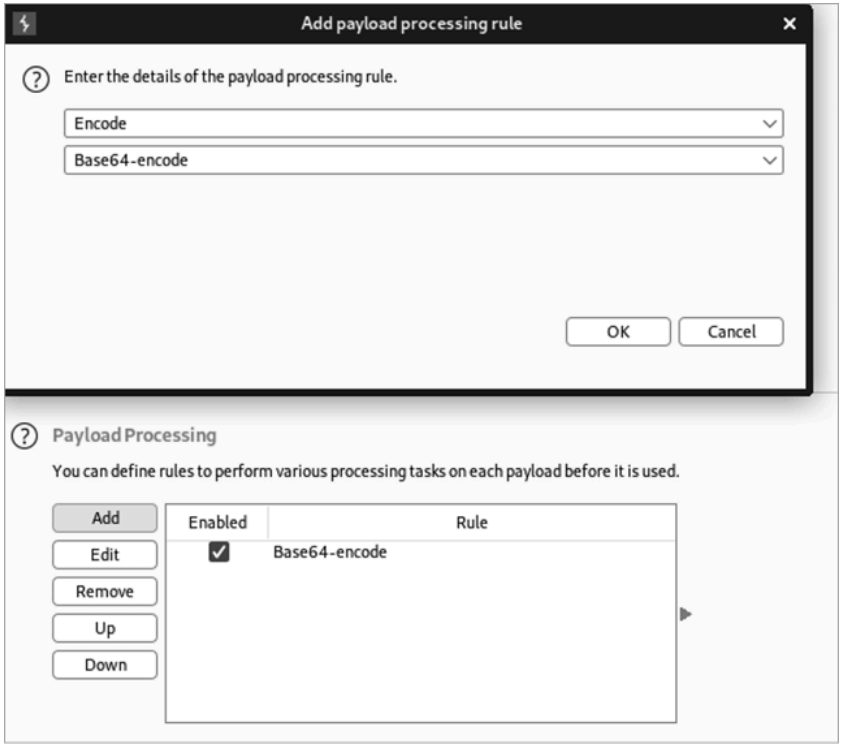

Figure 8-6: Adding a payload-processing rule to Burp Suite Intruder

Now your base64-encoded password-spraying attack is ready to launch.

Forging Tokens

When implemented correctly, tokens can be an excellent way for APIs to authenticate users and authorize them to access their resources. However, if anything goes wrong when generating, processing, or handling tokens, they'll become our keys to the kingdom.

The problem with tokens is that they can be stolen, leaked, and forged. We've already covered how to steal and find leaked tokens in Chapter 6. In this section, I'll guide you through the process of forging your own tokens when weaknesses are present in the token generation process. This requires first analyzing how predictable an API provider's token generation process is. If we can discover any patterns in the tokens being provided, we may be able to forge our own or hijack another user's tokens.

APIs will often use tokens as an authorization method. A consumer may have to initially authenticate using a username and password combination, but then the provider will generate a token and give that token to the consumer to use with their API requests. If the token generation process is flawed, we will be able to analyze the tokens, hijack other user tokens, and then use them to access the resources and additional API functionality of the affected users.

Burp Suite's Sequencer provides two methods for token analysis: manually analyzing tokens provided in a text file and performing a live capture to automatically generate tokens. I will guide you through both processes.

Manual Load Analysis

To perform a manual load analysis, select the **Sequencer** module and choose the **Manual Load** tab. Click **Load** and provide the list of tokens you want to analyze. The more tokens you have in your sample, the better the results will be. Sequencer requires a minimum of 100 tokens to perform a basic analysis, which includes a *bit-level* analysis, or an automated analysis of the token converted to sets of bits. These sets of bits are then put through a series of tests involving compression, correlation, and spectral testing, as well as four tests based on the Federal Information Processing Standard (FIPS) 140-2 security requirements.

NOTE

If you would like to follow along with the examples in this section, generate your own tokens or use the bad tokens hosted on the Hacking-APIs GitHub repo (https://github.com/hAPI-hacker/Hacking-APIs).

A full analysis will also include *character-level* analysis, a series of tests performed on each character in the given position in the original form of the tokens. The tokens are then put through a character count analysis and a character transition analysis, two tests that analyze how characters are distributed within a token and the differences between tokens. To perform a full analysis, Sequencer could require thousands of tokens, depending on the size and complexity of each individual token.

Once your tokens are loaded, you should see the total number of tokens loaded, the shortest token, and the longest token, as shown in *Figure 8-7*.

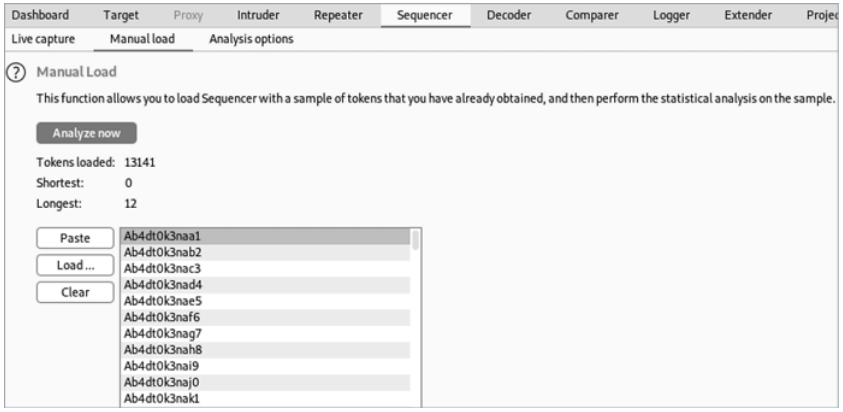

Figure 8-7: Manually loaded tokens in Burp Suite Sequencer

Now you can begin the analysis by clicking **Analyze Now**. Burp Suite should then generate a report (see *Figure 8-8*).

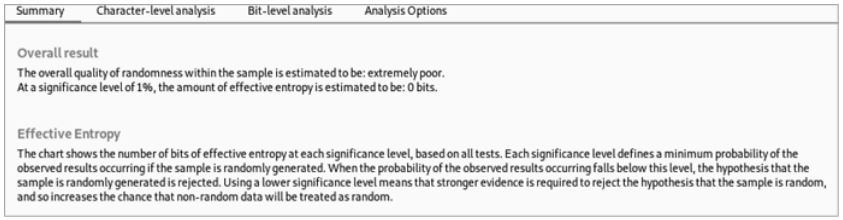

Figure 8-8: The Summary tab of the token analysis report provided by Sequencer

The token analysis report begins with a summary of the findings. The overall results include the quality of randomness within the token sample. In *Figure 8-8*, you can see that the quality of randomness was extremely poor, indicating that we'll likely be able to brute-force other existing tokens.

To minimize the effort required to brute-force tokens, we'll want to determine if there are parts of the token that do not change and other parts that often change. Use the character position analysis to determine which characters should be brute-forced (see *Figure 8-9*). You can find this feature under Character Set within the Character-Level Analysis tab.

As you can see, the token character positions do not change all that much, with the exception of the final three characters; the string `Ab4dt0k3n` remains the same throughout the sampling. Now we know we should perform a brute force of only the final three characters and leave the remainder of the token untouched.

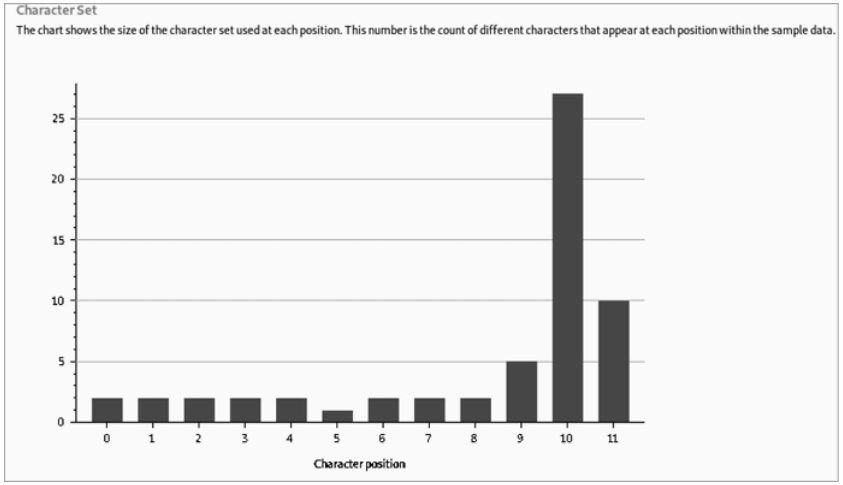

Figure 8-9: The character position chart found within Sequencer's character-level analysis

Live Token Capture Analysis

Burp Suite's Sequencer can automatically ask an API provider to generate 20,000 tokens for analysis. To do this, we simply intercept the provider's token generation process and then configure Sequencer. Burp Suite will repeat the token generation process up to 20,000 times to analyze the tokens for similarities.

In Burp Suite, intercept the request that initiates the token generation process. Select **Action** (or right-click the request) and then forward it to Sequencer. Within Sequencer, make sure you have the live capture tab selected, and under **Token Location Within Response**, select the **Configure for the Custom Location** option. As shown in *Figure 8-10*, highlight the generated token and click **OK**.

Select **Start Live Capture**. Burp Sequencer will now begin capturing tokens for analysis. If you select the Auto analyze checkbox, Sequencer will show the effective entropy results at different milestones.

In addition to performing an entropy analysis, Burp Suite will provide you with a large collection of tokens, which could be useful for evading security controls (a topic we explore in Chapter 13). If an API doesn't invalidate the tokens once new ones are created and the security controls use tokens as the method of identity, you now have up to 20,000 identities to help you avoid detection.

If there are token character positions with low entropy, you can attempt a brute-force attack against those character positions. Reviewing tokens with low entropy

could reveal certain patterns you could take advantage of. For example, if you noticed that characters in certain positions only contained lowercase letters, or a certain range of numbers, you'll be able to enhance your brute-force attacks by minimizing the number of request attempts.

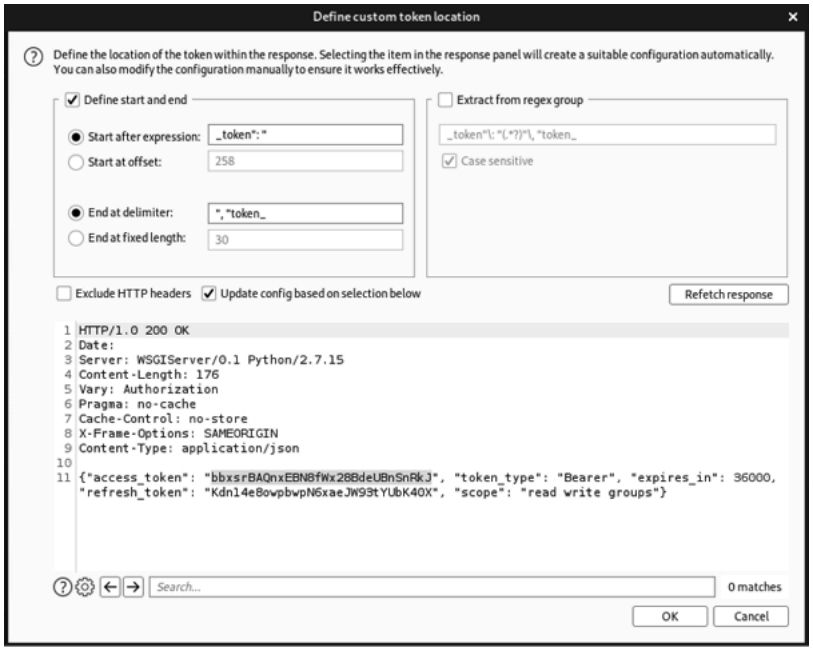

Figure 8-10: The API provider's token response selected for analysis

Brute-Forcing Predictable Tokens

Let's return to the bad tokens discovered during manual load analysis (whose final three characters are the only ones that change) and brute-force possible letter and number combinations to find other valid tokens. Once we've discovered valid tokens, we can test our access to the API and find out what we're authorized to do.

When you're brute-forcing through combinations of numbers and letters, it is best to minimize the number of variables. The character-level analysis has already informed us that the first nine characters of the token `Ab4dt0k3n` remain static. The final three characters are the variables, and based on the sample, we can see that they follow a pattern of *letter1* + *letter2* + *number*. Moreover, a sample of the tokens tells us that that *letter1* only ever consists of letters between *a* and *d*. Observations like this will help minimize the total amount of brute force required.

Use Burp Suite Intruder or Wfuzz to brute-force the weak token. In Burp Suite, capture a request to an API endpoint that requires a token. In *Figure 8-11*, we use a GET request to the */identity/api/v2/user/dashboard* endpoint and include the token as a header. Send the captured request to Intruder, and under the Intruder Payload Positions tab, select the attack positions.

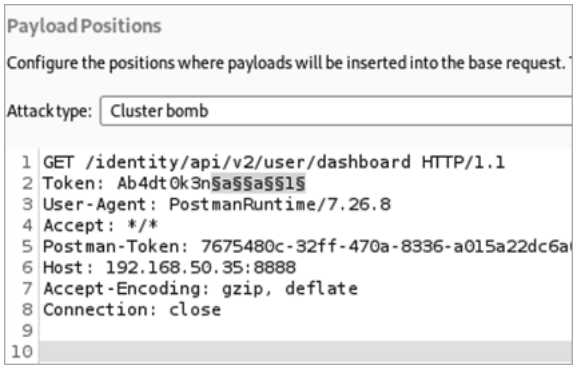

Figure 8-11: A cluster bomb attack in Burp Suite Intruder

Since we're brute-forcing the final three characters only, create three attack positions: one for the third character from the end, one for the second character from the end, and one for the final character. Update the attack type to **cluster bomb** so Intruder will iterate through each possible combination. Next, configure the payloads, as shown in *Figure 8-12*.

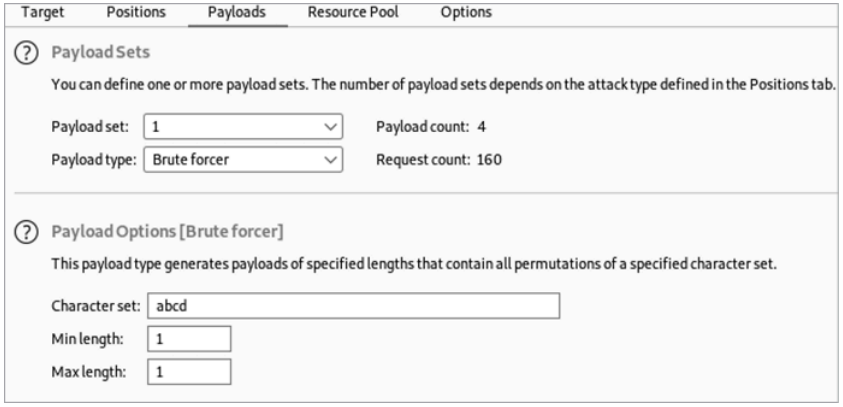

Figure 8-12: The payloads tab in Burp Suite's Intruder

Select the **Payload Set** number, which represents a specific attack position, and set the payload type to **brute forcer**. In the character set field, include all num-

bers and letters to be tested in that position. Because the first two payloads are letters, we'll want to try all letters from *a* to *d*. For payload set 3, the character set should include the digits 0 through 9. Set both the minimum and maximum length to **1**, as each attack position is one character long. Start the attack, and Burp Suite will send all 160 token possibilities in requests to the endpoint.

Burp Suite CE throttles Intruder requests. As a faster, free alternative, you may want to use Wfuzz, like so:

```
$ wfuzz -u vulnexample.com/api/v2/user/dashboard -hc 404 -H "token: Ab4dt0k3nFUZZFUZ2ZFUZ3Z1" -z list,a-b-c-d -z list,a-b-c-
================================================================================
ID           Response   Lines    Word            Chars       Payload
================================================================================
000000117:   200        1 L      10 W            345 Ch      " Ab4dt0k3nca1"
000000118:   200        1 L      10 W            345 Ch      " Ab4dt0k3ncb2"
000000119:   200        1 L      10 W            345 Ch      " Ab4dt0k3ncc3"
000000120:   200        1 L      10 W            345 Ch      " Ab4dt0k3ncd4"
000000121:   200        1 L      10 W            345 Ch      " Ab4dt0k3nce5"
```

Include a header token in your request using `-H`. To specify three payload positions, label the first as `FUZZ`, the second as `FUZ2Z`, and the third as `FUZ3Z`. Following `-z`, list the payloads. We use `-z list,a-b-c-d` to cycle through the letters *a* to *d* for the first two payload positions, and we use `-z range,0-9` to cycle through the numbers in the final payload position.

Armed with a list of valid tokens, leverage them in API requests to find out more about what privileges they have. If you have a collection of requests in Postman, try simply updating the token variable to a captured one and use the Postman Runner to quickly test all the requests in the collection. That should give you a fairly good idea of a given token's capabilities.

JSON Web Token Abuse

I introduced JSON Web Tokens (JWTs) in Chapter 2. They're one of the more prevalent API token types because they operate across a wide variety of programming languages, including Python, Java, Node.js, and Ruby. While the tactics described in the last section could work against JWTs as well, these tokens can be vulnerable to several additional attacks. This section will guide you through a few attacks you can use to test and break poorly implemented JWTs. These attacks

could grant you basic unauthorized access or even administrative access to an API.

NOTE

For testing purposes, you might want to generate your own JWTs. Use https://jwt.io, a site created by Auth0, to do so. Sometimes the JWTs have been configured so improperly that the API will accept any JWT.

If you've captured another user's JWT, you can try sending it to the provider and pass it off as your own. There is a chance that the token is still valid and you can gain access to the API as the user specified in the payload. More commonly, though, you'll register with an API and the provider will respond with a JWT. Once you have been issued a JWT, you will need to include it in all subsequent requests. If you are using a browser, this process will happen automatically.

Recognizing and Analyzing JWTs

You should be able to distinguish JWTs from other tokens because they consist of three parts separated by periods: the header, payload, and signature. As you can see in the following JWT, the header and payload will normally begin with `ey`:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJoYWNrYXBpcy5pbyIsImV4cCI6IDE1ODM2Mzc0ODgsInVzZXJuYW1lIjoiU2N1dHRsZXBoMXNoIiw
```

The first step to attacking a JWT is to decode and analyze it. If you discovered exposed JWTs during reconnaissance, stick them into a decoder tool to see if the JWT payload contains any useful information, such as username and user ID. You might also get lucky and obtain a JWT that contains username and password combinations. In Burp Suite's Decoder, paste the JWT into the top window, select **Decode As**, and choose the **Base64** option (see *Figure 8-13*).

Figure 8-13: Using Burp Suite Decoder to decode a JWT

The *header* is a base64-encoded value that includes information about the type of token and hashing algorithm used for signing. A decoded header will look like the following:

```
{
"alg": "HS256"
"typ": "JWT"
}
```

In this example, the hashing algorithm is HMAC using SHA256. HMAC is primarily used to provide integrity checks similar to digital signatures. SHA256 is a hashing encryption with function developed by the NSA and released in 2001. Another common hashing algorithm you might see is RS256, or RSA using SHA256, an asymmetric hashing algorithm. For additional information, check out the Microsoft API documentation on cryptography at *https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography*.

When a JWT uses a symmetric key system, both the consumer and provider will need to have a single key. When a JWT uses an asymmetric key system, the provider and consumer will use two different keys. Understanding the difference between symmetric and asymmetric encryption will give you a boost when performing a JWT algorithm bypass attack, found later in this chapter.

If the algorithm value is `"none"`, the token has not been signed with any hashing algorithm. We will return to how we can take advantage of JWTs without a hashing algorithm later in this chapter.

The *payload* is the data included within the token. The fields within the payload differ per API but typically contain information used for authorization, such as a username, user ID, password, email address, date of token creation (often called IAT), and privilege level. A decoded payload should look like the following:

```
{
  "userID": "1234567890",
  "name": "hAPI Hacker",
  "iat": 1516239022
}
```

Finally, the *signature* is the output of HMAC used for token validation and generated with the algorithm specified in the header. To create the signature, the API base64-encodes the header and payload and then applies the hashing algorithm and a secret. The secret can be in the form of a password or a secret string, such as a 256-bit key. Without knowledge of the secret, the payload of the JWT will remain encoded.

A signature using HS256 will look like the following:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  thebest1)
```

To help you analyze JWTs, leverage the JSON Web Token Toolkit by using the following command:

```
$ jwt_tool eyghbocibiJIUZZINIISIRSCCI6IkpXUCJ9.eyIzdW1101IxMjMENTY3ODkwIiwibmFtZSI6ImhBuEkgSGFja2VyIiwiaWFQIjoxNTE2MjM5MDIyf
Original JWT:
Decoded Token Values:
Token header values:
[+] alg - "HS256"
[+] typ - "JWT"
Token payload values:
[+] sub = "1234567890"
[+] name - "HAPI Hacker"
[+] iat - 1516239022 = TIMESTAMP - 2021-01-17 17:30:22 (UTC)
JWT common timestamps:
iat - Issuedat
```

```
exp – Expires
nbf - NotBefore
```

As you can see, `jwt_tool` makes the header and payload values nice and clear.

Additionally, `jwt_tool` has a "Playbook Scan" that can be used to target a web application and scan for common JWT vulnerabilities. You can run this scan by using the following:

```
$ jwt_tool -t http://target-site.com/ -rc "Header: JWT_Token" -M pb
```

To use this command, you'll need to know what you should expect as the JWT header. When you have this information, replace `"Header"` with the name of the header and `"JWT_Token"` with the actual token value.

The None Attack

If you ever come across a JWT using `"none"` as its algorithm, you've found an easy win. After decoding the token, you should be able to clearly see the header, payload, and signature. From here, you can alter the information contained in the payload to be whatever you'd like. For example, you could change the username to something likely used by the provider's admin account (like root, admin, administrator, test, or adm), as shown here:

```
{
   "username": "root",
   "iat": 1516239022
}
```

Once you've edited the payload, use Burp Suite's Decoder to encode the payload with base64; then insert it into the JWT. Importantly, since the algorithm is set to `"none"`, any signature that was present can be removed. In other words, you can remove everything following the third period in the JWT. Send the JWT to the provider in a request and check whether you've gained unauthorized access to the API.

The Algorithm Switch Attack

There is a chance the API provider isn't checking the JWTs properly. If this is the case, we may be able to trick a provider into accepting a JWT with an altered algorithm.

One of the first things you should attempt is sending a JWT without including the signature. This can be done by erasing the signature altogether and leaving the last period in place, like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJoYWNrYXBpcy5pbyIsImV4cCI6IDE1ODM2Mzc0ODgsInVzZXJuYW1lIjoiU2N1dHRsZXBoMXNoIiw
```

If this isn't successful, attempt to alter the algorithm header field to `"none"`. Decode the JWT, updating the `"alg"` value to `"none"`, base64-encode the header, and send it to the provider. If successful, pivot to the None attack.

```
{
"alg": "none"
"typ": "JWT"
}
```

You can use JWT_Tool to create a variety of tokens with the algorithm set to `"none"`:

```
$ jwt_tool <JWT_Token> -X a
```

Using this command will automatically create several JWTs that have different forms of "no algorithm" applied.

A more likely scenario than the provider accepting no algorithm is that they accept multiple algorithms. For example, if the provider uses RS256 but doesn't limit the acceptable algorithm values, we could alter the algorithm to HS256. This is useful, as RS256 is an asymmetric encryption scheme, meaning we need both the provider's private key and a public key in order to accurately hash the JWT signature. Meanwhile, HS256 is symmetric encryption, so only one key is used for both the signature and verification of the token. If you can discover the provider's RS256 public key and then switch the algorithm from RS256 to HS256, there is a chance you may be able to leverage the RS256 public key as the HS256 key.

The JWT_Tool can make this attack a bit easier. It uses the format `jwt_tool <JWT_Token> -X k -pk public-key.pem`, as shown next. You will need to save the captured public key as a file on your attacking machine.

```
$ jwt_tool eyJBeXAiOiJKV1QiLCJhbGciOiJSUZI1Ni 19.eyJpc3MiOi JodHRwOlwvxC9kZW1vLnNqb2VyZGxhbmdrzwiwZXIubmxcLyIsIm1hdCI6MTYYCJ
Original JWT:
File loaded: public-key. pem
jwttool_563e386e825d299e2fc@aadaeec25269 - EXPLOIT: Key-Confusion attack (signing using the Public key as the HMAC secret)
(This will only be valid on unpatched implementations of JWT.)
[+] ey JoexAiOiJK1QiLCJhbGciOiJIUZI1NiJ9.eyJpc3MiOiJodHRwOi8vZGVtby5zam91cmRsYW5na2VtcGVyLmSsLyIsIm1hdCI6MTYyNTc4NzkzOSwizhJ
```

Once you run the command, JWT_Tool will provide you with a new token to use against the API provider. If the provider is vulnerable, you'll be able to hijack other tokens, since you now have the key required to sign tokens. Try repeating the process, this time creating a new token based on other API users, especially administrative ones.

The JWT Crack Attack

The JWT Crack attack attempts to crack the secret used for the JWT signature hash, giving us full control over the process of creating our own valid JWTs. Hash-cracking attacks like this take place offline and do not interact with the provider. Therefore, we do not need to worry about causing havoc by sending millions of requests to an API provider.

You can use JWT_Tool or a tool like Hashcat to crack JWT secrets. You'll feed your hash cracker a list of words. The hash cracker will then hash those words and compare the values to the original hashed signature to determine if one of those words was used as the hash secret. If you're performing a long-term brute-force attack of every character possibility, you may want to use the dedicated GPUs that power Hashcat instead of JWT_Tool. That being said, JWT_Tool can still test 12 million passwords in under a minute.

To perform a JWT Crack attack using JWT_Tool, use the following command:

```
$ jwt_tool <JWT Token> -C -d /wordlist.txt
```

The `-C` option indicates that you'll be conducting a hash crack attack and the `-d` option specifies the dictionary or wordlist you'll be using against the hash. In this example, the name of my dictionary is *wordlist.txt*, but you can specify the direc-

tory and name of whatever wordlist you would like to use. JWT_Tool will either return "CORRECT key!" for each value in the dictionary or indicate an unsuccessful attempt with "key not found in dictionary."

Summary

This chapter covered various methods of hacking API authentication, exploiting tokens, and attacking JSON Web Tokens specifically. When present, authentication is usually an API's first defense mechanism, so if your authentication attacks are successful, your unauthorized access can become a foothold for additional attacks.

Lab #5: Cracking a crAPI JWT Signature

Return to the crAPI authentication page to try your hand at attacking the authentication process. We know that this authentication process has three parts: account registration, password reset functionality, and the login operation. All three of these should be thoroughly tested. In this lab, we'll focus on attacking the token provided after a successful authentication attempt.

If you remember your crAPI login information, go ahead and log in. (Otherwise, sign up for a new account.) Make sure you have Burp Suite open and FoxyProxy set to proxy traffic to Burp so you can intercept the login request. Then forward the intercepted request to the crAPI provider. If you've entered in your email and password correctly, you should receive an HTTP 200 response and a Bearer token.

Hopefully, you now notice something special about the Bearer token. That's right: it is broken down into three parts separated by periods, and the first two parts begin with `ey`. We have ourselves a JSON Web Token! Let's begin by analyzing the JWT using a site like *https://jwt.io* or JWT_Tool. For visual purposes, *Figure 8-14* shows the token in the JWT.io debugger.

Figure 8-14: A captured JWT being analyzed in JWT.io's debugger

As you can see, the JWT header tells us that the algorithm is set to HS512, an even stronger hash algorithm than those covered earlier. Also, the payload contains a `"sub"` value with our email. The payload also contains two values used for token expiration: `iat` and `exp`. Finally, the signature confirms that HMAC+SHA512 is in use and that a secret key is required to sign the JWT.

A natural next step would be to conduct None attacks to try to bypass the hashing algorithm. I will leave that for you to explore on your own. We won't attempt any other algorithm switch attack, as we're already attacking a symmetric key encryption system, so switching the algorithm type won't benefit us here. That leaves us with performing JWT Crack attacks.

To perform a Crack attack against your captured token, copy the token from the intercepted request. Open a terminal and run JWT_Tool. As a first-round attack, we can use the *rockyou.txt* file as our dictionary:

```
$ jwt_tool eyJhbGciOiJIUZUxMi19.eyJzdWIiOiJhQGVtYWlsLmNvbSIsImlhdCI6MTYYNTC4NzA4MywiZXhwIjoxNjI10DCzNDgzfQ. EYx8ae40nE2n9ec4
Original JWT:
[*] Tested 1 million passwords so far
[*] Tested 2 million passwords so far
[*] Tested 3 million passwords so far
[*] Tested 4 million passwords so far
```

```
[*] Tested 5 million passwords so far
[*] Tested 6 million passwords so far
[*] Tested 7 million passwords so far
[*] Tested 8 million passwords so far
[*] Tested 9 million passwords so far
[*] Tested 10 million passwords so far
[*] Tested 11 million passwords so far
[*] Tested 12 million passwords so far
[*] Tested 13 million passwords so far
[*] Tested 14 million passwords so far
[-] Key not in dictionary
```

At the beginning of this chapter, I mentioned that *rockyou.txt* is outdated, so it likely won't yield any successes. Let's try brainstorming some likely secrets and save them to our own *crapi.txt* file (see *Table 8-1*). You can also generate a similar list using a password profiler, as recommended earlier in this chapter.

Table 8-1: *Potential crAPI JWT Secrets*

Crapi2020	OWASP	iparc2022
crapi2022	owasp	iparc2023
crAPI2022	Jwt2022	iparc2020
crAPI2020	Jwt2020	iparc2021
crAPI2021	Jwt_2022	iparc
crapi	Jwt_2020	JWT
community	Owasp2021	jwt2020

Now run this targeted hash crack attack using JWT_Tool:

```
$ jwt_tool eyJhbGciOiJIUzUxMi19.eyJzdwiOiJhQGVtYWlsLmNvbSIsImlhdCI6MTYYNTC4NzA4MywiZXhwIjoxNjI10DCzNDgzfQ. EYx8ae40nE2n9ec4y
Original JWT:
[+] crapi is the CORRECT key!
You can tamper/fuzz the token contents (-T/-I) and sign it using:
python3 jwt_tool.py [options here] -5 HS512 -p "crapi"
```

Great! We've discovered that the crAPI JWT secret is `"crapi"`.

This secret isn't too useful unless we have email addresses of other valid users, which we'll need to forge their tokens. Luckily, we accomplished this at the end of Chapter 7's lab. Let's see if we can gain unauthorized access to the robot account. As you can see in *Figure 8-15*, we use JWT.io to generate a token for the crAPI robot account.

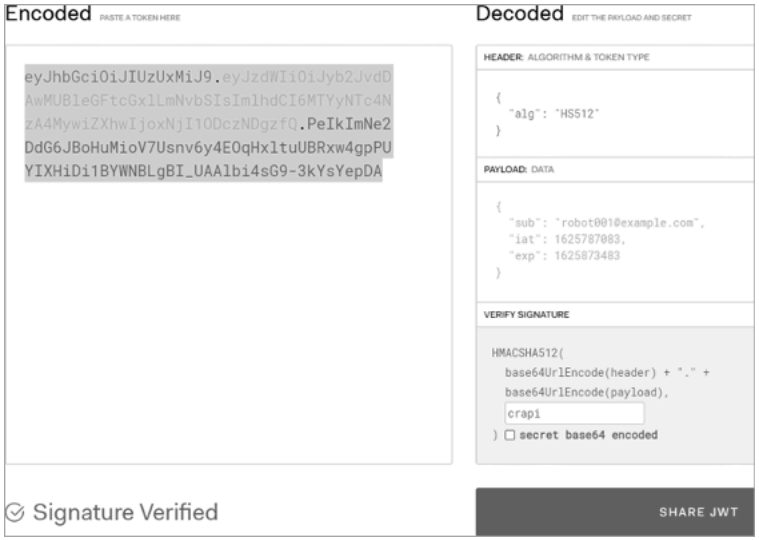

Figure 8-15: Using JWT.io to generate a token

Don't forget that the algorithm value of this token is HS512 and that you need to add the HS512 secret to the signature. Once the token is generated, you can copy it into a saved Postman request or into a request using Burp Suite's Repeater, and then you can send it to the API. If successful, you'll have hijacked the crAPI robot account. Congrats!