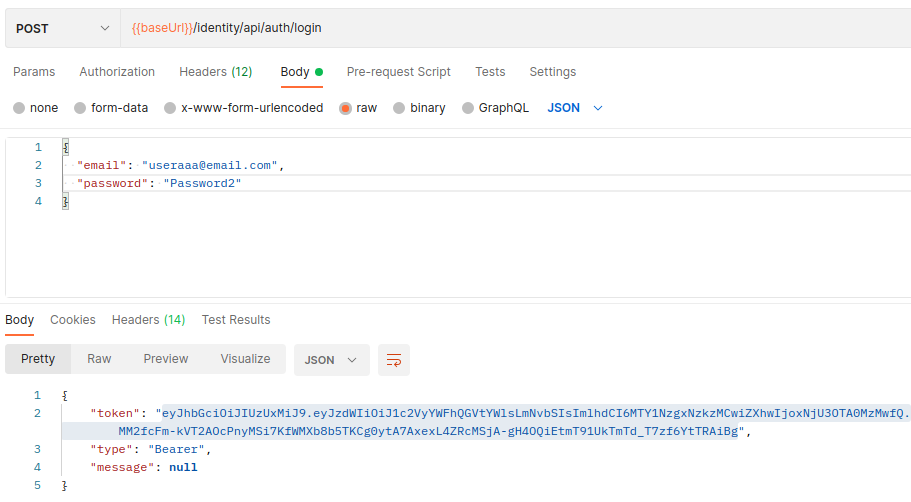
Classic Authentication Attacks

Classic authentication attacks include techniques that have been around for a while like brute forcing and password spraying. Both of these attacks are different methods of guessing username and password combinations. These are vulnerabilities that are inherent anytime basic authentication is deployed. To authenticate using this basic authentication, the consumer issues a request containing a username and password, then the provider performs a check to make sure that the combination matches records stored in a database. As we know, RESTful APIs do not maintain a state, so if the API were to leverage basic authentication across all endpoints, then a username and password would have to be issued every time. Instead, the provider may leverage basic authentication once using an authentication portal, then upon providing the correct credentials, a token would be issued to be used in subsequent requests. Let’s check out crAPI for an example of this process.



Thus, providers typically use basic authentication only as part of an initial login process. This is exactly what we have seen with crAPI. Following a successful POST request to /identity/api/auth/login users are provided with a Bearer Token. The classic authentication attacks in this section include password brute-force attacks with base64 encoding, password reset brute-force, and password spraying.

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 require base64 encoding.

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 the Endpoint Analysis module, 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 Intruder or Wfuzz. The following example uses Wfuzz with an old, well-known password list, rockyou.txt. Rockyou.txt does come as a standard wordlist on Kali Linux, however, it is currently zipped with Gzip. You can unzip rockyou.txt using the following command.

$gzip -d /usr/share/wordlists/rockyou.txt.gz

Before, we dive into using WFuzz let's treat this like a real test. Earlier we discovered an excessive data exposure vulnerability with a request to the crAPI community forum. Let's gather email addresses from that request to perform a password.

To start out with WFuzz, access the help menu and read the options to get a better idea of what can be done with this tool.  
$wfuzz --help

Important items to note for API testing include the headers option (-H), hide responses (--hc, --hl, --hw, --hh), and POST body requests (-d). All of these will be useful when fuzzing APIs. We will need to specify the content-type headers for APIs which will be 'Content-Type: application/json' for crAPI. The POST body for the crAPI login is '{"email":"a@email.com","password":"FUZZ"}'. Notice that FUZZ is the WFuzz attack position. Finally, if we are getting many irrelevant responses we can clean up our attack by hiding certain responses. For the following brute force attack, it is recommended that you only attack your own lab environment. If many students attempted brute force attacks against the hosted labs, it could cause the apps to fail.

$ wfuzz -d '{"email":"a@email.com","password":"FUZZ"}' -H 'Content-Type: application/json' -z file,/usr/share/wordlists/rockyou.txt -u http://127.0.0.1:8888/identity/api/auth/login --hc 405 ================================================================== ID Response Lines Word Chars Payload ================================================================== 000000007: 200 0 L 1 W 225 Ch "Password1!" 000000005: 400 0 L 34 W 474 Ch "win"

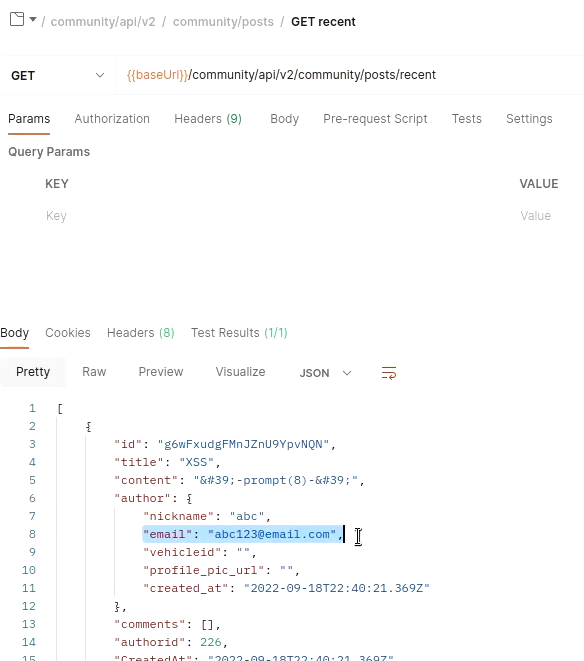
The -d option allows you to fuzz content that is sent in the body of a POST request. The curly brackets that follow contain the POST request body. To discover the request format used in this example, I attempted to authenticate to a web application using a browser, and then I captured the authentication 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 respond with 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 on 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 Spraying

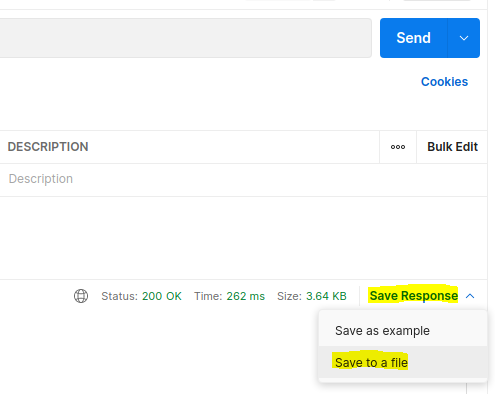
Many security controls could prevent you from successfully brute-forcing an API’s authentication. A technique called password spraying can evade many of these controls by combining a long list of users with a short list of targeted passwords. Let’s say you know that an API authentication process has 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 constraints 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. Use 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!@#$, Password1!, and the formula Season+Year+Symbol (such as Winter2021!, Spring2021?, Fall2021!, and Autumn2021?). The second type includes more advanced passwords that relate directly to the target, often including a capitalized letter, a number, a detail about the organization, and a symbol. Here is a short password-spraying list I might generate if I were attacking an endpoint for Twitter employees: Summer2022! Spring2022! QWER!@#$ March212006! July152006! Twitter@2022 JPD1976! [Dorsey@2022](mailto:Dorsey@2021)

The real key to password spraying is to maximize your user list. The more usernames you include, the higher your odds of compromising a user account with a bad password. Build a user list during your reconnaissance efforts or by discovering excessive data exposure vulnerabilities. Let's revisit the crAPI excessive data exposure that we discovered earlier in the course.

Postman Screenshot of Excessive Data Exposure.

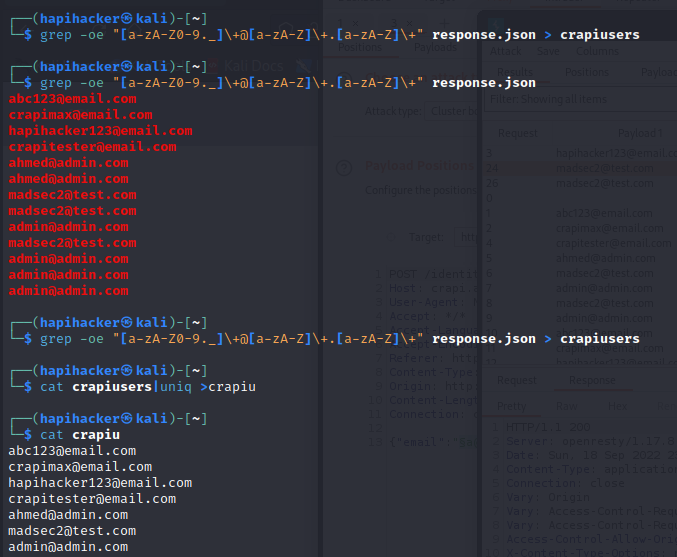


At the top right of the Postman Response pane, select "Save this Response" and then "Save to a file". Save the response containing an excessive data exposure as "response.json".

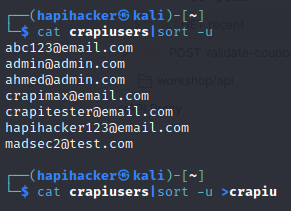


Next, you can use the power of grep and regular expressions to pull emails from the saved JSON response.

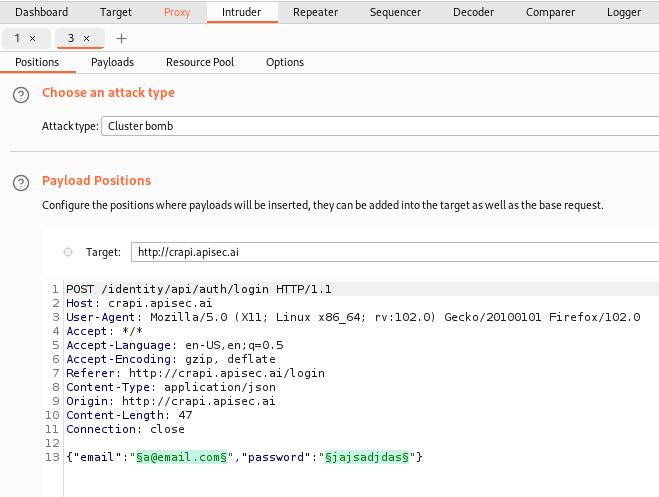
$grep -oe "[a-zA-Z0-9.\_]\+@[a-zA-Z]\+.[a-zA-Z]\+" response.json



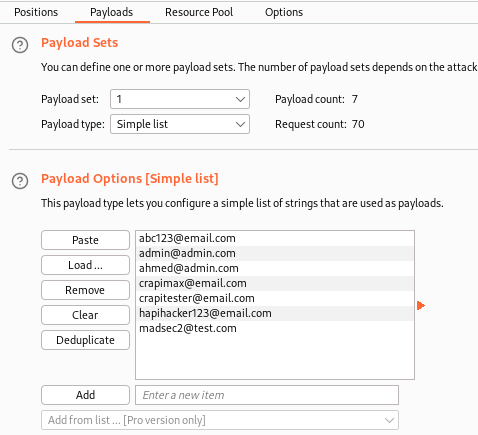
Using this grep command should pull everything out of a file that resembles an email. Then you can save those captured emails to a file and use that file as a payload in Burp Suite. You can then use a command like sort -u to get rid of duplicate emails.

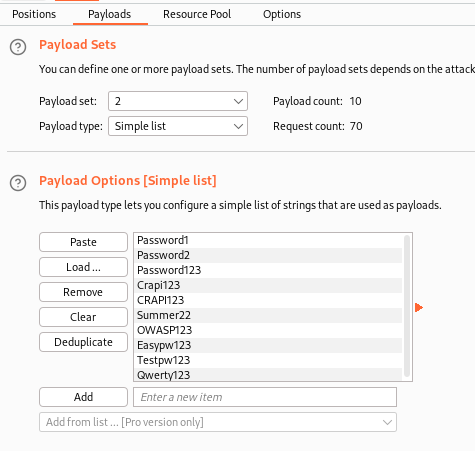


In Burp Suite’s Intruder, you can set up this attack in a similar manner to the standard brute-force attack, except you’ll use both a list of users and a list of passwords. Choose the cluster bomb attack type and set the attack positions around the username and password as seen below.

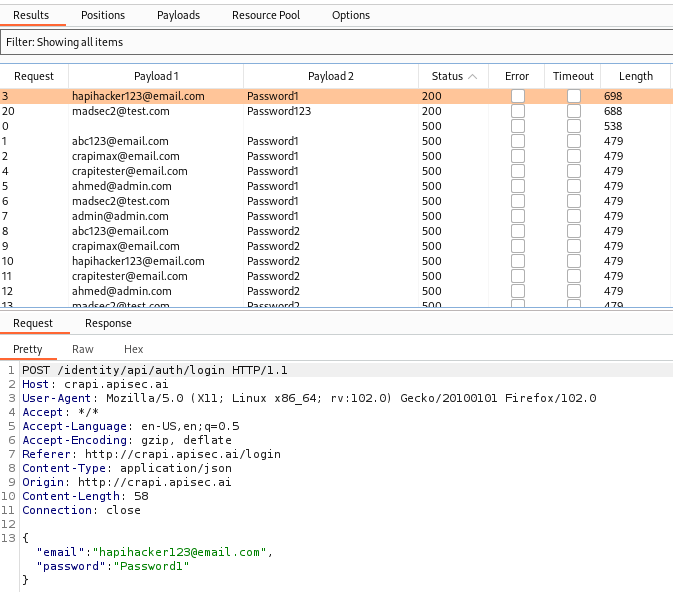


Notice that the first attack position is set to replace the username in front of @email.com, which you can do if you’ll only be testing for users within a specific email domain. Next, add the list of collected users as the first payload set and a short list of passwords as your second payload set. Once your payloads are configured, you’re ready to perform a password-spraying 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 and the response length of successful responses will be different than unsuccessful responses.



You can see a successful password-spraying attempt that has two anomalous features: a status code of 200 and a response length of 698. To help spot anomalies using Intruder, you can sort the results by status code or response length.