August 24, 2022 · API Hacking Fundamentals

The Beginner's Guide to API Hacking

If you're reading this, you probably want to know how to get started in API hacking. Well, you've come to the right place!

In this article, I'll discuss some basic concepts and give you a few tips on how to get started. Keep in mind that this is just a beginner's guide – there is much more to learn about API hacking than what we'll cover here.

My hope is to point you in the right direction on how to look at API security in a different way through multiple API attack techniques... how to test web APIs and how to abuse web applications that leverage these APIs to find those high payout API bugs.

What is an API?

An API is an application programming interface. In other words, it's a way for different software applications to communicate with each other.

APIs are everywhere – they power the interactions between our favorite apps, websites, and devices.

Many modern web applications rely on APIs. For example, when you order a product on Amazon, the company uses an API to communicate with your bank and process the payment. Likewise, when you check the weather on your phone, it's using an API to fetch data from a weather service.

In short, APIs today are what make the modern world go 'round!

And when it comes to API security, there are plenty of real world examples that showcase this as a prime place to look for vulnerabilities. Like the time the hacking group Anonymous (with support from the IT Army of Ukraine) hacked Russia's most popular taxi company's API to cause a huge traffic jam in the middle of Moscow.

Why would you want to hack an API?

API hacking is a type of security testing that seeks to exploit weaknesses in an API. By targeting an API endpoint, you as an attacker can potentially gain access to sensitive data, interrupt services or even take over entire systems.

> It's said that more than 80% of all web traffic is now driven through API requests.

Web API security testing can be lucrative, especially if you are into bug bounties. Many bug bounty programs have a well-defined scope of their web application programming interfaces that you can take advantage of to penetration test APIs.

A good security researcher can additionally offer penetration testing services around common API vulnerabilities to help their customers test web APIs. Be it an internal API or a public one, breaking web application programming through brute forcing allows you to perform common attacks in a methodical and consistent manner.

It's surprising just how easy it is to find high payout API bugs based on a simple methodology to breaking web application programming interfaces. We will discuss some of this later in this article.

Before we can do all that though, we need to level set on some key fundamentals around how the web works so we start hacking APIs.

HTTP Fundamentals

In order to test web APIs, you need to understand how they communicate. The most common way for applications to interact with each other is through HTTP requests.

HTTP is a protocol that allows web browsers/clients and servers to communicate with each other. When you type a URL into your browser, your computer sends an HTTP request to the server that hosts the website. The server then responds with HTML and other files that are required to display the website.

APIs work in much the same way. Usually though, instead of responding with HTML it sends data back and forth in a structured manner using JSON or XML. As an example, REST APIs and GraphQL API prefer to use JSON objects. SOAP APIs on the other hand requires you to use XML.

In fact, this is an excellent way to detect API info. They are easier to discover by looking at the request header and checking what the `Content-Type` is. If it's `application/json` or `application/xml`, there is a good chance that you have stumbled upon an API endpoint.

Since we are talking about request headers, we should probably cover the rest of what makes up a request.

An HTTP request consists of three parts: the Request URL, the Request Method, and the Request Body (if any). Let's take a closer look at each one:

- **Request URL:** The Request URL is the address of the resource that you want to access. It consists of two parts – the hostname and the path. The hostname is simply the domain name of the website (e.g. http://www.amazon.com). The path is the location of the resource on the server (e.g. `/products/shoes`).
- **Request Method:** The Request Method is used to specify how you want to access the resource. The most common types of Request Methods are GET, POST, PUT, and DELETE.
- **Request Body** (if any): Some Request Methods (such as POST) require a body in order to transmit data between the client and server. The body usually contains information about what type of data is being sent and how it should be processed.

Sometimes, you may hear the term CRUD when looking into APIs. In fact, sometimes you will see CRUD used interchangeably with REST when discussing APIs.

But that's not quite right.

CRUD is an acronym for *Create*, *Read*, *Update*, *Delete*. These operations are the basis for most web applications today. And CRUD is very closely related to the Request Methods that we just talked about.

In order to perform CRUD operations, we use the following Request Methods:

- Create: POST
- Read: GET
- Update: PUT/PATCH
- Delete: DELETE

REST and CRUD work together because CRUD can exist within a REST environment, and their functions often correspond to each other.

But they are not the same.

The best way to differentiate between them is to remember that REST is a **standard** (an API architecture), and CRUD is a **function**. Understanding this essential but straightforward difference is necessary for understanding both.

Now that you understand how HTTP requests work and how to identify API requests, let's talk about how to get started by building a streamlined API testing lab.

Building your own API testing lab

A great way to get familiar with hacking APIs is to target intentionally vulnerable APIs. This lets you perform common attacks against your own APIs as well as deliberately vulnerable systems that you can download over the Internet.

There are a few different ways to go about this. I prefer using the following methods:

- Docker: This is a great way to get started because it provides you with a self-contained environment in which you can experiment without fear of breaking anything.
- Self-hosted Virtual Machines: This is the second method that I prefer. If you are using a Mac, you can use Parallels Desktop or VMware Fusion to run a VM. Linux users can use VirtualBox. And if you are using Windows, HyperV, VirtualBox or VMWare all work well.
- Cloud-hosted Virtual Machines: Azure. AWS. Digital Ocean. Linode. You name it, there are plenty of cloud providers that you can spin up vulnerable apps on. Just remember to lock down the instance to only your IP address; failing to do that

will result in your VM being compromised in no time.

All of these methods have their pros and cons, but I tend to prefer Docker because it's more lightweight and easy to use. When that doesn't make sense, I use VirtualBox with snapshots.

But YMMV. Use what works for you.

Once you have decided how to run your hacking lab, you will need to choose an intentionally vulnerable API that you can brute force attack. For the purposes of getting started, I'd recommend you consider using one of the following two:

- OWASP crAPI: This is a "completely ridiculous API" that highlights the ten most critical API security risks. It can be installed using Docker.
- OWASP Juice Shop: This is probably the most modern and sophisticated insecure web application out there. It can also be installed using Docker.

Both of these apps are great for getting started. They are both easy to install and come with a wide range of vulnerabilities that you can exploit.

If you want more options to hack on, download the free resource guide I built around API hacking. In it is a whole bunch of additional deliberately vulnerable apps you can practice on. It even includes info on a private API hacking room I have made available for you on TryHackMe so you don't even need to host anything.

So now that we have a place to play and practice, let's actually put together a simple and easy-to-follow methodology for attacking APIs.

Actually, let's first set up your API hacking toolchain.

The hacking tools you actually need

So if you grabbed my resource guide you already know there are pages and pages of tools you can use for hacking APIs. There are so many tools in fact that I can't just do it justice in a beginner's guide like this.

So forget about all of them. Except for two.

When first starting out you can accomplish pretty much everything you need if you have a decent API client and a good intermediate web proxy.

For the API client, I recommend Postman. Get the standalone app, as it's much nicer to use than the web client. It is by far the most robust tool to create, share, test, and document APIs. And it becomes even more awesome when used with my next recommendation.

For your web proxy, you want Burp Suite. While many people will

recommend you get started with the free Community Edition, I am going to tell you to buy Burp Suite Professional as soon as you can, especially if you plan to offer penetration testing services.

Here's why.

Burp is the most comprehensive web penetration testing toolkit out there. If you are really getting serious about your API hacking tradecraft, you want the performance and capabilities of the professional edition. I'm not a developer evangelist for PortSwigger, but I believe in the tool *that* much.

With the professional edition it unlocks a ton of capabilities you just don't get in the free Community edition, namely:

- You get project files so you can save your work and come back to it later.
- You can orchestrate custom attacks using the full capabilities of the *Intruder* toolchain. You can't do practical brute force attacks using Community edition as it's crippled in speed/performance. Professional edition has that limitation removed.
- You get a built-in web vulnerability scanner, INCLUDING an API scanner.
- You get access to the pro-exclusive BApp extensions that deliver more functionality into Burp. Some of these extensions considerably speed up the identification and exploitation of vulnerabilities and offer protection bypass techniques.
- You have full search capabilities and content discovery
- You get out-of-band application security testing (OAST)

through *Burp Collaborator*. Very useful when doing blind injection testing.
- You have access to task scheduling and automation

I could go on, but I think I've made my point. Get Postman and Burp. And then go to town. Let me show you a great way to get them working better together.

Burp + Postman = #WINNING

OK, so you've downloaded and installed the only two tools you *really* need to break APIs. Let me show you a way to get them to work better together.

It's possible to configure Postman to use Burp as its proxy. In other words, you can set it up so that whenever you fire an API request using Postman it can be monitored and manipulated in Burp. This becomes immensely useful when you don't want to tamper with pre-defined Postman collections and want to leverage things like Burp's *Repeater* tool to alter a request and see how it responds.

Let me show you how.

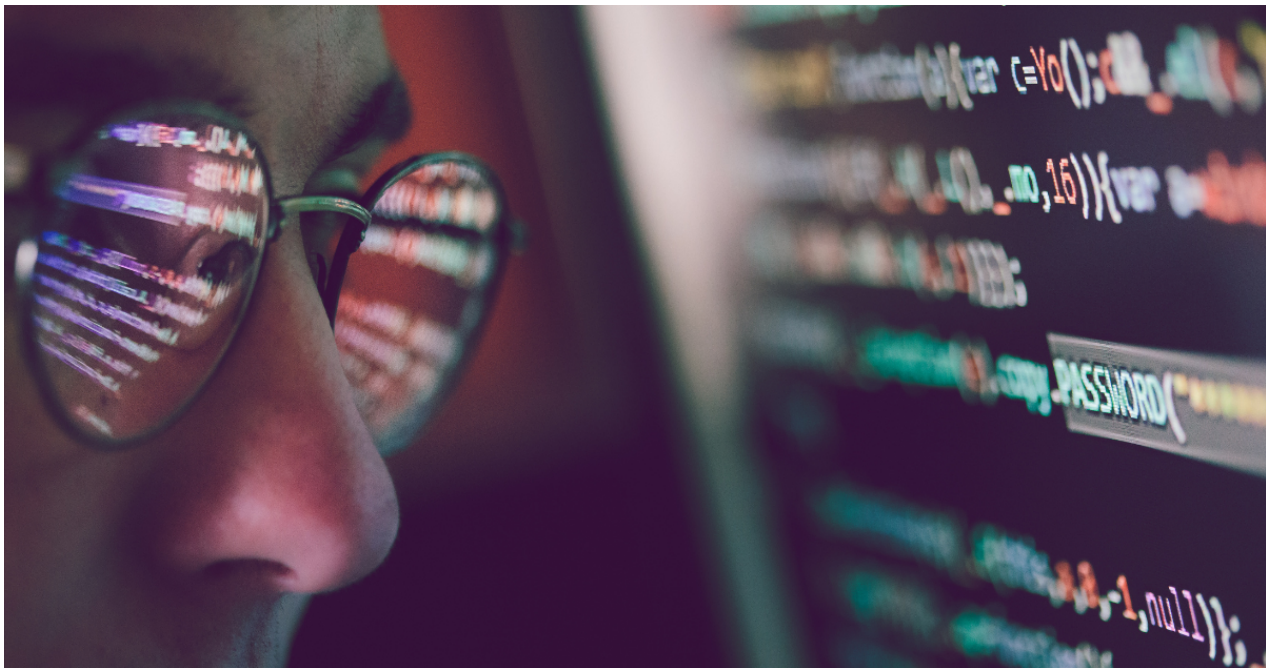
1. Open up Postman
2. Click the COG wheel on the top right side of the screen, and select **Settings**. In General disable "**SSL certificate verification**". This will allow Postman to work with Burp's self-signed cert used in the proxy.
3. In Proxy, disable "**System Proxy**" and enable "**Add a custom proxy configuration**". Ensure both HTTP and HTTPS are checked, and that you have set the proxy server to `localhost:8080`, which points to Burp's proxy.

4. Now start Burp
5. Back in Postman, go to an API collection you already have (or create a new request) and send it. You should see the request in the HTTP History in Burp.
6. Right-click on the history log item in Burp and select *Send to Repeater*.
7. Move to the Repeater tab in Burp and click the **Send** button. Notice you now have full control and the ability to hack the request.

OK, with Postman and Burp now set up, we can start actually poking at your own web API security or intentionally vulnerable APIs in your streamlined API testing lab.

Let's start by learning how to find APIs through reconnaissance (recon).

API Recon



When it comes to APIs, there are three main types to consider – public, partner, and private.

Public APIs are those that anyone can sign up for and use. For example, the Twitter API or Google Maps API. Usually, these APIs are well documented as they can be accessed by virtually anyone.

Partner APIs are ones where you have to have some sort of relationship with the provider in order to get access, but it's not something that is completely hidden from the public. A good example of this is the Stripe API. In many cases, there is good API documentation, but they are only available if you are a partner.

Private APIs are, as you would expect, completely hidden from the public eye and can only be accessed by those with explicit permission to do so. These are usually found within an organization that has built its own API for use internally. You will rarely ever find documentation for these APIs. This leaves you to have to reverse engineer the API to ever get an understanding of how it works. But that's a topic for another day, in another article.

Now that we know the types of APIs that exist, let's talk about how to find these APIs.

How to detect APIs

No matter if you want to hack a public, partner, or private API there are common techniques you can use to detect if a suspected endpoint is an application programming interface or not, and how it works.

The first way would be to find existing applications that consume the API you want to hack. By monitoring how it communicates with the API, you can detect URL paths to hopefully discover where the web APIs may exist. A simple way to monitor this would be to open

DevTools in your browser and simply watch the network tab for requests as you use the app. Another would be to use a web proxy and record all the communications.

A second way is to try to find the API documentation. There are several ways this can be accomplished:

- You can search for the API documentation directly from the vendor/publisher of the web API on their website or public source code repositories like GitHub.
- You can search for published API documentation within the API itself. This is commonly called the “swagger” or “OpenAPI” documentation.
- You can look for the API documentation within third-party API directories.

There is a great wordlist designed specifically for finding API documentation called swagger.txt, which is part of Daniel Miessler’s SecLists repository. You can use Burp Intruder to automatically test each known path via fuzzing to detect if the API documentation exists on a target.

Of course, sometimes it’s just as easy to look for default paths. A third method to detect APIs is to look for common paths like:

- https://target.domain/api
- https://target.domain/v1, https://target.domain/v2 etc
- https://target.domain/graphql
- https://target.domain/swagger
- https://target.domain/docs
- https://target.domain/rest

While you are fuzzing paths, you should also fuzz subdomains too. It's not uncommon to see APIs exist under a subdomain like **api**.target.domain. I'd recommend you grab a common subdomain wordlist like subdomains-top1million-5000.txt and tailor it to searching for APIs on your target.

As a fourth method, you can check public third-party API directories that catalog known APIs, and offer details where you can find API info. There are several API directories you can use, including:

- APIs Guru
- Postman Explore
- ProgrammableWeb API Directory
- Public APIs Github Project
- RapidAPI Hub
- SwaggerHub

Once you've determined where your API may reside and how it may work it's time to go deeper and discover more about the API attack surface. The first step is to do some passive recon through open-source intelligence (OSINT) that you have access to on the Internet.

Passive Recon

In this step, you will want to learn as much as possible about your target without actually interacting with it. This is called passive reconnaissance and can be very useful in the early stages of penetration testing of an engagement.

It's during this phase that you may not only discover endpoints and documentation, but artifacts of the application programming interface like versioning, keys, credentials, and tokens. All useful

intelligence that can help you decide how to approach your target.

Let's explore some of the more common passive recon techniques you can use to learn more about your target.

Google Dorking

I think it goes without saying that Google does a pretty damn good job of spidering pretty much everything. Use that to your advantage. Even something as simple as "<appname> api" may render you results and point you to API documentation.

But don't stop there.

You can combine search syntax like **inurl** with **site** to find API artifacts. An example, searching for `"inurl:/api/admin site:slack.com"` not only shows Slack's dedicated API site at *api.slack.com*, but you can discover that there is an endpoint at `https://slack.com/api/admin.users.list` that lists all user's in a slackspace. Useful. Very useful.

Here are a few other Google dorks you can try:

- `intitle:"index of" intext:"api"` ← Find API artifacts exposed on a web server, like keys and config
- `inurl:"/api/*" intext:"index of"` ← Find interesting API directories
- `intext:api filetype:env` ← Find possible environment variables relating to API, including keys and tokens
- `intitle:"index of" api_key OR "api key" OR apiKey -pool` ← Find potential API keys
- `intext:APIKey ext:js | xml | yaml | txt | conf | py intitle:"index of"` ← Find API keys in interesting

files.
- `intitle:"index of" "api.yaml"` ← Find potential API configs
- `"api" ext:log` ← Find log files that may be leaking information about API artifacts

TIP: Remember to also include the **site:** param to scope the search down to your target.

YMMV. Use your imagination. You'd be surprised what you can find via Google this way.

Shodan

Shodan is a search engine that lets you find specific devices and information on the Internet. It can be used to find vulnerable systems, such as servers with known vulnerabilities, or to identify specific devices on a network. Shodan also includes a feature called "Scanning" which allows users to scan for devices that meet certain criteria, such as port number or operating system.

Of course, you can use it to help find API info too.

Remember earlier when I said a good indicator that you have found an API is through its *Content-Type*? Well, with Shodan you can add a filter to your query to look for that. As well as things like port number and response codes. Here are a few Shodan dorks you can try:

- `port:80,443 http.status:200 "Content-Type: application/json"` ← Find web servers returning potential REST endpoints
- `"Content-Type: application/xml"` ← Find web servers

returning potential endpoints that use XML (ie: SOAP)
- `"X-*API*" hostname:"*.target.domain"` ← Find servers that contain custom headers related to "API". ie: X-API-KEY, X-API-VERSION, X-API-ENV, X-AMZ-API-PATH etc
- `ssl.cert.subject.cn:target.domain` ← Find servers who have been issued an SSL cert for *.target.domain
- `ssl:"Company Name"` ← Find servers who have been issued an SSL cert relating to the company you are targeting. Useful for certs generated by SaaS/cloud vendors offering services to the target (ie: AWS, Azure, Google, etc). This typically finds stuff in the Issued To organization fields.

You can get pretty creative here. If the scope of your engagement includes large blocks of IPs, you can have a field day finding undocumented endpoints throughout the infrastructure using Shodan dorks.

Have fun with it.

Active Recon

When it comes to active recon, there are tons of interesting tools out there. As I promised you that when getting started you can do pretty much everything with Postman and Burp, it only seems reasonable that we stick with that.

However, before I do I will say that there is value in having other tools in your toolchain for active recon. Here are just a few I use, that in time, you will probably find helpful:

- I use nmap to do port recon on my targets. More importantly, I use nmap's NSE scripts like *http-enum, http-methods,*

http-headers, etc to enumerate the web server in greater detail.

- I use feroxbuster to do directory enumeration, especially to find active endpoints and documentation. It's a wickedly fast content discovery tool that supports recursion. You can also tailor the threads and response speed to dead with rate limiting which is always helpful.
- I recently started moving to use kiterunner over feroxbuster, as it works to find even more API artifacts than raw directory bruteforcing does. It finds more resources as it extracts the actual routes and can detect the methods, headers, params, and values an API may have.

With that out of the way, let's use Burp's Intruder tool to query a server to find APIs through fuzzing techniques.

API discovery with Burp

Directory fuzzing is a process of bruteforcing directories and files on a web server to find potential APIs. This can be done with feroxbuster, kiterunner, or other similar tools. However, we can do this with Burp Suite as well.

To do so, we need to configure Intruder in Burp. For our example, let's say our target is *vulnapi.thm* and we want to find any API endpoints that might exist on the server. We would configure our attack like so:

1. Open Burp and launch the built-in browser
2. Visit http://vulnapi.thm to create the first request and populate the HTTP history.
3. Right-click on the log item in the history and select *Send to*

Intruder. Alternatively, you can use the keyboard shortcut **CTRL+I**.

4. Now open the Intruder tab. you should be on the **Positions** tab of your latest request.
5. For the "Attack Type", leave it as **Sniper**.
6. In the "Payload Positions" section, append the word **FUZZ** after "`GET /`". Then highlight the word, and click the **Add** button to set a *payload marker*. It should look something like this (`GET / § FUZZ § HTTP/1.1`):
7. Now select the **Payloads** tab.
8. Under "Payload Options", click the **Load** button.
9. Select a wordlist you have that you would like to fuzz with. As a starting point, the common-api-endpoints-mazen160.txt from SecLists is a good choice.
10. Once the wordlist is loaded, click the **Start attack** button to begin.
11. Once the attack completes, sort by the Status and/or Length column. If anything was found, you should be able to see it.

Now, this is where tools like feroxbuster and kiterunner are better. At this point, it can recursively continue searching down the routes. ie: `/api/v1` etc. With Burp, you would now need to update your payload to account for the newly found subdirectory and scan again.

Once you have found the path to where you believe the API may exist, you will want to actually try to discover routes through *objects* and *actions*. SecLists has some good wordlists you can use as a base called objects.txt and actions.txt respectively that can work with your fuzzing techniques.

Now you can use a powerful attack feature in Burp Intruder called the **cluster bomb**:

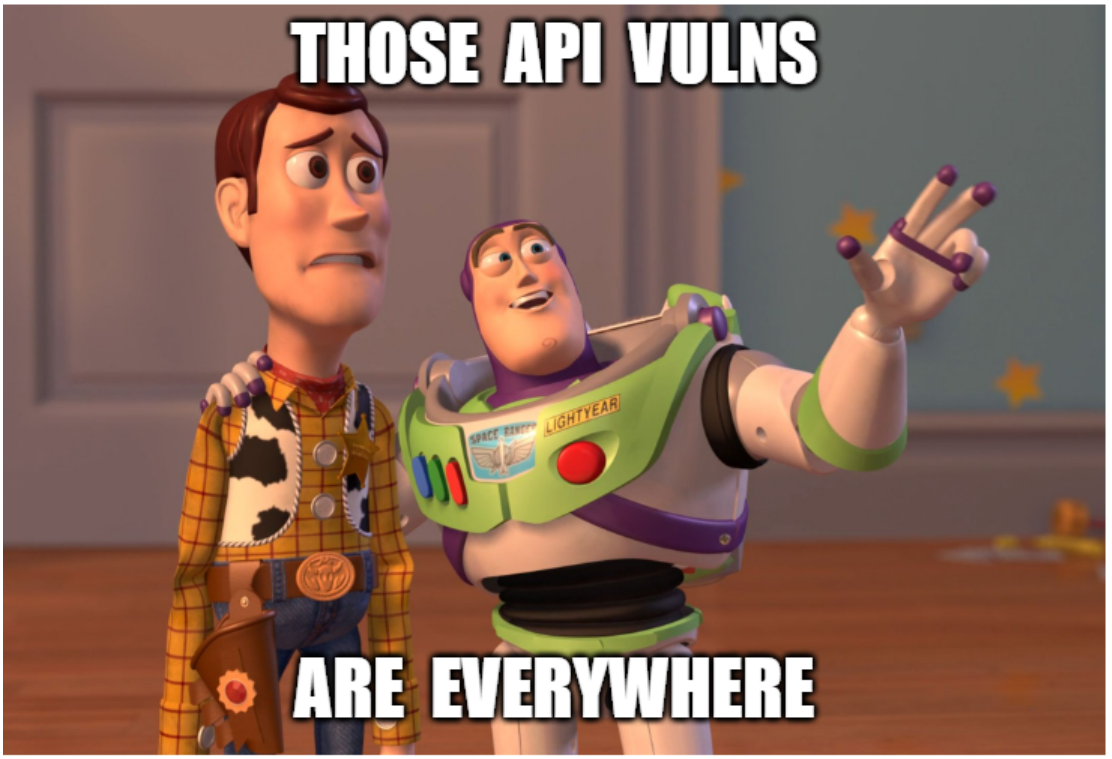
1. Send a successful request you've made in the HTTP history to Intruder from your newly found API directory. ie: `http://vulnapi.thm/api/`
2. Update the payload to set up two position markers. ie: `http://vulnapi.thm/api/ § FUZZ1 § / § FUZZ2 §`
3. On the "Positions" tab, set the Attack type to **Cluster Bomb**.
4. On the "Payloads" tab, select 1 for the first Payload set drop-down, then select a Payload type of "**Runtime file**" and navigate to the directory you downloaded these text files to. Select "actions.txt".
5. Repeat step 3 by setting Payload set 2 to "objects.txt".
6. Now click the **Start attack** button and go get a coffee. This will take a while.

After a while, with any luck, you will start to discover routes to the API that you may, or may not, be aware of. Welcome to the world of API discovery through bruteforcing!

With practice and patience, you will find you can accomplish a lot through Burp Intruder. It's a powerful tool, and one of the reasons why you should buy the professional edition.

Now that we have discovered the APIs, it's time to learn how to attack them and find vulnerabilities.

Focusing on the API attack types that matter



Now that you have found the web API endpoints that you want to investigate, you need to think about how you plan to approach your penetration testing.

There are multiple API attack techniques that can be used when hacking APIs. Knowing the most common attack types helps you to stay focused and allows you to build a simple methodology that's easy to follow.

OWASP has done a pretty good job documenting this in their OWASP API Security Top 10 list. However, when first starting out there are a few API attack types you should focus on first. These include:

1. Broken access control
2. Broken authentication
3. Improper data management
4. Weak input validation
5. Improper assets management

Let's explore each of these categories of attack.

Broken access control

One of the most common attacks against API security is to exploit broken access control. This type of attack occurs when an attacker can access resources or data that they should not have access to.

One way to exploit broken access control is to guess the URL for a resource that you should not have access to. If the API does not properly check permissions, you may be able to view, edit or delete data that you are not authorized to see or change.

When I say "guess a URL", I am referring to the ability to alter the URL in a way to bring back other data. As an example, if accessing your personal profile is done at `https://target.domain/users/1001/profile`, where `1001` looks like a possible user id, would switching it to `1000` bring back the administrator's profile? Or could `1002` bring back another user's profile?

If so, this can usually emphasize access control issues.

When you penetration APIs, this is called *Broken Object Level Authorization*, or BOLA. If you are used to hacking web applications, this is also known as *Insecure Direct Object Reference* (IDOR). BOLA... IDOR... call it whatever you like.

Another way to exploit broken access control is by using a technique called "path traversal." This technique takes advantage of vulnerabilities in the web server's file system. By submitting specially crafted input values, an attacker can navigate through the file system and access files and folders that they should not have access to.

As an example, imagine an API that fetches reports from a path like:

```
https://target.domain/api/reports/1001/abc123.pdf
```

Through path traversal, it might be possible to grab the server's passwd file using something like this:

```
https://target.domain/api/reports/%2e%2e/%2e%2e
/%2e%2e/%2e%2e/%2e%2e/etc/passwd
```

Finally, attackers can also exploit broken access control by using session hijacking attacks. A session hijacking attack allows an attacker to gain access to a user's session ID and use it to gain unauthorized access to the user's account. In the case of APIs, this is typically done by stealing the user's authorization token, typically a bearer token in a JSON Web Token (JWT) format.

Speaking of JWTs, let's discuss another huge attack type, which is broken authentication.

Broken authentication

Authentication in APIs is the process of verifying that API users, apps, services, or devices (sometimes called the ***subject***) are who they say they are.

There are many ways that you can exploit broken authentication, including guessing passwords, abusing leaked API keys, stealing session IDs, or exploiting vulnerabilities in the API authentication

process itself.

Broken authentication is a serious vulnerability that can allow you to gain unauthorized access to the target API. This type of attack occurs when you are able to bypass the authentication process and log in as a valid *subject*.

This could be done in a number of ways, including:

- Abusing a poorly designed password reset process
- Credential stuffing. This can work because API users often re-use their old usernames and passwords on different sites. So if their credentials leak from somewhere else online, they may now be used against the target API.
- Stealing sensitive data like passwords or tokens that can be used for authentication that are stored in GET parameters. This may even allow for man-in-the-middle (MiTM) attacks.
- Abusing APIs that do not validate the expiration date of auth tokens such as session tokens and JWTs.
- Abusing APIs that do not validate the digital signature of JWTs, or allow them to be used without being signed using the *None* algorithm.

Knowing an API's authentication process can help you figure out the best way to attack it, especially if you have access to the user logs.

API authentication process

There are a variety of ways to authenticate APIs. The four most common methods are:

1. HTTP Basic Auth
2. API Key

3. OAuth
4. No/NULL Auth

HTTP Basic Auth

HTTP Basic Auth is a simple authentication scheme in which a username and password are sent with each request, usually, Base64 encoded. This means that the username and password can be easily decoded, which is a major security concern. If the API isn't using SSL, it means anyone who can capture the request can access the credentials.

API Key

API Key authentication is a process of authorizing access to an API by using a unique secret key that is generated for each user. It usually is a difficult-to-guess string of numbers and letters – usually at least 32 characters in length, although there is no set standard for this. It is typically passed in the API authorization header, or as an additional header like X-API-KEY etc. A common attack vector is to recover these keys directly from source code (ie: on GitHub) or reverse engineering apps that use the API where the key may be statically compiled in.

OAuth

OAuth2.0 is a popular authorization framework that allows users to authenticate to APIs using their existing credentials from providers like Google, Microsoft, Facebook, and Twitter. The OAuth2.0 protocol defines how these authentication requests are made and how the resulting access token is used. To use OAuth2.0 for authentication, you first need to register your application with the chosen provider. Once your application is registered, you will be given a client ID and

client secret. These credentials are used to generate an access token (typically a JWT), which is then used to authenticate the requests to the API.

No/NULL Auth

Sometimes, an API may not use any authentication at all. This typically is seen when apps are expected to communicate internally, or when they expect to use an API management gateway for API security and is misconfigured.

Understanding how the API authentication process works allow you to manipulate those requests accordingly. Luckily, Burp does a great job exposing this and managing auth tokens for you. You can even leverage Burp extensions like Autorize, AuthMatrix, and JSON Web Token that can help you test and abuse authentication and authorization issues in APIs.

Improper data management

API developers must be careful to properly manage the data that they expose. If they don't, it can lead to a number of security issues.

There are two key ways that you can exploit improper data management. The first is by exfiltrating additional data that is returned when it shouldn't be – this is commonly called an "*excessive data exposure*" vulnerability.

The second way is to take advantage of how developers read and save data objects in their API. When they aren't careful it becomes possible to overwrite objects in a way to gain additional access, tamper with data, and bypass security mechanisms. This is typically called "*mass assignment*".

Excessive Data Exposure

Excessive data exposure occurs when developers inadvertently return more data than they should.

Modern programming languages these days like to work with *objects*. These are typically structured in a data model schema that makes it easy to move data around the service. Since most APIs are seen as direct data sources, developers sometimes implement them in a generic way without thinking of the sensitivity of the exposed data.

They expect the front-end application to filter out the information they don't need. And we can abuse that, giving us access to additional data we shouldn't have.

Chained together with other vulnerabilities like BOLA, we have seen major data breaches of popular services because of this. From leaking credentials to credit cards, looking for an excessive data

exposure vulnerability is a great place to start when hacking APIs.

Mass Assignment

So remember when I mentioned how developers like to work with *data objects*? Well, this pattern can be abused for more than information disclosure. If the API uses these same objects when creating and updating records, we can exploit this to tamper with the data. This might allow us to leverage this flaw for privilege escalation, or even entirely bypass built-in security controls.

As an example, consider a typical "User" object. It might contain things like your *name*, *email address*, and *password*. It might even include a field listing if you are an admin or not. Let's say for this example the field is a boolean (true/false) type called *isAdmin*. Now imagine that during the creation of a new user (ie: on sign up) it automatically sets it to *false*. It is here where we might be able to abuse mass assignment.

As the attacker, you could replay that user creation (ie the POST request) but alter the object to and set *isAdmin* to true. The result? The new user becomes an admin, even when they shouldn't. Securing APIs is hard when developers reuse and generalize data like this.

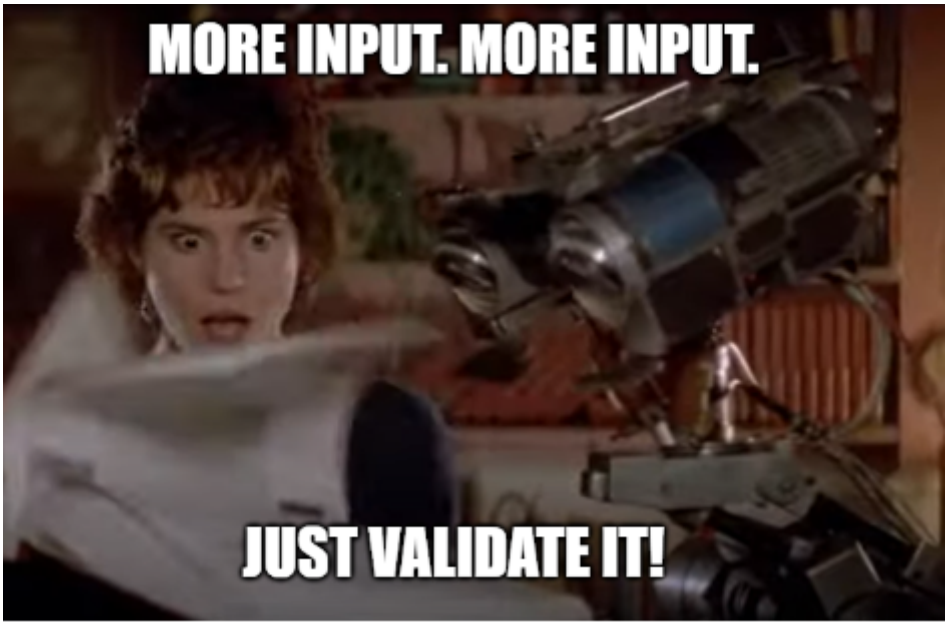
This becomes even more exciting when the API is vulnerable to excessive data exposure. When you can see the entire data model and understand how things are being stored, you can leverage all this improper data management to really abuse how the app works.

I've personally used these vulnerabilities to gain complete control of critical infrastructure, simply because developers weren't validating

the objects I was sending... giving me complete control of how the system was working.

Speaking of validation, let's go deeper into how weak input validation can be abused to do more than just tamper with data and bypass security controls.

Weak input validation



When it comes to writing secure code, a key rule is to NEVER trust user input and to always validate it as it crosses trust boundaries. However, it's common in APIs for developers to not heed this when working with data objects, which leads to injection vulnerabilities.

This can cause all sorts of issues. Here are just a few injection vulnerabilities commonly found that can be abused:

- SQL / NoSQL injection – you can alter the data in a way to manipulate queries allowing you additional access to the datastore or even allowing you to run commands directly from the database.
- Command injection – you can alter the data in a way to run

arbitrary commands on the host operating system.

- Template injection – you can alter the data in a way to confuse the template engine and get it to run code on the web server. This is usually called Server-Side Template Injection (SSTI).
- XSS injection – you can alter objects in a way to abuse how consumers of the API may render data in their front-end apps, taking advantage of potential cross-site scripting vulnerabilities.

It's very dangerous when developers don't sanitize input as it comes in and out of the API; not doing so allows you as the attacker a way to directly manipulate how the application functions and can even lead to a foothold on the server infrastructure.

Improper assets management

One of the benefits of modern API development is the agility and speed at which new code can be deployed. To ensure backwards compatibility these APIs typically use a versioning schema that allows older versions of the API to run in parallel with the new code.

We can take advantage of that.

As new APIs are written, data models may change. As will the way it's managed. At times you may be able to abuse that by calling an older version of the API which may be unpatched and not regularly maintained.

These are sometimes called *zombie APIs*.

Another problem is that of the *rogue API*. This is an API that was

written, but not properly documented or registered as an official API for the company. These rogue APIs can be dangerous because they may have never been security tested and could contain all sorts of vulnerabilities. Worse, if they are built by external third parties, the original data owners of the underlying APIs may not know their data is being exposed in this manner.

Zombie APIs... Rogue APIs... oh my.

Zombie APIs

The benefits of attacking zombie APIs come in the fact they are usually pretty static. They are typically documented, and you can compare versions to see what changes. And of course, you can use that as a recipe to see if you can manipulate requests based on those changes.

As an example, if you notice that new fields are added into a data object, but which require higher levels of access to read, see what happens when accessing those fields from an older version of the API. Sometimes security checks in new APIs don't exist in the older versions, which means if they read or write objects through the data store directly you might be able to tamper with the data through a mass assignment vulnerability.

This could lead you to gain additional access to data fields (information leakage) or allow you to manipulate the way the data and app works.

Rogue APIs

What makes rogue APIs dangerous is that many times they exist without the data owners even knowing. As an example, CRM data

may be exposed to a third-party marketing platform that gives them far too much access to the underlying customer records. This was exactly how major breaches from the likes of Facebook and Salesforce have occurred in the past.

Another example of rogue APIs comes in the form of dev and test instances of APIs that may not have the same security controls as the production instances. Stumbling upon these instances during your recon may allow you more time and access to abuse the API without fear of the NOC/SOC finding out.

Conclusion

In many ways, hacking APIs is very similar to hacking web applications. What differs is in the way developers trust APIs to move data around. If you can understand how they do that and have a clear understanding of how they work with the objects underneath, you can approach the target in a more offensive manner when you penetration test APIs.

This is a beginner's guide. I've shown you some of the key areas where APIs are typically weaker. Authentication. Authorization. Weak data management. Even weaker input validation. And of course, abusing those zombie and rogue APIs.

I encourage you to target intentionally vulnerable APIs in your own lab environment and practice abusing the multiple API attack techniques covered here. Then you can focus on web API security testing of your own APIs and infrastructure, or consider working with companies that offer bug bounties that have APIs in scope. HackerOne and Bugcrowd offer great search engines for their bug bounty programs you can use to find programs that might fit you.

Once you have that under your belt, you can start looking at finding vulns and reporting them. You might want to check out my article on exploiting APIs with cURL for a good starting point on how to write decent proof-of-concept (PoC) exploit scripts to attach to your reports.

As you continue hacking APIs there are tons of online resources that can help you get better at your tradecraft. I have consolidated some of the better ones into the most comprehensive guide to API hacking resources, which I continue to maintain for the community.

Download the free PDF of The Ultimate Guide to API Hacking Resources today!

P.S. Don't forget to check out my latest articles on API hacking.

DANA EPP

Hey, I'm Dana, aka SilverStr. I build and break software for a living, and am a Microsoft Regional Director and Developer Security MVP. I've spent decades as a security architect that focuses on helping secure software, data, and infrastructure on both blue and red teams. As of late, I have been focusing more on my offensive tradecraft to help developers and IT administrators see the impact of exploitation on vulnerabilities in their work. This blog is my chance to give back to the community by sharing my

experiences and war wounds from the trenches.

API Security Testing: How to Use OWASP guidance as your blueprint →