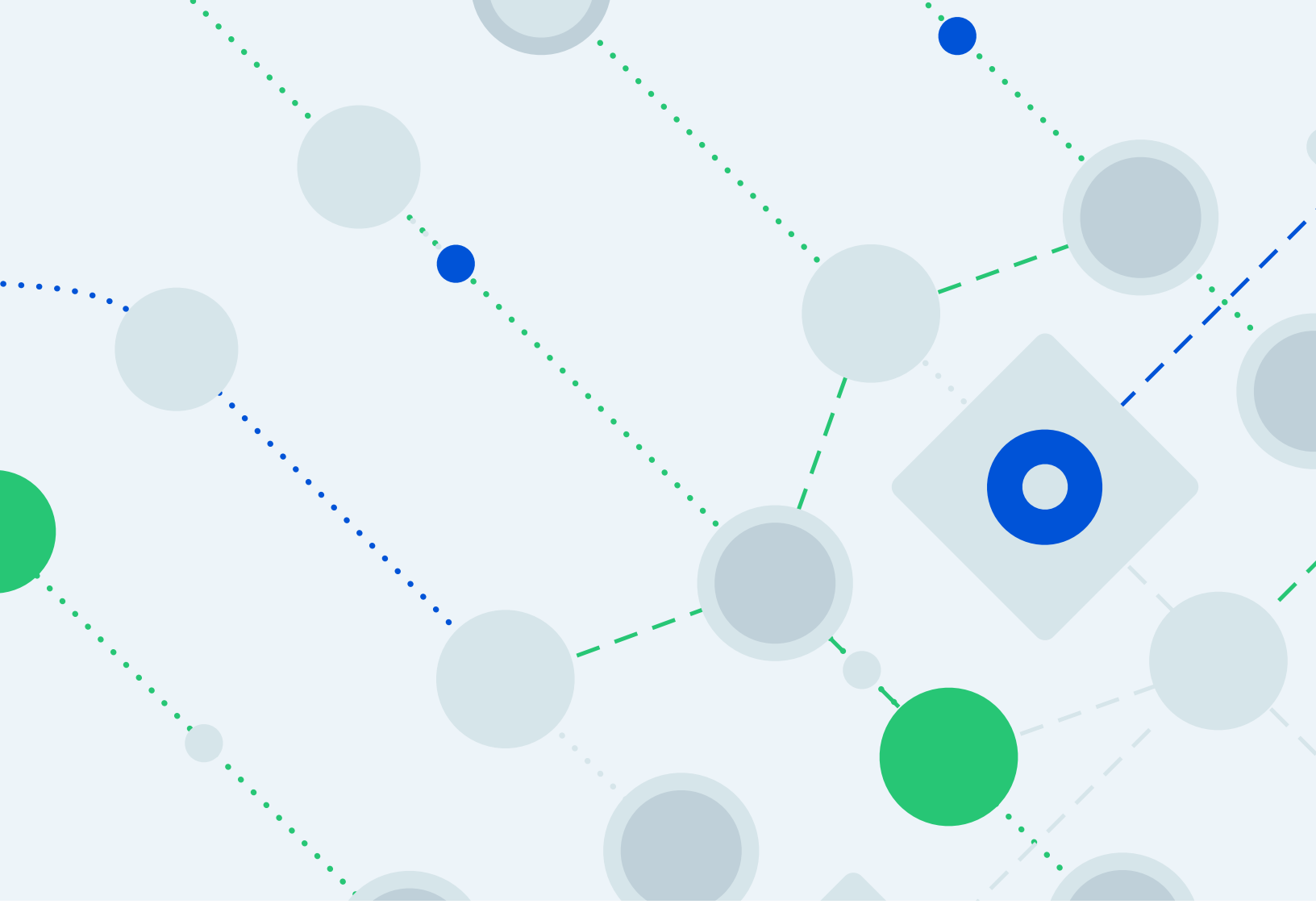


AuthN, AuthZ, and the Growing Menace of API Breaches

TRACEABLE

v 1.1

As distributed architectures become more popular, new API security vulnerabilities are on the rise. Here's how to design authentication and authorization systems to protect web applications from attack

A famous criminal was once asked why he robbed banks. "That's where the money is," he reportedly answered.

In today's computing environment, the cloud and cloud-native applications are where the money is, metaphorically speaking. For cybercriminals this means the tried-and-true methods used to breach traditional monolithic applications are in many ways no longer up to the task.

They've risen to the challenge. Attackers have upped their game to take on modern distributed web architectures. One trick: Use misconfigured or incomplete authentication and authorization systems as their entry — like walking through the bank's front door to get to the vault.

We'll explore this modern frontier of cybercrime, understand how incorrectly used authentication and authorization systems that guard web interfaces can be exploited, and learn what organizationscan do to strengthen their defenses against serious API attacks.

Authentication Versus Authorization

Let's compare the differences between traditional application development with application programming interface (APIs) and cloud-native development. These fundamental differences affect the tools and techniques used to secure these applications.

Authentication is the process of verifying a user's identity. Essentially, it means making sure that a user is who they say they are.

Authentication can be implemented using one or more of the following methods:

1. What a person knows (password or passphrase).
2. What a person has (one-time token or physical device).
3. What a person is (biometrics, fingerprint reader, facial recognition).

Authorization, by contrast, is ensuring that a logged-in user has the right to perform specific actions or view certain data. For example, you may have access to view your own personal information through a web interface, but you shouldn't be able to see other users' data or have access to administrative functions.

Both authentication and authorization are necessary for an application to be secure. Getting into a party (authentication), however, doesn't automatically get you access to the VIP lounge (authorization).

You may see authentication abbreviated to authN and authorization abbreviated to authZ. These are shorthand terms often used in the industry and are interchangeable with the longer words.

AuthN and AuthZ are different for APIs

Now that we understand the differences between authentication and authorization, it's time to dive into how they are different for APIs. There are three reasons.

Reason 1: APIs are Distributed, Not Monolithic

How a web application appears to the end-user doesn't reflect all of the pieces used to deliver its functionality. They experience a single interface that hides the complexity underneath. There could be hundreds of small microservices distributed in data centers worldwide doing the work necessary to display everything on the page or in your app.

Monoliths. Web applications were once monolithic. Essentially, that meant they existed as one chunk of code running on a server. The server did most of the work, and one page or deliverable was passed to the browser at once.

The authentication and authorization mechanism in such a site is simple. After the user logs in to the website, a single database holding user information verifies their identity. A session is created on the server, and all subsequent requests use the session to identify the user without another login required.

The rise of development frameworks made this process even easier for developers. Many frameworks handle session management out of the box, so developers didn't have to think much about it apart from wiring up the essential pieces.

Distributed. Fast forward to today where web applications consist of microservices distributed in cloud data centers. Each microservice is a self-contained server and data store bundled together but separate from the application's other functions. A client application, the one the user interacts with, makes API calls to the services it requires to do its job.

Authentication and authorization look entirely different under this new distributed model. Since each microservice has a data store, a session created in one has no meaning to another. API calls would constantly break if the application depended on a single session ID created by the first server an application happened to call.

Distributed APIs require a new way of distributed authN and authZ.

Reason 2: APIs Are Technology and Platform Agnostic

For many years a company was a "Java shop" or ".NET shop" using only those technologies. Now, developers use many frameworks and languages across the enterprise.

Developers create microservices using frameworks and languages that make sense for the problem they're solving. One microservice uses NodeJS with a MongoDB database. Another uses Scala and GraphQL. As long as each service adheres to the API it publishes for others to use, the implementation doesn't matter.

These differences between languages and frameworks are another reason why authentication and authorization must change for APIs. Each language and framework has its own session management implementation, and every microservice has a different datastore.

AuthN and authZ technology for APIs must work for any programming language.

Reason 3: New Technologies and Development Techniques Lead To New Vulnerabilities

New technologies and development styles traverse a repeating cycle of security. First, a new technology appears on the scene that solves a problem. It catches fire in the industry as more people discover and use it.

Unfortunately, when a new technology catches fire, security can become a secondary concern next to the problem it solves. Also, it's not evident how to secure the technology because it's unclear how attackers will break in or what vulnerabilities exist.

That's why malicious actors often have an initial advantage. They pick apart the technology and find new ways of breaking into applications and systems. The industry scrambles to catch up and seal the vulnerabilities discovered.

Traditional web applications have well-known weak spots. Over the years, many frameworks have built out-of-the-box protections for the most common of them. For example, frameworks such as Angular and .NET have built-in protection against cross-site scripting and cross-site request forgery.

The concept of APIs has been around for many years. But the technologies used to enable the recent boom in microservices are relatively new. These include:

- Containers
- Service meshes
- Container orchestration (i.e., Kubernetes)
- Service buses
- Serverless computing
- Cloud computing.

As these technologies are implemented, new ways to get around their defenses emerge. Along with the technologies themselves come new challenges with the logistics of distributed, cloud-native architectures.

As a distributed architecture becomes more popular, the need for new authentication and authorization methods increases. These methods aren't immune to the repeating cycle of security.

The Danger of Broken Authentication and Authorization in APIs

Technologies used to create web applications have fundamentally changed. Authentication and authorization techniques have to change with them.

We've discussed three reasons why:

- APIs are distributed, not monolithic.
- APIs are technology and platform agnostic.
- New technologies and development techniques lead to new vulnerabilities.

What happens when APIs have broken authentication and authorization?

Case Studies: The Consequences of Poor Authentication and Authorization Practices in APIs

Let's turn to real-world API vulnerabilities. Look through the examples to understand how attackers can breach your defenses and what to look for when designing and building your authentication and authorization system.

Shopify Breach #1: Broken Object Level Authorization in Kit App

You can find details on this vulnerability on Hacktivity. A hacker named Sandie found the flaw via bug bounty and was awarded $1,000 for his efforts. This vulnerability is a textbook Broken Object Level Authorization (BOLA) flaw, which happens to be No. 1 on the OWASP API Top 10 list.

BOLA occurs when an attacker changes an ID parameter in a Uniform Resource Identifier (URI) to view a resource they are not authorized to view. Imagine a medical application where you can change a parameter and see someone else's medical records.

The vulnerability found in Shopify's Kit app isn't quite as catastrophic as exposing medical records but still illustrates how BOLA can slip into any API.

The Kit app is an automation tool Shopify store owners use to manage marketing tasks like Facebook ads, emails, and integrated apps. The vulnerability allowed an attacker to obtain the authorization token for a high-permission user of Kit using a low-permission account.

The attack begins with a URI requesting an authorization token for Shopify Ping to talk to Kit.

/api/v1/arro_token?access_token=[redacted]&myshopify_domain=alwayzhack.myshopify.com&id=42668326968

This endpoint generates a token for the given account ID. The API expected that the currently logged in user would send their ID to the API, and all is well. Unfortunately, the hacker discovered that passing a high-privileged user's ID into this endpoint would return an authorization token for that user. The attacker could use this token to send requests to Kit as an administrator and view previous messages.

This type of vulnerability could be hazardous for any API. It allows an attacker to impersonate an admin-level user and perform any action they desire.

How can this attack be prevented? The issue is the "id" parameter used to identify the user requesting the token. The API assumed that the ID passed in was the ID of the corresponding user. Instead, it should've checked every request to ensure that the current user ID matched the ID given and that the ID passed in had authorization to operate the app or function.

Shopify Breach #2: Anyone Can Become a Collaborator Without the Store Manager's Permission

The appearance of another Shopify vulnerability isn't an indictment of their security. It's a compliment. They've embraced thorough testing to find these issues and disclose them so others can learn from these mistakes.

This Shopify bug exemplifies another vulnerability on the API Top 10, Broken Function Level Authorization (BFLA). BFLA occurs when non-privileged users can perform certain privileged operations.

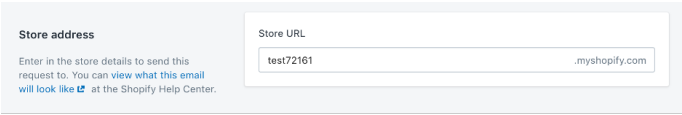
Shopify's partner program allows service providers to help store owners with tasks such as store design, build, and marketing. Typically, the collaborator enters the store URL they want to be associated with, and the store owner approves the request. Upon approval, the browser sends a request to a specific endpoint: **/admin/settings/account/approve/<id>.</id>**

Collaborators have full access to perform any action on the store, including reading customer data, changing inventory, and more. A single expert can be a collaborator on different stores.

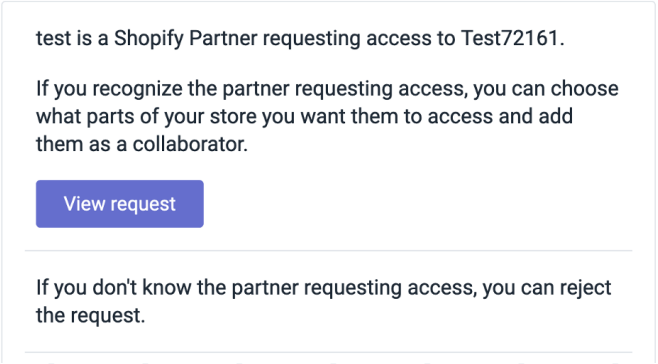
A security researcher named Uzsunny discovered that this endpoint didn't check if the API call came from a store manager. Any user could call the endpoint and approve the request, opening the door for anyone to give themselves administrative access to any store on the platform.

The process of requesting "collaborator" access to an existing store contained three steps:

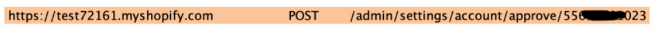
1. The expert enters the store URL.



2. The store manager receives an email about the access request.



3. Once the store manager approves the request their browser sends an API call to:

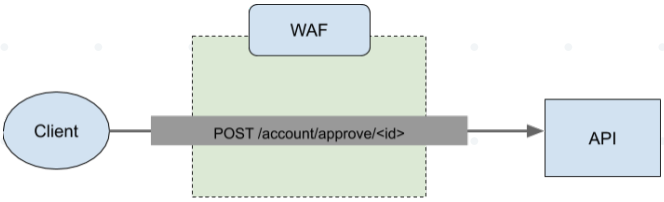


This API call approves the access request by the expert and converts him to a "collaborator."

The problem: The code did not validate that the API call was triggered by a store manager. In fact, any user could call the API endpoint and approve the access request, even if they don't have the right privileges.

Using this technique Uzsunny managed to "login to any store with full permissions." For his troubles, Uzsunny received a $20,000 bounty from Shopify, which reported: "We tracked down the bug to incorrect logic in a piece of code that was meant to automatically convert an existing normal user account into a collaborator account."

The exploit in step #3 looks simple and requires only a single HTTP request: "POST /admin/settings/account/approve/<id>". Authorization exploits usually look legit from a WAF/RASP perspective; they don't contain suspicious payloads or characters, weird HTTP headers, or abnormal structure. In fact, if the same exact HTTP call was sent by a different user who is a store manager it would be completely legit.</id>



In order to understand authorization exploits, a much broader context is needed. It's simply not enough to look at a single HTTP request.

BFLA flaws are sneaky and can be challenging to find. They usually exist due to a missing authorization check in the endpoint code or as the result of assumptions that come back to bite you.

Facebook Breach: Password Recovery API Allows Access To Any Account

Facebook had an authentication bug that allowed anyone to take over an account. It was an exploit of the password recovery functionality.

As part of a bug bounty program, the AppSecure cybersecurity research team found a vulnerability on the authentication mechanism of Facebook. It gave them the ability to potentially gain full control of the social media giant's more than 1 billion users. The team won a $15,000 bounty for its discovery.

This vulnerability was found on a niche API, which reminds us that in many cases the most interesting bugs don't exist on main APIs but on secondary ones that have fewer protection mechanisms in place.

Here's how it worked.

1. The user starts the "forgot password" process by using their email address.

2. Facebook sends a text message with a temporary 6-digit secret token to the user.

```
POST /recover/as/code/ HTTP/1.1
Host: facebook.com
secret=<6_digits_temp_code>
```

3. The user enters the received temporary code, and the browser sends an API call to "POST facebook.com/recover/as/code/" with the secret token.

This endpoint implemented rate limiting to prevent attackers from brute-forcing the reset code. However, several endpoints under the beta.facebook.com and mbasic.facebook.com domains hadn't enabled rate limiting.

An attacker could take over any account using this process:

1. Enter the victim's email address on the password recovery page.

2. Brute force the six-digit reset code by sending requests to the beta.facebook.com and mbasic.facebook.com endpoints.

Facebook had implemented an anti-brute force mechanism on this API that blocked the user after 10 failed attempts. However, during their research, the AppSecure team found that the same API endpoint existed on different API hosts, under "beta.facebook.com" and "mbasic.beta.facebook.com." These API hosts didn't implement the anti-brute force mechanism, allowing the attacker to easily iterate through the secret token and reset the victim's password.

Anand Prakash, CEO of AppSecure and credited with discovering the bug, explained what he found. "This gave me full access to other user accounts by setting a new password. I was able to view messages, their credit/debit cards stored under their payment section, personal photos, and other private information."

Even though this vulnerability was primarily due to forgotten endpoints containing old code, it illustrates that authentication processes, especially "forgot password" functionality, are prime targets for attackers. Any hole in your authentication processes could lead to a catastrophic breach.

Even though the vulnerability was discovered in 2016, similar weaknesses have been discovered — and in many cases exploited — ever since.

Uber

Uber Breach: Exploiting an API Authorization Vulnerability

In September 2019 a critical bug was discovered on Uber API, It allowed merchants, service providers, and others to offer ride-sharing services to customers. Uber had exposed a vulnerable API endpoint that allowed attackers to steal valuable data, including personally identifiable information (PII) records and authentication tokens, of riders and drivers. A leaked authentication token could be used to perform a full account takeover.

Luckily for the company, the vulnerability was discovered before harm could be done. But the case is another example of where traditional security systems fail to find potential threats because they lack the business context for the application's logic. Let's take a closer look at what happened and the implications.

When a new Uber driver joins the platform through a referral link, their browser communicates with the API host "bonjour.uber.com". The registration process triggers an API call to the API endpoint of:

POST /marketplace/_rpc?rpc=getConsentScreenDetails

```
POST /marketplace/\_rpc?rpc=getConsentScreenDetails HTTP/1.1
Host: bonjour.uber.com
Connection: close
Content-Length: 67
Accept: application/json
Origin: [https://bonjour.uber.com](https://bonjour.uber.com)
x-csrf-token: xxxx
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML,
DNT: 1
Content-Type: application/json
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: xxxxx
{"language":"en","userUuid":"xxxx-776-4xxxx1bd-861a-837xxx604ce"}
```

The API receives the "userUuid" parameter from the client and returns details about the user. This information is used to populate the consent screen on the client-side:

```
{
   "status":"success",
   "data":{
      "data":{
         "language":"en",
         "userUuid":"xxxxxx1e"
      },
      "getUser":{
         "uuid":"cxxxxxc5f7371e",
         "firstname":"Maxxxx",
         "lastname":"XXXX",
         "role":"PARTNER",
         "languageId":1,
         "countryId":77,
         "mobile":null,
         "mobileToken":1234,
         "mobileCountryId":77,
         "mobileCountryCode":"+91",
         "hasAmbiguousMobileCountry":false,
         "lastConfirmedMobileCountryId":77,
         "email":"xxxx@gmail.com",
         "emailToken":"xxxxxxxx",
         "hasConfirmedMobile":"no",
         "hasOptedInSmsMarketing":false,
         "hasConfirmedEmail":true,
         "gratuity":0.3,
         "nickname":"abc@gmail.com",
```

As part of a legit flow, the user should send only their own user ID.

The API endpoint is susceptible to two types of API vulnerabilities:

- BOLA. Because the program didn't validate that the client sending the request had access to the user represented by the user ID parameter, the client could access data of other users by changing the user ID.
- Excessive data exposure. The API response contained a large JSON object with all the user's details. The API returned this information, even though the client didn't use it.

These examples of real-world exploits show what can happen when companies don't implement authN and authZ correctly. To be clear, it's not a case of authN and authZ systems being exploited, but rather that they are not being used properly with APIs by developers. As long as humans are writing code, mistakes can creep in. But there are steps you can take to build secure web APIs and reduce risk.

API Authentication and Authorization the Right Way

Teams need to address three core elements to develop a simple yet scalable model for API security: safely managing logical state, support for distributed architectures built on containers and microservices. and enabling a web of authentication for linking loosely coupled services.

Modern tools and frameworks can address all three of these through the appropriate combination of the OAuth 2.0 Framework, OpenID Connect, and JSON Web Tokens (JWT).

Manage the logical state. Traditional web security evolved to simplify the user experience so developers found a way to use session cookies for managing the authorization state of a user. This reduced frustration with having to log in repeatedly; users only had to enter passwords once, or more if the credentials were stored. However, session cookies were vulnerable to session hijacking attacks that took advantage of limited security around cookies. A better practice is to securely manage the logical state using tokens instead of cookies.

Need for distributed authorization methods. Web applications assumed that one browser client would access one web application connected to one or more databases. The web page would be assembled in the middle and then passed to the user. But modern applications allow new architectures in which one client, like a mobile app, builds the user interface from multiple APIs. Each API, in turn, may manage interactions with multiple microservices. Early security frameworks like OAuth 1.0 supported direct access but did not scale for distributed architectures.

Web of authentication. Distributed security needs to strike the right balance between the number of authentication efforts on back end servers and the overhead and latency each call adds. In a complex authentication flow the client authenticates to an initial service, which in turn authenticates with another back end service, and so forth until the request is completed. One strategy is to use public-key cryptography to allow each service to validate new requests locally using a chain of interconnected public-keys on top of OpenID Connect.

OAuth 2.0 Provides Distributed Authorization

As websites began to take off, so did the number of security schemes for simplifying access using session cookies.

In late 2006, Blaine Cook, the chief architect at Twitter, began dreaming up the framework for a more generic approach that could be shared across websites, which evolved into OAuth 1.0. Unfortunately, there were ambiguous elements that could be implemented differently, and there was quite a bit of disagreement between mainstream websites and enterprise vendors on how it would work.

One big challenge was that the authentication scheme was baked into the specification, making it hard to support applications like mobile or microservice design patterns. So, work began on OAuth 2.0 spec, which was more generic but also lacked support for a specific way to manage the security state. OAuth 2.0 only shares the goal of OAuth 2.0 and is not backward compatible.

OAuth does a better job separating the roles of security servers and authorization servers. It introduces the notion of a client, authorization server, resource server, and resource owner. This makes it easier to describe the authorization flow that can protect sessions from being hijacked and reduces the threat of business logic attacks on the back end server.

There was some contention with OAuth 2.0 when vendors implemented different versions of the draft standard. Major vendors started implementing OAuth 2.0 after draft 10, and then another 22 revisions were made. Different vendors adopted parts of these that would not interoperate. Eventually, the maintainers of the standard pulled out the conflicting pieces and renamed the protocol a framework. Other pieces were required to support authentication, tokens, and claims.

Adding Authentication With OpenID Connect

To build consensus, many things were left out of OAuth 2.0, such as the token type and identity framework. OpenID Connect adds an interoperable protocol to OAuth 2.0. This complements OAuth's extensive library of flows used to manage access for sharing resources across services.

The significant innovation is that developers can authenticate users without creating and maintaining a separate password file. This improves security since these files are sometimes compromised. It is the third generation of technology. The first version was not widely adopted. The second version, OpenID 2.0, was more fleshed out but difficult to implement since it relied on XML.

OpenID Connect is much simpler and takes advantage of JSON, making it more accessible to modern developers. Popular security libraries and development tools natively support OpenID Connect, which further simplifies implementation. OpenID 2.0 required a customer signature system that was problematic and prone to errors. OpenID Connect introduced JSON Web Tokens, which are much easier to implement.

Replacing Cookies With JWT

In 2011 researchers began exploring how JSON could simplify web security in the same way it simplified APIs. John Bradley and Nat Sakimura introduced a simple signing mechanism for JSON, which evolved into the JWT framework spelled out in RFC 7519 in 2015. The core spec talks about representing "claims" digitally encoded inside a JSON payload as a token.

The token structure includes a header, payload, and signature. The header indicates the type of token and the signing algorithm. The payload includes the cryptographically signed claims. The signature is a hash generated by applying the sender's private-key to the payload.

The tokens are used to encrypt data between parties in a way that hides it from others or for applying digital signatures that allow the recipient to validate the integrity of claims in a communication. A claim is any statement issued by the appropriate source that can be cryptographically verified. Claims can be used to verify who issued the JWT, that the appropriate subject uses them, that they are delivered to the appropriate recipient, and when they expire. They may also include publicly registered claim names (i.e., Google) in a special JWT database or private claim names for use in a restricted flow.

JWT provides several benefits over token schemes like Simple Web Tokens (SWT) and Security Assertion Markup Language (SAML). SWT required symmetric security, which complicated the authorization flow. Both SAML and JWT can use public-key cryptography in which a pair of public/private keys can verify the source and hide the data. JWT is also more efficient than SAML, which reduces the overhead and packet sizes. JSON also aligns better with JSON API techniques. They are also easier to process.

Common Use Cases

The most common use case is authorization. After a user or service has been authenticated subsequent communications can use the JWT to access services permitted for that user or service. It is commonly used as part of a single sign on implementation since it can be used across multiple domains.

Secure information exchange is another common use case. In these instances, the JWT is used to sign and encrypt a transmission using a private/public key pair. The recipient can verify the source, and that the data has not been tampered with by using the public-key and its own private-key to decode the message.

Scopes provide a way of limiting appropriate access to a subset of resources. For example, one scope would give you access to the free tier of a nifty customer relationship management (CRM) service, while another scope would provide access to all the extra features available on the gold tier. Scopes can also limit access based on who owns the data. The scope could limit access to view all the enterprise's customers stored in the CRM system but not see records created by others.

Better Security Through Multiple Approaches

Enterprises can roll their own security by combining the appropriate encryption libraries. But this can add additional overhead for maintaining and updating these components. A much better practice is to combine industry-leading frameworks and tools such as OAuth 2.0 for authZ, OpenID Connect for authN, and JWT to implement encrypted tokens. The combination of these is well documented and can provide the best framework for protecting the API infrastructure.

More importantly, enterprises can benefit from the wide use of these tools as new threats are discovered and new best practices evolve.

Enterprises should also consider how to protect the business logic that operates across this infrastructure. Microservice architectures can expose more API endpoints to outsiders. Modern API observability tools like Traceable can provide another layer of protection at the business logic level that might be blocked by traditional authN and authZ tools.

Why Current Security Solutions Can't Detect It

Let's tackle the obvious question. Automated testing tools are becoming increasingly sophisticated. Why can't they find authN and authZ vulnerabilities while developers are writing code or in a testing environment?

Security testing tools like Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) aren't effective at finding authentication and authorization vulnerabilities. SAST tools scan source code with no knowledge of the architecture or business logic. DAST tools are great at finding vulnerabilities against running applications, but complicated business logic escapes them.

Interactive Application Security Testing (IAST) is probably the best option for discovering these types of vulnerabilities but still not a100 percent solution.

In addition, testing tools can't predict what unique set of steps might lead to a compromise. Authentication and authorization systems are complex with many steps.

There is frequent back-and-forth communication between the client and API. There are password recovery systems, login implementation, and service-to-service authentication. All of these moving parts lead to unexpected interactions.

When we look carefully at the exploitation flow in the Uber case, we find the attack involved a very subtle change in the API call, replacing one user ID with another. In order to detect such a small change in the traffic, a deep understanding of the business logic of the app is required. Many security solutions don't understand business logic.

How Traceable Solves the Problem

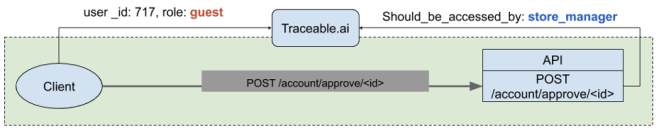
Keeping track of your APIs is not an easy task. Modern organizations might have dozens or even hundreds of API hosts for different environments, regions, or versions. Each API host can expose multiple endpoints related to authentication processes including login, forgot password, and one-time login link.

Many security solutions focus on protecting a system's main APIs and don't have enough visibility into less common or less used APIs, such as Facebook's beta API in the example above. Sophisticated attackers choose to target those niche APIs.

Our approach to detecting API attacks is very different from other solutions on the market. In a nutshell, we observe the data that passes through the API and the microservices of the app. We then use machine learning algorithms to discover the application's business logic.

We get full visibility into the users and their roles, the API endpoints they communicate with, and the resources the endpoints interact with behind the scenes. Now we can create a baseline understanding of a legit user's flow through the system.

The visibility into users helps us understand that the client is actually a guest user, and the visibility into the API helps us to recognize that the endpoint is an admin endpoint that should be used only by store managers. Then we can simply block malicious abnormal activity.



With this approach we can detect the most sophisticated and subtle API attacks, including Broken Function Level Authorization.

Current security solutions in the market lack that understanding, usually acting in the context of a single HTTP request. They don't understand deeply enough the important components of the application required to detect BOLA and other authorization vulnerabilities.

Conclusion

Even though the transition to digital was promoted to organizations with claims that the cloud would be more secure than traditional on-prem and network computer systems, it turns out that this was only partly true.

As is always the case, clever cybercriminals have adapted to the challenges of breaking into distributed architectures, in part by exploiting missing or incorrectly implemented authN and authZ protections.

Now it's up to your security system to leapfrog ahead. Again.

About the Authors

Justin Boyer, George Lawton, and Inon Shkedy contributed to this white paper.

About us.

Traceable was founded by third-time entrepreneur Jyoti Bansal and Sanjay Nagaraj. Bansal and Nagaraj saw the massive adoption of cloud-native architectures firsthand during their time at AppDynamics and founded Traceable as a result to protect applications from next-generation attacks.

Traceable applies the power of machine learning and distributed tracing to understand the DNA of the application, how it is changing, and where there are anomalies in order to detect and block threats, making businesses more secure and resilient.

Traceable.ai

TRACEABLE_