Abstract

Tokens have been the de-facto means that support scalable and performance-focused security solutions in the current technological environments. There are currently many options and operation modes for anybody thinking of implementing a token-based security measure with support of cryptographic mechanisms. With the web-attack landscapes changing to APIs, and with APIs managed as a decoupled data handling entity, APIs need to have stringent authorization decision engines that ensure an authorized entity requests each data retrieval. This article is mainly designed for a software architect to help understand the problems tokens tend to solve, with illustrations of different types of tokens in use. The article also explains the token implementation methods for an enterprise API mechanism and provide security best practices to be taken care of when implementing the same.

Authentication (AuthN) and authorization (AuthZ) have always been the essential security controls in a programmable system or entity. The program needs to identify who is using it and what features of the program are unrestricted to the user. There are many commercially available products, both as a software and a service, to handle the authentication security controls. The security community has striven to set many authentication policy definitions, such as password policies, user lockout policies, captcha frameworks, and provided many scan engines to check the implemented authentication policies. Enterprise organizations have prioritized their strategic plans concerning security by utilizing the best products to support the applications’ authentication and password handling requirements.

Implementing single sign-on with compulsory multi-factor authentication sequences has become a de facto for applications and web components in an enterprise environment. Also, organizations have taken the acceptable password policies seriously, and have taken steps to ensure their users are using only adequately complex passwords for their accounts. The password rotation policies in enterprise organizations are now aptly defined, and employees lose their access if the passwords are not changed before they expire. Furthermore, organizations have taken additional precautions to store sensitive information such as passwords, tokens, and private keys in centralized key storage locations, and procure them only during runtime. From an authentication perspective, there have been significant advancements in recent years,
and enterprise organizations are taking additional measures to close security gaps. So, does the buck stops here?

**Traditional authorization sequences**

One of the common security gaps prevalent in current enterprise environments is not to effectively build authorization definitions in API endpoints. Many API endpoints still use server-based authentication and authorization using basic authentication or master tokens for accessing APIs and other external resources. In this architecture scheme, the application or an API client uses a single username/password pair or a long-lived token to access the APIs. This design results in users having full control of the API endpoints, once authenticated, eventually making it impossible to define and establish authorization rules. Figure 1 considers how API endpoints authenticate using server-based authorization via master tokens or basic authentication [4].

![Figure 1 – Authentication to API servers using Basic Auth or master tokens](image)

Following are some of the security loopholes that may result from applications/API endpoints utilizing server-based authentication:

- **Inadequate checks in the API or data layer**: Data entities such as the API may never be able to deduce who is requesting the data and which part of the application is utilizing it. If the application has more than one role, like user and an administrator, the API entity cannot limit what data should be available in the response, and thus blindly sends all information back to the application. Eventually, the application decides whether to show this data to the end user or not (depending on the user’s role authenticated to the application front-end).

- **Not effectively utilizing the existing single sign-on**: Single sign-on (SSO) deployments at enterprise organizations support many approaches for APIs to authorize the user and retrieve the data. Using service account credentials or master tokens means the APIs would lose the authenticated users’ session context, eventually missing the point of utilizing SSO mechanisms for internal applications. Since most of the enterprise application’s authenticated users are under the SSO context, the authenticated users’ session context should be forward to the underlying APIs for them to authorize the user, and return appropriate data.

- **Growing dependency on passwords or master tokens**: The dependency on passwords or keys would limit developers from not being able to provide features that require some level of automation. This limitation eventually results in an administrative overhead of securing and recycling passwords. To overcome this limitation, developers store the passwords or master tokens and erroneously publish in GitHub. Furthermore, if attackers steal these passwords or master tokens, they would be able to retrieve all the data provided by the API.

- **Growing dependency on front-end applications**: If APIs do not handle authorization mechanisms on their own, they eventually depend on the front-end web or mobile applications to dictate the authorization sequences. To fix the created design and security gaps, architects would eventually make the APIs entirely dependent upon external ap-
Applications for all security-related activities, thus changing the established trust boundaries. If any strategic changes to the APIs are outlined, for example, to support or service a mobile application, the APIs should be positioned to be Internet facing. Eventually, the trust boundaries break, and they now have a very minimalistic authentication and no authorization mechanism.

- **Scalability limitations**: Due to the dependency of sessions, the application's scalability factor will always be at stake. With seasonal or incremental traffic influx, applications should have a way to increase the number of machines it uses or its computational power. If an application is dependent on user sessions, the mechanisms to add new devices to support traffic cannot be implemented or will be cumbersome.

- **Lacking defenses against phishing attacks**: If an application uses external authentication/authorization sequences, and if generated master tokens are used, the applications would never consent from their users to use the feature or perform actions on users’ behalf. This design gap can be utilized by attackers to formulate a successful phishing campaign, and eventually implement complicated cross-site request forgery attacks.

So, the solution is to ensure that we are utilizing short-lived dynamic tokens on all our APIs. Dynamic tokens (user tokens) make the APIs have the user context and are aware of whom they are serving. Hence, they can also be used to define and employ more stringent access controls, based on the API methods or functions or the data that those API methods are rendering back. Furthermore, dynamic tokens are valid for a short time, and hence, even if attackers gain access to them, they would not be able to gain full control of the application or retrieve all data. We have been using dynamic tokens in our day-to-day life. If you have authenticated to a third-party application by using your Google or Facebook account, then you have used dynamic tokens that are short lived.

**Introducing security tokens**

Security tokens are a random set of bits that are generated by a centralized tokenization system, whose randomization can be utilized to identify a person/entity or enable a feature for a specific object. In recent years, the use of tokens is primarily to reduce complexities in implementing security controls, which otherwise would have been cumbersome. In addition to providing security benefits, tokens have helped attain the user experience level developers plan for their applications.

Although security tokens are a random set of bits, their applicability has been vast in recent years. They are used to attain the “confidentiality” aspect of the CIA triad in cryptography, and in recent times they have been extended to the “integrity” aspect as well. Security tokens are not just limited to implementing authentication sequences; they are also extensively used for supporting authorization requirements. One of the most common use of tokens in authorization requirements is to communicate user’s access levels to all downstream and dependent applications. Also, the conventional methods of session management, utilizing cookies, have been replaced with tokenization systems that ensure granular access controls and independent implementations.

**Tokens and security controls**

While designing the security controls in our applications, we need the right tokenization system for the right token content. Below is a list of common tokenization mechanisms that relate to what security controls or the security problem it solves:

- **Session cookie**: Technically a type of a token, session cookies are used to ensure that an application maintains the user session after the user authenticates. Since HTTP is a stateless protocol, the application requires the contents of the cookies to validate every request from the user. Session identifiers are stored as cookies, and the application’s session management sequence verifies whether the user session is valid or not every time the browser submits a request. The cookies are used extensively in server-based authentication and are generally extended further for other API endpoints.

- **Authentication tokens**: Authentication tokens usually are used instead of the username/password combinations. The first time a user logs into an application using credentials, an authentication token is provided. The user can subsequently use authentication tokens to log in until they expire. For the ease of usage, authentication tokens were helpful during the initial mobile application development, as the tokens were stored in the mobile application key-store instead of the user’s credentials. Currently, there are better means of tokenization techniques such as OAuth2 + OIDC, with support from Google API, Facebook API, and Apple API to authenticate through their ecosystem.

- **Authorization tokens**: Authorization tokens are provided by the application’s authorization servers to update the API server or downstream applications regarding access details about the authenticated user. The token generated can be easily decoded to review the user’s contents, domain, authorization levels (admin, user, read, write). The token contains a signature hash, which validates that the token contents (with authorization details) are not changed. JWT tokens are extensively utilized for this measure as by design they support signing/hashing techniques. These tokens are not designed to maintain secrecy (as for authentication tokens). These tokens are to be shared with downstream applications to ensure that they know the authenticated user’s roles and provide access to the required resources.

- **Access tokens**: Access tokens allow users to authenticate to a resource and attain required access to retrieve them. In a general sense, resources can be a target API or an external entity to retrieve data. They are also called as grant tokens to support other applicability or implementation mech-
anisms. This token is only provided to an authenticated and authorized user and to access a specific API.

The illustration below helps to understand how each of these tokens represents. *This is a generalized example to explain each token format to the reader.* In this example, a user would like to check his balance on Bank A website. For this action, below are all the security-related transactions that have to follow successfully.

1. The user logs into https://banka.com and since he doesn't have session identifier or a session cookie, he is redirected to the login page.
2. The user enters username/password and is logged into the application. Bank A allocates a session identifier and stores it in a session cookie, which is stored in the user’s browser. If this was a mobile app, and if a user chooses to authenticate using the phone's biometrics, the server would send back an authentication token, which is stored in the phone. Next time the user logs in, he uses the authentication token instead of entering his username/password combination.
3. Once authenticated, the authorization server in enterprise environment checks its backend about the user, whether he is the primary member of the account or an authorized manager of the account.
4. Depending upon the user details, an authorization token is sent back to the user, with the details embedded in as a token. During each transaction, the user forwards the authorization token to the server, which is then exchanged with an access token.
5. When the user requests to access the BalanceAPI, the server sends the user an access token through which he can log in to the BalanceAPI and check his balance.

In the above illustration, each type of token is defined, based on the security control it solves. Most current applications do not contain all the tokens in a single implementation and may use a single token to perform all actions. Depending upon the application’s architecture and external entities, the token's pertinence may differ, but the security controls each solve do not change.

**Token implementation methods**

OAuth2 implementations mainly provide proper authorization controls and grant access to the required data and methods it supports. The OAuth2 acts as a delegation framework, wherein multiple internally developed applications define what the user is capable of performing. On the other hand, OpenID Connect (OIDC) is used to ensure that the application's authentication requirements are handled. In this mechanism, OIDC ensures that authentication scenarios are done throughout the enterprise environment, thus supporting single sign-on functionality.

**OAuth2**

In OAuth2, the method where applications get an access token to access resources is classified as grant types. OAuth2 defines many grant types methods, but the most common and agreeable grant type for an enterprise API architecture is the authorization code flow.

Below are the simple steps that formulate the authorization code flow in OAuth2 [1][6]. The same is outlined in figure 2:

1. The user tries to access that part of the application that requires authentication. The application opens a new browser tab and directs the user to access the OAuth2 server.
2. The OAuth2 prompt is shown to the user, and the user accepts or provides consent for the application to access it.
3. The user is brought back to the application with the authorization token from the OAuth2 server.
4. The application accepts the authorization token and generates an access token to retrieve data from respective APIs.

**OIDC - OpenID Connect**

OIDC is an authentication protocol and is developed by adding further steps to the OAuth2’s authorization code grant flow. The measures mainly cover authentication features, including some of the additional advantages listed below [8]:

- With OIDC the user is forced to authenticate even if already logged in OpenID providers.
- OIDC mandates the authentication and authorization mechanisms to utilize a signed token containing user's access information in the form of claims.
- OIDC provides a venue to use encryption and signatures, which can then be used for additional security controls and strengthens the protocol through the token generation process.
Security best practices

The following are some of the security design best practices to consider while implementing tokenized authentication and authorization (OAuth/OIDC) using APIs [7].

- **Authorization/access tokens are required to be forwarded to downstream applications or external APIs.** While using JWT as an access or authorization token, proper measures must be taken to ensure that sensitive user information is not embedded as part of the token.

- **Refresh tokens in OAuth2 and OIDC are considered to be a different entity and are not part of the initial specification/implementation.** Since refresh tokens are long-lived, they are required to be more secured than access tokens. Hence, proper measures to store them securely must be implemented.

- **The token revocation sequences supporting each authentication/authorization protocol's security requirement is altered to an agreeable level.** The most important aspect of OIDC Connect is step F in figure 3, the validation of the ID token as per the Open ID’s specification, section 3.1.3.7. If ID token validations are performed using proprietary or developed software, proper measures must be taken to ensure that the validation routine takes care of all the thirteen checks defined in the specification.

**JSON web tokens**

JSON web token (JWT) is an open-source standard (RFC 7519) that provides compact means to send and receive data securely. JWT tokens are designed for serving the integrity requirements of any implementation. Figure 4 is an example of a JWT token that can be utilized in the OIDC authorization code flow, in the form of an ID token. The JWT token illustrated in the example includes all the details required for the downstream applications or dependent entities. These entities in the ID token provide the right access, depending upon the assigned role in the application’s respective sectors.

**Figure 3 – OIDC Authorization Code Flow for APIs supporting SSO [2]**

**Security best practices**

The following are some of the security design best practices to consider while implementing tokenized authentication and authorization (OAuth/OIDC) using APIs [7].

- **Authorization/access tokens are required to be forwarded to downstream applications or external APIs.** While using JWT as an access or authorization token, proper measures must be taken to ensure that sensitive user information is not embedded as part of the token.

- **Refresh tokens in OAuth2 and OIDC are considered to be a different entity and are not part of the initial specification/implementation.** Since refresh tokens are long-lived, they are required to be more secured than access tokens. Hence, proper measures to store them securely must be implemented.

- **The token revocation sequences supporting each authentication/authorization schema should be aptly performed, and the revocation sequences should consider invalidating a specific token, for a particular user, for a role, or even a kill-switch where all the tokens are revoked.**

- **Moving the authorization aspects of an application to different components supported by APIs provides a distributed approach.** The application may not have any business logic around the authorization components, thereby limiting the application or user-agent to implement functional logic alone. Proper care should be taken to split these two logical components and not to overlap them.

- **With tokens, sharing roles and permissions of the users becomes easier.** Implementing selective permissions to downstream applications or external applications is an
important aspect going forward, so resource servers, such as APIs, are not exposed to the public by mistake.

- Leaked client secrets are detrimental to the security posture of the authentication/authorization schema. Hence, proper measures should be taken to safeguard the client secrets from incidentally storing in source code, binary, or configurations.

- Single sign-on with tokenization sequence ensures that users do not have to log in to each application and that applications do not have to handle user passwords. By using a single login platform for many applications, the SSO sequence is prone to phishing attacks. Although tokenization sequences do not provide any defenses against phishing attacks, the application users should be educated about these attacks and best practices to use only trusted URLs and verify certificates.

For additional information on threats and safeguards, please refer to the OAuth’s Working Group’s IETF draft submission [7].

Conclusion

Organizations are building up enterprise API solutions to support their increasing requirements related to data and features. With applications extensively relying on APIs to render data, and that most applications are internally supported by multiple functional APIs, there has been a need for a scalable security solution to ensure only authorized users get the requested resources. Even with stringent measures, implementing a seamless and secure enterprise API is not an easy task, not with the existing controls and techniques applicable for securing traditional, monolithic, and browser-centric applications. Security tokens solve this unique problem. However, it requires proper care during implementation, while following stringent security best practices and ensuring that no security flaws are introduced into the mechanism [3]. With the future moving towards a scalable implementation of security mechanisms, passwords have been found to be a deterrent to any automation efforts. Considering this, tokens, in any mode or function, will potentially be the primary building blocks of security implementations supporting enterprise API solutions.

References


8. OpenID, “Connect Core 1.0: Aggregated Distributed Claims” – https://openid.net/specs/openid-connect-core-1_0.html#AggregatedDistributedClaims.

About the Authors
Sandeep Jayashankar, CISSP, GIAC GMOB, is an experienced information security architect with a demonstrated history of supporting the finance industry. He has led many successful red team operations and ethical hacking campaigns against enterprise orga-

izations. Sandeep earned a Master’s degree in Information Security from the Johns Hopkins University and holds many certifications including Offensive Security’s OSCP, OSCE, OSEE, and OSWE. He may be reached at sandeep.jayashan-
kar@gmail.com.

Subin Thayyile Kandy is an industry veteran with more than a decade of experience in application security and offensive security. He has worked for several banking and financial organizations like Barclays and Capital One spanning both defensive and offensive security roles, and enjoys every aspect of it. He may be reached at Subin.kandy@gmail.com.

Up Close with Candy Alexander, ISSA International President
Continued from page 6

even did risk assessments with dumpster diving. I was smart enough and passionate enough to understand that everything I did and every task I performed would add to my knowledge base. I was like a sponge and just kept consuming the experiences and knowledge.

Drawing on my experience of starting a user’s group while I was a secretary, I coerced a dear friend of mine, Robin Wheeler (and fellow ISSA member), to start up a security users group, much like we have with the ISSA today. The security function within the company grew and blossomed, and so did the opportunities – I grew from a “security person,” to a compliance specialist, to security manager. Then I left the company and continued accepting various roles up the ladder to CISO and finally where I am today, a CISO, an executive advisor, and the ISSA International president.

Q: Did you have a mentor?
A: When we talk of having a mentor, I often have the image in my mind of a mentor and a mentee. I can say this: I never had one mentor; I’ve had several – at the same time. I have always solicited guidance and knowledge from those individuals with more knowledge than I have – and by topic.

I also sought out individuals whom I have the utmost respect for and listened to for their general words of wisdom. In my early years, I have had some very dear people brought into my life to help guide me. Don Evans comes to mind. He was one of the unsung champions of our profession. Don worked at United Space Alliance, and I had met him at an ISSA conference many years ago. He would never really give me the answer but rather guide me until I found the answer myself. I appreciated that approach and often use it with individuals I guide or coach.

Today, I follow the same approach. I have my “subject matter expert” mentors/guides, but then I have role models: little pieces of individuals that I try to emulate, such as Sandy Lambert for her general devotion to the ISSA and endless caring, Pat Gilmore for her knowing what is right and hav-

ing the courage to do it. And the luminaries such as Winn Schwartau and Spaf (Gene Spafford) to jump in and provide insights as to where the profession and technology are headed. Then there are those that have passed, such as Gene Schultz with his thoughtfulness and willingness to help out in any way he could and Howard Schmidt and his willingness to share his vast knowledge. The amazing thing about all of these individuals is they are humble and open to having a conversation with you, no matter your station in life.

Q: What gender-related opportunities or roadblocks did you experience during your career?
A: In looking back, I never really stopped and thought about gender-related issues. I can’t say that I was ignorant to the fact; it was just that I was so focused on the learning.

Perhaps it was how I was brought up. Growing up, my role model was my brother. I wanted what he wanted – bicycles, watching racing on TV, and eventually it was cars. I was never told “that is a ‘boy’ thing,” or I needed to act like a girl. It was more of the approach, if that’s what you like. So, I liked fast cars, motorcycles, as well as doing my nails and hair. Kinda simple, right?

Fast forward to entering into this profession. Yes, I was one of two women in the IT group and the other (Robin) went to another facility. I adapted. I competed and did the job. And more importantly, I learned from everyone I met and every situation I lived through, for I really believe that there is something to be learned as an end result. Good or bad.

I would say that one key element of the gender-related opportunities or roadblocks comes down to this simple truth: