Computer Science and Engineering 2020, 10(1): 22-30
DOI: 10.5923/j.computer.20201001.03

Implementing Granular Access Definitions in Log Records

Sandeep Jayashankar*, Subin Thayyile Kandy

USA

Abstract This document specifies a method of creation or generation of software logs that would further assist in building more granular access control definitions. The technique relies on including an authorization token within each log record, which is generated using a signed JSON Web Token (JWT). Each authorization token embeds all factual information that lets the log viewers set these access constraints, either using an external access control list or applying an access control list outlined in the log management platform.

Keywords JWT, Authorization, Tokens, Software Logging, Access Definition, Secure Log Records, Secure Logging Mechanism, JSON Web Tokens

1. Introduction

A logging mechanism in any software implementation is the core utility for debugging the behavior, either during a failure or during a successful transaction. As the software runs, the logging mechanisms continuously create a trail of event checkpoints, by capturing ample information about the software's state. For developers and system administrators who are tasked with debugging a specific software's behavior, these event trails are gold mines of valuable information.

Since logs serve as the source in the debugging process, all recorded information about the event should be available in log records at the time of perusal. However, this provision may result in software incidentally logging sensitive information, resulting in exposure of critical data to unauthorized personnel. There are also occasions when developers or system administrators are required to examine detailed log data which may contain sensitive information, however, restrictions could be enforced from accessing those logs. Therefore, an urgent requirement is warranted to implement precise and granular access controls in the log platforms. Even if some of the restrictions are already enforced, there is a need for a solution that can embrace the integrity conditions for the authorization definitions.

* Corresponding author:
sandeep.jayashankar@gmail.com (Sandeep Jayashankar)
Published online at http://journal.sapub.org/computer

2. Current Trends in Logging Mechanisms

Many software entities in enterprise environments require logging mechanisms to create and store appropriate logs. More importantly, the application and web servers, database servers, API endpoint servers, security solutions like firewalls, all require logging mechanisms to capture the event appropriately. For ease of use, logs from various sources are typically placed into centralized storage repositories for user consumption by users with appropriate access.

Implementing "*Least Privilege*" and "*Need-To-Know*" security principles may be cumbersome in an environment supporting a large organization. This constraint has led to many organizations redefining their security best practices to refrain from including sensitive or personal identifiable information in the records.

Another practice is to limit user access from viewing a set of logs altogether. Examples of restriction-based access controls are listed below:

- *Based on Log Level (error, warn, info, debug)*
- *Based on specific applications*
- *Based on origination (App, DB, Web, API)*
- *Based on data owners*

In all the above examples, the access control definitions can never achieve the granularity it suitably requires. Additionally, defining access controls for every log record has considerable administrative overhead, and could have storage and data processing implications. Hence, there is a need for a secure and lightweight mechanism that can embed each log record with access information.

3. Token Type and Its Features

JSON Web Tokens (JWT) is an open standard (RFC 7519) that defines an efficient way of encapsulating a set of data, and securely transmitting them in a JSON format. By design, JWTs are meant to support lightweight transactions, and formulating the whole token in a JSON format helps them achieve the primary goal of being uncomplicated. Considering other tokens such as SAML (Security Assertion Markup Language) tokens, which use XML, JWTs are compact due to the same reason. Moreover, JSON formats have extensive usage in the current technology landscape, and parser support is available in all programming languages. Hence, the lack of required escape or encoding characters keep the token compact and lightweight.

The second goal of securely transmitting data is possible by adding a digital signature to the token's contents. The digital signature is a mandatory part of any JWTs and signing token contents warrants the integrity provisions of a secure implementation. Unlike SWT (Simple Web Tokens), which only supports symmetric signing algorithms, JWT supports symmetric and asymmetric digital signatures. Depending upon the degree of integrity requirements, any suitable digital signature algorithm can be selected, thus achieving a proper balance between the lightweight token and the security aspect. Even though JWTs have provisions for encrypting the contents of the token and act as an authentication sequence, the usage of JWTs in the current article is mainly from the authorization perspective. Thus, we discuss more towards the usage of hashing algorithms and signing the contents using private keys.

4. Token Contents

JWTs, by default, contains mainly three parts, a header, payload, and the signature. The header and payload contents of a JWT token utilize an encoding format, "Base64URL", to ensure the fulfillment of token encapsulation criteria. The contents of a JWT is explained in more detail below:

Headers contain two main parts; the signature algorithm and the type of token used for generating the JWT. Concerning the token type used, JWTs always use the value "JSON" for a defined claim "typ" in RFC-1759. The RFC-7519 represents the claim "alg" to identify the digest hash or signature algorithm used during token generation. In addition to the two above claims, "cid" (correlation ID) can also be utilized to correlate a specific event and its pertaining logs from different solutions.

```
{
    "alg": "HS256",
    "typ": "JWT",
    "cid": "123456789"
}
```

The token's payload contains all the information required for the authorization sequence to ensure that the log record is only those who have permission to view it. The main contents of the payload, as per the initial design, is the following:

The below claims are as per the RFC-7519:

1. *iss (issuer):* The issuer defines the entity that has created the log record. The value can be an application name, infrastructure entity, or even another log server.
2. *iat (created date and time):* The date when the log record originates.
3. *exp (expiry date and time):* The date when the log record expires.
4. *aud (audience):* The audience can specify the centralized log server where logs collaborate.

Additionally, to define the Access Control Matrix or Access Control Lists, the below claims are utilized.

5. *acm (access matrix): This claim helps build a two-dimensional access matrix for each log record, with the precise access and the associated log level.*
6. *acl (access control list): This claim helps build a uni-dimensional access control list for each log record, with just the access defined.*

The illustration below shows the proposed JWT payload using an Access Control Matrix:

```
{
{
    "iss": "BalanceAPI",
    "iat": "1431706505",
    "exp": "1589559305",
    "aud": "https://centralized-logging.example.com/",
    "acm": {
        "AD_LOG_DEV_USER": "error, warn, info",
        "AD_LOG_DEV_ADMIN": "error, warn, info, exception",
        "AD_LOG_SECURITY": "sec",
        "AD_LOG_SRE": "error, warn, info"
    }
}
```

The final part of a JWT is the hash digest or the digital signature of the header and the payload. The following algorithms are supported:

- *Symmetric Algorithms:* JWTs use the HMAC algorithm with SHA cryptographic message digest. Depending upon the size of the SHA digest required, JWTs support 256, 384, and 512 sized HMAC algorithms.
- *Asymmetric Algorithms:* JWTs use RSASSa-PKCSv1.5 (RS), ECDSA (ES), and RSASSA-PSS (PS) algorithms with SHA cryptographic digests (256, 384 and 512), and a private key to sign the contents.

By using the header and the payload from the examples above, a JWT with HS256 algorithm is shown below (secret phrase being “secret”):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImNpZCI6IjEyMzQ1Njc4OSJ9.eyJpc3MiOiJCYWxhbmNlQVBJIiwiaWF0IjoiMTQzMTcwNjUwNSIsImV4cCI6IjE1ODk1NTkzMDUiLCJhdWQiOiJodHRwczovL2NlbnRyYWxpemVkLWxvZ2dpbmcuZXhhbXBsZS5jb20vIiwiYWNtIjp7I

kFEX0xPR19ERVZfVVNFUiI6ImVycm9yLCB3YXJuLCBpbmZvIiwiQURfTE9HX0RFVl9BRE1JTiI6ImVycm9yLCB3YXJuLCBpbmZvLCBleGNlcHRpb24iLCJBRF9MT0dfU0VDVVJJVFkiOiJzZWMiLCJBRF9MT0dfU1JFIjoiZXJyb3IsIHdhcm4sIGluZm8ifX0.rjG0AueEgx7NN4YWKhSsO47NittkykLBeWAqLZ_uzxw

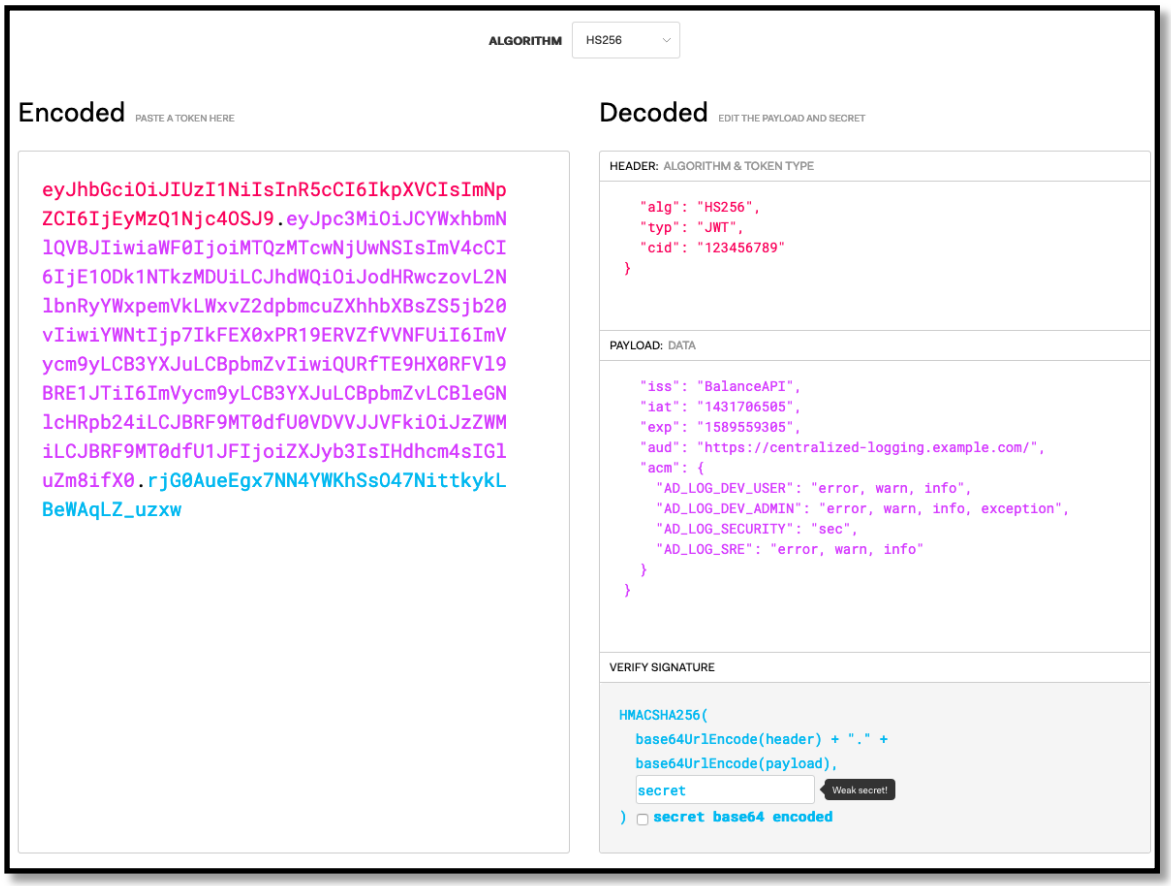

Figure 1. Screenshot from https://jwt.io showing the token generated with example values

5. Proposed Logging Mechanism

The proposed logging mechanism is a full-stack logging platform, which includes the Log Generation Process and a Log Viewer Verification process. Each illustrated process in the proposed logging mechanism contains multiple sub-components that can act as a separate software entity. The modularized design takes into consideration the software development challenges, and decoupling software modules help development efforts to be more manageable.

Log Creation Process

The Log Generation process acts as a middle layer between the software-generated logs and the centralized logging platform. In this pivotal position, the log creation process consolidates the retrieved log record and assigns the correct access definitions. Following are the sub-components as per the propose designed:

- *Log Processor:* In the proposed solution, the log processor is the single point of entry service that accepts all software generated logs for processing. The processing includes two aspects; correlating logs based on a specific event and collecting the information the ADE (AuthZ Decision Engine) requires. The Log Processor sends the original log record to the next step, where the logs concatenate with the generated authorization token to form the final log record.
- *Authorization Decision Engine (ADE):* The Authorization Decision Engine ensures that for every log record from the Log Processor, the right set of access definitions are formulated and provided to the Token Generator. Depending upon the environment in the network, the ADE can collect the access control and user information from Active Directory, a user-object repository, or from a mechanism that has an access control matrix/list compiled.
- *Token Generator:* Based on the Token Contents described in the previous section, the Token Generator requires the following from different sources:
  - o The Log Processor provides the required information for the formation of the claims, cid, iss, iat, exp, and aud.
  - o The ADS provide the claims, acm, or acl to the Token Generator.
  - o The claims, typ, and alg has to be set as an environmental variable or can be a constant value for implementation.

Once the respective sources transmit the required

information, the Token Generator utilizes a pre-existing library from many supported development languages to compile the complete token. This generated token acts as the authorization token for that specific log record.

Finally, the log record includes a column for the the authorization token which gets stored in the centralized log storage.

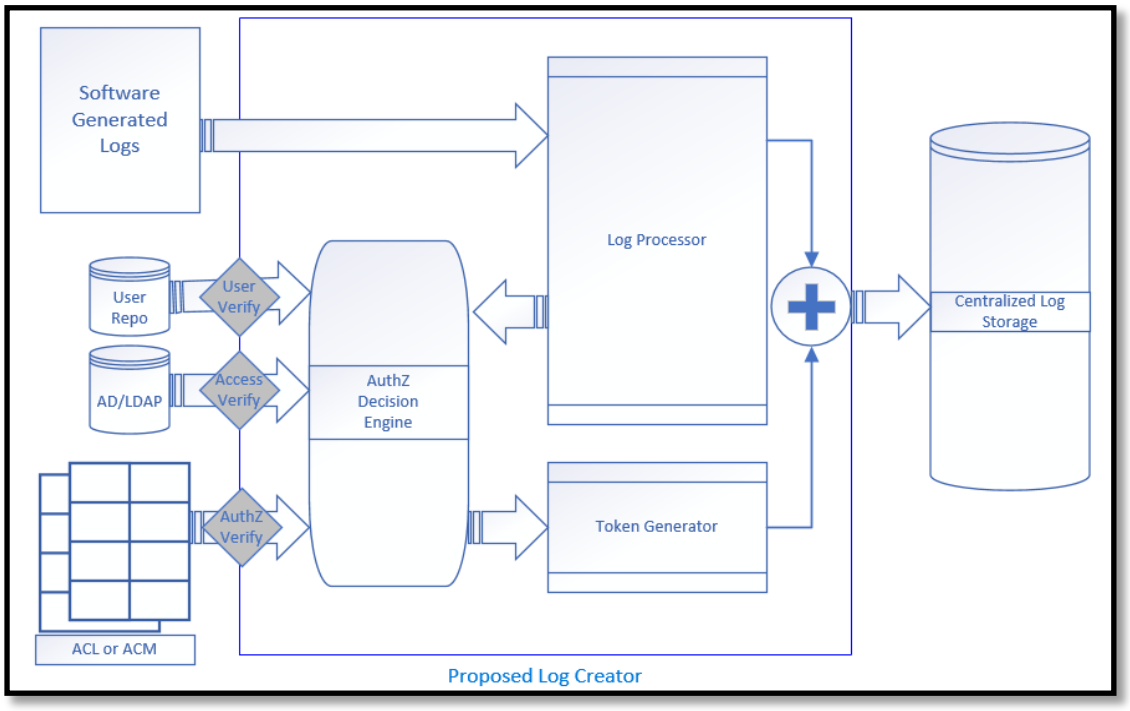

Figure 2. Process diagram showing the proposed Log Creator mechanism

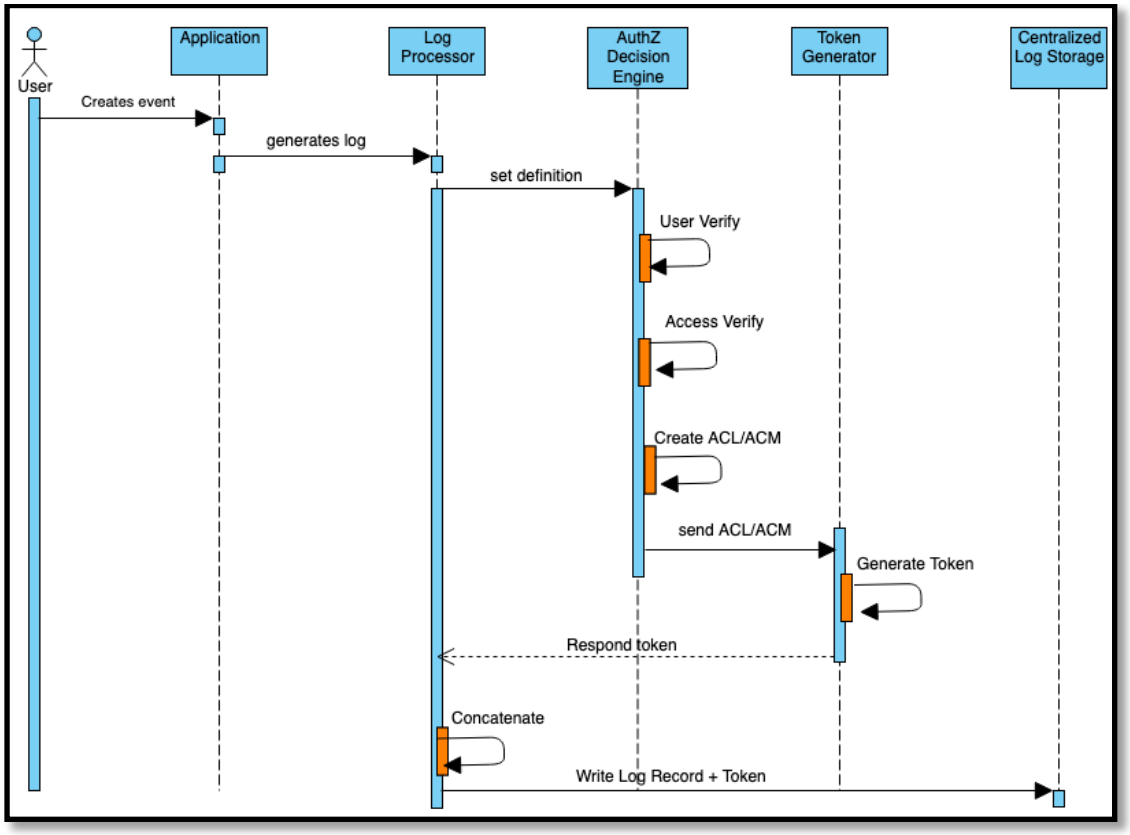

Figure 3. Sequence diagram showing the proposed Log Generator flow

The above sequence diagram demonstrates the entire Log Generation process starting with the user's creation of an application event and ending with the centralized log stored. In the illustration, the Log Processor calls the ADE to construct an access definition after verifying the ACL/ACM and the user. With the generated acm/acl claim, the Token Generator creates the token and stores it into the Centralized Log Storage along with the log record.

Log Viewer Verification Process

The Log Viewer Verification process acts as an intermediate barrier between the centralized log storage and the Log Dashboard software. In a practical scenario, the proposed mechanism is applied as an API endpoint that only returns log records authorized to the user. However, internal to the API endpoint, the process consists of some sub-components which mainly process and validate the authorization token.

The Log Viewer Verification process consists of two main sub-components, as per the proposed design:

- *Log Processor:* The Log Processor can be an API endpoint that accepts requests from the Log Dashboard and contacts the Centralized Log Storage for a specific user's log records. Additionally, the Log Processor is also responsible for populating only the authorized log records back to the Log Dashboard. Furthermore, the Log Processor can provide additional services such as:
  - masking sensitive data from logs,
  - correlating log records based on a specific event,
  - grouping log records based on preset filters, and
  - ensuring that expired log records are removed from the list.
- *Verification Engine:* The Verification Engine in the Log Viewer Verification process checks each log's authorization token and verifies it against the authenticated user. As per the proposed design, the Verification Engine consists of the following segments:
  - Signature and Token Verification: This segment validates whether the signature hash retrieved from the authorization token is confirmed and that there are no changes detected in the JWT payload. There are currently many open-source libraries in every development language that can efficiently perform signature and hash verification. The token verification segment can act as a separate entity assuming the role of parsing and debugging the authorization token's retrieved information.
  - Access Verification: The Access Verification segment reviews the acm/acl value embedded in the authorization token and validates if the authenticated user has access to the specific log record. Based on the identity and access management implementation in the organization's environment, the Access Verification process may consult Active Directory to check the authenticated users' group memberships and compare them with ACL/ACM definitions. Additionally, the access verification process can also consider the log origination and validate if the user has 'read' access to it. Furthermore, depending upon a specific view filter, access restriction can also be validated.

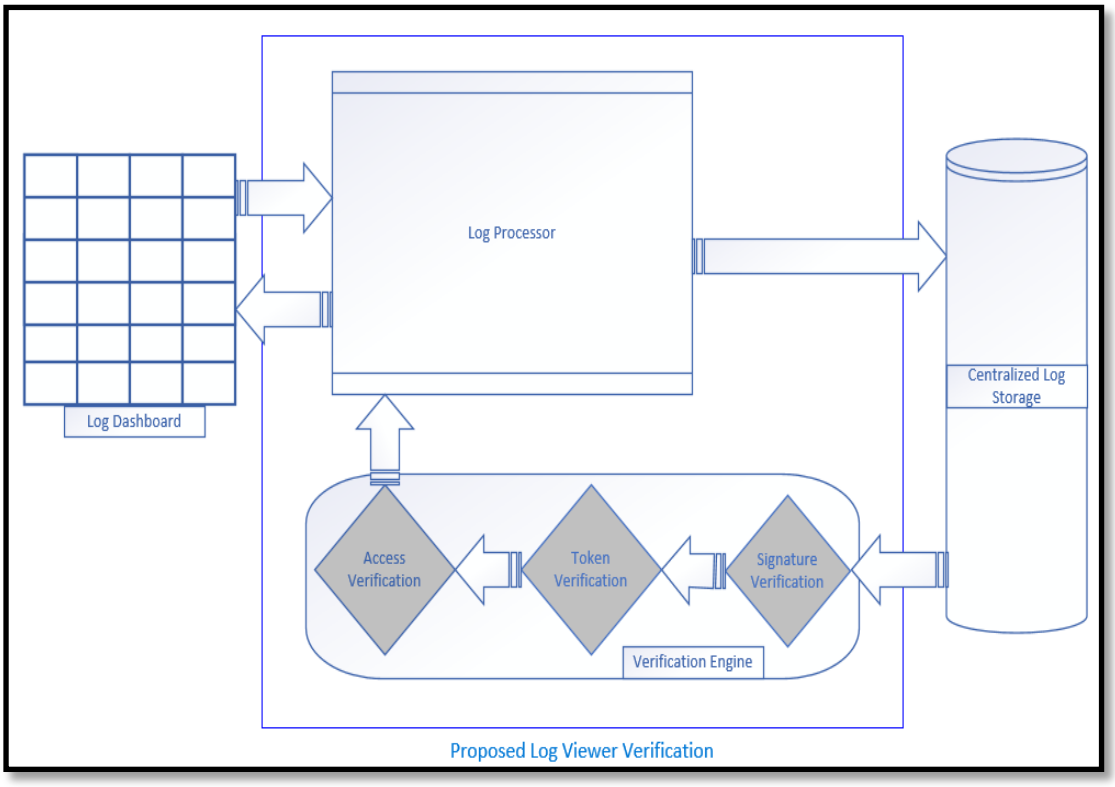

Figure 4. Process diagram showing the proposed Log Viewer Verification mechanism

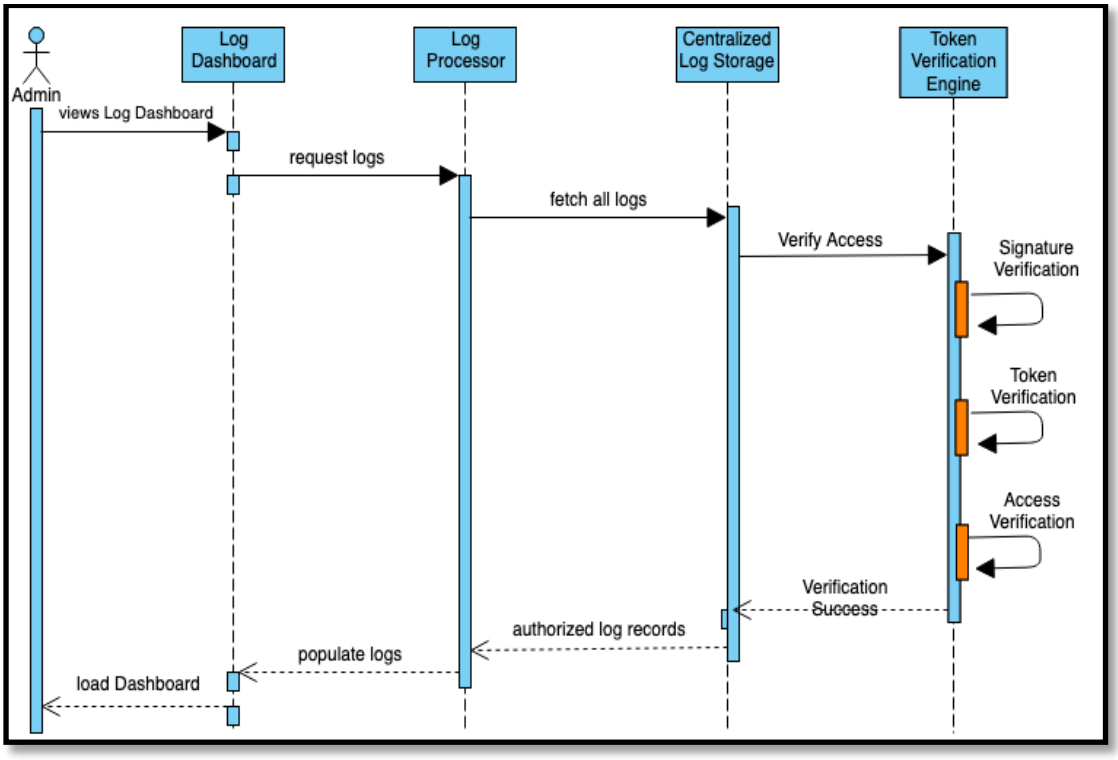

Figure 5. Sequence diagram showing the proposed Log Viewer Verification flow

- Once all defined validations are complete, the Verification Engine responds with its decision to either show or hide the log record depending upon the access definition.

Finally, the Log Dashboard retrieves the response from the Log Processor and displays the list of log records authorized to the end-user.

The above sequence diagram shows the log records' verification process before being presented on the Log Dashboard. In the illustration, the Log Dashboard calls the Log Processor to retrieve all the log records present for the authenticated user. The Log Processor sends a request to the Centralized Log Storage to respond with log records that are authorized to the end-user. The Centralized Log Storage uses the Token Verification Engine to check the token, and signature and verify if the specific log record is permitted to be viewed by the end-user. The Log Processor then responds to the request with a list of authorized users from the Centralized Log Storage.

Log Authorization Process

A user without access to any particular logs can request access through Identity and Access Management platforms, Active Directory, or any other directory service that is in line with organizational policies and processes. The proposed Authorization Decision Engine includes an LDAP listener feature, which, if enabled, polls any changes to the directory services based on the ADE's configurations. Administrators should also be able to configure the polling frequency between the ADE and the AD. This extensive feature helps organizations set their AD polling requirement, depending upon their requirements of either needing real-time enforcement or a more relaxed daily polling condition.

Once ADE is notified of LDAP listeners' changes, it synchronizes objects that map to the recently updated user profiles. As part of this process, ADE identifies any profile with modifications in the logs' access requirements. The process also includes the notification of Token Generator with an account list, for which the corresponding tokens are to be updated or reissued to reflect the current authorization changes. This step ensures that the verification engine accurately authorizes users to access the logs during the log viewer process. Any changes in authorization matrices, either the user is enrolled, modified, or revoked, will be reflected in the respective tokens.

The below illustrates how the proposed Logging mechanism acts during the user lifecycle management process:

- *New User enrollment:* As a new user joins an organization, as part of his onboarding, he may request access to several applications, and their logs for debugging purposes. Once the approval process and records creation are complete in the organization's directory services, the proposed LDAP listener identifies the user addition instantly (or as configured by the administrators). Once notified, the Authorization Decision Engine creates a corresponding user profile object, and details all the roles assigned to the user in AD. When the user accesses the Log Dashboard, the Token Generator creates a new token, after utilizing the Verification Engine to validate the user's access with the directory

services.

- *Existing User Modifications:* As per the proposed mechanism, the existing user's access modifications work about the same as that of the new user enrollment process. However, when the LDAP listener identifies any differential in the user access matrices, active tokens are replaced with the updated tokens containing all the changes to the user access.
- *User Termination:* As part of the user exit process, the user's profile and roles in AD are set to expire on a specific date and time. Once the changes take effect in the directory services, the LDAP listener identifies the differential and updates the Authorization Decision Engine. The Token Generator utilizes the token payload's claim, "exp" to set the tokens to expire as per AD profile details, and update the existing tokens. As part of the token's records cleaning process, any expired tokens should automatically be deleted from the record. As the tokens expire, the Log Verification process fails the validation and does not let any further unauthorized actions on behalf of the user.

6. Implementation Requirements

The Implementation requirements mainly attempt to propose the general provisions of the algorithms and not to set a mandatory requirement to implement the proposed mechanisms effectively. It is up to the implementation software to dictate what algorithms are advisable to use for each implementation design. However, proper care is recommended during the algorithm selection by embracing the understanding that any flexibility in the security requirements for convenience or performance may result in the implementation to be ineffective.

The following are the security best practices, or recommendations, concerning the algorithm selection and the key sizes. The security level required for the software implementations can be finetuned by changing the SHA key size to each algorithm. The below description shows how each algorithm can provide benefits over the others:

- Software solutions can utilize *RSAPSSSHA384 (RSA Signature with the probabilistic signature scheme with SHA-384)* to implement a JWT signature mechanism with the highest security setting. The probabilistic signature scheme is proven not to be susceptible to the same type of attacks that affect the PKCS and other RSA implementation flavors. The "alg" claim values for DSA with EC are PS256, PS384, or PS512, depending upon the SHA's digest size.
- The *ECDSASHA512 algorithm (Elliptic Curve Digital Signature Algorithm with SHA-512)* is a close second option for software solutions that require a very high degree of security setting. The DSA algorithm is proven to be faster at signing digital signatures compared to RSA. However, RSA is proven to be faster at verifying digital signatures. Hence, depending upon the implementation requirements, the software solutions can choose either ECDSA or RSA-PSS. The "alg" claim values for DSA with EC are ES256, ES384, or ES512, depending upon the SHA's digest size.
- The *RSASHA512 algorithm (RSA signature with PKCS#1 v1.5 with SHA-512)* is a standard option for software solutions that require interactions with other third-party systems. The PSS flavor of RSA is not as prevalent as PKCS#1 v1.5, but since the algorithm is deterministic, it is prone to many practical attacks. Also, RSA PSS is a complex algorithm to implement, and it takes a considerable amount of time to compute. Hence, using the RSASHA512 algorithm may reduce the security posture of the implementation, but it is beneficial from a performance and support aspect. The "alg" claim values for RSA with PKCS#1 v1.5 are RS256, RS384, or RS512, depending upon the SHA's digest size.
- The *HMACSHA512 (HMAC with SHA-512)* uses a symmetric algorithm with a secret key shared between two parties. Since there are no public-private key pairs, but just a secret key, there is a considerable gap in the security posture if the secret key is compromised. However, the HMAC algorithm is known to be fast and simplistic and very advantageous in the performance aspect. The "alg" claim values for the HMAC algorithm are HS256, HS384, or HS512, depending upon the SHA's digest size.

The below implementation requirements are for the proposed process:

- *Log Processor*: As per the proposal, the Log Processors are envisioned as an API endpoint that allows software entities to call when an event trigger. The implementation can act as a separate entity to the main software solution, such as a provisioned software-as-a-service or can be part of the software solution. However, proper measures must be taken to ensure that the API endpoints are authenticated adequately and have appropriate network zoning restrictions so that it is not exposed to the external environment.
- *AuthZ Decision Engine (ADE):* The ADE is an integral part of the proposed implementation, as it assumes the responsibility of proper access definitions to each log record. Hence, it is recommended to implement secure channels for its interactions with Active Directory, ACM/ACL providers, and the User databases.
- *Token Generator*: While generating the tokens, proper measures must be taken to ensure that the token claim, "alg" is assigned with the correct cryptographic algorithm. Since the token generation process mainly utilizes an open-source library or a separate software entity/framework, it is recommended that care be taken to ensure there are no pre-existing vulnerabilities in the token generation mechanism.
- *Verification Engine*: The software solution utilized for

this measure should verify that the authorization tokens conform to the JWT's RPC structure, with appropriate JSON object schema. Additionally, the Base64URLDecode of the authorization token should result in a fully formed header, payload, and signature digest. Furthermore, it is advisable to have the header and payload go through a canonicalization sequence. This progression ensures that the header and payload are verified after eliminating any encoding formats, thus enforcing uniformity in the data before verification. Finally, before authorizing the token, the Verification Engine should validate the RPC specified claims such as exp (token expiration) and iss (issuer/issuing authority).

7. Interoperability Considerations

To achieve an acceptable and interoperable deployment of the proposed mechanism, the sub-components and their underlying system entities should be in sync with each other. Any changes in the configurations without updating the others may result in breaking the entire proposed mechanism.

The following are some of the configurations that are critical from an interoperability perspective:

- *"iss" and "aud" claim in the payload*: The Issuer identifies where the log record originated. This information is essential for the ADE to assign the appropriate access definition to the suitable end-users, and for the verification engine to provide access to authorized end users. Hence, it is vital to set prerequisite configurations that identify issuers and audience values accurately.
- *Storing sensitive information securely*: The software-sensitive information in this proposed solution is the private keys used for generating the JWT signature. The Token Generator requires access to these private or secret keys and storing them in plaintext may pose a security risk. Hence, proper measures to store these private or secret keys in an encrypted format is recommended.
- *Network Zoning in an enterprise environment*: The Network environment must be zoned based on the defined trust boundaries and the sub-components' sensitivity. In this proposed solution, the Log Processor API must be network accessible only to the software that generates logs. The other sub-components, such as ADE and Verification Engine, can be designated to be at the Operations Zone. Since the Token Generator is an integral part of the implementation, and since it handles the private keys for generating a signature, it can be designated to the restricted zone.
- *Server Authentication between sub-components:* Each sub-component defined in this logging mechanism are designed to be decoupled with each other. Implementing an effective server authentication and a secure transmission channel (preferably mutual TLS) is essential as attackers can target sub-components and formulate a successful man-in-the-middle attack.
- *Supporting Centralized Log Storage:* The proposed implementation acts as an intermediate framework between the software generating logs, the centralized log storage, and the Log Dashboard. Hence, there is a need for currently available centralized log storage solutions and Log Dashboard to support the feature, either by providing a calling API service or invoking proposed API implementations.

8. Security Considerations

The proposed mechanism extensively utilizes the JSON Web Tokens as an authorization token. While JWTs are secure by design, many attacks, vulnerabilities, and security gaps have been identified in the implementation and configurations. Some of the reported public vulnerabilities can be mitigated by following the recommendations below:

- *Algorithm Verification Bypass*: The "None" Algorithm vulnerability resulted due to applications not verifying if signature algorithms are specified in the token headers. As a result, attackers crafted malicious JWTs by assigning the "alg" claim to "none", which resulted in JWTs getting verified even without signature hashes assigned to them. Hence, the Token Generator should explicitly specify which algorithm is utilized, and the Verification Engine must confirm that the "alg" claim is assigned with strictly approved values.
- *HMAC "Verify" Attack*: The signature verification part of many JWT Libraries was found not to accept the verification algorithm as a function parameter. As a result, attackers could use the public key to formulate a signature and replace the algorithm to be HS256. Currently, JWT libraries mandate algorithm values to be sent as a parameter in the “verify” function. In accordance, the Verification Engine in the proposed mechanism should also mandate the algorithm claim ("alg") in the token, and also accept the algorithm parameter for all supported verification functions/ methods.
- *Sensitive Data in Tokens:* The proposed mechanism generates JWTs that are not intended to handle confidentiality, but instead support the integrity of the token contents. Hence, it is not advisable to contain any application or user sensitive data as part of the token. If any instance of instantaneous information leakage occurs, proper care must be exercised to delete the token and generate a new token.
- *Using a robust symmetric key*: HMAC signature digests are prone to brute-force or dictionary attacks if the keys are not robust enough. There are currently many JWT attacking tools such as JohnTheRipper and JWTBrute, which conduct dictionary attacks to extract the key from the HMAC signature. Hence, it is prudent to

utilize robust keys that are generated by securely initialized/seeded pseudo-random number generators (PRNGs).
- *Use the latest TLS versions*: The proposed sub-components should extensively utilize TLS protocols to establish any secure transmission.

Terminology

For ease of understanding, this paper defines the following key terminology as:

- *End-User: Unless specified to be the user of a log generating application, the end-user in this proposal refers to those who view the Log Records.*
- *Log Records or Logs: Log Records or Logs are a snippet of information recorded during events while running software, or while a specific entity is performing any actions to achieve a goal.*
- *Log Dashboard: A dashboard with a table of all the log records that is viewable to a user.*
- *User Repo: A repository or a database containing users' identifiable information.*
- *Active Directory: More precisely, Active Directory Domain Service (AD DS) is Microsoft's domain and user service platform providing user account information and their applicable access/authorizations in a Windows environment.*
- *ACM: Access Control Matrix is a matrix of roles vs. which log type can the user access. The ACLs are generally used to check if a specific user has access to viewing a particular log record.*
- *ACL: Access Control List is a complete list of all the roles for whom the log record is accessible.*

REFERENCES

[1] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, doi: 10.17487/RFC, May 2015, http://www.rfc-editor.org/info/rfc7515.

[2] Ahmed, S., & Mahmood, Q. (2019). An authentication based scheme for applications using JSON web token. 2019 22nd International Multitopic Conference (INMIC). doi: 10.1109/inmic48123.2019.9022766.

[3] Siriwardena, P. (2014). JWT, JWS, and JWE. Advanced API Security, 201–220. doi: 10.1007/978-1-4302-6817-8_13.

[4] Haekal, M., & Eliyani. (2016). Token-based authentication using JSON Web Token on SIKASIR RESTful Web Service. 2016 International Conference on Informatics and Computing (ICIC). doi: 10.1109/iac.2016.7905711.

[5] Sheffer, Y., Hardt, D., & Jones, M. (2020). JSON Web Token Best Current Practices. doi: 10.17487/rfc8725.

[6] JWT: The Complete Guide to JSON Web Tokens https://blog.angular-university.io/angular-jwt/.

[7] JSON Web Token Introduction https://jwt.io/introduction.

[8] IETF JSON Web Token (JWT) Draft https://tools.ietf.org/html/rfc7519.

[9] JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants https://www.hjp.at/doc/rfc/rfc7523.html.

[10] The JWT Handbook by Sebastian Peyrott, Auth0 https://auth0.com/resources/ebooks/jwt-handbook.

[11] JWT Signing using RSASSA-PSS in .NET Core https://www.scottbrady91.com/C-Sharp/JWT-Signing-using-RSASSA-PSS-in-dotnet-Core.

[12] DSA vs. RSA Encryption https://www.jscape.com/blog/bid/82975/Which-Works-Best-for-Encrypted-File-Transfers-RSA-or-DSA.

[13] Application Logging: What, When, How https://dzone.com/articles/application-logging-what-when.

[14] Critical vulnerabilities in JSON Web Token libraries https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/.

[15] JWT Brute https://github.com/jmaxxz/jwtbrute.