UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Gregor Eesmaa

Authorization of Web Requests Based on Merkle Trees

Bachelor's Thesis (9 ECTS)

Supervisor: Kristjan Krips, MSc

Tartu 2020

Authorization of Web Requests Based on Merkle Trees

Abstract:
People should not have access to unauthorized data. Web applications can employ many different authentication and authorization schemes to accomplish this. To prove user permissions, session IDs or signed sets of claims are often used. However, scalability and efficiency are increasingly important in microservice architecture. It is also beneficial to decrease privacy risks when communicating with unknown parties. Thus, we propose a way of signing and selectively transmitting a large set of claims using the Merkle tree. In addition, we implement a JavaScript library based on the concept, that is optimized for many permissions and helps enhance user privacy when using microservices.

Keywords:
authorization, claims, efficiency, privacy, Merkle tree, cryptographic hash tree

CERCS: P175

Veebipäringute autoriseerimine Merkle'i puu abil

Lühikokkuvõte:
Veebirakenduste kasutajad ei tohi pääseda ligi andmetele, mille kasutamiseks neil pole volitust. Selleks kasutatakse veebirakendustes erinevaid autentimise ja autoriseerimise skeeme. Kasutaja õiguste tõestamiseks kasutatakse tavaliselt sessiooni-identifikaatorit või allkirjastatud õiguste paketti. Skaleeruvus ja efektiivsus mängivad aga arenevas mikroteenuste arhitektuuris aina suuremat rolli. Lisaks on tarvilik vähendada privaatsusriive mõju võõraste teenustega suhtlemisel. Pakume välja lahenduse suure hulga õiguste allkirjastamiseks ja valikuliseks edastamiseks Merkle'i puu abil. Implementeerime ka sellele põhineva JavaScript teegi, mis on optimeeritud paljude õiguste jaoks ja aitab tagada kasutaja privaatsust mikroteenustel põhinevas arhitektuuris.

Võtmesõnad:
autoriseerimine, õigused, optimeerimine, privaatsus, Merkle'i puu, krüptograafiline räsipuu

CERCS: P175

Contents

1 Introduction

When communicating over the internet, one might need to prove their permissions to the other party. On the web, access control is usually split into authentication and authorization – this means that user's identity is validated only once (authentication) and a token is then handed out to the client – be it by a session token or a signed set of claims. The latter can be used to verify user permissions with each request (authorization check).

User state usually needs to be shared between all instances of a service, or perhaps even multiple services. A mediator would decrease scalability and stability of the whole system. Having the client provide the state as a signed set of claims with every request solves these problems. However, using a signed JSON Web Token (JWT) with too many claims leads to much overhead on every request, as the JWT can only be verified as a whole. Also, privacy concerns might arise when communicating with a third-party service. These services would obtain knowledge of irrelevant data, some of which may be identifiable.

This paper proposes a way of proving authorization using the Merkle tree, which enables the client to supply only the claims needed for a request. When the authentication service initially emits the claims using the proposed method, it can sign everything without regard to any specific use cases. This increases the freedom of the client – the recipient of the claims – later on. The proposed mechanism is compatible with the existing JWT standard.

The main outcome of this work is a theoretical description of a solution along with a library, which uses the Merkle tree to decrease the request size in comparison with existing solutions, and to decrease the importance of third party trustworthiness in the context of authorization. We begin by describing the problems of communication efficiency and privacy regarding JWT-based authorization. Then, the Merkle tree data structure is introduced and used to solve these problems. Different details of its usage are described. Finally, the proposed approach is analyzed with regards to usage, security, privacy and efficiency.

2 Background

Understanding the domain of the problem requires some context about authorization on the modern web. This chapter provides an overview about the concept of authorization and the JSON Web Token standard.

2.1 Authorization

It is important to distinguish authentication from authorization. Authorization occurs upon every request based on information obtained from authentication. For example, for an application to query a person's Instagram profile, it must first get an access token by having the user authenticate – sign in. Then the profile request can be authorized by the token. Only when the token is correct, will the API respond with the user's profile [Fac].

Historically, the web was powered by monoliths – single-instance servers running on reliable and expensive hardware [Sum19]. This meant the user's session – user state – could rather safely be stored in-memory on the server. Authorization could then be performed by providing a session identifier, which was too long to randomly guess. However this kind of architecture ran into scalability problems as internet usage grew [Her18].

To battle this, nowadays, larger web applications are often split into multiple smaller, easily scalable microservices. For example, Netflix, a pioneer in microservice architecture, runs hundreds of different microservices in tens of thousands of instances [Nai17]. As a result, authorization schemes have also evolved.

Sharing the user state between microservices decreases the scalability and stability of the system. The mediator – the process that is used for sharing the state – becomes a bottleneck when scaling up, and is a single point of failure to the whole system. To prevent data loss, the information that makes up the state does have to be stored on the server-side, but some constant data, such as permissions, can be queried upon authentication and cached on the client-side. The state could then be provided with every request, decreasing the dependence on the server-side storage and mitigating the scalability and stability problems. However, in order to ensure authorization, the state needs to be verifiable. To accomplish this, it can be signed by the authorizing party – the authentication service.

2.2 JSON Web Token

Described in RFC 7519 [JBS15], the JSON Web Token (JWT) is a signed representation of claims. The claims are provided by one party as proof to another. The JWT contains Base64-encoded user information (see Figure 1).

The JWT is a widely used standard, that allows for services to store user permissions on the client-side. This has the benefit of the server, or even multiple servers, not having

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJ1c3IiOjEsIm5hbWUiOiJGb28gQmFyIn0.
    bnVH_kXtE4luv3jlR_nlKc1J9Rxdz1Um1m3I1I5iplM
```

Figure 1. An example JWT with some user information encoded in Base64.

to keep track of the authenticated users. Also, every incoming request contains sufficient proof for authorization, so no cascading queries are required.

This is achieved by providing the client a token serialized in the following format: `<header>.<payload>.<signature>`. The header is a Base64-encoded JSON structure that describes the signing algorithm, the payload is a Base64-encoded JSON structure containing the claims to be signed, and the signature is a Base64-encoded result of the algorithm specified in the header (see Figures 2, 3 and 4).

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

Figure 2. The header of the JWT in Figure 1 decoded into JSON.

```
{
  "usr": 1,
  "name": "Foo Bar"
}
```

Figure 3. The payload of the JWT in Figure 1 decoded into JSON.

```
HMACSHA256(base64enc(header) + "." + base64enc(payload), secret)
```

Figure 4. Pseudocode of the signing function of the JWT in Figure 1, where the algorithm is specified as "HS256".

Given the secret is not known by third parties, having a signed JWT is sufficient proof that an authority has at some point in time been convinced of the user's identity and/or their permissions.

3 Problem Statement

Using the JWT is convenient as it is stored and provided by the user. This removes the need to share user state between servers or to lock the user to a single server instance. Yet not everything can be stored in a JWT. For example, larger sets of data could add unnecessary overhead to requests, as JWTs have to be intact to verify their authenticity. Also, the whole payload of a JWT is visible to any request-handling party, so they could see and perhaps log personally identifiable details.

A single web request is usually limited in scope and does not require knowledge of every possible permission the user might have. This leads to two main objectives for a better solution: communication efficiency in case of many claims and enhanced privacy when requests are made to an untrusted service.

3.1 Communication Efficiency

Consider the following use-case. A user of a video streaming platform owns or has moderation access to thousands of videos. A token could be generated and cached for all of the videos. When the user attempts to modify a single video, database queries need not be done and a single token could be used to prove user's access to the resource. However, if this were a JWT with a thousand claims, it would be unreasonable to transport with a single request.

3.2 Privacy

Consider the following use-case. An internet service provider (ISP) could provide IP-based authentication for content-serving online services, such as news providers, video streaming platforms or even delivery services. A token could be generated for the user by the ISP with multiple pieces of information: all their service subscriptions, user information, user address. The token could be used to verify the user's access to the corresponding services and could be used to auto-fill user's address for package deliveries. If a single JWT were used, it would be bad for user's privacy, as any malicious third-party service could access that personally identifiable information. To combat this, every service would need their own corresponding JWT with the information the user is willing to share. A simpler solution would be to design an authorization scheme where information could be shared granularly. This would also increase separation of concerns by decoupling back-end data structure from the front-end use cases.

4 The Solution

To solve the mentioned shortcomings of JWTs, we propose a solution in this chapter, that uses a Merkle tree to store claims. A subtree of that tree could then be used to prove authorization. This minimizes request overhead and enhances user privacy, while leaving the responsibility of providing the correct authorization proof to the client. Comparing it with real life, it is similar to a person having to know which document to present to a government official. The root node of a Merkle tree built on the claims can be signed to provide sufficient proof of all the contained claims.

4.1 Merkle Tree

The Merkle tree [Mer82] is a binary cryptographic hash tree. It is constructed by recursively combining hash value pairs of an array of raw values, until a single root hash value is left. Due to properties of cryptographic hash functions and trees, the Merkle Tree has the following properties:

- The root hash value depends on all of the initial raw values in leaf nodes.
- The raw values in leaf nodes cannot be deduced from the root hash value more easily than by brute-forcing.
- Proving the existence of a single leaf node in the tree does not require knowledge of other leaf nodes – instead only the otherwise unknown nodes (or hashes) on the path from the raw value leaf to the root node need to be known.

While the Merkle tree was initially described with respect to authentication of signatures, the patent also mentions, that "it may be used to authenticate a piece of information in a list of information", which better matches with our use case.

4.2 Storing Claims

The claims can be stored as the leaves of a Merkle tree. We shall denote claims as C_i, where i is the index of the claim. In a Merkle tree, the leaves themselves should be hashed with a salt to ensure confidentiality of neighbouring values when constructing proofs. Recursively, all node pairs are then hashed in a determined manner. For example $H(a, b) = H(a \parallel . \parallel b)$, where a and b are values to be hashed and H is an agreed-upon hash function. Finally, only one node, the root node is left (see Figure 5).

As cryptographic hash functions have the one-wayness property, there is no formula to find the input based on the hash value other than brute-forcing the input space. Thus, although the value of the root node directly depends on the values of the claims, the claims themselves cannot be deduced from the root node in case the input space is

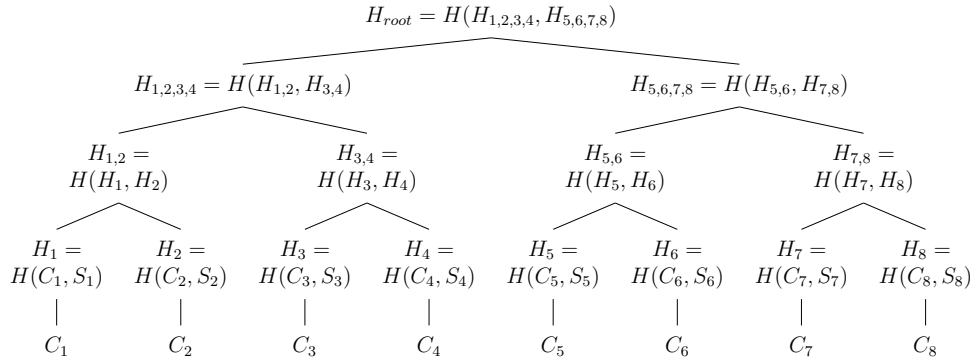

Figure 5. An example Merkle tree with 8 claims. Here, C_i is a claim, S_i its salt value, $H(x, y)$ a two-argument hash function.

sufficiently large. The root node could then be signed by the authentication service, and provided by the client with proofs for validation.

4.3 Salting

In order to prove a claim's existence in the tree, hash values are used to construct the root hash. As hash values H_i are used in proofs, claims C_i would become brute-forceable if salts S_i were a known constant or if any hash H_i would be created solely from claim C_i. For example, a boolean value has a small enough value space, that hashing it without a salt would require up to two comparisons to brute-force. In order to protect confidentiality of claims C_i in all cases, the following salting mechanism is proposed:

$$S_i = H(C_i, i, P)$$

Here, the three-argument hash function H is constructed similarly to the two-argument hash function, for example: $H(a, b, c) = H(a \parallel . \parallel b \parallel . \parallel c)$. P is a sufficiently long random value or general-purpose pepper supplied by the authenticating party with the entire tree along with the root hash. P is not to be forwarded by the client, but used only to locally calculate the proof tree. S_i should be supplied along with C_i to enable calculation of the corresponding H_i.

As P is unknown by receiver of the proof, brute-forcing a non-supplied claim C_j from a given H_j becomes very difficult. This is because $H_j = H(C_j, S_j)$, where even if value space of C_j is small, the value space of $S_j = H(H(C_j, j), P)$ is sufficiently large, since P is an unknown and sufficiently long random value. Also, in case $C_i = C_j$ where $i \neq j$, we do not want $H_i = H_j$ or it is possible to deduct that the values C_i and C_j are equal – so the indices i and j are also used in generating the salt values S_i and S_j respectively resulting in differing hashes H_i and H_j. OWASP recommends using at least

32 randomly generated characters as a password pepper [OWA], being similar in nature, it should also be enough for this use-case.

Salting increases all proof paths by one hash, effectively doubling the amount of leaves on the proof tree. When privacy of claims is not sought for, proof size could be decreased by having $H_{i,j} = H(C_i, C_j)$ (see Figures 6 and 7).

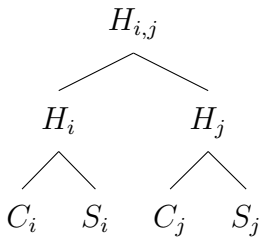

Figure 6. A Merkle tree with claims C_i, C_j and their respective salts S_i, S_j.

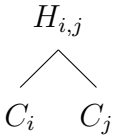

Figure 7. A Merkle tree with two claims C_i, C_j and no salts.

4.4 Verification

To verify, that a piece of data – a claim – is a leaf in a Merkle tree, a proof path from the leaf to the root node needs to be provided. This would contain the claim to be proven and the hash values required to compute the node hashes up to the root node. The path does not contain the hash values, that can be computed using the claim and other provided hash values.

4.4.1 Proof Tree

As multiple claims may need to be provided for authorization of a web request, it becomes inefficient to provide multiple overlapping paths for verification. Instead, a proof tree could be provided which can be easily verified (see Figures 8 and 9).

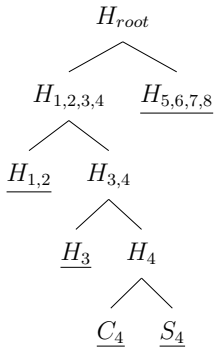

Figure 8. A proof tree for claim C_4. Only the underlined values are needed to find H_{root}.

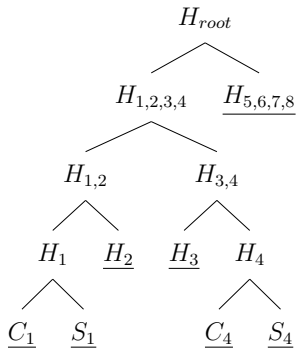

Figure 9. A proof tree for claims C_1 and C_4. Only the underlined values are needed to find H_{root}.

4.4.2 Metadata and Signature

Providing a Merkle tree's root hash and the proof tree is not sufficient to prove validity of some claims. The root hash also needs to be signed by a trusted authority.

It would be simple to use HMAC for signing and verifying the root hash with a predefined secret key. In the real world, that key cannot always be shared between services, so asymmetric cryptography is used which makes it possible to verify the signature with a public key. Additionally, it might be useful to include some metadata, for example the expiration time of the claims. Fortunately, the existing JWT standard can be relied upon to sign the Merkle tree's root hash and any other metadata, as it supports multiple different signature algorithms.

4.4.3 Limiting the Audience

As such proof trees are potentially used with multiple services, it would be wise to include the service identifiers into the claims in the token. Otherwise, a malicious service could re-use the same token to perform a request with another service.

For example, the client could receive the claims with a list of permitted audience, but supply only one of these with the request as the proof (see Figure 10). The services would need to assert the existence of their respective values in the proof for this scheme to work. This would ensure that malicious third-party services could not cause harm elsewhere.

In an ideal world, independent services shouldn't need knowledge about other services. However, such list of permitted audience needs to be known during token generation in the authentication service, requiring revision when the service dependencies are changed.

```
claims: {
  ...,
  audience: ["service1", "service2"]
}
```

Figure 10. An example set of claims received, with permitted audience. Supplying a proof only with the claim value of "service1" would prohibit the same set of claims be used with a service requiring "service2".

4.5 Claim Structure

User permissions can be represented in different formats. Stringified permissions could be represented as an array, while a more complex permissions could be represented as an object, containing multiple different types of claims and even nested objects.

4.5.1 Arrays

To store an array of data as the claims, an ordering scheme would need to be determined to ensure the same tree could be built from the same data. The simplest way to do so would be to use alphabetical ordering (see Table 1).

4.5.2 Objects

A more complex data structure could also be represented as claims (see Figure 11). JSONPath [Goe07] could be used to flatten and serialize the data as an array containing key-value pairs. The JSONPath notation can be used to represent every possible key in an

Table 1. Claims for an array: foo, bar, baz, qux, quux, corge, grault, garply.

Claim	Value
C_1	bar
C_2	baz
C_3	corge
C_4	foo
C_5	garply
C_6	grault
C_7	quux
C_8	qux

object's structure. The claim itself could be constructed as follows: "key=value" (see Table 2). The receiving service could use JSONPath to re-build the object from available data for convenience. Such object is enough for validation when only the provided claims are required to perform the intended action by the service (see Figure 12).

```
{
    foo: "bar",
    baz: 1,
    qux: true,
    quux: null,
    corge: ["grault", "garply", "waldo"],
    fred: {
        plugh: "xyzzy",
    }
}
```

Figure 11. An example data object.

4.6 In Practice

The technical description of the proposed authorization scheme does not limit its usage to the web. However, as the solution can be easily applied to web applications, only that will be explored in this work.

4.6.1 Reference Implementation

A reference implementation as a JavaScript library is supplied along with a demonstration application. These are published as a GitHub repository [Ees20]. The library makes it easy to use all of the features discussed beforehand in a real-world scenario.

Table 2. Claims for the object in Figure 11.

Claim	Value
C_1	$.baz=1
C_2	$.corge[0]="grault"
C_3	$.corge[1]="garply"
C_4	$.corge[2]="waldo"
C_5	$.foo="bar"
C_6	$.fred.plugh="xyzzy"
C_7	$.quux=null
C_8	$.qux=true

```
{
    foo: "bar",
    corge: [<1 empty item>, "garply"],
    fred: {
        plugh: "xyzzy",
    }
}
```

Figure 12. The object generated by claims C_3, C_5 and C_6. The claims are described in Table 2.

The library provides methods for signing arrays or objects, and offers the option to use salting. The demo consists of two layers – back-end and front-end applications. There is an endpoint for providing the claims along with a signature and the pepper, computed by the library. Also, there are a few endpoints that require different combinations of the initially provided claims to demonstrate usage of single- and multi-claim proofs. The front-end is configured to send requests along with the proof trees constructed by the reference library.

In the demonstration, proofs are passed as JSON-encoded trees in request headers. In a real-world scenario, the claims and the signature can also be persisted in browser's storage to minimize server load. These need only be cleared when it becomes known that the user permissions have changed or the user itself has changed.

4.6.2 OAuth 2.0

OAuth 2.0 is a commonly used multiple-party access-control framework. Described in RFC 6749 [Har12], the OAuth 2.0 Authorization Framework is in itself a complete framework, but the specification leaves room for customization. The token used for access control can be substituted for a Merkle-tree-based approach, to achieve more standardized third-party communication.

Using the framework, an end-user first obtains an access token, which can then be used for authorization when querying resources. According to the standard, the access token is "a string denoting a specific scope, lifetime, and other access attributes" and the format is not strictly set. Also, it is specified, that "a single authorization server may issue access tokens accepted by multiple resource servers".

While the intention of the OAuth 2.0 standard seems to be that complete control over the access token scope lays on the side of an authorization server, the access token could be constructed in the form of the proposed Merkle tree mechanism. This would allow the end-user to omit certain properties either for the sake of efficiency or privacy when requesting a resource from a resource server.

5 Analysis

The Merkle-tree-based authorization method is inspired by shortcomings of the JWT. Thus, it is useful to analyze security, privacy and efficiency by comparing the two.

5.1 Security and Privacy

As the JWT is meant to be supplied with requests in one piece, it also allows for encryption [JBS15]. Doing so, any sensitive claims can be kept from the client or any unintended audience. The proposed Merkle-tree-based approach can not be fully encrypted to hide information from the client, as the client needs to be able to identify individual claims for building proof trees. Still, the proof trees could be encrypted by the client for protection from unintended audience using a public key, but that is not the scope of this work. Unnecessary values can simply be omitted from the proofs, which reduces the impact of such encryption scheme.

Consider a setup consisting of an authentication service, a third-party service and a client (see Figure 13). After authentication, the client receives a signed set of claims. A subset of these claims can be used to build a proof tree, which can be provided along with a request to a third party. To validate the proof, the third party needs to verify the signature using the verification key from the authentication service.

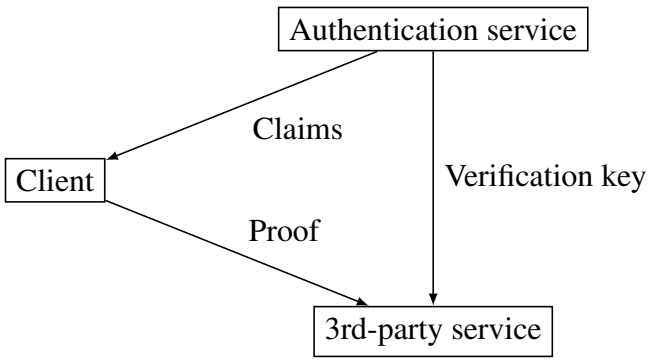

Figure 13. The parties and communications involved.

Next, we will take a look at all possible ways the system can be misused. The following analysis also applies to the JWT, especially in the case where a JWT with as few claims as possible is generated for every third-party service by the authentication service. By having the client selectively choose the claims, the Merkle tree approach merely simplifies the process of limiting the information available to third-party services.

5.1.1 Compromised Authentication

Consider, that the authentication service is compromised. This would mean, that the attacker could issue false keys for verification or false claims for authorization. Having such power over all the following communications, the authentication service is a cornerstone and a single point of failure of the whole architecture. For example, the attacker could issue claims to impersonate every user to every service that is part of this architecture. It is important to prevent a breach in the authentication service, as it could have potentially devastating effects.

It should be noted that with third-party services, the mechanism for the signature generation and verification should employ asymmetric cryptography. Otherwise, the key used for proof validation could also be used to sign new claims. So if only the public key were shared, it could only be used to validate the claims, not sign them.

In case the authentication service has recovered from an attack, changing the signing key should be enough to invalidate all signatures. The dependent services should also be informed of the key change. Doing so, forces all clients to authenticate again.

Provider of the authentication service should also have a great desire to not violate the trust of their users and the dependent third-party services. A breach would decrease their trustworthiness and risk their business. This is why we can trust the authentication service.

5.1.2 Compromised Client

Consider, that the client is compromised. This would mean, that the attacker could perform malicious requests on behalf of the user. The damage here is much more limited compared to an attack on the authentication service. As it is impossible to distinguish the real user from an attacker, it is impossible to protect the client from taking malicious actions. Any potential way of revoking specific signatures reduces the benefits of using signed claims, as a central service would need to be queried, which is what we are trying to avoid with the proposed scheme.

It is possible to mitigate the mentioned risks by introducing multi-factor confirmation for actions with a greater impact. Also, by having a reasonably small expiry time for the claims, the malicious actions would need to be taken quickly. This would more likely be noticed by the client, who would then become aware of an attack and could employ means to revert the damage. Regarding the data transfer, on the modern web, all requests can be encrypted using the TLS protocol, so the request and response integrity should be ensured.

If the user itself were malicious, they could only perform actions they had authorization for, as determined by the authentication service generating the signed set of claims. Thus, unauthorized actions cannot be taken without compromising the authentication service or the 3rd-party service.

5.1.3 Compromised Third Party

Consider, that the 3rd-party service is compromised. If it receives a proof tree which fixes the audience to this 3rd-party service, the proof would not be usable with other services, limiting the amount of potential harm. However, it is up to the other services to enforce the audience requirement, as otherwise nothing prevents the proof from being reused.

Due to having less trust with third parties, the proofs would have to contain only absolutely necessary information for the request. This inherently decreases the risk of a privacy breach by an attacker.

The third party should have a desire to not be breached and should take reasonable precautions on their own. The risks related to dependence on a third-party service can be mitigated by limiting the information it receives.

5.2 Efficiency

The JWTs are effectively signed payload objects. The Merkle-tree-based authorization consists of hashing claim pairs to form a binary tree that can be used to find proof paths or proof trees. Such binary tree constructed of n leaves consists of $2n - 1$ nodes or $n - 1$ non-leaf nodes and $\lceil log_2(n) \rceil + 1$ levels.

5.2.1 Time Complexity

JWT signature generation and verification consist of only one operation on the serialized payload. Thus, they run in constant time.

Building the Merkle tree from n claims runs in linear time, as $n-1$ hashing operations need to be performed to construct the non-leaf nodes. Computing the root node of the smallest possible proof tree – proving a single value – runs in logarithmic time, as there is a single hashing operation required for each of the $\lceil log_2(n) \rceil + 1$ levels of the tree. Computing the root node of the largest possible proof tree – proving all values in the Merkle tree – runs in linear time, as $n - 1$ hashing operations need to be performed to construct the non-leaf nodes.

The Merkle-tree-based approach also involves computation on the client side. It takes time to find sufficient proof for root hash verification, and also to re-generate the Merkle tree on the client side to optimize finding the proofs. Finding sufficient proof for a single value runs in logarithmic time, as there are $\lceil log_2(n) \rceil + 1$ levels in the tree. Finding sufficient proof for near-all values runs in linear time, as the proof tree can be constructed by adding proof paths of the values one-by-one one node at a time, stopping at already added nodes, with up to $2n - 1$ additions. Finding the proof for all values can be further optimized as it is identical to the underlying Merkle tree. Re-generation of the Merkle tree is of the same time complexity as the initial generation.

Signing and verifying the root node of the Merkle tree run in constant time, as they consist of only one operation on the hash value. Using a JWT here does not change the time complexity.

Combination of the operations to generate and sign a Merkle tree, runs in linear time. Combination of the operations to compute and verify the root node of a proof tree, runs from logarithmic to linear time, depending on the amount of proven claims.

Optionally salting the values to ensure their confidentiality in proofs, effectively doubles the amount of leaves in the Merkle tree. In addition, calculation of the pepper is a single operation during the token generation and computation of the salt values for n claims is a total of n hashing operations. While adding to the time, these do not change the time complexity of signing and verifying the Merkle tree.

The overall time complexity of the proposed authorization scheme is worse in comparison with the JWT. However, the proposed scheme excels in communication complexity, which we consider to be more important. This is because internet communication is generally orders of magnitude slower and potentially more expensive, than the computational resources of a single system.

5.2.2 Communication Complexity

To minimize request overhead, JWTs are often used with fixed audience with as few claims as possible. The Merkle-tree-based authorization scheme is meant to be more generic, containing more different kinds of values. While analyzing the communication complexity, we can ignore these differences in usage patterns to show that the request overhead of the proposed scheme doesn't grow as fast.

A JWT contains a fixed-length header, Base64-encoded claims and a fixed-length signature. It is linear in size in regards to the amount of claims.

A Merkle tree can be rebuilt from only a given set of claims, and requires a fixed-length signature and optionally a pepper for verifiable proofs. Thus, the authentication response can also be linear in size in regards to the amount of claims.

The minimal proof tree of a Merkle tree proves a single value. Such proof tree contains the value and one hash from each level of the tree. As there are $\lceil log_2(n) \rceil + 1$ levels in a Merkle tree of n claims, such proof tree provides logarithmic communication complexity.

The maximal proof tree of a Merkle tree proves every value. Such proof tree contains all of the leaf values, which can be used to compute all other levels of the tree, including the root hash. Such proof tree provides linear communication complexity.

The communication complexity achieved by the Merkle tree approach is as good, if not better than the communication complexity achieved by the JWT. Assuming only a small subset of the claims are actually needed for request authorization, the proof trees of the proposed approach achieve near logarithmic communication complexity. However, since the hash values can be quite long, the benefit is not obvious with fewer claims.

5.2.3 Achieving Request Overhead Smaller than the JSON Web Token

The proposed authorization mechanism does produce proofs larger than JWTs when there are few claims. Let's calculate roughly when a JWT of n values would surpass a minimal proof tree of a Merkle tree with n values in size. Only the minimal proof tree is analyzed, as the Merkle-tree-based proofs are meant to contain only a small subset of claims. Efficiency of proving all claims is similar to a JWT. We are looking for n, where the following is true:

$$l(\text{JWT of } n \text{ values}) > l(\text{minimal proof tree of a Merkle tree with } n \text{ values})$$

Let's define a function l which results in the length of the component specified as the argument. Let's also define the following parts of a JWT: He is the header, P_n is the payload with n values and S is signature, C is an average claim with the JSON structure. Finally, let's define the following parts of a proof tree of a Merkle tree: N_n is the minimal proof tree of a Merkle tree with n-leaves, and Ha is a hash value. We can express the following approximation:

$$l(\text{JWT of } n \text{ values}) = l(He) + l(P_n) + l(S)$$
$$l(\text{minimal proof tree of a Merkle tree with } n \text{ values}) = l(\text{JWT of 1 hash}) + l(C) + l(N_n)$$
$$l(\text{JWT of 1 hash}) = l(He) + l(H) + l(S)$$
$$l(P_n) = n * l(C)$$
$$l(N_n) = log_2(\lceil n \rceil) * l(Ha) \approx log_2(n) * l(Ha)$$

By substituting the values in the original inequality, we get:

$$l(He) + n * l(C) + l(S) > l(He) + l(Ha) + l(S) + l(C) + log_2(n) * l(Ha)$$
$$n * l(C) > l(Ha) + l(C) + log_2(n) * l(Ha)$$
$$(n-1) * l(C) > (log_2(n) + 1) * l(Ha)$$
$$\frac{n-1}{log_2(n)+1} > \frac{l(Ha)}{l(C)}$$

We will take reasonable assumptions to provide an estimation for reference. Let's assume we require SHA-3 with 256-bit output as the hashing algorithm, which would be encoded into a 64-character string. When combined with the data structure in JSON, totaling 11 symbols (ignoring the subtree) as such: $\{"l" :< subpath >, "R" :< hash > \}$. Then $l(Ha) = 64 + 11 = 75$ for a single level of hash information in the proof tree. Assuming reasonably, that an average value in the claim can be conveyed as 6-digit integers in a string format, with a short, for example a 3-letter key name with 3 structural

symbols, such as "usr" : 123456, then $l(C) = 6 + 3 + 3 = 12$. We get the following equation:

$$\frac{n-1}{log_2(n)+1} > \frac{75}{12}$$

Solving this, we get, that $n \gtrapprox 40$. This means, that the efficiency of the Merkle-tree-based authorization mechanism becomes apparent at around an estimate of 40 claims. When only a small subset of all claims is needed with each request and efficiency is sought for, the Merkle tree approach should be a good candidate, as the effectiveness becomes apparent rather early relative to the amount of claims used.

5.3 Existing Applications

Simple monolithic web services usually keep permissions in user state in system memory. A way to increase scalability is to externalize user state, and the Merkle tree authorization mechanism could be used to ease this transition. This would make migration to a stateless architecture easier for developers, especially if modification or invalidation of user state is not required.

It is also reasonable to use the proposed authorization mechanism as a validated cache of information. Theoretically, a signed user identifier should be sufficient to derive user's permissions and other authorized data, for example by querying other services. However, in practice, performing such subqueries can take significant time. The proposed authorization scheme could be used to cache the necessary information on the client side, which could then be supplied along with subsequent requests. A JWT could be used for similar purposes, but when lists of values are involved, it would be safer to use the Merkle tree approach, as token size is limited by network speed and the standards involved.

6 Conclusion

As a part of this work, an authorization scheme, that uses the Merkle tree was proposed. Combining the existing JSONPath specification with the concept of proof trees and an efficient salting mechanism, entire objects could be used as claims that can be included in web requests.

The proposed authorization scheme improves upon the scalability and privacy of the JSON Web Token standard. A reference implementation of such authorization scheme was also provided. The intended effects are beneficial in multiple real world scenarios and should integrate well with existing technologies.

While this paper explored the use of Merkle trees in modern web authorization schemes, some aspects were left unexamined, thus, future work could expand this idea in multiple different ways. Firstly, the proof size could be further reduced by picking a hash function with smaller output, so more work could be done finding a balance between proof size and security requirements. Secondly, the library could be implemented in other programming languages and optimized for higher developer convenience. Thirdly, better ways of fixing the audience of a proof should be explored, as the current solution requires the permitted audience to be known when signing the claims.

References

[Ees20] Gregor Eesmaa. Library for merkle tree based authorization on the web. Accessed 8 May 2020, https://github.com/gregoreesmaa/merkle-auth, 2020.

[Fac] Facebook. Instagram Basic Display API Overview. Accessed 8 May 2020, https://developers.facebook.com/docs/instagram-basic-display-api/overview.

[Goe07] Stefan Goessner. JSONPath - XPath for JSON, February 2007. Accessed 8 May 2020, https://goessner.net/articles/JsonPath/.

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. Internet Requests for Comments, RFC 6749, 2012. Accessed 8 May 2020, http://www.rfc-editor.org/rfc/rfc6749.txt.

[Her18] Eran Hertz. The Rise and Fall of Server-Side Sessions: Part 1 - When the cookie crumbles, May 2018. Accessed 8 May 2020, http://codematters.tech/the-rise-and-fall-of-server-side-sessions-part-1/.

[JBS15] Mike Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). Internet Requests for Comments, RFC 7519, 2015. Accessed 8 May 2020, http://www.rfc-editor.org/rfc/rfc7519.txt.

[Mer82] Ralph C. Merkle. Method of providing digital signatures. U.S. Patent 4309569, January 1982.

[Nai17] Mayukh Nair. How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play. Medium, October 2017. Accessed 8 May 2020, https://medium.com/refraction-tech-everything/3a40c9be254b.

[OWA] OWASP Foundation. Password Storage Cheat Sheet. Accessed 8 May 2020, https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[Sum19] Fajar Sumirat. From Monolith to Microservices, October 2019. Accessed 8 May 2020, https://fajarsumiratmuhrip.wordpress.com/2019/10/31/from-monolith-to-microservices/.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Gregor Eesmaa**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Authorization of web requests based on Merkle trees,
   (title of thesis)

   supervised by Kristjan Krips.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Gregor Eesmaa
08/05/2020