Authorization Tokens- cookies and macaroons

Tahera Fahimi
Univeristy of Calgary - Computer Science Department
Calgary, Canada
tahera.fahimi@ucalgary.ca

ABSTRACT

Authentication is one of the most complicated problems in web security; yet, on the Web, and in the Cloud, sharing is still based on rudimentary mechanisms [2]. In order to solve this problem, there are several bearer credentials. These tokens do not consider different security policies and access control in distributed systems or controlled sharing systems. We search for a flexible and easy way to implement authorization conditions in tokens with macaroons.

This paper research macaroons based on [2]: "flexible authorization credentials for Cloud services that support decentralized delegation between principals. Macaroons are based on a construction that uses nested, chained MACs in a manner that is highly efficient, easy to deploy, and widely applicable" [2].

Although macaroons are bearer credentials, like Web cookies, macaroons embed caveats that attenuate and contextually confine when, where, by who, and for what purpose a target service should authorize requests [2]. This paper describes macaroons and motivates their design, compares them to other credential systems, such as cookies, and checks a specific scenario with sequence architecture for application considerations.

KEYWORDS

Bearer credential, controlled sharing, sequence architecture

1 INTRODUCTION

In all access services systems, we have twin authentication and authorization problems. In simple, for the authentication, we take the username and password of the user and check their validity. For the authorization, we make sure that the user actually allowed to make this specific request. We set each route up with a policy like "read", "write" and then permitted individual users to operate within each policy so when the request comes in, we check the policy on the route, and then we will figure out if we have granted this user permission by checking that token stuck in the context. In other words, we would set up this set of access control lists inside our service. However, this approach has so many problems that people are wrestling with them.

If we separate authorization problems, we will see two difficulties: capability vs. identity. In capability, we directly grant someone the capability to do something. In identity, ask who someone is, and then check whether the person can do it or not.

For example, consider a car that can have a key. In capability, the car's owner can pass the key to the valet without any permission checking. It is dangerous because the valet can steal the car, so it is not a good idea to have a token with so much power. On the other hand, if it is an identity-based car, it should be programmed based on different valets. Based on the ACL model[3], relying on identity is also dangerous when there are more than two principals exists so in other words, if there is a restricted action and a deputy with authority to take that action, the deputy can get confused and act on behalf of another actor who does not have that authority, like a CSRF attack[7].

Macaroon could safely combine the strengths of both the capability and the identity models. An example is the car key could be attenuated or limited to make it much safer, like the car key only works for an hour or one mile or if the person wears the valet company uniform. So these are some caveats based on the context that can be assigned to the token attenuate it.

To understand the behavior of the macarons, we should understand the basic authentication(auth). In basic auth, a client sends a request that its authorization header field contains username and password. In this case, we face two problems. First, the username and password should be provided on every request. The second issue is that the auth only addresses the authentication issues, so we still need some identity-based access control list logic to limit what users can do.

If we use a token with HMAC, we solve the first problem, but we still have the second problem, so here is where macaroons come in handy. Macaroons have a way of baking that authorization rules into the token as caveats. Macaroons can add different authorization principles as different layers in a cookie.

"Clouds as modern software are often constructed as a decentralized graph of collaborative, loosely-coupled services" [2]. Those services comprise different protection domains, communication channels, execution environments, and implementations—with each service reflecting the characteristics and interests of the different underlying stakeholders[2]. Thus, security and access control are critical, especially as Clouds are commonly used to share private, sensitive end-user data.

Unfortunately, controlled sharing in Clouds is founded on basic, rudimentary authorization mechanisms, such HTTP cookies that carry pure bearer tokens [4]. Thus, today, it is practically impossible for the owner of a private, sensitive image stored at one Cloud service to email a URL link to that image, safely such that the image can be seen only by logged-in members of a group of users that the owner maintains at another, unrelated Cloud service [2]. This has given many opportunities for impersonation and eavesdropping Currently. Macaroons can solve this problem with contextual caveats which are authorization principals in Cloud.

2 PREVIOUS WORKS

There are different security tokens in the web application. These tokens are usually temporary strings that grant access to some resource, and we usually get them by exchanging a long-term credential like a password.

When using credential tokens, we need to validate them before using their contents. Otherwise, the token can be untrustable. Basically, we replace a cookie with a bearer token in the first format.

So instead of checking the cookie from the database, we can use cryptography to verify the token locally. The verification depends on the format of the token.

" JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification" [6]. JWT is the most popular token. When we are talking about the JWT, we actually mean JWS. JWS is consists of header, payload, and signature. Each of these sections is base64 encoded, and they are all separated by a period. We can use the shared secret to create HMAC for the signature part, public key, or none. The algorithm that we use for signature is defined in the header. We can add different features by defining them in the header.

Paseto is created as a response to the complexity of the Jose specs. With Paseto we only have two local and public versions, and it is much like JWT. It is broken up into four parts (the version, the purpose, the payload, and the optional footer). Local Passeto tokens are encrypted with a preshared key. Public tokens are not encrypted. They are equivalent to a JWS and use public-key pair to generate the signature.

The authorization rules are not inside the token in all of the above tokens. So we still need to verify the authorization for each token as a simple auth in the server.

Also, Most authentication mechanisms based on public-key certificates are not directly suited to the Cloud, since they are based on more expensive primitives that can be difficult to deploy and define long-lived, linkable identities, which may impact end-user privacy. [2] For example, JWT token while last long for more than one session.

Even so, the inflexibility of current Cloud authorization is quite unsatisfactory. "Most users will have first-hand experience of the resulting frustrations" [2]. For example, because they have clicked on a shared URL, only to be redirected to a page requesting account creation or sharing of their existing online identity [2].

So we find that cookies(and similar tokens) are ubiquitous in Cloud authorization. Also, they are easy to steal, carry broad authority, and lack flexibility. With macaroons, we are trying to introduce better cookies with arbitrary caveats, i.e., restrictions on access.

3 MACAROONS

"Macaroons are authorization credentials that provide flexible support for controlled sharing in decentralized, distributed systems. Macaroons are widely applicable since they are a form of bearer credentials—much like commonly-used cookies on the Web—and have an efficient construction based on keyed cryptographic message digests" [2].

Macaroons are designed for the Web, mobile devices, and the related distributed systems collectively known as the Cloud [2].Macaroons aim to combine the best aspects of using bearer tokens and using flexible, public-key certificates for authorization by providing the wide applicability, ease of use, and privacy benefits of bearer credentials based on fast cryptographic primitives, also, precise restrictions on how, where, and when credentials may be used.

"Macaroons allow the authority to be delegated between protection domains with both attenuation and contextual confinement (we will talk about attenuation and contextual confinement in Concept section in details.). For this, each macaroon embeds caveats which are predicates that restrict the macaroon's authority, as well as the context in which it may be successfully used" [2]. For example, such caveats may attenuate a macaroon by limiting what objects and what actions it permits, or contextually confine it by requiring additional evidence, such as third-party signatures, or by restricting when, from where, or in what other observable context it may be used [2].

Also, Macaroons allow Cloud services to authorize resource access using efficient, restricted bearer tokens that can be delegated further, and, via embedded caveats, attenuated in scope, subjected to third-party inspection and approval, and confined to be used only in certain contexts [2].

Conceptually, macaroons make an assertion about conditional access to a target service, along these lines: "The bearer may perform a request as long as predicates $C_1, \ldots, C_n$ hold true in the context of that request."[2]

In distributed systems, the complexity of such assertions can grow surprisingly quickly. Due to their flexible, efficient HMAC-based construction, macaroons can enforce such complex assertions despite tight freshness constraints and the need to guard privacy closely.

3.1 Concepts

To restrict the authority of a derived macaroon, caveats are added and embedded within it. "Each macaroon caveat states a predicate that must hold true for the context of any requests that the derived macaroon is to authorize at the target service— with that predicate evaluated by the target service in the context of each particular request" [2]. For example, every macaroon is likely to contain one or more validity-period caveats whose predicates restrict the time (at the target service) during which the macaroon will authorize requests; similarly, each macaroon is likely to have caveats that restrict the arguments permitted in requests [2].

It is straightforward to add and enforce such first-party caveats that confine the context observed at the target service when a macaroon is used. First-party macaroons naturally support attenuation and delegation.

Here is an example of delegation and attenuation: Consider SA as a provider for the SB, so SA creates a macaroon (call it M1) that contains the ID of SB as a caveat and passes it to the SB. For every client that requests service from SB, SB adds the matched requested caveat to M1 and sends it to the client. Now the client can send back this new macaroon to SA, and SA, after verification, sends the requested data or service to the client. In this example, we attenuate M1 by adding caveats of SB.

Macaroons' flexibility stems mostly from third-party caveats, their main innovation, which allow a macaroon to specify and require any number of holder-of-key proofs to be presented with authorized requests. For example, instead of using time-based caveats,

macaroons might use caveats for a third-party revocation-checking service, and require authorized requests to present fresh proofs discharging those caveats.

"Such third-party caveats can ensure that each request authorized by a macaroon is subject to any number of extra steps for authentication, authorization, or audit, to name just a few, checks of network time, authentication at identity providers, auditing and provenance tracking at document repositories, scanning by antivirus engines, and vetting by a DoS-prevention and risk-analysis service"[2].

We should know that even though the macaroon can enforce different and complex policies, the end-user is utterly unaware of this construction. So the macaroons, like any other bearer tokens, can be created and used with only software awareness.

3.2 Structure and Mechanism

A macaroon is constructed from a sequence of messages and a chain of cryptographic digests derived from those messages. This digest is computed using HMAC function. Each macaroon contains only the final HMAC value of the chain, which serves as the macaroon's signature and can be used in the verification step.

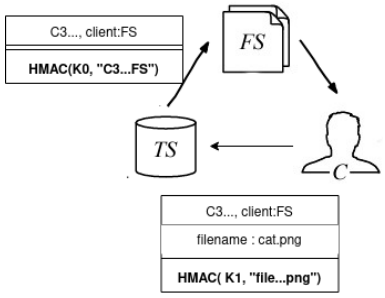

Figure 1: First-party macaroons. TS gave FS first macaroon(top right macaroon), which FS can guarantee its users access to service by adding additional caveats and sending it to users. Then user, in order to gain access, sends back the macaroon(down right macaroon) to TS, and after validating the macaroon by TS, gives access to the user[2].

The first message in a macaroon is required to be a public, opaque key identifier that maps to a secret root key known only to the target service[2].(Such key identifiers can be implemented using random nonces, indices into a database at the target service, keyed HMACs, or using public key or secret-key encryption). Starting with this root key, a macaroon's chain of HMAC values is derived by computing, for each message, a keyed HMAC value, nested so that each such value itself is used as the HMAC key for the next message. Thus, a macaroon with two messages $MsgId$ and $MsgCav$ and root key K_R could have the signature $HMAC(K_{tmp}, MsgCav)$, where K_{tmp} is $HMAC(K_R, MsgId)$ [2].

An example of this concrete macaroon construction is shown in Figure 1

So far, only the target server is trusted for authorization decisions. The third-party caveats in macaroons are requirements for holder-of-key proofs: assertions of one or more predicates signed by the

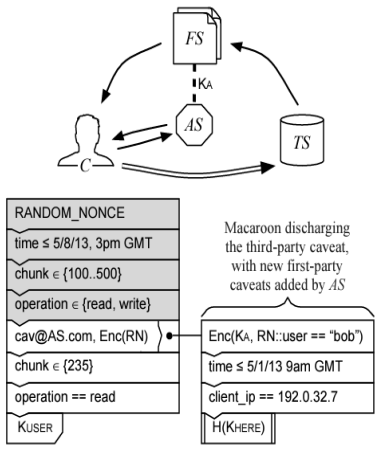

Figure 2: Two macaroons that together give client C limited-time read access to chunk 235 at the key-value storage service TS. Using its macaroon from TS, the "forum service" FS derives for C a macaroon (on the left) that embeds a third-party caveat for an authentication service AS, confining access to clients C where a user "bob" is logged into AS. That caveat contains an encrypted root key and a key identifier. Using that key identifier, AS derives for C a macaroon (on the right) that asserts, for only a limited time, and for only one IP address, that AS has verified client C's user[2].

holder of a key. Third-party caveats require another service to check restrictions.

The main advantages and innovative aspects of macaroons are illustrated by the example of Figure 2, in combination. In the Figure 2, we add the third-party caveats to the delegation and attenuation example. From its existing macaroon for the service TS, the forum service FS can derive a macaroon for the client that attenuates what aspects of TS are accessible; simultaneously, FS can delegate the authentication of users to a third-party service AS, using the key K_A it holds for AS. (Of course, for this, FS must know AS, and trust it to perform authentication.) The AS service can evaluate the request from client however it chooses, before returning a macaroon that discharges the $user == "bob"$ caveat. Notably, this caveat discharge macaroon may be very short lived, due to the efficiency and low overhead macaroons' HMAC-based construction [2].

3.3 Design

This section formally defines macaroons' notation, structure, and operations, precisely defines macaroon-based authorization.

Macaroons use primitives whose domain of values are $Keys$ for all cryptographic keys, $Locs$ for the locations of all services, and $BitStrs$ for all bit-string values. Keys and locations are all bit-strings; thus $Keys \cup Locs \subseteq BitStrs$. Macaroons are based on a secure, keyed cryptographic hash function MAC, and its output message authentication codes, as well as secure encryption and decryption functions Enc and Dec, with the following type signatures: [2]

$$MAC : Keys \times BitStrs \rightarrow Keys$$

$$Enc : Keys \times BitStrs \rightarrow BitStrs$$

$$Dec : Keys \times BitStrs \rightarrow BitStrs$$

We consider that each key output from a MAC operation can be used as a key for another MAC operation in chained-MAC, or for a symmetric-key-based *Enc* and *Dec* operations. So If we consider a list of bit-strings as $[b_1, \ldots, b_n]$, under a key k, we can define $MAC(...MAC(MAC(k, b_1), b_2)..., b_n)$ as the final signature in the macaroon for verification, where $MAC(k, b_1)$ is the key for computing MAC of b_2, and so on.

Principals are each associated with a location, a set of keys and macaroons, and the ability to manage secrets; the universe of all principals is denoted *Prncs* [2]. For each secret, principals are assumed to be able to construct a public key identifier that confidentially, and appropriately conveys this secret. Such key identifiers have long been used in security protocols, with each acting as a certificate that reminds a principal P of a certain secret it holds; their specifics may vary across principals, with possible implementations including secret-key certificates based on symmetric, shared-key encryption as well as simple, ordinal identifiers that index into a principal's persistent, confidential database[2].

Every macaroon has a root key that the target service principals mint. The structure of a macaroon is like this: it begins with a key identifier for the target, followed by a list of caveats and a macaroon signature (which is computed using the root key for chained-MAC). The key identifier or the macaroon identifier conveys the secret root key. "Both first-party and third-party caveats are defined using the same structure that consists of two identifiers: a caveat identifier meant for the caveat's discharging principal and a verification-key identifier meant for the embedding macaroon's target service, with the embedding macaroon's signature ensuring the integrity of both identifiers" [2]. The structure of a macaroon is defined as follows:

Defenition[2]: A macaroon M is a tuple of the form

$$macaroon_{@L}\langle id, C, sig\rangle$$

where,

- $L \in Locs$ (optional) is a hint to the target's location.
- $id \in BitStrs$ is the macaroon identifier.
- C is a list of caveats of the form $cav_{@cL}\langle cId, vId\rangle$, where
  - $cL \in Locs$ (optional) is a hint ti a dischange location.
  - $cId \in BitStrs$ is the caveat identifier.
  - $vId \in BitStrs$ is the verification-key identifier.
- $sig \in Keys$ is a chained-MAC signature over the macaroon identifier id, as well as each of the caveats in C, in linear sequence.

In third-party caveats, the caveat identifier encodes one or more predicates and a root key; its construction may vary across caveat-discharging principals. The verification-key identifier encodes only the caveat root key, encrypted using the current signature from the embedding macaroon; the target decrypts this identifier to obtain the root key since it can always recover all intermediate signatures in a macaroon. You can see a third-party caveat with a caveat identifier and verification-key identifier in the Figure 3. The third-party creates the discharged macaroon with the purpose of authentication and authorization can be seen in the Figure 4.

Figure 5 shows the operations to create and extend macaroons, prepare them for use in a request, and verify them at a target service. The operations are written as pseudocode. For a macaroon M, the

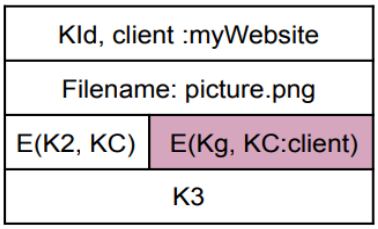

Figure 3: The main provider can verify the whole macaroon with the root key from first caveat and then gain the k_c, then verify the discharge macaroon caveats.

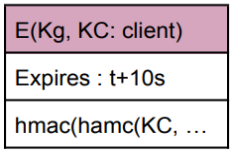

Figure 4: The discharge macaroon that is created by the third-party with the purpose of authentication and authorization.

notations $M.id$, $M.cavs$ and $M.sig$ are used to refer to its identifier, list of caveats, and signature, respectively [2].

Creating macaroons: The function *creating macaroon* creates a simple macaroon with no caveat.

Adding caveats: Third-party and first-party caveats can be added to a macaroon M using the methods $M.AddThirdPartyCaveat$ and $M.AddFirstPartyCaveat$ respectively. $M.AddThirdPartyCaveat$ takes as input a caveat root key cK, a caveat identifier cId, and a location cL of the caveat's discharging principal. It first computes a verification-key identifier vId by encrypting the key cK with the macaroon signature of M as the encryption key. Next, using the method $M.addCaveatHelper$, it adds the caveat $cav_{@cL}\langle cId, vId\rangle$ to the caveat of M, and then calculate the new signature for the macaroon from $vId :: cId$ using the existing signature as the MAC key and then updates the signature [2].

The $M.AddFirstPartyCaveat$ operation takes as input an authorization predicate a, and adds it using the $M.addCaveatHelper$ method, as the caveat $cav_{@T}\langle a, 0\rangle$ to the caveat list of M [2].

Preparing requests: While making an access request, a client is required to provide an authorizing macaroon along with discharge macaroons for the various embedded third-party caveats[2]. If the discharged macaroon is not bounded to the main macaroon, the client (or any attacker that have access to the discharged macaroon can simply misuse it, because it is possible to makes a request to a principal other than the original target.

To prevent this problem, all discharge macaroons are required to be bound to the authorizing macaroon before being sent along with a request to the target. This binding is carried out by the method $M.PrepareForRequest(M)$, which binds the authorizing macaroon M to each discharge macaroon M' the list M by modifying their signature to $M.bindForRequest(M')$. Here, $M.bindForRequest(M')$ is kept abstract, but one possible implementation would be to hash

```
CreateMacaroon(k, id, L)
  sig := MAC(k, id)
  return macaroon_@L⟨id, [ ], sig⟩

M.addCaveatHelper(cId, vId, cL)
  macaroon_@L⟨id, C, sig⟩ ← M      // ← is pattern matching
  C := cav_@cL⟨cId, vId⟩
  sig' := MAC(sig, vId :: cId)      // :: is pair concatenation
  return macaroon_@L⟨id, C ▷ C, sig'⟩      // ▷ is list append

M.AddThirdPartyCaveat(cK, cId, cL)
  vId := Enc(M.sig, cK)
  return M.addCaveatHelper(cId, vId, cL)

M.AddFirstPartyCaveat(a)
  return M.addCaveatHelper(a, 0, ⊤)

M.PrepareForRequest(M)
  M^sealed := ∅
  for M' ∈ M
    macaroon_@L'⟨id', C', sig'⟩ ← M'
    sig'' := M.bindForRequest(sig')
    M^sealed := M^sealed ∪ {macaroon_@L'⟨id', C', sig''⟩}
  return M^sealed

M.Verify(TM, k, A, M)
  cSig := MAC(k, M.id)
  for i := 0 to |M.C| − 1
    cav_@L⟨cId, vId⟩ ← M.C[i]
    cK := Dec(cSig, vId)
    if (vId = 0)
      assert (∃a ∈ A : a = cId)
    else
      assert (∃M' ∈ M : M'.id = cId ∧ M'.Verify(TM, cK, A, M))
    cSig := MAC(cSig, vId :: cId)
  assert (M.sig = TM.bindForRequest(cSig))
  return true
```

Figure 5: The operations essential to authorization using macaroon credentials.[2]

together the signatures of the authorizing and discharging macaroons, so that $M.bindForRequest(M') = H(M'.sig :: M.sig)$. This structure is used in the implementation [9] and the pseudocode.

Verifying macaroons: In order to verify each macaroon, no matter whether it has discharged macaroon or not, all the caveats should be satisfied. So, if we have a first-party macaroon, then the caveats must be satisfied in the target, and if we have a third-party macaroon, its caveats and discharged macaroon's caveats should be satisfied. For the purpose of formalization, this task is simplified by assuming that the target service first generates a set A of all embedded first-party caveat predicates that hold true in the context of the request whose macaroon is to be verified. To authorize the request, the target service invokes the method $TM.Verify(TM, k, A, M)$ where k is the root key of macaroon TM. The method iterates over the list of caveats in TM and checks each of them. For each embedded first-party caveat $cav_{@T}\langle a, 0\rangle$, it checks if the predicate a appears in A. For each embedded third-party caveat $cav_{@L}\langle ccId, vId\rangle$ in TM, it extracts the root key cK from vId, and then checks if there exists a macaroon $M' \in M$ such that (i) M' has cId as its macaroon identifier and (ii) M' can be recursively verified by invoking $M'.Verify(TM, cK, A, M)$. Finally it checks that the signature of the current macaroon is a proper chained-MAC signature bound to the authorization macaroon TM.

4 THE FORMALIZATION OF MACAROONS IN AUTHORIZATION LOGIC

Here we present a formalization of macaroons using a variant of Abadi's authorization logic from [1]. A macaroon is seen as an assertion made by a target service, saying that the holder of certain keys can speak for the target service regarding all access requests, as long as all relevant predicates for the macaroon's embedded caveats can be seen to be valid [2].

The effects of caveat predicates cannot be directly captured in standard authorization logic because of their variety and format. Therefore, the logic is extended with two new types of principals—predicate principals and request principals —and a special axiom that characterizes their behavior[2].

An Extended Authorization Logic for Macaroons: Here, an extended authorization logic is defined for the purpose of formalizing the semantics of macaroons and the assertions made by the principals relevant to a target-service request that is authorized using macaroons.[2]

First, the set *PropCavs* is assumed to contain all atomic propositions relevant to target service requests, including request parameters, such as chunk and operation, and contextual attributes, such as time [2]. Examples of such propositions might be*"chunk is in [100, 500]"*, *"action is read"*, etc.

The syntax, axiom and deduction rules for the authorization logic are defined in 6. Principals $A, B, \ldots$ can be atomic principals from $P \in Prncs$, keys $k \in Keys$, compound principals of the form $A \wedge B$, or special predicate or request principals. Here, p ranges over the set of propositions in $PropCavs$, and X ranges over the set of propositional variables. A says Φ meaning that principal A asserts Φ. While $\supset$ is used for logical implication, the statement $A \Rightarrow B$ expands to A speaks-for B, for principals A and B, meaning that B also makes all assertions that A makes. A compound principal $A \wedge B$ makes an assertion if, and only if, it is also made by both A and B. To establish that a request is authorized, a proposition valid is included, with A says valid meaning that principal A considers the request to be valid[2].

This logic is a normal modal logic that includes the [DISTRIBUTION] axiom, the [NECESSISTION] rule, the [MODUS-PONENS] rule, and all standard axioms of propositional constructive logic, as used in [1].

Also included, as in [8], are the standard $[SPEAKS-FOR]$ axiom, the $[HANDOFF]$ axiom, and the expected axioms for characteristics of principal conjunction. From these, the monotonicity of $\wedge$ over $\Rightarrow$ and the transitivity of $\Rightarrow$, follow:

$$\dashv (A \Rightarrow B) \supset (A \wedge C) \Rightarrow (B \wedge C)$$

Syntax:

$$\begin{array}{rrl} \text{Compound principals } A, B & ::= & P \mid k \mid A \wedge B \mid \widehat{\phi} \\ \text{Formulas } \psi, \phi & ::= & \mathsf{valid} \mid \mathsf{true} \\ & & \mid p \mid X \mid \forall X.\phi \\ & & \mid \psi \wedge \phi \mid \psi \vee \phi \mid \psi \supset \phi \\ & & \mid A \text{ says } \phi \mid A \Rightarrow B \end{array}$$

Axioms:

All the axioms of propositional constructive logic [PROP]

$\vdash (A \text{ says } (\psi \supset \phi)) \supset (A \text{ says } \psi \supset A \text{ says } \phi)$ [DISTRIBUTION]

$\vdash (A \text{ says } B \Rightarrow A) \supset B \Rightarrow A$ [HANDOFF]

$\vdash A \wedge A \equiv A$ [CONJ1]

$\vdash A \wedge B \equiv B \wedge A$ [CONJ2]

$\vdash (A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ [CONJ3]

$\vdash (A \wedge B \text{ says } \phi) \equiv (A \text{ says } \phi) \wedge (B \text{ says } \phi)$ [CONJ4]

$\vdash A \Rightarrow B \equiv (A \wedge B = A)$ [SPEAKS-FOR]

$\vdash (TR \text{ says } \phi) \supset (\widehat{\phi} \text{ says } \mathsf{valid})$ [PPRIN]

Rules:

$$\frac{\vdash \psi \quad \vdash \psi \supset \phi}{\vdash \phi} \text{ [MODUS-PONENS]}$$

$$\frac{\vdash \phi}{\vdash A \text{ says } \phi} \text{ [NECESSITATION]}$$

Figure 6: The syntax, axioms and rules for an authorization logic suitable for macaroons[2]. Here, TR and $\hat{\Phi}$ are request and predicate principals, respectively, $P \in Prncs$, $k \in Keys$, and X is a variable in a proposition $p \in PropCavs$.

$$\dashv (A \Rightarrow B) \wedge (B \Rightarrow C) \supset (A \Rightarrow C)$$

4.1 Request principals and predicate principals

A request principal TR is a special principal that makes the strongest possible assertion—using propositions from $PropCavs$—that a target service can see to be true for the context of a request[2]. For example, TR says *chunk is 250* $\wedge$ *operation is read* $\wedge$ *IP is 172.12.34.4* indicates that the target service is seeing a request to read chunk 250 being made from the IP address 172.12.3.44 [2].

A predicate principal $\hat{\varphi}$ is a special principal introduced to model the effect of a first-party caveat with predicate φ[2]. Informally, the principal $\hat{\varphi}$ says that a request is valid, and the caveat is satisfied, if the request principal TR asserts φ. This characteristic is formalized by the axiom [$PPRIN$].

By combining the axiom [$PPRIN$], the [$DISTRIBUTION$] axiom and the [$NECESSITATION$] rule, the following rule can be derived[2]:

$$\frac{\dashv \psi \supset \varphi \quad \dashv TR \text{ says } \psi}{\dashv \varphi \text{ says } \mathbf{valid}} [\text{FC}AV\text{DISCHARGE}].$$

This rule means that if TR asserts a predicate for a request, and this predicate logically implies a weaker predicate in a first-party caveat, then that caveat is satisfied for this request.

4.2 Macaroon formulas

In this logic, macaroons are defined by formulas that describe restricted speaks-for delegations from the target service to certain caveat principals (corresponding to embedded first-party caveats), and certain keys (corresponding to the root keys of embedded third-party caveats). A request made using macaroon credentials is authorized if it can be proven—from the macaroon's formulas, and the request principal TR's assertion about the request context—that the root key of the macaroon says valid, which implies that the macaroon signatures verify using this root key, and that all embedded caveat predicates are satisfied. Before formally defining macaroon formulas, it's worth giving some examples [2].

Consider a macaroon $M_1 := macaroon_{@L}\langle kId, [], k_1\rangle$ that is minted from root key k_0 without any caveats. This macaroon M_1 represents a complete delegation from the key k_0 to the empty caveat principal $\widehat{true}$, which considers all requests valid. Thus, M_1 is modeled by the formula k_0 says $\widehat{true} \Rightarrow k_0$.

A macaroon $M_2 := macaroon_{@L}\langle kId, [cav_{@T}\langle\varphi, 0\rangle], k2\rangle$ can be obtained by extending M_1 with a first-party caveat whose predicate is φ, to represent a delegation from the root key k_0 to the caveat principal $\hat{\varphi}$. This macaroon M_2 is modeled by the formula k_0 says $\hat{\varphi} \Rightarrow k_0$.

Finally, the macaroon M_2 can be extended with a third-party caveat whose root key is c_K, to obtain the macaroon

$$M_3 := macaroon_{@L}\langle kId, [cav_{@T}\langle\varphi, 0\rangle, cav_{@l}\langle cId, vId\rangle], k_3\rangle$$

.

This macaroon M_3 represents a delegation from the root key k_0 to the conjunction of the cK and $\hat{\varphi}$ principals—i.e., that k_0 says that a request is valid only if both cK and $\hat{\varphi}$ say so. Thus, M_3 is modeled by the formula k_0 says $cK \wedge \hat{\varphi} \Rightarrow k_0$. The formula for an arbitrary macaroon M can now be defined.

Definition 1 (Macaroon formulas)[2]: A macaroon M whose root key is k_0, embedding first-party caveats whose predicates are $\varphi_1, \ldots, \varphi_m$, and embedding third-party caveats whose root keys are $cK_1, \ldots, cK_n$, is modeled using the formula

$$\alpha(M) := k_0 \text{ says } (\widehat{\varphi_1} \wedge \cdots \wedge \widehat{\varphi_m} \wedge cK_1 \wedge \cdots \wedge cK_n) \Rightarrow k_0$$

For a set μ, the formulas are

$$\alpha(\mu) := \{\alpha(M) | M \in \mu\}$$

4.3 Macaroon verification

To verify that a request is authorized by a macaroon M, the target service must verify−−using a root key k_0 for M, known only to the target service−−the set μ of macaroons presented with the request, including M and all third-party discharges, and establish that all their embedded first-party caveats are satisfied. The verification of a first-party caveat with a predicate φ is modeled by having the request principal TR assert the strongest possible formula ψ_{req} about the request context, and, if $\psi_{req} \supset \varphi$, apply the derived rule [FCAVDISCHARGE] to show that the principal $\widehat{\varphi}$ considers the request to be valid.In general, a request accompanied by such a macaroon set μ is authorized if the formulas $\alpha(\mu)$ together with the formula TR says ψ_{req} imply that k_0 *says* **valid** for the root key k_0. Recursively, this requires that μ contain discharge macaroons

for any third-party caveats involved, and that TR says ψ_{req} allows cK *says* **valid** to be established for the root key cK of each of those discharge macaroons.

Definition 2 (Macaroon verification): A set of macaroons μ,whose distinguished authorizing macaroon M has the root key k_0, authorizes a request whose target-service context is described by the propositional assertion ψ_{req}, if, and only if

$$\dashv (TR \text{ says } \psi_{req} \wedge \bigwedge_{M \in \mu} \alpha(M)) \supset (k_0 \text{ says } \textbf{valid})$$

5 PROBLEMS WITH MACAROONS

The [10] talks about the architecture that macaroons seem to be legit, but after implementation, it causes many problems. In this scenario, we simplify a product system called sequence in chain systems, as it is shown in the Figure 7.

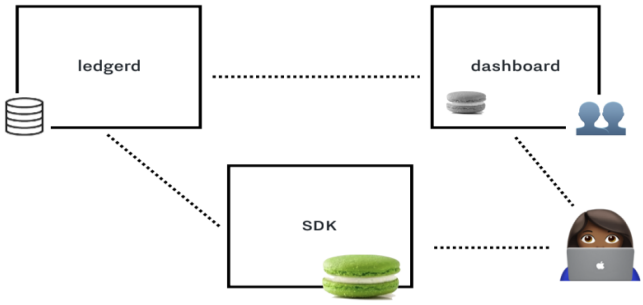

Figure 7: Services in simplified sequence architecture [10]

In Figure 7, we have a couple of services. The first service is called legerd, and it owns all the blockchain logic and provides the API for the core product. The other element is the dashboard, a rails app that provides a nice web interface for users and does all the user management, so the dashboard is the app that knows about users, emails, etc. Customers often use the dashboard, but they can also use an SDK in their client application that talks to legerd directly. The main problem in this architecture arises that the authorization owner is not one single service, because the dashboard is the only service that knows about users, but legerd has API endpoints on it that need to be protected. In this kind of situation where the dashboard needs to make auth decision for action executed inside legerd this is like a recipe for a confused deputy, and macarons are perfect for confusing the deputy.

If the legerd as our target service mint each service, it can create a most potent macaroon with no caveat (we call it golden macaroon). When legerd creates a golden macaroon t can hand the golden macaroon to the dashboard, which could then attenuate the golden macaroon with third-party caveats to restrict it to a certain user, at which point it is no longer a golden macaroon. This attenuated macaroon was then shared with the end-user who could stick it in their SDK, and so when a client application would make an API request, it would present this attenuated macaroon. Moreover, the macaroon had a third-party caveat, so the client application would also need to present a discharged macaroon alongside the main macaroon, and our SDK handled this behind the scenes, so they just quietly fetched a new discharge macaroon before the old one expired.

The problems arise based on the following reasons:

1- we created availability dependence on the dashboard, so if the dashboard went down and could not produce a discharge macaroon, users would not be able to access the API. We expect API to be more stable and reliable, but if the dashboard went out, the API would not work either.

2- Difficult to use locally: it is very cumbersome to use the API, especially without an SDK, so we had to manually fetch a discharged macaroon and provide that alongside the main macaroon for every request.

3- confusing behavior for users: In a system, we like users to have roles like admin, and when someone changes their user role in the dashboard, their token will stay the same, so unless they rotated out their token as well, their permission would not actually change and in real systems. This problem is confusing to users and occasionally also to the engineering team.

4- Impossible to revoke immediately: On a similar note, revoking a credential took up to five minutes(depending on the expiration time of the macaroon), so if we want to revoke a macaroon, we need to tell the dashboard to stop issuing discharged macaroon, but any given discharged macaroon was valid for five minutes, and we could not clot it back once it had been issued so we ultimately solved this problem by making each request check-in with the dashboard so each request should have to go to the dashboard as well to make sure that the discharged macaroon had not been revoked. This solution defeated the whole point of using the macaroon.

5- Tricky format: most of the time, macaroons are very long to fit on a single line in any environment, making it hard to visually inspect them.

6 CONCLUSION

We have presented macaroon, a credential token for decentralized authorization systems. We have described the different types of tokens, first-party and third-party macaroons, that contain first-party and third-party caveats, respectively, and then we define operations that can be operated to create, verify, or add caveats to them.

We used Abadi's authorization logic to formalize the macaroon. At last, we propose the advantages and disadvantages of macaroons in a simplified sequence architecture.

Macaroons are an exciting technology, but it is quite young and will improve over time. The current solution was not the right fit for Chain. The main reason for the hardness of using macaroons is that they are not commonly used. So there are not many providers of third-party caveats in real life. Also, the caveats format is still unspecific. Also, it is good to know that even though adding macaroons to the system is easy (because it does not have a complicated cryptographic algorithm) but removing them from the service is hard because the macaroons are in users' systems, and the service should wait for all users to replace their macaroons with simple authentication tokens.

ACKNOWLEDGMENTS

Macaroons originated in the Belay research project at Google [5].

REFERENCES

[1] [n.d.].

[2] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*.

[3] Tyler Close. 2009. ACLs don't. *HP Laboratories Technical Report* (2009).

[4] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S. Wallach. 2012. Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 317–331. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/dietz

[5] Google Inc. 2012. Belay reasearch project.

[6] Michael Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Signature (JWS). RFC 7515. https://doi.org/10.17487/RFC7515

[7] KirstenS. 2021. *Cross Cite Request Forgery.* Retrieved July 14, 2021 from https://owasp.org/www-community/attacks/csrf

[8] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. 1992. Authentication in Distributed Systems: Theory and Practice. *ACM Trans. Comput. Syst.* 10, 4 (nov 1992), 265–310. https://doi.org/10.1145/138873.138874

[9] Matt Williams Roger Reppe, Artiom Diomin. 2018. macaroon. https://github.com/go-macaroon/macaroon.

[10] Guillaume J. Charmes Tess Rinearson. 2018. An over-Engineering Disaster with Macaroons. https://about.sourcegraph.com/go/gophercon-2018-an-over-engineering-disaster-with-macaroons/.