Scaling Backend Authentication at Facebook

Kevin Lewi, Callen Rain, Stephen Weis, Yueting Lee, Haozhi Xiong, and Benjamin Yang

Facebook

Abstract

Secure authentication and authorization within Facebook's infrastructure play important roles in protecting people using Facebook's services. Enforcing security while maintaining a flexible and performant infrastructure can be challenging at Facebook's scale, especially in the presence of varying layers of trust among our servers. Providing authentication and encryption on a per-connection basis is certainly necessary, but also insufficient for securing more complex flows involving multiple services or intermediaries at lower levels of trust.

To handle these more complicated scenarios, we have developed two token-based mechanisms for authentication. The first type is based on certificates and allows for flexible verification due to its public-key nature. The second type, known as "crypto auth tokens", is symmetric-key based, and hence more restrictive, but also much more scalable to a high volume of requests. Crypto auth tokens rely on pseudorandom functions to generate independently-distributed keys for distinct identities.

Finally, we provide (mock) examples which illustrate how both of our token primitives can be used to authenticate real-world flows within our infrastructure, and how a token-based approach to authentication can be used to handle security more broadly in other infrastructures which have strict performance requirements and where relying on TLS alone is not enough.

1 Introduction

Requiring proper access control and identity management in a large-scale distributed network of systems is important to maintaining infrastructure security. Typically, infrastructures use a mix of authentication, authorization, and encryption to control access to sensitive data or systems. A common and successful approach to providing meaningful access control involves distributing certificates for identities and relying on a secure authenticaton protocol, such as Kerberos or TLS, to enable authenticated connections between equally trusted hosts.

Designing secure authentication at Facebook's scale presents numerous additional challenges. One challenge throughout our efforts in secure authentication is that we do not equally trust all of the machines within our network. In such a trust model, the compromise of a single member of the network could enable an attacker to impersonate and act on behalf of any other machine in our fleet.

This can be quite troublesome: for example, we have machines which lie closer to the edge of our network security boundary which are typically exposed to more adversarial threats, like side-channels attacks, than the machines which we have more physical control over. In fact, we can categorize Facebook's machines into several tiers of trust, with the highest level of trust associated with the machines which, for example, have access to Facebook's master secret keys. Because of the amount of trust we must place in these servers, we add extra security countermeasures to these servers and closely monitor all direct accesses to these machines.

However, these extra countermeasures generally do not scale well, and as a result, it is important for our root of trust to remain as small as possible. As we will see, enforcing authorization in a complex network which is based on a small root of trust places many constraints on the types of protocols we can use to authenticate connections across machines internally. We first expand on what exactly is contained within our root of trust.

Central root of trust. Establishing a central authority responsible for assigning secrets to entities offline is the first step to having secure identities, authentication, and authorization. Facebook's root certificate authority (CA) associates secure identities with machines by producing and securely distributing on-disk certificates which can be verified by anyone with access to the CA's public key. These certificates contain the identity for the service running on the machine, with the assurance that all machines belonging to a service have certificates which represent that service's identity. In addition to associating identities with physical hosts and backend services, we also tie unique identities to all Facebook users, pages, and other entities which require a Facebook login. Since these entities are not associated with individual hosts, we link identities with sessions. Internally, the session information for each such entity is then signed by a special service which gates all user logins. These signed sessions are tied to a client-side secret and carried with user requests to authenticate the request on behalf of the logged-in user. A service which needs to check the acting user for a request can then use the login service's public key to verify the associated session signature. Our root of trust also contains our key management servers which hold Facebook's master keys, and those which handle sensitive cryptographic operations for clients are also included in the root of trust, along with our service responsible for processing logins and distributing signed sessions to users. See Figure 1 for an illustration of how the root CA and login service interact to authenticate user sessions to our backend services internally.

Keeping this root of trust as small as possible while transitively extending its security to the rest of the machines in our fleet is the motivation behind most of our efforts in securing our infrastructure. The vast majority of our machines (over 99.9%) are not included in this root of trust, and it is important that we can continue to maintain this ratio, even as our fleet continues to expand.

With the certificates from our certificate authority in place, trusted hosts can authenticate with one another over a secure channel by using TLS.

TLS authentication and tokens. The main type of secret that the root certificate authority distributes to hosts in our network are on-disk X.509 certificates which are used to set up TLS connections. These certificates allow for hosts to communicate across secure channels, as long as they can trust our root certificate authority.

However, relying on TLS and X.509 certificates alone for host-level authentication is insufficient for securing Facebook's infrastructure as a whole. Often, a service must communicate with another backend service through one or more layers of proxies, which lie at a much lower layer of trust. One proposal to address this kind of limitation involves credential delegation for TLS [1]. However, in our setting, authentication for these communications cannot be limited to the connection, due to issues with connection pooling and re-use of credentials from connections across entities which have different levels of privilege. Hence, there is a motivation for *per-request authentication* which can be performed without having to elevate the untrusted intermediate proxies to a higher level of trust.

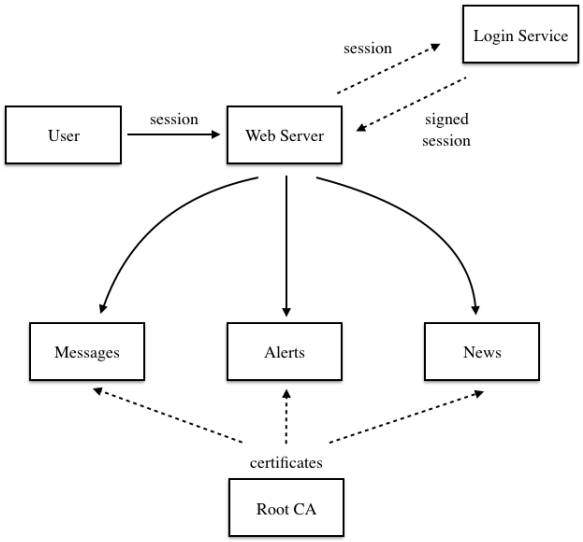

Figure 1: When a logged-in user loads the Facebook website, the user's session is signed by the login service, and this signed session is used to help authenticate to Facebook's backend services. The backend services are issued certificates from the root certificate authority (CA) in an offline step.

1.1 Authentication Tokens

We introduce the framework for our token-based approach for authentication, which includes two token-based mechanisms able to scale to a very high volume of requests:

- Certificate-based tokens: A token type based on certificates that extends authentication beyond what is provided by direct TLS connections, while still relying on public-key infrastructure.
- Crypto auth tokens (CATs): A symmetric-key based token that efficiently derives shared secrets between two communicating parties, and works independently but coexists with our public-key infrastructure.

Together, these two token types allow for secure authorization within our network and help to improve security for the platform as a whole.

Certificate-based tokens. Consider the example of a client connecting with TLS through an intermediate proxy to its final destination, as depicted in Figure 2. The client is typically unable to authenticate to the destination by relying solely on its connection to the proxy, since TLS typically only provides authentication between neighboring hosts in the network. Furthermore, we cannot allow the proxy to simply impersonate as the client to the server, especially in cases where the client is more trusted than the proxy. To address these problems, we allow hosts to create certificate-based tokens for propagating authentication, which contain a signature over the host certificate and some metadata, signed using the host's private key.

The client's outbound requests can attach these tokens, authenticating to a distant destination, while maintaining the flexibility of using network proxies. Furthermore, the client can embed

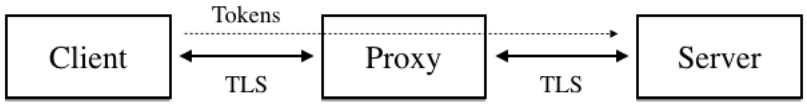

Figure 2: In our token-based authentication model, authentication tokens are passed from client to server across one or more proxies, which allow the client to authenticate to the server. The peer connections are secured with TLS.

additional request-specific data into the token, which is covered by a signature and limits the power of the authentication it provides by scoping down its access and inherently linking the identity carried by the token to only the actions it can perform. This is necessary for enforcing fine-grained access controls as it reduces the impact of a compromised or replayed token.

Since certificate-based tokens rely on the existing certificates used for TLS, these extra authentication mechanisms are inherited essentially "for free" from a dependency perspective[1], without having to rely on any additional setup procedures introducing any online interaction with the central authority. In practice, certificate-based tokens are extremely reliable and simple to use.

These certificate-based tokens are used across our backend infrastructure to securely propagate authentication. However, due to the public-key nature of certificates, there are limitations to how much we can scope access granted by these tokens without either hitting performance ceilings or utilizing aggressive caching. Furthermore, not all entities (e.g. external users) can be easily assigned a concrete certificate which lives on disk, yet we still need to provide access control for these entities.

Crypto auth tokens. To address these limitations, we also employ an approach which is not based on certificates, but instead relies on private symmetric keys. Consider the client-server interaction consisting of a Facebook user Alice making a request to access a resource controlled by some Facebook internal service, for example Messages.[2] Both parties have associated identities (one a user identity, and the other a service identity), but if we apply a purely public-key approach to authorize the request, the service must perform signature validations over the request contents. Given that more than 2 billion people use Facebook, all of whom need to be supported by most internal services, such an approach would be computationally infeasible, since each client-server interaction would require a public-key signature to be evaluated.

In general, utilizing a generic pre-shared secret between Alice and Messages would address the performance issues, but is also equally infeasible for other reasons. The service would then have to manage large lists of shared secret keys for each user, and these lists would have to be adaptively updated as users join and leave the platform, which requires online interaction with the central authority.

To address the authorization problem in this common scenario between a user and service, we employ a hybrid approach, which relies on the use of a keyed pseudorandom function [17] PRF which takes as input a key and an arbitrary string, and outputs a string which we use as

[1]At the cost of an extra signature validation over the request-specific metadata.

[2]Here, we are presenting a vastly simplified view of our infrastructure for the purposes of motivating the need for crypto auth tokens.

a "derived" private key. The central authority holds a master secret key K^*, and in an offline phase (or, as each user and service join the platform), the central authority computes and securely distributes the derived key constructed as $\mathsf{PRF}(K^*, \text{“Messages”})$ to Messages, and the derived key $\mathsf{PRF}(\mathsf{PRF}(K^*, \text{“Messages”}), \text{“Alice”})$ to Alice. In other words, for each user-service pair, the user holds a service-specific secret key which the service can locally derive from its own private key. The user can then use this shared secret to construct a MAC [3] over a specific request to encode details about the resource being accessed. When the service receives this MAC, along with the identity "Alice", it can reconstruct the shared secret key by doing a local computation of the PRF to verify the MAC.

Applications. In Section 4, we review more concrete examples of how taking a token-based approach to authentication, and more specifically, how certificate-based tokens and crypto auth tokens, can be used to help secure various parts of our infrastructure.

1.2 Related Work

There is an abundance of existing work spanning entire fields of research on authentication in distributed systems [24, 33] which cannot be fully covered here. Instead, we highlight several areas of study with an emphasis on how they directly relate to our contributions.

Authentication protocols. The Kerberos protocol [27, 31, 22] has served as the basis for providing authentication for various protocols including Microsoft's Active Directory, SAML 2.0 [20], OAuth 2.0 [19], and OpenID Connect. TLS [10, 9] is the most widely-used encryption protocol on the internet, and it can also be used to provide one-way or mutual authentication between hosts which rely on a certificate-based public-key infrastructure for security. These protocols can be used to provide authentication for a connection established between two hosts. The scope of our work is to augment the efficacy of the authentication beyond the connection in the presence of untrusted intermediaries, and is built off of the assumption that the channels between all communicating parties can be encrypted and authenticated at their endpoints. Hence, our contributions can be seen as an extension to these authentication protocols which can help to establish authorization on large-scale platforms.

Bearer credentials. A common solution to authentication, especially across the web, are bearer credentials: strings which are used to prove ownership or identity of a user to external services [23, 21, 14, 11, 30]. These tokens are simple and powerful in their design, since any party which obtains a bearer credential for a user can impersonate and act on behalf of the user, usually without needing access to any cryptographic key material or other secrets. However, Facebook's internal architecture involves billions of requests all passing through shared resources (caches, load balancers, and other proxies) which usually lie at lower levels of trust. It is undesirable for the compromise of a single one of these proxies to result in a compromise of all users whose requests pass through the compromised host. As a result, we generally want to limit the use of raw bearer credentials for authentication internally.

There are also numerous existing public-key certificate mechanisms similar to our certificate-based token approach for tackling decentralized authentication [2], decentralized trust management [5], decentralization using certificates [7], and extensible authorization for distributed services [25].

Macaroons. Macaroons [4, 26] are flexible authorization credentials that support decentralized delegation between hosts and help to enable fine-grained authorization. Macaroons are constructed as bearer credentials which can be augmented by supporting attenuation via nested applications of MACs. This is useful in situations where, for example, a service may require a client to provide proof that its requests have been audited and approved by an abuse-detection service, and come from a specific device with a particular authenticated user. The crypto auth tokens we introduce are similar to macaroons in that both constructions use the output of a PRF as the key to another PRF evaluation to produce a credential.

However, crypto auth tokens are used to establish shared secret keys for authentication between a client and a service, whereas macaroons focus on applying "caveats" [4, Section 2A] to credentials *after* a shared secret or root key between client and target service has already been established. This is typically because the intermediate service which creates the caveats must at least be partially trusted by the verifier. If the threat model does not allow for such delegation of authority, then the caveats provided by macaroons become less useful.

Encrypted database solutions. In general, the purpose of pursuing strong authentication across services is to allow for fine-grained control of access to data. There is a large field work which takes a parallel approach to solve this problem by encrypting the data using private keys held by clients [28, 29, 32]. A variety of advanced encryption techniques can be applied to enable the server to perform operations on the encrypted data it holds, including: attribute-based encryption [18], garbled circuits [34], fully-homomorphic encryption [15, 16], and functional encryption [6]. However, the complexity of all operations which would need to be supported at Facebook's scale make these approaches infeasible as a general-purpose solution to the authentication problem within our infrastructure.

1.3 Notation

We use || to represent the string concatenation operator. We use PRF to denote a pseudorandom function on two input spaces: a key space and an input string space. We use $\mathsf{PRF}(k, m)$ to represent the evaluation of the pseudorandom function on key k and input string m. A message authentication code (MAC) function has two inputs, a key and an input string, and produces an output string. We use $\mathsf{MAC}(k, m)$ to represent the evaluation of a MAC function on key k and input string m. For a signature algorithm with a public key and secret key pair $(\mathsf{pk}, \mathsf{sk})$, we use Sign as an algorithm over a signing key sk and input string m to produce a signature, which can be verified using the public key pk. We write KDS to represent the central key distribution server.

2 Token-Based Authentication

Consider the setting of two backend services Messages and Alerts, each with distinct identities, that wish to communicate with one another through an untrusted intermediary Proxy.[3] In an offline phase, all three entities Messages, Alerts, and Proxy each receive their own certificate from the root certificate authority, along with their private key. Then, when Messages initiates a new request whose final destination is intended for Alerts, Messages uses its certificate and private key

[3]Again, this is meant to illustrate the high-level infrastructure layout and not to provide a concrete example of any interactions between real services.

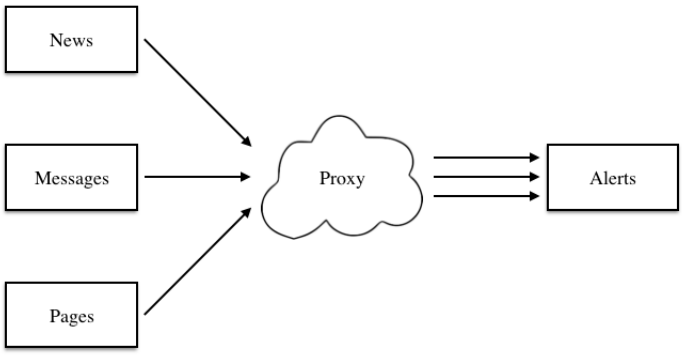

Figure 3: Typically, services send requests to other services in between layers of proxies. Although we have authentication for the individual connections between machines, we need a general mechanism which provides authentication in a way that allows us to not rely on trusting the proxy layer. For example, Alerts should be able to verify that a request originated from Messages without having to make assumptions on how the proxy routed the request. Handling request-level authentication between trusted services that communicate between less-trusted proxies is our primary motivation for pursuing token-based authentication in general.

to establish a TLS connection with Proxy, securely transmitting the request. Then, Proxy processes the incoming request and establishes a TLS connection with Alerts in a similar manner, forwarding the request to Alerts.

We can think of Proxy as being a member of a large collection of machines responsible for general request routing. Typically these machines are used to perform load balancing, process requests generically, or reroute traffic across less-congested regions. Due to the nature of the functions performed by Proxy, we want to minimize the amount of trust we must place in Proxy to forward requests honestly to Alerts without malicious modification. At the same time, it is also important that Proxy remains as a distinct entity from Messages and Alerts, in that it should not be able to impersonate either of these more-trusted servers. For example, granting access to the private key of Messages or Alerts to Proxy would greatly reduce the amount of trust we can place in both Messages and Alerts, since its attack surface would have drastically increased. We illustrate this example in Figure 3 and expand more in Section 4.2.

Motivation. The central issue we want to resolve here is that Alerts cannot trust that the request originated as intended from Messages without having to trust that Proxy did not modify the request. To address this, we use *token-based authentication*, which describes our practice of creating some form of a cryptographic token that is attached to the request's headers and allows for Alerts to verify that the request came from Messages unmodified.

Our simplest approach to implementing token-based authentication involves piggybacking off of the certificate and TLS infrastructure which is already used to secure direct connections between machines. With tokens based off of certificates, we can simply use the server's TLS private key to sign the request. The resulting signature is placed, along with the server's publicly-verifiable certificate, within the token that is passed along in the headers of the request. We refer to this entire structure together as a *certificate-based token*.

2.1 Certificate-Based Tokens

In our model, we have a key distribution server (KDS) which is responsible for distributing certificate and private key pairs to authenticated clients. The server KDS has a master public key mpk which is published to every host. A client can interact with KDS over a secure channel by submitting a valid authentication proof for the client's identity, to which KDS replies with a certificate cert and private key sk specific to the client.

A certificate-based token which authenticates a request with metadata `req_data` is constructed as

$$\mathsf{token} = \mathsf{cert} \,||\, \texttt{req\_data} \,||\, \mathsf{sig},$$

where $\mathsf{sig} = \mathsf{Sign}(\mathsf{sk}, \mathsf{cert} \,||\, \texttt{req\_data})$.

A verifier can then use the master public key mpk to validate the certificate cert, extracts the certificate's public key pk, and uses it to validate the signature over the token data cert || `req_data`. Note that this involves two signature validations.

Token expiration. Since these certificate-based tokens rely heavily on the security of the underlying certificates they are built off of, we can also bind the rotation and expiration of these tokens to the rotation and expiration of their associated certificates. However, the tokens themselves can be restricted to have an expiration which is much smaller than the expiration of their certificates.

We note that our certificate-based token approach is built upon existing works which have explored a similar approach to assocating certificates with identities, most notably SPKI [13, 8, 12].

2.2 Limitations of Certificate-Based Tokens

The public-key nature of certificate-based tokens can be problematic in high-performance settings. If Alerts expects to receive millions of requests per second from Messages, having to validate signatures on each such request can quickly consume the computation time of Alerts. To further complicate matters, Alerts also serves requests initiated by users, which are not associated with physical hosts and hence do not rely on the public-key infrastructure or any X.509 certificates for authentication.

Caching signature validations. To address the performance issue, we utilize caching of signature validations on the server side. Rather than associating a unique certificate-based token for each request, we can instead sign a piece of the request which is common amongst multiple requests which access the same set of resources. This cache is kept in memory on each of the service's machines in order to limit the risk of an attacker being able to maliciously modify the cached signature validations. When combined with a deterministic signature scheme, this means that each host belonging to the server needs to only validate the signature belonging to each unique certificate-based token once. By placing a hash of the validated tokens in a cache, we can avoid the computationally-expensive operations needed for validating each request, at the expense of using RAM to keep track of the validated tokens. For many applications, this tradeoff is desirable, and so the caching of these signature validations is beneficial.

However, note that the caching still comes at a cost to security (depending on the granularity of the caching), as we must extend the validity period of each of these tokens in order to take advantage of the cache properly. Enforcing a very short timeout can have an adverse effect on the server's cache hit rate, and as a result, we avoid using certificate-based tokens for applications which

have strict memory requirements and do not allow for the granular caching that we would need in order to maintain security.

These limitations motivate a different token primitive which is not only symmetric-key, but also flexible enough to support authentication for user sessions (which are not tied to physical hosts).

3 Symmetric-Key Tokens

In order to deal with the limitations of taking a public-key approach to token-based authentication, we also use a symmetric-key alternative. *Crypto auth tokens* (CATs) are not based on certificates, but instead are naturally more similar to the design of the Kerberos authentication protocol.

At a high level, we split secrets into two types of private keys: service keys and session keys. A service key is assigned to each service which is responsible for validating requests from its clients, whereas a session key is associated with a client-service pair. In our model, service keys are much more powerful than session keys, in that a single service key can be used to derive any session keys which are associated with the service. This feature of CATs allows services to perform validations without having to repeatedly re-authenticate to the master key server, and also without having to resort to public-key primitives. This asymmetry between service keys and session keys for CATs can also be seen as a performance optimization which leverages the fact that the number of clients which request for session keys far exceeds the number of services which request for service keys within our infrastructure. Indeed, we observe that CATs would be much less performant in an "inverted" infrastructure model where the number of services far exceeded the number of clients.

Decoupling the token-based authentication mechanism from on-disk certificates also means that we can support more flexible identities which may be implicitly tied to requests rather than being tied to machines and filesystems. Formally, there are three types of keys involved in the CAT authentication protocol.

- **Master key:** The master secret key msk is selected uniformly at random and is kept secret, so that only the key distribution server has access to it.
- **Service keys:** A service with identity Server is associated with its service key constructed as $\mathsf{vk} = \mathsf{PRF}(\mathsf{msk}, \text{“}\mathsf{Server}\text{”})$.
- **Session keys:** A client with identity Client that wishes to communicate with a server with identity Server is associated with a session key $\mathsf{sk} = \mathsf{PRF}(\mathsf{PRF}(\mathsf{msk}, \text{“}\mathsf{Server}\text{”}), \text{“}\mathsf{Client}\text{”})$.

Again, note that any entity with access to a service key vk can compute any session key for any client that wishes to communicate with it. This is both a performance optimization and also a potential for key compromise impersonation.

Key distribution. The key distribution server is responsible for distributing service keys to entities with identities. To do this, the key distribution server responds to requests to obtain service keys and/or session keys for entities which can supply authentication proofs to prove their identities.

Token creation. The crypto auth token cat between Client and Server, with the session key sk, consists of the following fields:

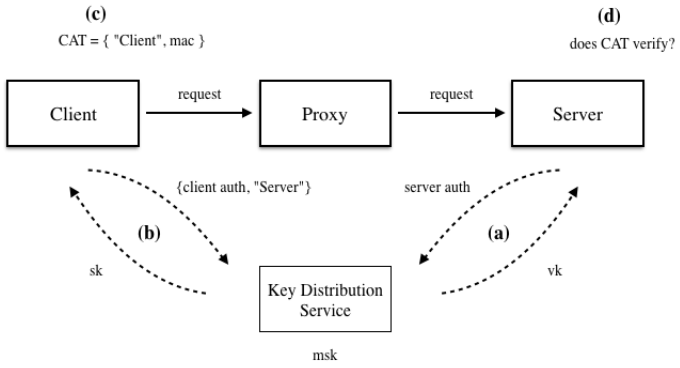

Figure 4: In the presence of a key distribution service KDS with a master secret key msk, CATs can be used to authenticate requests between Client and Server as follows: **(a)** In an offline phase, Server provides the proper authentication to KDS and retrieves a service key vk constructed as PRF(msk, "Server"). **(b)** The client provides the proper authentication, along with the server name "Server" to KDS and retrieves a session key sk constructed as PRF(PRF(msk, "Server"), "Client"). **(c)** When the client makes a request to the server, it uses its session key sk to construct a MAC over the request data "data", computed as MAC(sk, "data"). This MAC, along with the string "Client", is what composes the CAT, and is sent alongside the request. **(d)** When the proxy server receives this request along with the CAT, it uses its server key vk to determine whether or not the request is authenticated by checking if the MAC is equal to MAC(PRF(vk, "Client"), "data").

- **Signer/verifier identities**: This simply contains the literal strings "Client" and "Server".
- **Token creation and timeout**: This field consists of a creation timestamp and the length of validity for the token, set by the client.
- **Additional data**: This field can be used to store additional request-specific data that the verifier is expected to check.
- **MAC**: This contains a string $\mathsf{cat}_{\mathsf{mac}}$ which is computed as $\mathsf{cat}_{\mathsf{mac}} = \mathsf{MAC}(\mathsf{sk}, \mathsf{cat}_{\mathsf{data}})$, where $\mathsf{cat}_{\mathsf{data}}$ represents all of the other (above) fields in this token.

Token verification. The structure of this crypto auth token cat contains the literal strings "Client" and "Server", an additional data field $\mathsf{cat}_{\mathsf{data}}$, and the MAC $\mathsf{cat}_{\mathsf{mac}}$. To verify cat, the service can use the available fields to verify that $\mathsf{cat}_{\mathsf{mac}} = \mathsf{MAC}(\mathsf{PRF}(\mathsf{vk}, \text{"Client"}), \mathsf{cat}_{\mathsf{data}}))$. The complete process is also described in Figure 4.

Key Rotation. This construction can easily extend to support expiration and rotation of these service keys and session keys through inheriting the properties of the rotation and expiration mechanism for the master secret key which is used to derive the service and session keys. Typically, there is a key-rotation mechanism which associates versions with master secret keys, which can be periodically incremented. This version number can be propogated as versions for the derived service and session keys and used for rotation in a similar manner.

Security. The CAT session keys are constructed so that the only hosts which can obtain a valid session key between a client and a verifier are the hosts can authenticate to the key distribution server as being either the client or the verifier. Without access to either of these authentication proofs, an adversary which is attempting to impersonate a request to a target service with identity Server on behalf of a target client with identity Client would need to produce a valid session key between Client and Server. However, since this session key is the output of a PRF evaluation based off of the master secret key held by the key distribution server, the session key is uniformly and independently distributed from all other session keys the adversary has access to, by the security of the PRF used to derive the session key. Without being able to learn any information about the target session key, we can then use the security of the MAC algorithm to establish that the adversary is unable to produce a forged crypto auth token for a request between Client and Server with any noticeable probability.

Note, however, that if an adversary were to gain access to a service key for which it does not have an authentication proof for, then the adversary could use this key to create valid session keys for arbitrary clients which interact with the service whose key was compromised. We rely on the periodic rotation of these service keys to help mitigate such scenarios in which a service key gets leaked to an adversary that does not have the proper authorization to access the key normally.

4 Applications

In this section, we show how we can use the authentication primitives established in the previous sections in order to secure real interactions within Facebook.

4.1 Users Accessing Their Own Data

When a Facebook user visits and logs in to `https://www.facebook.com`, a production web server is responsible for completing the request to display the Facebook page to the user. This single request can trigger hundreds of smaller requests to backend services to fetch the user's private data. As a concrete running example, let's focus on a user-triggered request for that user's private message conversations with their friends. This request queries the storage layer and infrastructure which holds all user messages, where the contents of the request specify the parameters which determine whose messages to obtain.

Without any additional authentication mechanisms on top of simply enabling TLS between all endpoints involved in serving the request, the storage layer has to fully trust that the production web server is fetching the correct user's messaging data. Even worse, since these high-performance requests are routed through layers of proxies and caching, the storage layer must fully trust every intermediate between the web server that initiates the request and itself.

We can use crypto auth tokens to provide authentication for this flow. Upon login, the web request first interacts with the login service in an offline phase to obtain a signed user session for the request. This signed session can be used to authenticate an online request to the key distribution server to obtain a crypto auth token key, bound with the user identity as the signer identity and the message backend infrastructure as the verifier identity. With this key, the web server can create a crypto auth token by issuing a MAC over the request it sends to the messaging backend. See Figure 5 for more details.

When the messaging backend receives this request, it can use its service key to derive the

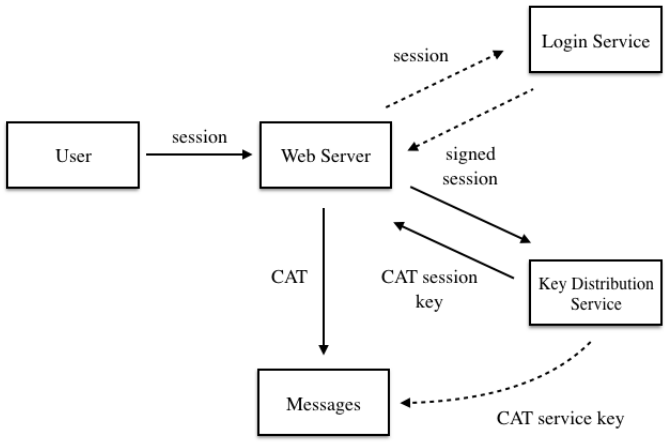

Figure 5: In this example, a user initiates a request which accesses the Messages backend. Similar to the example described in Figure 1, the user authenticates with the login service to obtain a signed session in an offline phase. Then, for an online request, the signed session is then used to obtain a session key from the key distribution service. This key is then used to construct a MAC over the request to the Messages service. This MAC, along with its metadata, is what we refer to as the crypto auth token (CAT). The Messages service can validate this MAC using its service key which was loaded in an offline phase from the key distribution service.

appropriate session key with the user as the signer. It can then check the request details against the crypto auth token to verify that the request has not been adversarially modified by any intermediate proxies or caching layers.

Reliability and performance. One of the motivations behind pursuing this kind of authentication flow is that the messaging backend infrastructure must be able to maintain high reliability and performance, even in disaster situations. With respect to authentication, this means avoiding any online dependencies or expensive operations when serving messaging content. Crypto auth tokens achieve this by allowing the messaging backend to perform local PRF evaluations to recreate session keys. The only dependency needed here is a call to the key distribution service on startup, and an occasional polling thread to listen for updates (such as rotations) to the service key. Note that relying on public-key signature validations here is a non-starter, since it would be infeasible to verify signatures on every message request, and a cache for these signature validations would have a low hit rate. Furthermore, calling an external service on every message request would also be infeasible.

4.2 Servers Accessing Other Servers Through Layers of Proxies

Another common access pattern within backend infrastructure involves a set of distinct services which access a set of distinct storage layers, while sharing a pool of (sometimes multiple layers of) intermediate proxies. We can think of these proxies as services which help to route connections for load balancing reasons, or which serve as intermediary caches, for example.

Typically, each of the storage services keep an independent list of requesting services which are

they are allowed to serve. However, when a requesting service sends a request to a specific storage service through layers of proxies, if the only form of authentication is TLS, then the storage service must trust all of the intermediate proxies to correctly maintain and forward the identity of the requesting service to its destination.

In this situation, both crypto auth tokens and certificate-based tokens are applicable in supplying the necessary authentication to reduce the amount of trust placed in the intermediate proxies. However, these two solutions also have their own tradeoffs.

By using crypto auth tokens, each requesting service can authenticate and obtain a session key from the key distribution server which can be used to authenticate the requests sent to its destination. The proxies can faithfully route the request along with its crypto auth token to the destination service, which can then use its own service key to locally rederive the session key and validate the request. However, a disadvantage to this solution is that only the destination service is able to validate this request. In particular, intermediate proxies cannot recreate the session key to validate the request. This inflexibility can be resolved by having the requesting service send multiple crypto auth tokens, where each verifier identity is associated with each of the layers of proxies that the request is routed to. However, this is undesirable as it requires knowledge of the network structure, which can be adaptively modified and requires modifying the authentication flow every time the network structure is adjusted.

We can circumvent this inflexibility of authentication tokens by instead relying on certificate-based tokens. Due to their public-key nature, verification no longer requires knowledge of a shared secret, and hence the intermediate proxies are able to validate the requests and reject them before they ever reach the destination service.

5 Conclusions

Token-based authentication becomes a useful tool in an infrastructure with granular access control, but where not all servers can be equally trusted. The two approaches we have covered, certificate-based tokens and crypto auth tokens, play a role in Facebook's authentication infrastructure and we believe that they can be applied more generally to any large-scale platform which enforces a similarly aggressive internal trust model. To conclude, we present two directions for further study:

1. In our setup, a single client which uses CATs to authenticate to N backend services must retrieve and manage N distinct keys from the key server. However, a single backend service which accepts authentication from an arbitrarily large number of clients need only manage a single key corresponding to the service. Is it possible to reduce the number of keys that the client must manage without resorting to public-key primitives?

2. As described in Section 3, CATs are subject to key compromise impersonation, which can be problematic when multiple entities are responsible for verifying CATs which are bound to the same verifier identity. Is it possible to design an authentication token which is not subject to key compromise impersonation, yet which still mirrors from the efficiency of CATs, both in allowing the server to locally derive its verification keys for each client, and while still avoiding public-key primitives?

Acknowledgments

The primitives described in this work are the result of an on-going work with authentication tokens. We would like to thank all of the members of Facebook's security teams for their guidance in the design of certificate-based tokens and crypto auth tokens. Many engineers also contributed to the continuous refinement and deployment of multiple iterations of these ideas over several years.

References

[1] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla. Delegated credentials for TLS. `https://tools.ietf.org/id/draft-ietf-tls-subcerts-00.html`, 2017.

[2] M. Y. Becker, C. Fournet, and A. D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4), 2010.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[4] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, 1996.

[6] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, 2011.

[7] N. Borisov and E. A. Brewer. Active certificates: A framework for delegation. In *NDSS*, 2002.

[8] D. E. Clarke, J. Elien, C. M. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[9] T. Dierks. The transport layer security (TLS) protocol version 1.2, 2008.

[10] T. Dierks and C. Allen. RFC 2109: HTTP State Management Mechanism. `https://www.ietf.org/rfc/rfc2246.txt`, 1999.

[11] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX*, 2012.

[12] C. M. Ellison. SPKI. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1243–1245. Springer, 2011.

[13] C. M. Ellison, B. Frantz, B. W. Lampson, R. Rivest, B. Thomas, and T. Ylönen. SPKI certificate theory. *RFC*, 1999.

[14] K. Fu, E. Sit, K. Smith, and N. Feamster. The dos and don'ts of client authentication on the web. In *USENIX*, 2001.

[15] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[16] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, 2011.

[17] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 1986.

[18] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, 2006.

[19] D. Hardt. The OAuth 2.0 authorization framework. *IETF RFC 6749 (Informational)*, 2012.

[20] J. Hughes and E. Maler. Security assertion markup language (SAML) v2.0 technical overview. *OASIS SSTC Working Draft sstc-saml-tech-overview-2.0-draft-08*, 2005.

[21] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, 2001.

[22] J. T. Kohl and B. C. Neuman. The kerberos network authentication service (V5). *RFC*, 1993.

[23] D. Kristol and L. Montulli. RFC 2109: HTTP State Management Mechanism. `https://www.ietf.org/rfc/rfc2109.txt`, 1997.

[24] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. In *SOSP*, 1991.

[25] C. Lesniewski-Laas, B. Ford, J. Strauss, R. T. Morris, and M. F. Kaashoek. Alpaca: extensible authorization for distributed services. In *CCS*, 2007.

[26] A. López-Alt. Cryptographic security of macaroon authorization credentials. In *New York University, Tech. Rep. TR2013-962*, 2013.

[27] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 1978.

[28] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *ACM SOSP*, 2011.

[29] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *NSDI*, 2014.

[30] F. B. Schneider. `http://www.cs.cornell.edu/fbs/publications/chptr.CredsBased.pdf`, 2015.

[31] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference. Dallas, Texas, USA, January 1988*, 1988.

[32] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*, 2016.

[33] E. Wobber, M. Abadi, M. Burrows, and B. W. Lampson. Authentication in the taos operating system. In *SOSP*, 1993.

[34] A. C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.