OpenAutonomy Authentication

Open Autonomy Inc.
January 2014

Abstract:

A system of federated components, such as OpenAutonomy, requires a protocol to authenticate messages between servers so that a message's receiver can verify the sender is who they claim to be. This document outlines the approach used for building and rebuilding the sequence of authentication keys used to authenticate successive messages.

Introduction:

Internet services based on monolithic walled gardens have complete control over every component of the system, within a closed environment. This means that they never need to authenticate the sender of a message, from one component to another, as they are already guaranteed that the sender is within their controlled environment. In that context, an invalid sender would be a software bug as opposed to a deliberate authentication breach. Federated systems, such as OpenAutonomy, cannot rely on a closed environment so they cannot rely on this assumption.

OpenAutonomy relies on 3 different mechanisms to implement this functionality:

1. **Deterministic key sequence** - Both the sender and receiver have agreed upon a private sequence of keys to send with their messages.

2. **Synchronous verification** - The receiver can call back to the sender to verify they are who they claim to be.

3. **Trusted third-party** - Sender gets a token from a third-party which can be verified by the receiver, assuming they both trust the same third-party.

This document outlines the high-level logic of how and when these mechanisms are applied and discusses some potential weaknesses which may require resolution if they are deemed feasible attack vectors and are not resolvable via HTTPS.

Design Goals:

The authentication system governing communication between two OpenAutonomy components must have the following properties:

- In the general case, it should not incur appreciable overhead in an RPC. This refers to both round-trip time and payload size.
- It must be robust against intrusion. That is, a third-party should not be able to cause a denial-of-service for the legitimate applications.
- The user should not be required to do any work to facilitate communication between components. The exception to this is non-web applications which have no other way of being reached.
- It should not be feasible for a third-party to succeed with a brute force attack.

Permitted Limitations:

In order to keep the intent of this authentication component clear, the following limitations are stated as a reminder that they are outside the scope of this concern:

- Exceptional (first or infrequent) messages are permitted to add substantial overhead to their RPC.
- Non-web applications will require some amount of user interaction for initial authentication.
- No encryption is explicitly added (the system should be run over HTTPS if this is a concern).
- There is no signature protection against man-in-the-middle attacks (again, use HTTPS to mitigate this problem - cache and compare target certificates).

Mechanism Details:

- **Deterministic key sequence**: In this approach, A (sender) and B (receiver) both contain an algorithm which, starting from a given seed value, will generate an infinite sequence of seemingly random values. When A calls an RPC exposed by B, it passes the *next* value in this sequence as an argument. Since B has the same generator, it can compare this value against what it expected in order to determine if the call is genuine.
  While this approach is fast and relatively secure, it has the obvious bootstrapping problem of how this private key is communicated between A and B. Another noteworthy limitation is that all communication between these applications is synchronous (the messages must be received in the same order they are sent) and half-duplex (since both sides could try to send "next" at the same time).
  The specific key sequence applied in OpenAutonomy is generated like so:
  key_{n+1} := sha1(string_concatenate(key_{n}, private))

The initial key exchange includes **key_{0}** and **private**. The key exchange is of the form "**version-key_{0}-private**", where **version** is always 0, **key_{0}** and **private** are both non-empty lower-case ASCII strings.
The **sha1** function must return a **lower-case hex string** of the digest.

- **Synchronous verification**: In this approach, A (sender) calls B (receiver), to request an RPC. It passes its URL and a token as arguments to the RPC. B, in order to verify that A is the sender, will call a special verification RPC on A (using the URL), passing the token it received as an argument. A knows which tokens it has produced so it can verify if this is one of these tokens. If so, it returns true and B now knows it is talking to A.
  While this approach is reasonably secure, it is **very slow** and can introduce spurious failures due to network failures on the reverse call. Thus, it isn't appropriate in the general case. It also makes an assumption that A is a reachable web application which is an invalid assumption once non-native applications have been added to the system.

- **Trusted third-party**: In this approach, A (sender) first calls T (a third-party), to request a trusted token. A then calls B (receiver), passing this token as an argument to the RPC. B calls T in order to ask for whom it created the token, passing the token as an argument. T returns information used to identify A. Thus, B now knows the RPC was sent by A.
  While this approach is reasonably secure, it is **very slow** and can introduce spurious failures due to network failures surrounding the calls to T. It also introduces this additional party to the communication. Thus, it isn't appropriate in the general case. It does have some useful properties which are worth noting. It is similar to synchronous verification except it doesn't assume that A can receive calls. Additionally, T can provide richer information regarding A than simply its location.

Application-Identity Communication:

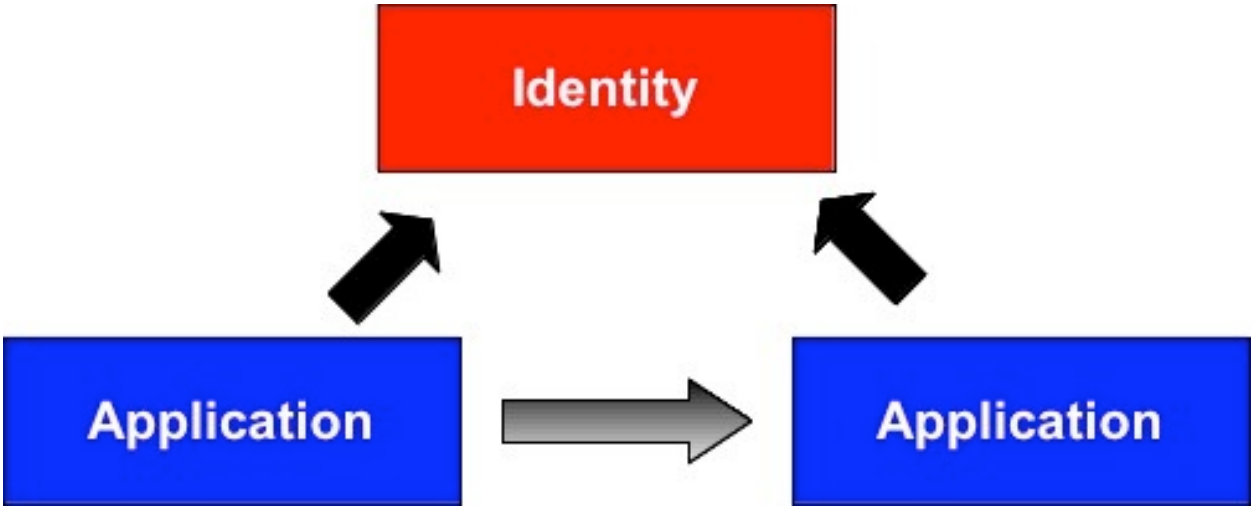
When an application contacts an identity, it uses a combination of **deterministic key sequence** and **synchronous verification**. The vast majority of calls use the key sequence but the initial key exchange is done using the verification approach.

The case where it cannot use synchronous verification is when the application is a non-web application, since it cannot be called back. In that case, the initial key exchange must be done by the user.

Application-Application Communication (same creator):

When an application contacts another application with the same creator identity, it uses a combination of **deterministic key sequence** and **trusted third-party**. The key sequence is used for all but the first message between them. The initial key exchange (as well as additional sender meta-data such as name and trust level) is done via a third-party where the identity acts as their mutually-trusted third party.

Note that this works the same way, no matter whether the sender is a web application or non-web.



2 applications with the same creator.

Application-Application Communication (differing creator):

When an application contacts another application with a different creator identity, it uses a combination of **deterministic key sequence** and 2 instances of **trusted third-party**. The key sequence is used for all but the first message between them. The initial key exchange (as well as additional sender meta-data such as name and trust level) is done via a third-party approach which involves both of their creator identities, as they share no mutually-trusted third-party.

The sequence is complicated so here are the steps involved:

1. A (sender) contacts X (its creator identity) to request a **referral token**.
2. A contacts Y (B's creator identity) to request access to its applications, passing the referral token.

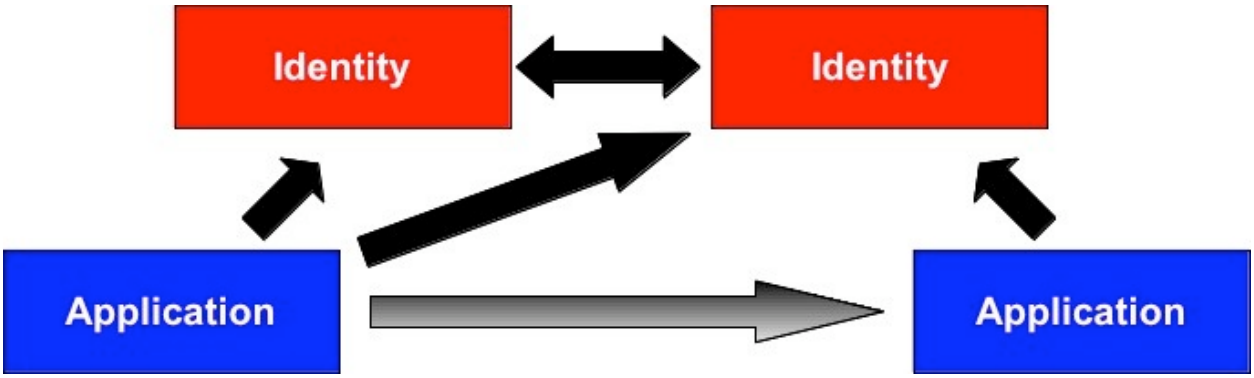
3. Before Y returns, it first contacts X to request more information about the application represented by the referral token it was given (that is, A). It then returns the token to contact B.

4. A contacts B (receiver), passing it the token it received from Y.

5. B contacts Y to verify the token and receives information about A, including its creator identity.

The communication can now progress using a key sequence derived from this token information.

Similarly to the case of the same creator, this works the same way, no matter whether the sender is a web application or non-web.



2 applications with different creators.

Conclusion:

This paper has provided a high-level overview of the authentication mechanisms used by OpenAutonomy along with the situations where they apply. The explanation demonstrated how these mechanisms support all 3 of the relationship types required by the federated components: identity contact, local applications, and foreign applications. These relationship styles facilitate the complex relationships required to support the various trusted multi-user services required by modern internet-based applications.