METASPEX

Metaspex Technical Insights

VI - Security

Introduction

This technical insight will be slightly different from the others. Security is a very sensitive issue, so this paper will dive deeper into some details of implementation than the previous papers. In particular it will mention exactly which security standards are used at a low level.

It is not the purpose of a framework like Metaspex to receive a certification, but every precaution has been taken in the implementation of Metaspex to ensure that the overall application relying on Metaspex will qualify for stringent certificates, such as the highest level of PCI-DSS.

Metaspex's security is an optional matter. Metaspex can be used in full trusted mode, meaning without any security checks. In this mode, Metaspex applications can, for instance, sit behind a security gateway performing all the security checks. When used, Metaspex security can take care of all security matters, from **user** management, **role**s and permissions, to their enforcement in web services calls.

Security is addressed through a set of types, which are part of the Foundation Ontology. These types can be handled with Metaspex built-in optional web services. There are approximately 200 of these web services, just for security. Many of these were discussed in an earlier technical insight dedicated to web services.

Security policy and configuration file

The security policy to be enforced is defined in the Metaspex configuration file. While Metaspex allows a fine-grained selection of which built-in security-related web services are embedded in an application, configuration file security policy directives allow an even finer level of control.

There are more than 30 configuration file security policy directives supported by Metaspex. The keywords all start with "secpol_". Without listing them all, we can mention that they allow you to control a variety of behaviors, such as:

- The constraints set on user passwords, such as the minimum length, and the required number of symbols, digits, and uppercase letters. The directives also allow you to set the maximum age, if any, a password can have before a user has to change it. You can indicate how many previous passwords Metaspex will retain to check that a new password does not reuse an older one;

- How many unsuccessful logins can be performed before a user is automatically locked;
- Whether locks are temporary or permanent;
- Whether two-factor authentication is enforced;
- Whether users, after creation, must be activated by clicking a link in an email sent by the backend. Metaspex will then take care of sending that email;
- Whether an inactivity timeout must be enforced to check that a session token is still valid;
- Whether OAuth 2.0 is accepted, and which behavior it should obey when activated (e.g., whether Google or Apple are accepted as JSON Web Token authorities, or whether they are considered as masters for the claims contained in JWT to automatically update the user documents stored by Metaspex, and whether attempting to create an already-existing user with a JWT is authorized);
- Etc.

These directives have a default behavior which is documented in Metaspex's reference guide.

Obviously, whether the application is B2C or B2B deeply drives the profile of the security policy. For instance, in a B2B application, OAuth 2.0 is probably not desirable.

Please consult the reference guide and its section on the configuration file to see all the directives enabling you to customize the security policy.

The directory

All the documents pertaining to built-in Metaspex security, **user**s, **organization**s, **affiliation**s, **solution**s, **role**s, **license**s, etc. are stored in a specific database, called the directory. The directory can be created in any of the database products supported, while the remainder of the application can use other products, if necessary.

The directory's technical characteristics are defined in the configuration file, which is read at application startup. It's there that you indicate whether the directory is stored in Couchbase, MongoDB or some other vendor database.

Any Metaspex backend can be enabled to read the directory. Adding security support to any backend does not constitute a significant overhead for binary modules, given the extremely small size of Metaspex-based backends. Even if all the built-in web services supporting security are included, the overhead is minimal.

All types participating in the security ontology are “first class citizens”, in the sense that they can be integrated in any user-defined ontology through **link** relationships. Take for instance a booking type in an ontology performing some kind of reservation, the booking type in the ontology can bear a strong **link** towards the user that made it. Referential integrity applies and if a user is removed, all their bookings will be removed too. An integration of security at ontology level is significantly more powerful than a mere integration at web services level through some identity management system microservice doing user validation.

Users

A central type within security is the one describing users. Each user is materialized by a document in the directory.

A core principle of Metaspex’s security is that users “exist in themselves,” independently from the organizations they can be affiliated to. This is in place to support use cases where single sign-on is needed, possibly with multitenancy. When a user logs in, Metaspex assigns a session token, which is presented by the back end as a cookie, to facilitate the integration of Metaspex web services in a browser. Browsers will automatically present the session cookie, without anything needed in client code. Non-browser clients implement their own cookie jar to process Metaspex’s session token.

The session token does not grant any right to call security-enforced web services. Once logged in, a user can only leverage web services without security, if any have been defined.

In order to issue security-enforced web services, the user will have to select an organization it is affiliated to. This is performed through a built-in service allowing users to do that after log in. Organizations will be covered later in this document.

The “_login” service is, like the “_logout” service built into Metaspex. The “_selco” service allows the user to select the affiliation the user wants to leverage for subsequent web services calls which are subject to security checks.

The root user

Amongst users one has a specific role: the **root user**. The root user is automatically created at directory creation. It cannot be removed.

The **root user** possesses special privileges. For instance, it is capable of removing other **user**s. Please consult the reference documentation for further details about what the **root user** can do.

User creation and activation

Depending on the specific security policy defined for the application, **users** can be created through a variety of methods:

- By using the **_user_create** service built into the Foundation Ontology. The security policy can specify whether **user**s can create themselves or if only the **root user** can do it.
- By using the **_user_create_oauth2** service. If the security policy allows it, the OAuth 2.0 protocol alongside JWT can be used to create **user**s.
- By using the **hx2a mkuser** utility.

The security policy can specify whether **user**s need email activation after their creation. If email activation is required, an email is sent to the **user**'s email address to confirm they requested a **user** creation. The security policy can specify that **user**s are immediately activated at creation.

To create a **user**, a number of pieces of information are needed, among which:

- A **user** id, which will be used at login
- A family name and a given name
- A preferred language
- A physical address
- A password

The password supplied must comply with the **user**-defined security policy constraints on passwords. A minimum length can be specified, along with how many uppercase characters, lowercase characters or symbols must be present.

In compliance with security standards such as PCI-DSS, Metaspex does not store **user**s' passwords in their document in the directory. Metaspex stores only a hashed value. That way, even in case hackers gain access to **user**s' data in the directory, they will not be able to reverse engineer passwords, and they will not be able to log in.

Passwords are hashed using the industry-standard PBKDF2 cryptographic algorithm based on SHA-512, 512-bit salt, and 10,000 iterations, making password data impenetrable. The

512-bit secret key is user-defined and specified in the configuration file, with protected access.

Document identifiers

Although the automatic allocation of unique document identifiers by Metaspex is not strictly part of security, it is security-related.

Metaspex uses Universal Unique Identifiers (UUIDs) to assign document identifiers automatically. These UUIDs are generated randomly using a Mersenne twister generator.

Communities and organizations

Users can be affiliated to **organization**s. Before we talk about affiliations themselves, let's talk about organizations. The type **organization** is part of Metaspex's Foundation Ontology.

Organizations are documents stored in the directory. They are hierarchical: each **organization** belongs to one upper **organization** only. The only exception is the so-called **slash organization** which has no upper **organization**.

Some **organization**s are more important than others; these are called communities. Communities are not specific types. Instead, they are **organization**s with a specific community flag. Communities can contain simple sub-**organization**s which are not communities. Simple sub-**organization**s can in turn contain other simple sub-**organization**s, forming a tree of **organization**s. Communities can contain other communities, either directly or transitively (through sub-**organization**s).

The concept of community supports Metaspex's automatic multitenancy. Communities provide data isolation against other communities.

When a user is affiliated to an **organization**, it is therefore by extension affiliated to the community their **organization** immediately belongs to. By "immediately," we mean it is just a matter of following the path of parent **organization**s from the **organization**s **user**s are affiliated to, to find the first occurrence of a community. In practice, this traversal is not necessary as **affiliation**s are **entities**, so they are subject to multitenancy isolation. This means that **affiliation** documents bear a direct strong **link** to the community they pertain to.

Affiliations

A given user can have multiple **affiliation**s to **organization**s and therefore communities. A **user** can be affiliated with only one **organization** in a given community, though.

Affiliations are captured as Metaspex **entity** documents in the directory. Every **affiliation** document contains **permission**s for the **role**s a **user** is entitled to when performing activities on behalf of a community they selected. We'll come back to that later.

Affiliations are contracts, and if necessary they can be assigned an expiration time. Beyond that time, the **affiliation** will not be valid any longer. As contracts, the security policy can require that **affiliation**s are signed (in the sense of being validated through a Metaspex Foundation Ontology web service) by the **user** and by another **user** with special privileges also affiliated to the same community.

An **affiliation** can also grant administrative rights to a **user** for the corresponding community. Administrative rights authorize specific operations, detailed in the reference documentation. A **role** with fewer rights can be assigned instead if desired: the built-in **legal role**. The legal **role** merely authorizes a **user** to sign affiliations on behalf of the community.

As **affiliation**s are **entities**, they bear a strong **link** towards the community they pertain to. This means that Metaspex's referential integrity applies, and if an **organization** or an entire community is removed, the **affiliation**s to them will be removed too.

Affiliations bear a strong **link** towards the **user** they apply to. Here also, referential integrity applies: if a **user** is removed, all their **affiliation**s will be removed too.

Depending on whether a community has been created open or not, creating **affiliation**s can be performed directly by **user**s who desire to join a community, or can only be created by specific privileged **user**s affiliated to the community.

In the case in which **user**s can join communities by their own action, Metaspex supports the creation of **ban**s. **Ban**s allow communities to prevent **user**s from creating **affiliation**s to the **organization**s that banned them.

Affiliations can be created through built-in web services or with the **hx2a** utility supplied with Metaspex.

Solutions

Metaspex's security ontology captures the concept of **solution**. A **solution** is an application published by a community. It is leveraged by users affiliated to **organization**s possessing a **license** for that **solution**. We'll cover **license**s in the next section.

As one example of a **solution,** take, for instance, an event reservation application published by a software firm. The software firm is represented as a community in the directory. The Metaspex-powered products they publish are captured by various **solution**s, including the event reservation application.

Solutions are Metaspex **entities**, meaning that they are subject to multitenancy isolation.

Solution documents appear, like the **organization**s they are published by, as documents in the directory.

Each **solution** document contains a list of **role**s which can be granted to affiliated **user**s.

Solutions can be created through built-in web services or with the **hx2a** utility supplied with Metaspex.

Licenses

Licenses are Metaspex **entities** which capture the existence of a contract between a community publishing **solution**s and an **organization** contracting to use some of these **solution**s.

Without a valid **license**, a user will not be able to perform **solution**-related activities requiring a **solution role** in their **affiliation**.

In order to be valid, a **license** must be created and signed by the **solution** provider community and the **organization** using the **solution**. When an activity is performed by a user, the corresponding **license** must also be within its window of validity. A **license** can be created with an expiration date, and if the **license** has expired, the activity will not be permitted by Metaspex.

Licenses can also be created "global." This allows sub-communities within communities to leverage parent communities' **license**s.

Licenses can be created through built-in web services or with the hx2a utility supplied with Metaspex.

Roles and permissions

Roles are Metaspex **entities** capturing a specific user profile allowed when performing activities pertaining to a **solution**. **Role**s bear a strong link to their corresponding **solution**. If a **solution** is removed, Metaspex referential integrity will remove the corresponding **role**s automatically.

Affiliations contain a list of **permission**s the corresponding user has been granted. A **permission** bears a strong **link** towards the **role** they correspond to. The permission contains also the timestamp capturing the moment when the permission was granted.

Security-enforced web services adopt a very simple pattern, they require the user calling them to possess the **role** they are associated with.

Roles and **permission**s in **affiliation**s can be created through built-in web services or with the hx2a utility supplied with Metaspex.

Logging in

Metaspex offers three different ways to perform login:

- Standard login using the user's user id and password. An optional flag indicates whether the user wants to be "kept logged in". If the flag is not set, a session cookie is sent back by the backend. If it is set, the session cookie is persistent.
- OAuth 2.0 login, using JWT. The configuration file allows to fine tune the behavior, including considering which authentication platform is data master.
- Login from email. Same as standard login, except that it takes the email address of the user.

As we'll see, the configuration file can alter the behavior of all these login methods. Also, it can impose two-factor authentication, limiting the usage of the application on some devices.

When login is successful, a session token is automatically sent back by the backend.

The session token

Whether the client performing login is a browser, another type of GUI or some other system, after a successful login, the backend returns a session token as a cookie. If the client is a browser, the cookie will be presented automatically again and again at every interaction with the backend. In other situations, the client will have to implement its own cookie jar technique.

The session token is a completely opaque string of characters that the client cannot tamper with. In order to make it invulnerable to attacks or spoofing, the session token uses symmetric encryption with industry-standard high-end AES-256, with 128-bit block size. The 256-bit secret key is user-defined and stored in the read-protected configuration file. Checksums are cryptographic, using SHA-256 hashing with 256-bit secret key HMAC. The 256-bit key used is also user-defined and stored in the read-protected configuration file.

Metaspex offers ways for a backend to cascade web services calls to other backends (or microservices) passing along the session token automatically and transparently.

Selecting the community

We saw that users exist “in themselves”, independently from the various affiliations they can have with different organizations.

When calling security-enforced web services, Metaspex will check whether the user possesses the corresponding **role**. To do so, Metaspex needs to inspect the affiliation the user has with their “current” community. That community is selected using a Metaspex built-in web service called “**_selco**” (for “selection community”).

In other words, the sequence a user has to follow to call a security-enforced web service is:

1. Perform login
2. Select a community among the ones they are affiliated to
3. Perform the succession of the security-enforced web services calls they need

As a given user can have several **affiliation**s, steps 2 and 3 can be repeated after login. After each community selection, a sequence of web services calls can be performed “on behalf” of the selected community.

Of course, Metaspex will only allow community selection if the **user** possesses a valid **affiliation** with that community.

Foundation Ontology built-in services

Login and community selection are not the only built-in web services offered by Metaspex. There are more than 200 built-in web services around security. They are as diverse as services which enable:

- Reporting that the **user** has lost their password
- Requiring a change in **user** data
- Performing logout
- Listing all **user**s by name
- Listing all **affiliation**s to a community by various criteria
- Etc.

Each of these built-in web services follows its own security restrictions. Please consult the Metaspex reference guide for details about the full range of web services offered and the restrictions associated with each one.

Minimum setup to enforce roles

The sophistication of the security-related part of Metaspex's Foundation Ontology is meant to cover the most complex needs. In many situations the problem is simpler.

For instance, you might have only one **organization**, in which case each **user** has only one **affiliation**, and no multitenancy applies.

Another typical situation is that there is only one **solution**, which translates into only one **license** if there is one **organization**.

In these cases the implementation of the backend can be fast tracked, and only a minimal number of documents need to be created in the directory.

But in any case, if **role**s need to be enforced, they need to be created in the directory, and every user logging in must subsequently select an **organization**.

Enforcement of a role in web services

It is time to look at some code. Let’s describe a web service with a simple “ok” endpoint, taking the empty JSON object, doing and returning nothing, which can be called only if the **user** calling it possesses the "person_admin" **role**:

```
#include "hx2a/role_checker_prologue.hpp"
#include "hx2a/service.hpp"

using namespace hx2a;

auto _ok_service =
  service<"ok", role_checker_prologue<"person_admin">>([](){
    // Doing nothing.
  });
```

The name of the **role** is "person_admin". That name must be defined during application implementation in the configuration file, following a definition such as:

```
id "person_admin" "4faba2992e84467f8a9b972351caab3e"
```

The **role_checker_prologue** functor takes care of checking whether the “current” community selected by the **user** invoking the web service grants that **user** in their affiliation the **role** with document identifier "4faba2992e84467f8a9b972351caab3e".

Please consult the technical insight (later in this series) pertaining to the configuration file for more details on the “id” keyword it accepts.

Paginated web services follow the same pattern:

```
#include "hx2a/paginated_services.hpp"
#include "hx2a/role_checker_prologue.hpp"

// Declaration of MySource.
#include "mysource.hpp"

// Some declaration of MySourceRow, which contains the relevant details from
MySource to be
// returned as rows when a pagination service is called.
// ...

using namespace hx2a;

rfr<MySourceRow> compute_mysource_row(const rfr<MySource>& source){
   // Some code creating a MySourceRow with make from the source reference, and
returning it.
   // ...
}

paginated_services<"my_paginated_service", MySource, compute_mysource_row,
role_checker_prologue<"person_admin">>
   _my_paginated_services(config::get_id("my_database"), "my_index");
```

Here the **role_checker_prologue** is used in a very similar manner to its use in the previous example

The audit database

Metaspex comes with an optional feature: logging every web service interaction along with cookies, including specifically the session cookie and the JSON query and reply payloads.

This logging is made in a specific location called the “audit database.” Here as well, just as for the directory, it is possible to define precisely with which database product and with which physical characteristics this database is to be used by the backend.

Once defined in the configuration file, the database will begin to be populated with documents containing the rich information above. No specific indexing is prepared in

advance; it is up to the application designer to decide which kinds of queries are to be performed for auditing. Obviously, the audit database logs timestamps to allow time interval queries.

Some specific kinds of **slots** are provided to avoid logging very sensitive information such as credit card numbers. This kind of sensitive data can be sanitized in the audit database.

15 validation steps for web services

Metaspex backends are compiled as web server modules running inside the web server of your choice.

As a result, a substantial part of the security policy is handled at the web server configuration level. For instance, the ability of the backend to answer HTTPS web services is completely configured in the web server, using familiar techniques to activate SSL/TLS.

Zero-trust measures with, for instance, X.509 certificates can also be put in place at the web server level.

Metaspex security is not limited to web server-level security configuration. It supplements that lower-level security configuration and it incorporates it, as Metaspex security stacks above what the web server is already doing.

Now let’s assume that Metaspex is confronted with an attempt by a **user** to perform a web service call, and the web service requires that the **user** has a specific **permission** for a **role**. Here are the 15 steps that Metaspex performs every time to verify that the web service can go through:

1. Metaspex checks that the **user** invoking the service is logged in (a prior call to the **_login** service must have been made). To perform this Metaspex checks that a session cookie was presented. An exception of type **must_login** is thrown in case the check fails.
2. After session cookie decryption, Metaspex checks that the value of the cookie is valid (in particular, that it has not been tampered with). Metaspex does this with the cryptographic checksum found within the cookie. An error response corresponding to the exception of type **invalid_session_token** is made to the client in the case where the check fails. In that case, the cookie will also be suppressed by the server.
3. Metaspex checks that the service endpoint name exists. If it doesn't, the HTTP request will be answered with an HTTP error code which is web server dependent, but in practice will be most of the time 404 (Resource Not Found).

4. Metaspex checks that the corresponding **user** still exists. If it does not (the **user** might have been removed since they logged in), an error response corresponding to the exception of type **user_logged_in_does_not_exist** is made to the client.
5. Metaspex checks that the **user** is still active. Even if the **user** successfully logged in, a subsequent login in another session might have put it in an inactive state. Depending on the state of the **user**, the following exceptions might be thrown: **user_is_not_yet_activated**, **password_expired**, **user_is_locked**, **user_is_suspended**, **password_must_be_changed**. If no exception is thrown, the **user** is considered "logged in".
6. Metaspex checks that the **user** logged in has selected a community on behalf of which he is operating (a prior call to the **_selco** service must have been made). To do that Metaspex inspects the session community to see if a community document identifier is present. If it is not, a response corresponding to the exception of type **no_community_selected** is made to the client.
7. Metaspex checks that the community document identifier is a valid UUID. If it is not, an error response corresponding to the exception of type **community_cookie_invalid** is made to the client.
8. Metaspex checks that the community is an **organization** that exists in the directory. In case it does not, an error response corresponding to the exception of type **organization_does_not_exist** is made to the client. In that case the community cookie is changed by the server to be the **user**'s unipersonal community. (The reference guide provides details on this type of community.) If the community exists, a check is made that it is a community, and not just a sub-**organization**. If it is not a community, an error response corresponding to the exception of type **organization_must_be_a_community** is made to the client.
9. A check is made that the community is still active, that the **user** logged in is (still) affiliated with the community, that the **affiliation** is signed (which means it is not deactivated), and that the **affiliation** is not expired. Note that the **root user** has an active **affiliation** with any community without the need for an explicit **affiliation**. If one of these checks fails, an error response corresponding to the exception of type **user_not_affiliated** is made to the client.
10. If the **user** logged in is not one of the safe **user**s (as declared in the configuration file), the session maximum duration is checked (see the configuration file), and if the session has exceeded its maximum duration, an error response corresponding to the exception of type **maximum_session_duration_exceeded** is made to the client.
11. If the **user** logged in is not one of the safe **user**s, the inactivity timeout is checked (see the configuration file), and if the session has timed out, an error response corresponding to the exception of type **session_timed_out** is made to the client.
12. Metaspex checks that the **role**'s physical identifier corresponds to an actual **role** stored in the directory. If the check fails, a response corresponding to the exception type **role_does_not_exist** is made to the client.

13. Metaspex checks that the community the **user** is operating for has an active **license** for the **solution** corresponding to the **role**. Parent organizations with global **license**s are checked too. The contract corresponding to the **license** is considered active if it is signed (which means it is not deactivated), and it is not expired. Note that the **root user** has such a **license** with any explicit contract. If the check fails, a response corresponding to the exception of type **user_organization_does_not_have_license** is made to the client.
14. Metaspex checks that the **user** is the root user, is an administrator of the community, or that their **affiliation** to the community contains the **permission** for the **service**'s **role**. In case their affiliation lacks the needed permission, a response corresponding to the exception of type **user_not_authorized** is made to the client.
15. Last, the JSON payload is parsed, and a number of errors can happen at UTF-8 level, at JSON syntax level or when deserializing slot types. If, for instance, a string is supplied in a JSON payload when a numeric slot is expected, a parsing error will be reported.

After all these checks, if they are successful, Metaspex will automatically trigger the code in the lambda expression assigned to the **service**.

Two-factor authentication

Metaspex supports two-factor authentication. When activated in the security policy specified in the configuration file, every time a **user** attempts to log in from a new terminal, an email will automatically be sent by the backend to the user to validate that the login should be authorized. Only after validation by the **user** will the login be authorized.

Database passwords

With Metaspex, whether you work with one or several databases, you can distinguish the database administrators from the database users who are only allowed to interact with the databases used by the backend. When you separate the two, the Metaspex backend will not need the administrator user id and password. It will only need the database user id and password. You can restrict the permissions for that user through database administration tools.

The user id and password of the database user are completely absent from the code of the application. They are purely stored in the configuration file. Please consult the technical insight dedicated to configuration for further details.

The configuration file itself can be protected at operating system-level so that the operating system user running the web server (and consequently the backend Metaspex module itself) is the only user with access to the configuration file and the database credentials specified inside of it.

Encryption within the database

Sometimes, in addition to the encryption capabilities of the operating system and the ones databases can offer, we need specific document **slots** to be encrypted. Metaspex allows that through a dedicated abstraction which replaces the usual **slot**: **slot_encrypted**. Encrypted slots are declared exactly like regular slots; Metaspex will ensure that the corresponding data inside the database is encrypted using AES-256 with 128-bit block size. Decryption is automatic. When documents are handled within a Metaspex specification, they are manipulated in-memory in decrypted form. The secret 256-bit key used is user-defined and securely stored in the read-protected configuration file.

Conclusion

Metaspex Foundation Ontology types supporting security are first-class citizens. They can be integrated into user-defined ontologies to describe applications never seen before. Microservice-based integration with identity management solutions pales in comparison to

ontology-based integration, which benefits from Metaspex's automatic powerful referential integrity.

With **solution**s and **license**s, Metaspex eliminates some of the barriers between CRMs and security to offer new ways to manage the relationship between suppliers and customers.

Metaspex supports industry-record levels of security, providing, for instance, 10,000-iteration **user** password hashing. Also, in addition to classical HTTP-level security, Metaspex implements up to 15 security steps to allow **user**s to initiate web service calls. In spite of this record level of verification, the overhead is minimal thanks to Metaspex's raw speed, keeping the response time of most web services calls instant.