Security_4_Plugfest – How To

Author: Oliver Pfaff, Siemens AG, CT RTC ITS

Introduction

This document assumes that you will (or consider to) participate in the Plugfest during the Nice F2F of the W3C IG WoT and you are developing any of the following components:

- A **resource server** (short: RS) responding to requests via **HTTP** or **CoAP**
- A **client** (short: C) sending requests via **HTTP** or **CoAP**

This document provides a how-to for the security-enabling of these interactions. An overview of the security-enabling is provided in an accompanying slide-deck.

Unprotected Interactions

Adding security is regarded optional for the Nice Plugfest. So your client and/or server should be prepared to accept resp. provide requests via HTTP-plain resp. CoAP-plain without supplying or requiring a security token. This document does not provide further details about unprotected interactions. Do consult other Plugfest documents for that.

Protected Interactions

The following describes the *basic* protection of the interactions during the Nice Plugfest. It is intentional to offer a *low entry*-barrier to encourage many security-enabled Plugfest implementations. For that purpose it is fully intentional to take various shortcuts (e.g. static settings/configurations, 'allow-all'/liberal policies, minimal token contents, optional transport-level security etc.). For production use additional considerations and additional security mechanisms/checks will typically be required: ***the basic protection utilized for Plugfest security shall not be used 1:1 in production setups***

This protection comprises following security services:

- Request authorization and (indirect) caller authentication: this is achieved by means of security tokens. For the Nice Plugfest this is the **primary security goal**. This is detailed in the first part of this HowTo.
- (Optional) message authentication and encryption: this is done by means of TLS (for HTTP) and DTLS (for CoAP). It presents a **subordinate security goal** for the Nice Plugfest. This is detailed in the second part.

Simple Request Authorization and Caller Authentication

The goal is to authorize requests and (implicitly) authenticate callers by means of bearer[1] security tokens (JWT[2]) with minimal contents.

Communications via HTTP

This section applies to Cs and RSs which support HTTP as a means of their interaction.

RS Developers

Configuration

Configure the RS component with

- RS id: `NicePlugfestRS`[3]
- AS issuer name `NicePlugfestAS`
- AS public signature verification key for ES256[4]. This is following JWK object[5]:

```
{
  "keys": [
    {
      "kty": "EC",
      "use": "sig",
      "crv": "P-256",
      "kid": "PlugFestNice",
```

[1] A shortcut: for the protection of high-value resources, PoP resp. HoK models may be required
[2] Another shortcut: JWTs are standard (RFC 7519), well-understood among Web developers and well-covered in implementation (see jwt.io). JWTs match RFC 7228 class 2 devices. Equivalent means for class 1 or 0 are not yet available - not yet as standards and also not as (mature) implementations
[3] Another shortcut: static/invariant RS identifier values for all RSs
[4] See appendix for the utilization of alternative JWT token signature schemes
[5] Another shortcut: this is a plain public key for verifying JWT signature values

```
            "x": "CQsJZUvJWx5yB5EwuipDXRDye4Ybg0wwqxpGgZtcl3w",
            "y": "qzYskD2N7GrGDSgo6N9pPLXMIwr6jowFGyqsTJGmpz4",
            "alg": "ES256"
          }
        ]
      }
```

Registration

Skipped[6] for RSs that support ES256

Operation

When receiving a HTTP request at a protected endpoint[7]:

1. Check if the request contains an `Authorization` header. Respond with a 401 error if not
2. Check if the request contains an `Authorization: Bearer`-header with non-null/empty contents. Respond with a 401 error if not
3. Check if the value of the `Authorization Bearer`-header is a JWT object. Respond with a 401 error if not
4. Check if the JWT object is signed. Respond with a 401 error if not
5. Check if the signature of the JWT object is valid. This is to be checked with AS public signature verification key (see above). Respond with a 401 error if invalid
6. Check the contents of the JWT object[8]:
    - Check if the value of "`iss`" is `NicePlugfestAS`. Respond with a 401 error if not
    - Check if the value of "`aud`" is `NicePlugfestRS`. Respond with a 401 error if not
7. Accept the request as well as "`sub`" as the originator of the request and process it as usual[9]

For more background see RFC 6750 (HTTP Bearer tokens) and RFC 7519 (JWT), RFC 7517 (JWK). For JWT libraries in various programming languages see http://jwt.io/ section "*Libraries for Token Signing/Verification*". For JSON libraries see http://json.org/. Depending on the chosen strategy (JSON vs. JWT library) and instance, the steps 4-6 will be done by a 3rd party library

[6] Another shortcut

[7] It is up to the RS (aka servient) implementation whether resources at a specific URL (e.g. http(s)://<host>:<port>**/protected** where a security token is mandatory vs http(s)://<host>:<port>**/public** where it is not), a specific port or all resources of a servient are protected. In any case the RS or servient implementation shall specify one or more URLs at which protected resources can be reached in order to facilitate 3rd party client or caller components.

[8] Another shortcut: further checks would apply in production environments including an "`aud`" check which depends on RS registration (that was skipped for simplicity)

[9] Another shortcut that implements a naive and very coarse access control: presenters of a valid security token (no detailing anything about the resource) may access any available resource with any method. One would not do this in production systems at least not without careful consideration.

C Developers

Configuration

Not required

Registration

Registration may be done programmatically (default) or manually[10] (fallback). Programmatic registration is done according RFC 7591 and described in the following:

Create a HTTP request[11] with JSON request content as in the following prototype and send it via TLS to the AM[12]:

Request:

```
POST /iam-services/0.1/oidc/am/register HTTP/1.1
```

Request headers:

```
Host: ec2-54-154-59-218.eu-west-1.compute.amazonaws.com
Content-Type: application/json
Accept: application/json
```

Request body[13]:

```
{
      "client_name": "yourClientName",
      "grant_types": ["client_credentials"]
}
```

Security: note that the registration endpoint is unprotected for the purpose of the Plugfest[14]. To comply with its TLS profile, configure the client-side TLS engine as follows:

- **Protocol version**: TLSv1.2, 1.1 or 1.0 (note that the usage of SSLv3.0 is deprecated by the IETF)
- **Cipher suite**: any ECDHE_ECDSA cipher suite. Note that ECDSA is used as public key algorithm during server authentication
- **Server authentication**: configure the trusted root CAs to Plugfest root CA certificate

---

[10] For manual registration send a mail with *yourClientName* to the author of this document. You will get <c_id> and <c_secret> values in a response mail.

[11] If the C component is implemented as a browser-based app that wants to utilize XMLHttpRequest for interactions with AM (registration, token request) the CORS rules apply (see http://www.w3.org/TR/cors/). The AM does set HTTP response headers `Access-Control-Allow-Origin: *` to facilitate calls by browser-based apps where the app code (.js for instance) is loaded from another origin as (`https, ec2-54-154-59-218.eu-west-1.compute.amazonaws.com, 443`). This presents another shortcut and it is suggested that client credentials needed to acquire tokens (client_id/client_secret) are not imprinted into the app code (the mobile app should make a registration to obtain client_id/client_secret and then use these dynamically acquired credentials in subsequent token requests)

[12] URL: https://ec2-54-154-59-218.eu-west-1.compute.amazonaws.com/iam-services/0.1/oidc/am/register

[13] Naming convention: replace all values prefixed "*your*" with your value i.e. use any string of your choice instead "*yourClientName*"

[14] Another shortcut that one would not do - at least not without careful consideration- in production

Response:

Check that you get a 201 Created response and extract the value of “client_id” (this value is called <c_id> in the following) and the value of “client_secret” (called <c_secret> in the following) from the JSON response body. A response body prototype[15] is:

```
{
  "client_id": "889d02cf-16dd-4934-9341-a754088faxyz",
  "client_secret": "ahd5MU42J0hIxPXzhUhjJHt2d0Oc5M6B644CtuwUlE9zpSuF14-kXYZ",
}
```

Store <c_id>[16] and <c_secret>[17] for use during the token acquisition.

Token Acquisition

Create a HTTP request as in the following prototype and send it via TLS to the AM[18]:

Request:

```
POST /iam-services/0.1/oidc/am/token HTTP/1.1
```

Request headers:

```
Host: ec2-54-154-59-218.eu-west-1.compute.amazonaws.com
Content-Type: application/x-www-form-urlencoded
Accept: application/json
Authorization: Basic Base64(<c_id>:<c_secret>)
```

Request body:

```
grant_type=client_credentials
```

Security: same as in registration

Response:

Check that you get a 200 OK response and extract the value of the `access_token` member from the JSON response body. A response body prototype is:

```
{
       "access_token":"
eyJhbGciOiJFUzI1NiJ9.eyJhdWQiOiJhYjliN2NiNy02YzQwLTQ3ZjUtOTBiOC0xNTlmMzQ0MDIx
OTYiLCJzdWIiOiJhYjliN2NiNy02YzQwLTQ3ZjUtOTBiOC0xNTlmMzQ0MDIxOTYiLCJpc3MiOiJod
HRwczpcL1wvYW0uY29tcGFueS1zLmNvbTo4NDQzXC9pYW0tc2VydmljZXNcLzAuMVwvb2lkY1wvYW
1cLyIsInR5cCI6Im9yZzp3Mzp3b3Q6and0OmFtOmFzLXdyYXA6bWluIiwiYXNfdG9rZW4iOiJleUp
oYkdjaU9pSkZVekkxTmlKOS5leUpoZFdRaU9pSk9hV05sVUd4MVoyWmxjM1JTVXlJc0luTjFZaUk2
SW1GaU9XSTNZMkkzTFRaak5EQXRORGRtTlMwNU1HSTRMVEUxT1dZek5EUXdNakU1TmlJc0ltbHpje
Uk2SWs1cFkyVlFiSFZuWm1WemRFRlRJaXdpZEhsd0lqb2liM0puT25jek9uZHZkRHBxZDNRNllYTT
ZiV2x1SWl3aWFuUnBJam9pWTJZeU5UTm1PV1l0WXpVME5pMDBZbVF4TFRoa1ptUXROamcyTURabE1
6UmhOalEwSW4wLmZlcVZFZF9JdXk1Q2FPNDhhb0RuWUhfOTlPX1ZlYncxMDdLdXpSclRpc1RBeHZk
Y1JaWjZaS3RPVWZkOUNlclYzaG1rQWF0LUNXUElCckxMOUZXTFhRIiwianRpIjoiN2MyMjFmMWItZ
```

[15] The actual response object (JSON) will provide more information. The JSON response members that shall be processed are documented here. Additional information may be ignored.
[16] In the prototype <c_id>=889d02cf-16dd-4934-9341-a754088facdb
[17] In the prototype <c_secret>= ahd5MU42J0hIxPXzhUhjJHt2d0Oc5M6B644CtuwUlE9zpSuF14-kF8Q
[18] URL: https://ec2-54-154-59-218.eu-west-1.compute.amazonaws.com/iam-services/0.1/oidc/am/token

```
jNkNC00ZTc4LWE4ZWQtOGRmOWUxNGNjMWM5In0.Le_Y90zsO92MyDgbKxUCr3eUNU8Z7-
QKc0u0RSr26MHN1za2EUQ1wOoJhLDXR2dFo9geFf7mBbiM77EP6h0ldA",
       "token_type":"bearer",
       "expires_in":3600
}
```

Decode the value of the `access_token` value. This provides a JWT structure. Optionally validate it (see above for JWT validation hints). Extract the value of the `as_token` member in the JWT payload. This value is called `<as_token>` in the following.

Operation

Attach an `Authorization: Bearer <as_token>` header to HTTP requests to RS.

On an optional basis: supply C functionality that allows requesting a resource from a protected endpoint without a security token or an invalid security token (change some bytes to break the signature). This allows showing the effect of RS endpoint protection.

Communications via CoAP

This section applies to Cs and RSs which support CoAP as a means of their interaction.

RS Developers

Configuration

Same as for HTTP communications. See above

Registration

Same as for HTTP communications. See above

Operation

When receiving a CoAP request at a protected endpoint[19]:

1. Check if the request contains a CoAP option 65000 with non-null/empty contents. Respond with a 4.01 error if not
2. Check if the value of the CoAP option 65000 content is `Bearer <jwt_token>` with a non-null/empty `<jwt_token>`. Respond with a 4.01 error if not
3. Check if the JWT object is signed. Respond with a 4.01 error if not
4. Check if the signature of the JWT object is valid. This is to be checked with AS public signature verification key (see above). Respond with a 4.01 error if invalid
5. Check the contents of the JWT object[20]:
    a. Check if the value of "`iss`" is `NicePlugfestAS`. Respond with a 4.01 error if not
    b. Check if the value of "`aud`" is `NicePlugfestRS`. Respond with a 4.01 error if not

[19] It is up to the RS (aka servient) implementation whether resources at a specific URL (e.g. coap(s)://<host>:<port>**/protected** where a security token is mandatory vs coap(s)://<host>:<port>**/public** where it is not), a specific port or all resources of a servient are protected. In any case the RS or servient implementation shall specify one or more URLs at which protected resources can be reached in order to facilitate 3rd party client or caller components.

[20] Another shortcut: further checks would apply in production environments including an "`aud`" check which depends on RS registration (that was skipped for simplicity)

6. Accept the request as well as "`sub`" as the originator of the request and process it as usual

C Developers

Registration
Same as for HTTP communications. See above

Token Acquisition
Same as for HTTP communications. See above

Operation
Attach `Bearer <as_token>` as value of the CoAP option 65000 to CoAP requests to RS.

On an optional basis: provide C functionality that allows requesting a resource from a protected endpoint without a security token or an invalid security token (change some bytes to break the signature). This allows showing the effect of RS endpoint protection.

Advanced Request Authorization and Caller Authentication
The AM and AS components also support a more advanced way of request authorization that supplies a JWT bearer security token with actual access control information in style of AIF (draft-bormann-core-ace-aif-03). If you want to move to this level: send an email to the author.

Message Authentication and Encryption
Message authentication and encryption is done by means of TLS (for HTTP) and DTLS (for CoAP). Utilizing TLS and DTLS for the interactions between C and RS is an optional part of security-enabling[21].

Communications via HTTP
This section applies to Cs and RSs which support HTTP as a means of interaction between C and RS and which opt-in to utilize message authentication and encryption. The goal is HTTP-over-TLS with server authentication[22].

RS Developers
Expose a port at the default port number 443[23] which accepts HTTP-over-TLS according RFC 2818[24]. Configure the TLS engine as follows:

- **Protocol version**: all supported TLS versions (note that the usage of SSLv3.0 is deprecated by the IETF, see RFC 7568)
- **Cipher suite**: all available ECDHE_ECDSA cipher suites. Note that ECDSA is used as default public key algorithm during server authentication

[21] Rationale: i. to allow the inclusion of esp. RSs without native support for TLS or DTLS resp. ii. to allow participants to focus on authorizing actions and authenticating actors
[22] This is a shortcut for telling the RS what to do while telling the C to accept all. Okay for a Plugfest, should not be done in any production setup.
[23] http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt
[24] Java containers esp. Servlet engines contain such implementations.

- **Server authentication**: public/private key pair for ECDSA where the public key is certified by the Plugfest Root CA. To obtain a server certificate that is signed by the Plugfest Root CA send a PKCS#10 request to the author. This shall contain:

```
-----BEGIN CERTIFICATE REQUEST-----

Base64(PKCS#10(serverName, serverPublicKey))

-----END CERTIFICATE REQUEST-----
```

Note that common tools such as the Java KeyTool or OpenSSL can be used for ECDSA key pair as well as PKCS#10 certification request generation

C Developers

When encountering a resource access at a https-URL send the HTTP request over a TLS-protected channel according RFC 2818. Configure the TLS engine to run:

- **Protocol version**: any supported TLS version (note that the usage of SSLv3.0 is deprecated by the IETF)
- **Cipher suite**: any ECDHE_ECDSA cipher suite (default: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256)
- **Server authentication**: configure the trusted root CAs to contain the Plugfest Root CA

Communications via CoAP

This section applies to Cs and RSs which support CoAP as a means of interaction between C and RS and which opt-in to utilize message authentication and encryption. The goal is CoAP-over-DTLS with server authentication[25].

RS Developers

Expose a port at the default port number 5684[26] which accepts CoAP-over-DTLS in style of RFC 2818 (there is no equivalent to RFC 2818 in the CoAP/DTLS ecosystem). Configure the DTLS engine as follows:

- **Protocol version**: all supported DTLS versions[27]
- **Cipher suite**: all available ECDHE_ECDSA cipher suites[28]
- **Server authentication**: public/private key pair for ECDSA where the public key is certified by the Plugfest Root CA. To obtain a server certificate that is signed by the Plugfest Root CA send a PKCS#10 request to the author. This shall contain:

```
-----BEGIN CERTIFICATE REQUEST-----

Base64(PKCS#10(serverName, serverPublicKey))
```

[25] This is a shortcut for telling the RS what to do while telling the C to accept all. Okay for a Plugfest, should not be done in any production setup.
[26] http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt
[27] For Scandium 1.0.0 this is DTLS v1.2
[28] For Scandium 1.0.0 this list is: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256, TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

```
-----END CERTIFICATE REQUEST-----
```

Note that common tools such as the Java KeyTool or OpenSSL can be used for ECDSA key pair as well as PKCS#10 certification request generation

C Developers

When encountering a resource accesses at a coaps-URL send the CoAP request over a DTLS-protected channel in style of RFC 2818. Configure the TLS engine to run:

- **Protocol version**: any supported DTLS version[29]
- **Cipher suite**: any ECDHE_ECDSA cipher suite (default: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256)
- **Server authentication**: configure the trusted root CAs to contain the Plugfest Root CA

Plugfest Root CA Certificate

Following self-signed root CA certificate is used for signing RS certificates for TLS and DTLS-protected communications:

```
-----BEGIN CERTIFICATE-----

MIIBgjCCASigAwIBAgIEVm6vGDAKBggqhkjOPQQDAjA3MQwwCgYDVQQKDANXM0Mx

DzANBgNVBAsMBldvVCBJRzEWMBQGA1UEAwwNTmljZSBQbHVnZmVzdDAgFw0xNTEy

MTQxMjAwNDFaGA8yMDY1MTIxNDEyMDA0MVowNzEMMAoGA1UECgwDVzNDMQ8wDQYD

VQQLDAZXb1QgSUcxFjAUBgNVBAMMDU5pY2UgUGx1Z2Zlc3QwWTATBgcqhkjOPQIB

BggqhkjOPQMBBwNCAARi7l1JQPjYpjCPpHRNC4nTwRj+vEWXunSiawpD9a2rKgJi

4g1+jLDmrxztrJa7e9NI9BkD9vZP2DgU6DfAU2czoyAwHjAPBgNVHRMECDAGAQH/

AgEDMAsGA1UdDwQEAwICBDAKBggqhkjOPQQDAgNIADBFAiBbYQOvT7yjmm+V9L2e

Dg6eF1s4uHeorqgS1P51NbGgsgIhAJs9KWtXkFxflCXV9vafNk02xcV1lOYn7WRW

A7NR3A19

-----END CERTIFICATE-----
```

Appendix: Alternative AS Token Signature Scheme

The default AS token signature scheme is ES256. RSs that can use JWT libraries which support ES256 can proceed as described above.

If ES256 support is not available an alternative signature scheme is needed. In this case HS256 should be considered as a fallback. This has following implications:

[29] For Scandium 1.0.0 this is DTLS v1.2

- Signalizing the use of an alternative JWT signature scheme (to the AS) is performed as part of an RS registration at AS (not needed for ES256)
- Requesting the issuance of a JWT with an alternative signature scheme requires to supply the RS identifier in the token request which implies the issuance of a “normal” AS token

If an alternative AS token signature scheme send a mail to the author for further instructions.