SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Call Identifier:	FP7–ICT–2011–7
Project Number:	287305
Project Acronym:	OpenIoT
Project Title:	Open source blueprint for large scale self-organizing cloud environments for IoT applications

D5.2.1 Privacy and Security Framework a

Document Id:	OpenIoT-D521-130905-Draft
File Name:	OpenIoT-D521-130905-Draft.docx
Document reference:	Deliverable 5.2.1
Version:	Draft
Editor:	Robert Gwadera
Organisation:	EPFL
Date:	2013 / 09 / 10
Document type:	Deliverable (Prototype)
Dissemination level:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Robert Gwadera	EPFL	26/08/2013	ToC Proposal and Content
V02	Nikos Kefalakis	AIT	28/08/2013	Technical Review. Provided comments to be followed.
V03	Robert Gwadera	EPFL	30/08/13	The prototype described by Mehdi Riahi
V04	Nikos Kefalakis	AIT	03/09/2013	Technical Review – Second Round Provided comments
V05	Robert Gwadera	EPFL	05/09/2013	Final Edits and Comments Addressed
V06	Martin Serrano	DERI	2013/09/05	Circulation for Approval
V07	Robert Gwadera	EPFL	2013/09/10	Minor Edits / Typo Amendments
V08	Martin Serrano	DERI	2013/09/10	Approved
Draft	Martin Serrano	DERI	2013/09/10	EC Submitted

TABLE OF CONTENTS

1 INTRODUCTION 7
1.1 Scope 7
1.2 Audience 7
1.3 Methodology 7
1.4 Related Documents 8
1.5 Structure 8

2 SECURE MESSAGING 9
2.1 Message digest 9
2.2 Private key cryptography 9
2.3 Public key cryptography 10
2.4 Digital signatures 11
2.5 Digital certificates 12
2.5.1 Keytool and keystore 12
2.5.2 CertPath API 13

3 SECURITY PROTOCOLS 14
3.1 IEEE802.15.4 14
3.2 IPsec 14
3.3 TSL 15
3.4 HTTPS 17

4 ACCESS CONTROL 19
4.1 Fundamental concepts 19
4.2 Lattice-Based Access Control Models 21
4.2.1 Information flow policy 21
4.3 Access control models-based on information flow 22
4.3.1 Mandatory access policy 23
4.3.2 Confidentiality 23
4.3.3 Integrity 24
4.3.4 Combined 24

5 THE SECURITY ARCHITECTURE IN OPENIOT 25
5.1 Overview of the architecture 25
5.2 Access matrix 26
5.3 Mandatory access rules 27
5.4 Security protocol stack 27

6 AUTHORIZATION FRAMEWORK 28
6.1 Introduction 29
6.2 Roles 30
6.3 OAuth Protocol Flow 31
6.3.1 Authorization Grant 31
6.3.2 Authorization Code 31
6.3.3 Implicit 33
6.3.4 Resource Owner Password Credentials 33
6.3.5 Client Credentials 34
6.4 Access Token 34
6.5 Refresh Token 34
6.6 TLS Version 36
6.7 HTTP Redirections 36
6.8 Interoperability 36
6.9 Security considerations 36
6.9.1 Beyond Basic 36
6.9.2 Credentials 37
6.9.3 Signature and Hash 38
6.9.4 Secrets Limitations 38
6.9.5 Timestamp and Nonce 39
6.9.6 Signature Methods 40
6.9.7 Signature Base String 40

7 TRUSTWORTHINESS OF SENSOR READINGS 41
7.1 Notation 41
7.2 Problem definition 42
7.3 Overview of the method 42
7.4 Algorithm 44

8 IMPLEMENTATION IN JAVA 45

9 THE PROTOTYPE 47
9.1 Trust-Module in the OpenIoT Architecture 47
9.2 Trust-Module Implementation 48

10 CONCLUSIONS 50

11 REFERENCES 51

LIST OF FIGURES

FIGURE 1. SECURITY ARCHITECTURE IN OPENIOT. 25

FIGURE 2. ABSTRACT PROTOCOL FLOW.......... 30

FIGURE 3. AUTHORIZATION CODE FLOW. 32

FIGURE 4. REFRESHING AN EXPIRED ACCESS TOKEN. 35

FIGURE 5. TRUST OF SENSOR STREAM REPRESENTATIONS.......... 43

FIGURE 6. TRUST-MODULE IN OPENIOT.......... 48

LIST OF TABLES

TABLE 1: AN EXAMPLE ACCESS MATRIX 20

TABLE 2: A FRAGMENT OF THE ACCESS MATRIX IN OPENIOT 27

TABLE 3: SECURITY CLASSES AND ROLES IN OPENIOT 27

TABLE 4: THE SECURITY PROTOCOL STACK IN OPENIOT 27

TERMS AND ACRONYMS

AIT	Athens Information Technology
CAS	Central Authorization Server
CMC	Config/Monitor Console
DERI	Digital Enterprise Research Institute
EPFL	Ecole Polytechnique Fédérale de Lausanne
EU	European Union
FP7	Framework Program 7
ICO	Internet Connected Object
IoT	Internet of Things
JDK	Java Development Kit (version >= 1.4)
LSM	Sensor Data Cloud Database
PS	Physical sensor
SDUM	Service Delivery and Utility Manager
SCH	Scheduler
SCH	Scheduler
TM	Trust Module
VS	Virtual sensor
X-GSN	Extended Global Sensors Network

1 INTRODUCTION

1.1 Scope

This deliverable specifies and implements the OpenIoT security/privacy and trustworthiness framework. The aim is to ensure that internet-connected objects contributing to the OpenIoT platform and serving users' request will provide trusted data, while the data to be exchanged will be secure (according to the target security/confidentiality level specified by the user).

This deliverable is the first of a series of two describing the overall privacy and security functional framework of OpenIoT. It is the first of two releases planned for August 2013 and August 2014.

1.2 Audience

This privacy and security framework report and prototype deliverable addresses the following audiences:

- Technical Developers, for sharpening the privacy and security framework planning and development;
- Business Developers, by taking into account the described privacy and security components in order to design, implement and fine-tune the framework among other components of the OpenIoT architecture, semantic infrastructure, management framework, middleware and proof-of-concept applications in OpenIoT;
- The European Commission, in order to assess the OpenIoT progress regarding privacy and security perspectives.

1.3 Methodology

Security, privacy and trust issues in Internet of Things (IoT) are of fundamental importance and guaranteeing the highest standards in those respects is necessary to advance the application of (IoT). First of all, any future large-scale deployment of OpenIoT will only be possible if it complies with corresponding legal security/privacy and reliability requirements (e.g., recent related EU regulations on privacy and security). Clearly, apart from the legal requirements, a lack of proper security/privacy and trust mechanisms in IoT may lead to disastrous consequences as a result of hacking OpenIoT-based systems. As examples, consider pacemakers or insulin pumps that start functioning differently or smart meters, where as a result of hacking the bills start going up. One of the biggest challenges in designing a proper security/privacy and trust mechanism in OpenIoT is the fact that most existing IoT infrastructures consist of separately connected elements, having very differing security/privacy and reliability standards (e.g., fixed versus mobile devices).

1.4 Related Documents

It is assumed that the reader is familiar with the OpenIoT technology as described in the following deliverables:

- D2.2 OpenIoT Platform Requirements and Technical Specifications.
- D2.3 OpenIoT Detailed Architecture and Proof-of-Concept Specifications.
- D3.1.1 Semantic Representations of Internet-Connected Objects a.
- D3.1.2 Semantic Representations of Internet-Connected Objects b.
- D3.2 Semantic Communication Protocols (Object/Object and People/Object).
- D4.1 Service Delivery Environment Formulation Strategies.
- D4.3.1 Core OpenIoT Middleware Platform a

1.5 Structure

This document describes the proposed two main modules: the security/privacy module and the trustworthiness (trust) module.

The security module consists of the following sub modules: (I) secure protocols for information exchange (secure messaging) and (II) authentication and authorization. The challenges in designing the authentication and authorization model for OpenIoT stems from fact that it has to accommodate differing requirements from the distributed components of the OpenIoT platform.
The trustworthiness module evaluates trustworthiness of sensor streams based on streams of neighbouring sensors.

This document presents the first part of the security/privacy and trust specification (deliverable) and the second part, due in a year, will include more implementation details.

The content of this document is a follows: Section 2 reviews the theory of secure messaging, Section 3 reviews applicable security protocols, Section 4 reviews the theory of access control (authentication and authorization), Section 5 presents the proposed security module in OpenIoT, Section 6 presents details of the proposed authorization framework, Section 7 presents the proposed trust module and Section 8 discusses secure implementation in Java. Section 9 presents the prototype.

2 SECURE MESSAGING

In this section we review the fundamental concepts designed to provide secure messaging that are crucial in understanding secure protocols discussed in the following sections. In particular we review the following ones:

- **Message digests**. Coupled with message authentication codes, a technology that ensures the integrity of your message.
- **Private key encryption**. A technology designed to ensure the confidentiality of your message.
- **Public key encryption**. A technology that allows two parties to share secret messages without prior agreement on secret keys.
- **Digital signatures**. A bit pattern that identifies the other party's message as coming from the appropriate person.
- **Digital certificates**. A technology that adds another level of security to digital signatures by having the message certified by a third-party authority.

2.1 Message digest

A message digest is a function that ensures the integrity of a message. Message digests take a message as input and generate a block of bits, usually several hundred bits long that represents the fingerprint of the message. A small change in the message (say, by an interloper or eavesdropper) creates a noticeable change in the fingerprint.

The message-digest function is a one-way function. It is a simple matter to generate the fingerprint from the message, but quite difficult to generate a message that matches a given fingerprint.

Message digests can be weak or strong. A checksum, which is the XOR of all the bytes of a message, is an example of a weak message-digest function. It is easy to modify one byte to generate any desired checksum fingerprint. Most strong functions use hashing. A 1-bit change in the message leads to a massive change in the fingerprint (ideally, 50 percent of the fingerprint bits change).

If a key is used as part of the message-digest generation, the algorithm is known as a message-authentication code.

2.2 Private key cryptography

Message digests may ensure integrity of a message, but they can't be used to ensure the confidentiality of a message. For that, we need to use private key cryptography to exchange private messages.

Consider this scenario: Alice and Bob each have a shared key that only they know and they agree to use a common cryptographic algorithm, or cipher. In other words, they keep their key private. When Alice wants to send a message to Bob, she encrypts the original message, known as plaintext, to create ciphertext and then sends the ciphertext to Bob. Bob receives the ciphertext from Alice and decrypts the ciphertext with his private key to re-create the original plaintext message. If Eve the eavesdropper is listening in on the communication, she hears only the ciphertext, so the confidentiality of the message is preserved.

The JDK supports the following private key algorithms:

- **DES**. DES (Data Encryption Standard) was invented by IBM in the 1970s and adopted by the U.S. government as a standard. It is a 56-bit block cipher.
- **TripleDES**. This algorithm is used to deal with the growing weakness of a 56-bit key while leveraging DES technology by running plaintext through the DES algorithm three times, with two keys, giving an effective key strength of 112 bits. TripleDES is sometimes known as DESede (for encrypt, decrypt, and encrypt, which are the three phases).
- **AES.** AES (Advanced Encryption Standard) replaces DES as the U.S. Standard. It is a 128-bit block cipher with key lengths of 128, 192, or 256 bits.
- **RC2**, **RC4**, and **RC5**. These are algorithms from a leading encryption security company, RSA Security.
- **Blowfish**. This algorithm is a block cipher with variable key lengths from 32 to 448 bits (in multiples of 8), and was designed for efficient implementation in software for microprocessors.
- **PBE**. PBE (Password Based Encryption) can be used in combination with a variety of message digest and private key algorithms. The Cipher class manipulates private key algorithms using a key produced by the KeyGenerator class.

2.3 Public key cryptography

Private key cryptography suffers from one major drawback: how does the private key get to Alice and Bob in the first place? If Alice generates it, she has to send it to Bob, but it is sensitive information so it should be encrypted. However, keys have not been exchanged to perform the encryption.

Public key cryptography, invented in the 1970s, solves the problem of encrypting messages between two parties without prior agreement on the key. In public key cryptography, Alice and Bob not only have different keys, they each have two related keys. A message encrypted with one key can only be decrypted with the other and vice-versa. One key is private and must not be shared with anyone. The other key is public and can be shared with anyone. When Alice wants to send a secure message to Bob, she encrypts the message using Bob's public key and sends the result to Bob. Bob uses his private key to decrypt the message. When Bob wants to send a secure message to Alice, he encrypts the message using Alice's public key and sends the result to Alice. Alice uses her private key to decrypt the message. Eve can eavesdrop on both public keys and the encrypted messages, but she cannot decrypt the messages because she does not have either of the private keys.

The public and private keys are generated as a pair and need longer lengths than the equivalent-strength private key encryption keys. Typical key lengths for the RSA algorithm are 1,024 bits. It is not feasible to derive one member of the key pair from the other.

Public key encryption is slow (100 to 1,000 times slower than private key encryption), so a hybrid technique is usually used in practice. Public key encryption is used to distribute a private key, known as a session key, to another party, and then private key encryption using that private session key is used for the bulk of the message encryption.

The following two algorithms are used in public key encryption:

- **RSA**. This algorithm is the most popular public key cipher, but it's not supported in JDK and one needs to use a third-party library like BouncyCastle to get this support.
- **Diffie-Hellman**. This algorithm is technically known as a key-agreement algorithm. It cannot be used for encryption, but can be used to allow two parties to derive a secret key by sharing information over a public channel. This key can then be used for private key encryption.

2.4 Digital signatures

In this section, we examine digital signatures, the first level of determining the identification of parties that exchange messages.

Clearly, the public key message exchange described has the following problem. How can Bob prove that the message really came from Alice? Eve could have substituted her public key for Alice's, and then Bob would be exchanging messages with Eve thinking she was Alice. This is known as a **Man-in-the-Middle attack**.

We can solve this problem by using a digital signature, i.e., a bit pattern that proves that a message came from a given party.

One way of implementing a digital signature is using the reverse of the public key process. Instead of encrypting with a public key and decrypting with a private key, the private key is used by a sender to sign a message and the recipient uses the sender's public key to decrypt the message. Because only the sender knows the private key, the recipient can be sure that the message really came from the sender.

In actuality, the message digest, not the entire message, is the bit stream that is signed by the private key. So, if Alice wants to send Bob a signed message, she generates the message digest of the message and signs it with her private key. She sends the message (in the clear) and the signed message digest to Bob. Bob decrypts the signed message digest with Alice's public key and computes the message digest from the cleartext message and checks that the two digests match. If they do, Bob can be sure the message came from Alice.

Note that digital signatures do not provide encryption of the message, so encryption techniques must be used in conjunction with signatures if confidentiality is also needed. We can use the RSA algorithm for both digital signatures and encryption. A U.S. standard called DSA (Digital Signature Algorithm) can be used for digital signatures, but not for encryption.

The JDK supports the following digital signature algorithms:

- **MD2/RSA**.
- **MD5/RSA**.
- **SHA1/DSA**.
- **SHA1/RSA**.

2.5 Digital certificates

In this section, we discuss digital certificates, i.e., the second level to determining the identity of a message originator. We look at certificate authorities and the role they play. We examine key and certificate repositories and management tools (keytool and keystore) and discuss the CertPath API, a set of functions designed for building and validating certification paths.

Clearly, there is a problem with the digital signature scheme. It proves that a given party sent a message, but how do we know for sure that the sender really is who she says she is? What if someone claims to be Alice and signs a message, but is actually Amanda? We can improve our security by using digital certificates which package an identity along with a public key and are digitally signed by a third party called a certificate authority or CA.

A **certificate authority** is an organization that verifies the identity, in the real-world physical sense, of a party and signs that party's public key and identity with the CA private key. A message recipient can obtain the sender's digital certificate and verify (or decrypt) it with the CA's public key. This proves that the certificate is valid and allows the recipient to extract the sender's public key to verify his signature or send him an encrypted message. Browsers and the JDK itself come with built-in certificates and their public keys from several CAs. The JDK supports the X.509 Digital Certificate Standard.

2.5.1 Keytool and keystore

The Java platform uses a keystore as a repository for keys and certificates. Physically, the keystore is a file (there is an option to make it an encrypted one) with a default name of keystore. Keys and certificates can have names, called aliases, and each alias can be protected by a unique password. The keystore itself is also protected by a password; you can choose to have each alias password match the master keystore password.

The Java platform uses the keytool to manipulate the keystore. This tool offers many options including generating a public key pair and corresponding certificate, and viewing the result by querying the keystore.

The keytool can be used to export a key into a file, in X.509 format, that can be signed by a certificate authority and then re-imported into the keystore. There is also a special keystore that is used to hold the certificate authority (or any other trusted) certificates, which in turn contains the public keys for verifying the validity of other certificates. This keystore is called the truststore. The Java language comes with a default truststore in a file called cacerts . If one searches for this filename, he will find at least two of these files. One can display the contents with the following command issued from Linux (operating system) command shell:

```
keytool -list -keystore cacerts

Use a password of "changeit"
```

2.5.2 CertPath API

The Certification Path API is a set of functions for building and validating certification paths or chains. This is done implicitly in protocols like SSL/TLS (see the description in the corresponding section) and JAR file signature verification, but can now be done explicitly in applications with this support.

As mentioned in the section on digital certificates, a CA can sign a certificate with its private key, and if the recipient holds the CA certificate that has the public key needed for signature verification, it can verify the validity of the signed certificate. In this case, the chain of certificates is of length two, the anchor of trust (the CA certificate) and the signed certificate. A self-signed certificate is of length one, the anchor of trust is the signed certificate itself.

Chains can be of arbitrary length, so in a chain of three, a CA anchor of trust certificate can sign an intermediate certificate; the owner of this certificate can use its private key to sign another certificate. The CertPath API can be used to walk the chain of certificates to verify validity, as well as to construct these chains of trust.

Certificates have expiration dates, but can be compromised before they expire, so Certificate Revocation Lists (CRL) must be checked to really ensure the integrity of a signed certificate. These lists are available on the CA Web sites, and can also be programmatically manipulated with the CertPath API.

3 SECURITY PROTOCOLS

In this section we review the most important security protocols that are essential in OpenIoT. We start from the lowest level and follow by higher levels.

3.1 IEEE802.15.4

IEEE802.15.4 is a standard that specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs). IEEE 802.15.4 nodes can operate in either secure mode or non-secure mode. Two security modes are defined in the specification in order to achieve different security objectives: Access Control List (ACL) and Secure mode. The MAC sublayer offers facilities that can be harnessed by upper layers to achieve the desired level of security. Higher-layer processes may specify keys to perform symmetric cryptography to protect the payload and restrict it to a group of devices or just a point-to-point link; these groups of devices can be specified in access control lists. Furthermore, MAC computes *freshness checks* between successive receptions to ensure that presumably old frames, or data that is no longer considered valid, does not transcend to higher layers.

In addition to this secure mode, there is another, insecure MAC mode, which allows access control lists merely as a means to decide on the acceptance of frames according to their (presumed) source.

3.2 IPsec

Internet Protocol Security (IPsec) [1] is a technology protocol suite for securing Internet Protocol (IP) communications by authenticating and/or encrypting each IP packet of a communication session. IPsec also includes protocols for establishing mutual authentication between agents at the beginning of the session and negotiation of cryptographic keys to be used during the session.

IPsec is an end-to-end security scheme operating in the Internet Layer of the Internet Protocol Suite. It can be used for protecting data flows between a pair of hosts (*host-to-host*), between a pair of security gateways (*network-to-network*), or between a security gateway and a host (*network-to-host*).

Some other Internet security systems in widespread use, such as Secure Sockets Layer (SSL), Transport Layer Security (TLS) and Secure Shell (SSH), operate in the upper layers of the TCP/IP model. In the past, the use of TLS/SSL had to be designed into an application to protect application protocols. In contrast, since day one, applications did not need to be specifically designed to use IPsec. Hence, IPsec protects any application traffic across an IP network.

The IPsec suite is an open standard and it uses the following protocols to perform various functions:

- Authentication Headers (AH) provide connectionless integrity and data origin authentication for IP datagrams and provides protection against replay attacks.

- Encapsulating Security Payloads (ESP) provide confidentiality, data-origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic-flow confidentiality.
- Security Associations (SA) provide the bundle of algorithms and data that provide the parameters necessary to operate the AH and/or ESP operations. The Internet Security Association and Key Management Protocol (ISAKMP) provides a framework for authentication and key exchange, with actual authenticated keying material provided either by manual configuration with pre-shared keys, Internet Key Exchange (IKE and IKEv2), Kerberized Internet Negotiation of Keys (KINK), or IPSECKEY DNS records.

Cryptographic algorithms defined for use with IPsec include:

- HMAC-SHA1 for integrity protection and authenticity.
- TripleDES-CBC for confidentiality
- AES-CBC for confidentiality.

3.3 TSL

Transport Layer Security (TSL) [2] and its predecessor, **Secure Sockets Layer** (**SSL**), are cryptographic protocols that provide communication security over the Internet. They use public-key cryptography for authentication of key exchange, private-key encryption for confidentiality and message authentication codes for message integrity. TSL operates in the Application Layer of the IP Protocol Suite. The TLS protocol allows client-server applications to communicate across a network in a way designed to prevent eavesdropping and tampering.

Since protocols can operate either with or without TLS (or SSL), it is necessary for the client to indicate to the server whether it wants to set up a TLS connection or not. There are two main ways of achieving this; one option is to use a different port number for TLS connections (for example port 443 for HTTPS). The other is to use the regular port number and have the client request that the server switch the connection to TLS using a protocol specific mechanism (for example STARTTLS for mail and news protocols).

Once the client and server have decided to use TLS, they negotiate a stateful connection by using a handshaking procedure. During this handshake, the client and server agree on various parameters used to establish the connection's security:

1. The client sends the server the client's SSL version number, cipher settings, session-specific data, and other information that the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, session-specific data, and other information that the client needs to communicate with the server over SSL. The server also sends its own certificate, and if the client is requesting a server resource that requires client authentication, the server requests the client's certificate.
3. The client uses the information sent by the server to authenticate the server. If the server cannot be authenticated, the user is warned of the

problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client proceeds to the next step.

4. Using all data generated in the handshake thus far, the client (with the cooperation of the server, depending on the cipher in use) creates the pre-master secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in step 2), and then sends the encrypted pre-master secret to the server.
5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case, the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.
6. If the server has requested client authentication, the server attempts to authenticate the client. If the client cannot be authenticated, the session ends. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret, and then performs a series of steps (which the client also performs, starting from the same pre-master secret) to generate the master secret.
7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity (that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection).
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.

The SSL handshake is now complete and the session begins. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

This is the normal operation condition of the secure channel. At any time, due to internal or external stimulus (either automation or user intervention), either side may renegotiate the connection, in which case, the process repeats itself.

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the key material until the connection closes. If any one of the above steps fails, the TLS handshake fails and the connection is not created.

In step 3, the client must check a chain of "signatures" from a "root of trust" built into, or added to, the client. The client must *also* check that none of these have been revoked; this is not often implemented correctly, but is a requirement of any public-key authentication system. If the particular signer beginning this server's chain is trusted, and all signatures in the chain remain trusted, then the Certificate (thus the server) is trusted.

3.4 HTTPS

Hypertext Transfer Protocol Secure (**HTTPS**) [3] is a communications protocol for secure communication over a computer network, with especially wide deployment on the Internet. Technically, it is not a protocol in and of itself but rather it is the result of simply layering the Hypertext Transfer Protocol (HTTP) on top of the SSL/TLS protocol, thus adding the security capabilities of SSL/TLS to standard HTTP communications.

In its popular deployment on the internet, **HTTPS** provides authentication of the web site and associated web server that one is communicating with, which protects against man-in-the-middle attacks. Additionally, it provides bidirectional encryption of communications between a client and server, which protects against eavesdropping and tampering with and/or forging the contents of the communication. In practice, this provides a reasonable guarantee that one is communicating with precisely the web site that one intended to communicate with (as opposed to an imposter), as well as ensuring that the contents of communications between the user and site cannot be read or forged by any third party.

Historically, HTTPS connections were primarily used for payment transactions on the World Wide Web, e-mail and for sensitive transactions in corporate information systems. In the late 2000s and early 2010s, HTTPS began to see widespread use for protecting page authenticity on all types of websites, securing accounts and keeping user communications, identity and web browsing private.

A site must be completely hosted over HTTPS, without having some of its contents loaded over HTTP, or the user will be vulnerable to some attacks and surveillance. For example, having scripts etc. loaded insecurely on an HTTPS page makes the user vulnerable to attacks. Also having only a certain page that contains sensitive information (such as a log-in page) of a website loaded over HTTPS, while having the rest of the website loaded over plain HTTP will expose the user to attacks. On a site that has sensitive information somewhere on it, every time that site is accessed with HTTP instead of HTTPS, the user and the session will get exposed. Similarly, cookies on a site served through HTTPS have to have the secure attribute enabled.

HTTPS is a URI scheme which has identical syntax to the standard HTTP scheme, aside from its scheme token. However, HTTPS signals the browser to use an added encryption layer of SSL/TLS to protect the traffic. SSL is especially suited for HTTP since it can provide some protection even if only one side of the communication is authenticated. This is the case with HTTP transactions over the Internet, where typically only the server is authenticated (by the client examining the server's certificate).

HTTPS creates a secure channel over an insecure network. This ensures reasonable protection from eavesdroppers and man-in-the-middle attacks, provided that adequate cipher suites are used and that the server certificate is verified and trusted.

Because HTTPS piggybacks HTTP entirely on top of TLS, the entirety of the underlying HTTP protocol can be encrypted. This includes the request URL (which particular web page was requested), query parameters, headers, and cookies (which often contain identity information about the user). However, because host (web site) addresses and port numbers are necessarily part of the underlying TCP/IP protocols, HTTPS cannot protect their disclosure. In practice this means that even on a correctly configured web server eavesdroppers can still infer the IP address and port number of the web server (sometimes even the domain name e.g. www.example.org, but not the rest of the URL) that one is communicating with as well as the amount (data transferred) and duration (length of session) of the communication, though not the content of the communication.

Web browsers know how to trust HTTPS websites based on certificate authorities that come pre-installed in their software. Certificate authorities (e.g. VeriSign/Microsoft/etc.) are in this way being trusted by web browser creators to provide valid certificates. Logically, it follows that a user should trust an HTTPS connection to a website if and only if all of the following are true:

- The user trusts that the browser software correctly implements HTTPS with correctly pre-installed certificate authorities.
- The user trusts the certificate authority to vouch only for legitimate websites.
- The website provides a valid certificate, which means it was signed by a trusted authority.
- The certificate correctly identifies the website (e.g., when the browser visits "https://example.com", the received certificate is proper for "Example Inc." and not some other entity).
- Either the intervening hops on the Internet are trustworthy, or the user trusts that the protocol's encryption layer (TLS/SSL) is sufficiently secure against eavesdroppers.

HTTPS is especially important over unencrypted networks (such as WiFi), as anyone on the same local network can "packet sniff" and discover sensitive information. Additionally, many free to use and even paid for WLAN networks do packet injection for serving their own ads on webpages or just for pranks, however this can be exploited maliciously e.g. by injecting malware and spying on users.

Another example where HTTPS is important is connections over Tor (anonymity network), as malicious Tor nodes can damage or alter the contents passing through them in an insecure fashion and inject malware into the connection. This is one reason why the Electronic Frontier Foundation and Torproject started the development of HTTPS Everywhere, which is included in the Tor Browser Bundle.

4 ACCESS CONTROL

In this section we review the access control framework pertaining to OpenIoT.

4.1 Fundamental concepts

Access control is a selective restriction of access to resource and includes the following components **authentication**, **authorization, access approval** and **accountability**.

In access control models, the entities that can perform actions in the system are called **subjects**, and the entities representing resources to which access may need to be controlled are called **objects.** Subjects and objects should both be considered as software entities, rather than as human users: any human user can only have an effect on the system via the software entities that they control.

Authentication is the process of verifying that an identity is bound to the entity that makes an assertion or claim of identity (i.e., verifying that "you are who you say you are"). Authenticators are commonly based on "so*mething you know"*, such as a password or a personal identification number (PIN). This assumes that only the owner of the account knows the password or PIN needed to access the account.

Authorization is the act of defining access rights for subjects. An authorization policy specifies the operations that subjects are allowed to execute on the system.

Access approval is the function that actually grants or rejects access during operations. During access approval the system compares the formal representation of the authorization policy with the access request to determine whether the request shall be granted or rejected.

Accountability uses such system components as *audit trails* (records) and *logs* to associate a subject with its actions. The information recorded should be sufficient to map the subject to a controlling user. Audit trails and logs are important for detecting security violations and re-creating security incidents

Two most widely recognized access control models are **Discretionary Access Control (DAC)** and **Mandatory Access Control (MAC)***.*

Discretionary Access Control (DAC) is a policy determined by the owner of an object. The owner decides who is allowed to access the object and what privileges they have. Two important concepts in DAC are:

- File and data ownership: Every object in the system has an *owner*. In most DAC systems, each object's initial owner is the subject that caused it to be created. The access policy for an object is determined by its owner.
- Access rights and permissions: These are the controls that an owner can assign to other subjects for specific resources.

Mandatory Access Control (MAC) *refers to allowing access to a resource if and only if rules exist that allow a given user to access the resource*. It is difficult to manage but its use is usually justified when used to protect highly sensitive information. Examples include certain government and military information.

Management of MAC is often implemented by using **sensitivity labels**. In such a system subjects and objects must have labels assigned to them. A subject's sensitivity label specifies its level of trust. An object's sensitivity label specifies the level of trust required for access. *In order to access a given object, the subject must have a sensitivity level equal to or higher than the requested object.*

Two methods are commonly used for implementing mandatory access control using sensitivity labels:

- **Rule-based access control**: This type of control further defines specific conditions for access to a requested object. A Mandatory Access Control system implements a simple form of rule-based access control to determine whether access should be granted or denied by matching:
    1. An object's sensitivity label
    2. A subject's sensitivity label
- **Lattice-based access control**: A lattice is used to define the levels of security that an object may have and that a subject may have access to. The subject is only allowed to access an object if the security level of the subject is greater than or equal to that of the object. A lattice model is a mathematical structure that defines greatest lower-bound (meet) and least upper-bound (join) values for a pair of elements, such as a subject and an object. For example, if two subjects *A* and *B* need access to an object, the security level is defined as the meet of the levels of *A* and *B*. In another example, if two objects *X* and *Y* are combined, they form another object *Z*, which is assigned the security level formed by the join of the levels of *X* and *Y*.

Access Control Matrix or **Access Matrix** is an abstract, formal security of protection state in computer systems, that characterize the rights of each subject with respect to every object in the system. More formally, access matrix is defined as a set of objects *O*, that is the set of entities that needs to be protected (e.g., processes, files, memory pages) and a set of subjects *S*, that consists of all active entities (e.g., users, processes). Further there exists a set of rights *R* of the form *r(s, o)*, where *s* in *S*, *o* in *O* and *r(s, o)* in *R*, where a right specifies the kind of access a subject is allowed to process object. Consider the following example of a matrix where there exist two subjects (*Role1* and *Role2*) the following objects: *asset1*, *asset2*, file and a device. Table 1 presents an example access matrix.

Table 1: An example access matrix

	asset 1	*asset 2*	*file*	*device*
Role1	*read, write, execute, own*	*execute*	*read*	*write*
Role2	*read*	*read, write, execute, own*		

In general, all subjects are objects but the inverse is not true. Thus, the cell for row *s* and column *o*, *[s, o]*, denotes the set of rights of subject *s* to perform an operation on object *o* (e.g., *read in [s, o]*). Thus, all users in the access matrix are represented by their corresponding subjects. The access matrix is a dynamic entity and its individual cells can be modified by subjects. For example, if subject *s* is the owner of object *o* then *s* can modify the content of cells corresponding to *o*. In such a case the owner of the object has complete discretion regarding the access to the owned object by other subjects. Such an access control model is called discretionary. The access matrix is usually sparse and is stored in a system using access control lists, capabilities, relations or another data structure suitable for efficient sparse matrix storage.

An **access control list** (**ACL**), with respect to a computer system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. Each entry in a typical ACL specifies a subject and an operation. For instance, if a *file* has an ACL that contains (*Alice*, *delete*), this would give *Alice* permission to delete the *file*.

4.2 Lattice-Based Access Control Models

In this section we present foundations of lattice-based access control models given the importance of lattice-based access control systems.

A security of computer system has the following three interdependent objectives:

- **Confidentiality** (or secrecy) related to disclosure of information, i.e., preventing users from learning about data of other users.
- **Integrity**, related to modification of information, i.e., preventing a user from changing data of other users.
- **Availability** related to denial of access to information, i.e., ensuring that requested data is delivered on time.

Lattice-based access control models were developed to deal with information flow in computer system. Although developed for the defence sector they can be used in most cases where information flow is critical. Therefore they are a key component of computer security.

4.2.1 Information flow policy

Information flow policies are concerned with the flow of information from one security class to another. In particular in a computer system this information flaws from one object to another, where object is defined as a container of information (e.g., files and directories in an operating system).

Information flow is controlled by assigning every object a security class also called a security label. If information flows from object *x* to object *y,* it implies information flow from the security class of *x* to the security class of *y*. Thus, an information flow from one class to another concerns the corresponding objects.

An information flow policy can be formally defined as follows:

- *SC* is the set of security classes
- *A ->B* is a binary relation on *SC* called "flows", where *A*, *B* is in *SC*. Thus, the information flows from security class *A* to security class *B*.
- *A + B = C* is a binary class combination or join operation, where *C* is in *SC*. Thus, the join operation specifies how to label information obtained by combining information from two security classes *A* and *B*, where *C* is the resulting class.
- '>=' is the dominance operator, where *A >= B* (*A* dominates *B*) if and only if *B->A*. The strict dominates relation > is defined by *A > B* if *A >=B* and *A <> B*. Thus, if *A > B* then *B->A.*
  Then the definition of *A + B* is just the maximum with respect to *A* and *B*. Thus, if information from two security classes is combined the label of the higher of the two is used for the result.

As an example consider High-low policy defined as follows: *SC={H, L}*, the flow relations is *H->H, L->L, L->H* and the join is defined as follows: *H+H=H*, *L+H=H*, *L+L=L*.

As another example of security classes consider *SC={TS: top secret, S: secret, C: confidential, U: unclassified}* and the total (linear) ordering of the security classes as follows: *U->S->C->TS* meaning *TS >S>C>U* and *A+B* is the maximum with respect to the dominance relation.

4.3 Access control models-based on information flow

Recall, that a user is defined as a human being assigned a unique user *id* in the system. A subject is a process in the system (a program in execution), where each subject is associated with a single user. In general a user can have several subjects concurrently running in the system. Thus, every time a user logs into the system it does so as a particular subject. (Note that access control models assume that identification and authentication of users takes place in a secure and correct manner. Different subjects associated with the same user can obtain different sets of access rights. For example, top secret user Bob logs in at the secret level. Then Bob can have subjects running at different levels dominated by the top secret class.

The discretionary access control model is not adequate for enforcing information flow policies because they provide no constraints on copying information from one object to another. For an illustration of this property consider the following example.

EXAMPLE 1. Suppose that *Tom*, *Dick* and *Harry* are users and *Tom* has a confidential file *Private* that he wants *Dick* to read but does not want *Harry* to read. *Tom* can authorize *Dick* to read the file by entering *read(Dick, Private*) in the access matrix. *Dick* can easily subvert *Tom*'s intention by creating a new file called *Copy-of-Private* and copying the contents of *Private* into it. As the creator of *Copy-of-Private*, *Dick* has the authority to grant read access for it to any user including *Harry*. Thus, *Dick* can enter r*ead (Harry,Copy-of-Private)* in the access matrix. Then *Harry* can read *Private*.

4.3.1 Mandatory access policy

The key idea of the mandatory access control model that we present in this section *is to augment discretionary access controls with mandatory access controls to enforce information flow policies.*

Thus, we take a two step approach to access control. First a discretionary access matrix *D*, whose contents can be modified by subjects is used, where authorization in *D* is not sufficient for an operation to be carried out. Second, the operation must be authorized by the mandatory access control policy over which users have no control.

4.3.2 Confidentiality

Mandatory access control policy is expressed in terms of security labels attached to subjects (**security classification**) and objects (**security clearance**). Thus, a user labelled *secrete* can run the same program such as text editor, as a subject labelled *secret* or as a subject labelled *unclassified*. Even though both subjects run the same program on behalf of the same user they both obtain different privileges due to their security labels. The assumption called tranquillity says that the security labels on subjects and objects cannot be changed.

Let *L* be the security label (confidentiality) of a given subject or object. Then mandatory access rules can be expressed as follows:

- Subject *s* can read an object *o* only if *L(s) >= L(o)*, meaning *L(o) -> L(s)*, i.e., *o* flows to *s.*
- Subject *s* can write object *o* only if *L(s) <= L(o)*, meaning *L(s)->L(o)*, i.e., s flows to o.

The mandatory checks are only applied if the checks of the discretionary matrix have been satisfied. If the matrix does not authorize the operation then we do not check the mandatory controls.

These security requirements apply to humans and programs equally. The write property is not applied to humans but to programs. Human users are trusted not to leak information. A secret user can write unclassified document because we assume that he will put only unclassified information in it.

Programs are not trusted because they can have embedded Trojan Horses. The write property prohibits a program running at a secrete level from writing to unclassified objects even if it is permitted to do so by discretionary access control. A user labelled secret can write to an unclassified object must log as an unclassified subject.

Now consider how the presented properties impact Example 1. Clearly the *read* property will prevent *Harry*'s subjects from being able to directly read the file *Private*. The presented *write* property will ensure that *Harry*'s subjects cannot surreptitiously read *Copy-of-Private* because it will either be labelled secret or will not contain any information from Private.

4.3.3 Integrity

The presented model can be extended to handle integrity. The concept of integrity says that low-integrity information should not be allowed to flow to high-integrity objects. Let W denote the integrity label of a subject or object. A particular mandatory integrity rules can be expressed as follows:

- Subject *s* can read an object *o* only if *W(s) <= W(o)*, meaning *W(s) -> W(o)*, i.e., *s* flows to *o.*
- Subject *s* can write an object *o* only if *W(s) >= L(o)*, meaning *W(o) -> W(s)*, i.e., *o* flows to *s.*

4.3.4 Combined

It is often suggested that the confidentiality and integrity models could be combined in situations where both confidentiality and integrity are of concern. If a single label were used for both confidentiality and integrity then a model would impose conflicting constraints. Therefore, a model with independent confidentiality and integrity labels is more useful in practice. In such a model each security class consists of two labels: a confidentiality label *L* and an integrity label *W* with independent controls applied to them. We assume that that in both lattices high confidentiality and high integrity are at the top.

Example combined confidentiality and integrity mandatory rules can be expressed as follows:

- Subject *s* can read an object *o* only if *L(s) => L(o)* and W(s) <= W(o)
- Subject *s* can write an object *o* only if *L(s) <= L(o)* and W(s) >= W(o)

This popular combined model has been implemented in several operating system, database and network products specifically built to meet requirements of the military sector. Thus this model amount to the simultaneous application of two lattices, with information flow, occurring in opposite directions (going upward in the confidentiality lattice and downward in the integrity lattice).

5 THE SECURITY ARCHITECTURE IN OPENIOT

In this section we present the proposed security architecture in OpenIoT that accommodates the specifics/requirements of the OpenIoT platform.

5.1 Overview of the architecture

Clearly, the OpenIoT platform consist of several cooperating distributed standalone applications (e.g., SDUM, X-GSN, LSM, etc.) that require individual security (authentication and authorization) services because of differing subjects and objects that they deal with. Therefore, we propose a central authorization and authentication server that provides authentication and authorization services for all relevant OpenIoT applications running on behalf of different subjects.

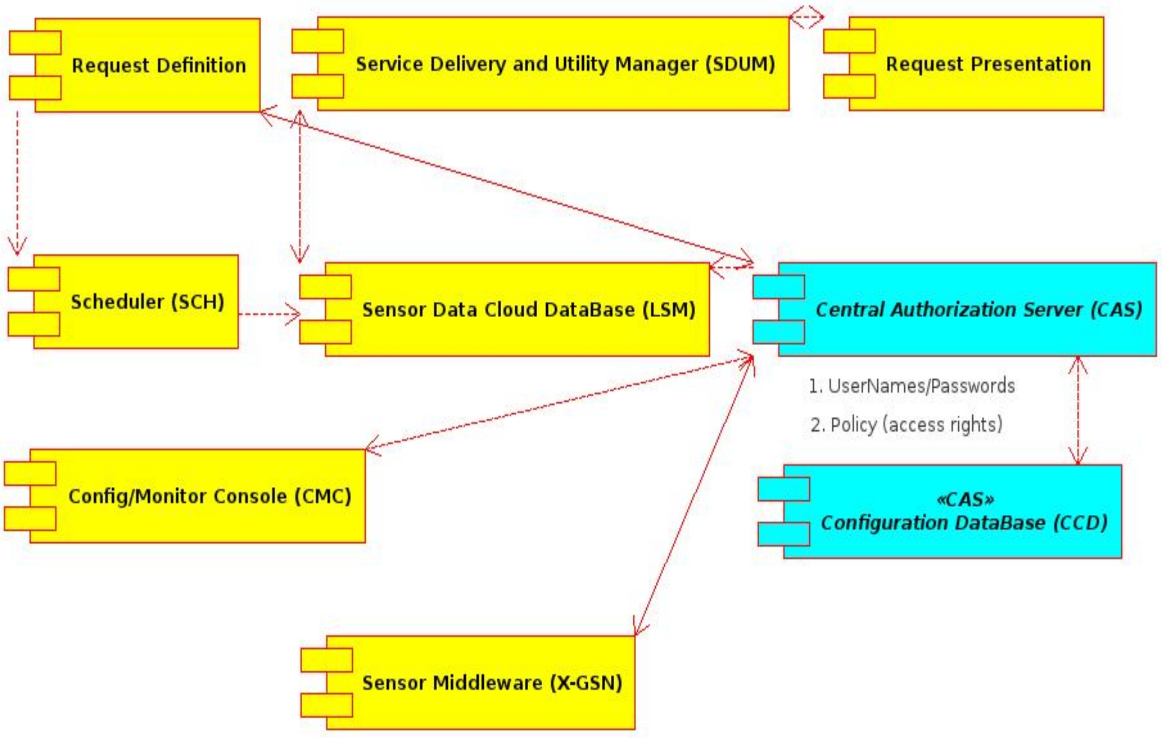

Request Definition
Service Delivery and Utility Manager (SDUM)
Request Presentation
Scheduler (SCH)
Sensor Data Cloud DataBase (LSM)
Central Authorization Server (CAS)
1. UserNames/Passwords
2. Policy (access rights)
Config/Monitor Console (CMC)
«CAS» Configuration DataBase (CCD)
Sensor Middleware (X-GSN)

Figure 1. Security Architecture in OpenIoT.

The main feature of this architecture is that user credentials (username/password) are only checked and maintained by Central Authorization Server (CAS) and it authorizes applications running on behalf a user by granting them an access token with a given time to live. This prevents any circulation of the credentials throughout OpenIoT components. Furthermore, the architecture has to be open to external security schemes that are more flexible or secure (or both) than the default security model. Note that the security and performance are usually orthogonal features and strong security means lower performance and vice-versa. Therefore, our ultimate objective is to design a scheme where security can be traded for performance.

Figure 1 presents an overview of the security architecture in the OpenIoT architecture.

In particular, CAS architecture considers the following roles:

- Resource owner: e.g., an owner of sensor data in LSM that grants access to the data for Service Delivery and Utility Manager (SDUM).
- Resource server: e.g., Sensor Data Cloud Database (LSM).
- Client: e.g., SDUM querying LSM on behalf of a resource owner. In general, Client consists of the distributed set of OpenIoT components, that use the corresponding token to get access to the data of the corresponding Resource owner in LSM.
- Authorization server: CAS issuing access tokens to a client after successfully authenticating the resource owner and obtaining authorization.

The Clients that directly authenticate with CAS are:

- Request Definition: where a user defines a request (query) and upon authentication the other clients (e.g., SCH, LSM, Request Presentation) get a corresponding token to accomplish their tasks.
- CMC: where the administrator is authenticated to accomplish his tasks.
- X-GSN: where sensor data providers are authenticated to stream the corresponding data to LSM.

The details of the CAS architecture are presented in Section 6.

5.2 Access matrix

The utilized access matrix consists of the following subjects and objects. The roles are as follows:

- Administrator: This role gains access to the entire OpenIoT platform. All the actions of the different modules will be available to administrators. Administrators will also have access to all available GUIs.
- User: The user role is the most common and the most used role of the access control model. After creating an account, the user will gain access to a specific list of possible actions. The user will have access to the request definition and the request presentation GUIs.

The considered objects are as follows:

1. Physical sensor (PS).
2. Virtual sensors (VS).
3. Internet Connected Object (ICO).
4. Services related to the components (e.g., SDUM, SCH and LSM operations).

A fragment of the access matrix as a relation is presented in Table 2.

Table 2: A fragment of the access matrix in OpenIoT

Role	Relation r(o)
Administrator	setup(VS), setup(PS), setup(LSM), setup(SCH), setup(SDUM), write(LSM), read(LSM), ..., etc.
User	read(VS), read(PS), read(GUI), ..., etc.

5.3 Mandatory access rules

We consider the following security classes

1. TS: top secret
2. C: confidential.

Security classes assignment to roles is presented in Table 3.

Table 3: Security classes and roles in OpenIoT

Role	Security class (confidentiality label)
Administrator	TS: top secret
User	C: confidential

We adapt the confidentiality rules as trust rules from Section 4. 3.

5.4 Security protocol stack

The security protocol stack in OpenIoT is presented in Table 4, where

The layers refer to corresponding communication between the following components: Layer 1: sensors and GSN, Layer 2: GSN and LSM and Layer 3: other OpenIoT components (e.g., LSM and SDUM).

Table 4: The security protocol stack in OpenIoT

	Layer/Link	Security/privacy solution
1.	Sensor->GSN	Ipsec (wired networks), IEEE802.15.4 (wireless networks)
2.	GSN->LSM	TSL, HTTPS
3.	Application (OpenIoT)	TSL, HTTPS

Clearly, the OpenIoT platform relies entirely on the TSL/HTTPS protocol to ensure secure (encrypted) messaging, while IEEE802.15.4, Ipsec guarantees secure sensor data.

6 AUTHORIZATION FRAMEWORK

In this section we present our authorization framework that is based on the OAuth [4,5] framework and provides authorization to OpenIoT modules on behalf of a user.

We chose OAuth as our authorization framework for the following reasons:

- From the point of view of OpenIoT the most important fact about OAuth is that it describes a method for providing authorization in a distributed environment, where distributed client applications get access to owner's resources using time-stamped tokens to avoid transmitting credentials (username, password) to the client applications.
- OAuth is an open standard for authorization, i.e., publicly available, and developed, approved and maintained via a collaborative and consensus driven process.
- OAuth is more like a framework (not a defined protocol), which leaves a lot of implementation freedom that we need because of non-standard requirements in OpenIoT. For example, for some OpenIoT applications involving mobile phones as sensors it is important to have time stamped and location-restricted access tokens.
- OAuth is the only framework in its genre and is widely used for similar applications.

Furthermore, OAuth is a result of standardization and combined wisdom of many well-established industry protocols. It is similar to other protocols currently in use (Google AuthSub, AOL OpenAuth, Yahoo BBAuth, Flickr API, Amazon Web Services API, etc). Each protocol provides a proprietary method for exchanging user credentials for an access token or ticker. OAuth was created by carefully studying each of these protocols and extracting the best practices and commonality that allow new implementations as well as a smooth transition for existing services to support Oauth.

An area, where OAuth is more evolved than some of the other protocols and services are its direct handling of non-website services. OAuth has built in support for desktop applications, mobile devices, set-top boxes, and of course websites. Many of the protocols today use a shared secret hardcoded into software but such an approach may pose an issue when the service trying to access private data is open source.

6.1 Introduction

In the traditional client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party.

This creates several problems and limitations:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
- Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token a string denoting a specific scope, lifetime, and other access attributes. An authorization server with the approval of the resource owner issues access tokens to third-party clients. The client uses the access token to access the protected resources hosted by the resource server.

For example, an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo-sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo-sharing service (authorization server), which issues the printing service delegation-specific credentials (access token).

6.2 Roles

OAuth defines four roles:

1. **Resource owner**: an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
2. **Resource server**: the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
3. **Client**: an application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
4. **Authorization server**: the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

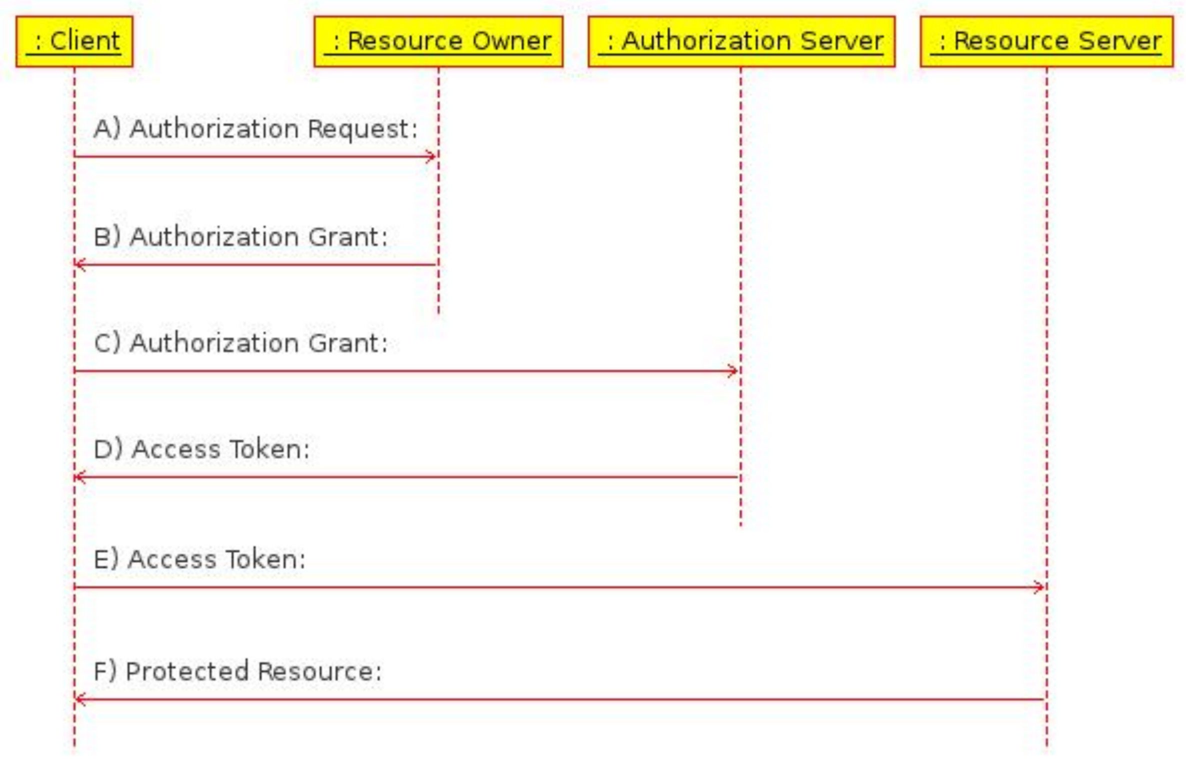


Figure 2. Abstract Protocol Flow.

6.3 OAuth Protocol Flow

The abstract OAuth flow illustrated in Illustration 2 describes the interaction between the four roles and includes the following steps:

A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.

B) The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

D) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

F) The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorization grant from the resource owner (depicted in steps (A) and (B)) is to use the authorization server as an intermediary as in Figure 3.

6.3.1 Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials as well as an extensibility mechanism for defining additional types.

6.3.2 Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server, which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner.

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

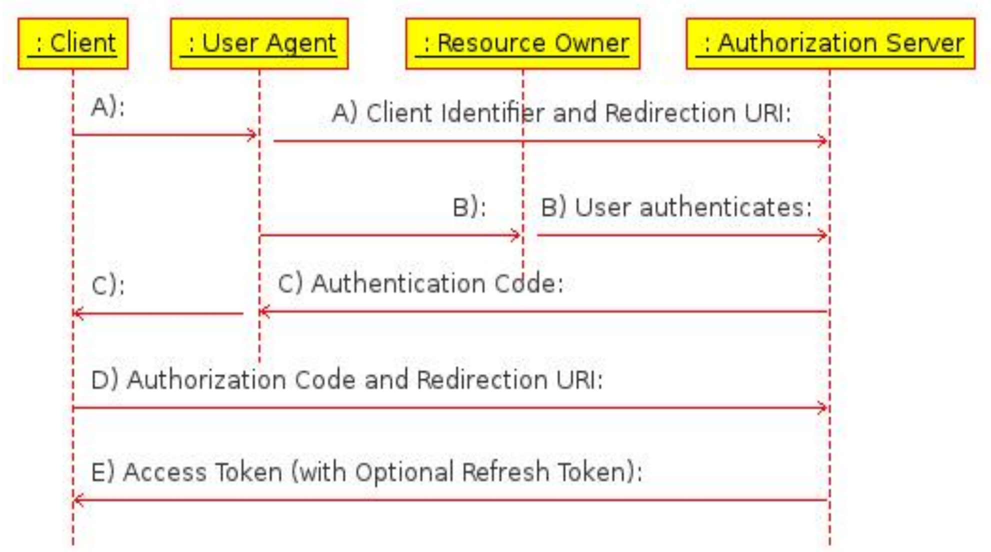

: Client
: User Agent
: Resource Owner
: Authorization Server
A):
A) Client Identifier and Redirection URI:
B):
B) User authenticates:
C):
C) Authentication Code:
D) Authorization Code and Redirection URI:
E) Access Token (with Optional Refresh Token):

Figure 3. Authorization Code Flow.

The flow illustrated in Illustration 3 includes the following steps:

A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

D) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

E) The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

6.3.3 Implicit

The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly (as the result of the resource owner authorization). The grant type is implicit, as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application), since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants, especially when the authorization code grant type is available.

6.3.4 Resource Owner Password Credentials

The resource owner password credentials (i.e., username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g., the client is part of the device operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

6.3.5 Client Credentials

The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorization previously arranged with the authorization server.

6.4 Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g., username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g., cryptographic properties) based on the resource server security requirements.

6.5 Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (D) in Illustration 2).

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

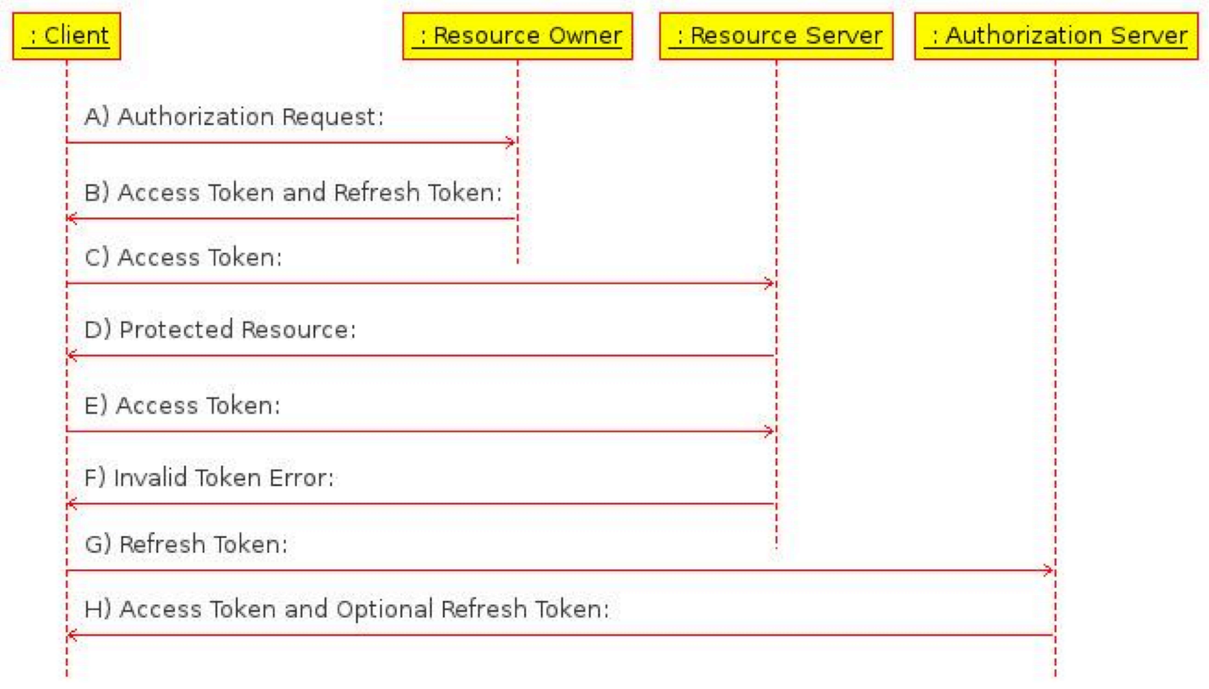


Figure 4. Refreshing an Expired Access Token.

The flow illustrated in Illustration 4 includes the following steps:

A) The client requests an access token by authenticating with the authorization server and presenting an authorization grant.

B) The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and a refresh token.

C) The client makes a protected resource request to the resource server by presenting the access token.

D) The resource server validates the access token, and if valid, serves the request.

E) Steps C) and D) repeat until the access token expires. If the client knows the access token expired, it skips to step G); otherwise, it makes another protected resource request.

F) Since the access token is invalid, the resource server returns an invalid token error.

G) The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorization server policies.

H) The authorization server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).

6.6 TLS Version

Whenever Transport Layer Security (TLS) is used by this specification, the appropriate version (or versions) of TLS will vary over time, based on the widespread deployment and known security vulnerabilities.

Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

6.7 HTTP Redirections

This specification makes extensive use of HTTP redirections, in which the client or the authorization server directs the resource owner's user-agent to another destination. While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

6.8 Interoperability

OAuth 2.0 provides a rich authorization framework with well-defined security properties. However, as a rich and highly extensible framework with many optional components, on its own, this specification is likely to produce a wide range of non-interoperable implementations.

In addition, this specification leaves a few required components partially or fully undefined (e.g., client registration, authorization server capabilities, endpoint discovery). Without these components, clients must be manually and specifically configured against a specific authorization server and resource server in order to interoperate.

This framework was designed with the clear expectation that future work will define prescriptive profiles and extensions necessary to achieve full web-scale interoperability.

6.9 Security considerations

In this section we consider the security of the OAuth framework.

6.9.1 Beyond Basic

HTTP defines an authentication scheme called ‘Basic’ which is commonly used by many sites and APIs. The way ‘Basic’ works is by sending the username and password in plain text with each request. When not used over HTTPS, ‘Basic’ suffers from significant security risks. First, it transmits a password unencrypted that allows anyone listening to capture and reuse those credentials. Second, there is nothing linking the credentials to the request which means once compromised, they can be used with any request without limitations. Third, ‘Basic’ does not provide a placeholder for delegation credentials and only supports a single username-password pair. Delegation requires being able to send both the credentials of the caller (client) and those of the party delegating its access (resource owner). The OAuth architecture explicitly addresses these three limitations.

The OAuth signature method was primarily designed for insecure communications mainly non-HTTPS. HTTPS is the recommended solution to prevent a man-in-the-middle attack (MITM), eavesdropping, and other security risks. However, HTTPS is often not available. When OAuth is used over HTTPS, it offers a simple method for a more efficient implementation called PLAINTEXT which offloads most of the security requirements to the HTTPS layer. It is important to understand that PLAINTEXT should not be used over an insecure channel. Therefore we focus on the methods designed to work over an insecure channel: HMAC-SHA1 and RSA-SHA1.

6.9.2 Credentials

In everyday web transactions, the most common credential used is the username-password combination. OAuth's primary goal is to allow delegated access to private resources. This is done using two sets of credentials: the client identifies itself using its client identifier and client secret, while the resource owner is identified by an access token and token secret. Each set can be thought of as a username-password pair (one for the application and one for the end-user).

However, while the client credentials work much like a username and password, the user is represented by an access token which is different than their actual username and password. This allows the server and resource owner greater control and flexibility in granting client access. For example, the resource owner can revoke an access token without having to change passwords and break other applications. The decoupling of the resource owner's username and password from the access token is one of the most fundamental aspects of the OAuth architecture.

OAuth includes two type of tokens: temporary credentials and access token. Each type has a very specific role in the OAuth flow. While mostly an artefact of how the OAuth specification evolved, the two-token design offers some usability and security features which made it worthwhile to stay in the specification.

OAuth operates on two channels: a front-channel which is used to engage the resource owner and request authorization, and a back-channel used by the client to directly interact with the server.

By limiting the access token to the back-channel, the token itself remains concealed from the resource owner and its browser. This allows the access token to carry special meanings and to have a larger size than the front-channel temporary credentials which are exposed to the resource owner when requesting authorization, and in some cases needs to be manually entered (mobile device or set-top box).

The request signing workflow treats all tokens the same and the methods are identical. The two tokens are specific to the authorization workflow, not the signature workflow which uses the tokens equally. This does not mean the two token types are interchangeable, just that they provide the same security function when signing requests.

6.9.3 Signature and Hash

OAuth uses digital signatures instead of sending the full credentials (specifically, passwords) with each request. Similar to the way people sign documents to indicate their agreement with a specific text, digital signatures allow the recipient to verify that the content of the request has not changed in transit. To do that, the sender uses a mathematical algorithm to calculate the signature of the request and includes it with the request.

In turn, the recipient performs the same workflow to calculate the signature of the request and compares it to the signature value provided. If the two match, the recipient can be confident that the request has not been modified in transit. The confidence level depends on the properties of the signature algorithm used (some are stronger than others). This mechanism requires both sides to use the same signature algorithm and apply it in the same manner.

A common way to sign digital content is using a hash algorithm. In general, hashing is the process of taking data (of any size) and condensing it to a much smaller value (digest) in a fully reproducible (one-way) manner. This means that using the same hash algorithm on the same data will always produce the same smaller value. Unlike compression which aims to preserve much of the original uncompressed data, hashing usually does not allow going from the smaller value back to the original.

By itself, hashing does not verify the identity of the sender, only data integrity. In order to allow the recipient to verify that the request came from the claimed sender, the hash algorithm is combined with a shared secret. If both sides agree on some shared secret known only to them, they can add it to the content being hashed. This can be done by simply appending the secret to the content, or using a more sophisticated algorithm with a built-in mechanism for secrets such as HMAC. Either way, producing and verifying the signature requires access to the shared secret, which prevents attackers from being able to forge or modify requests.

The benefit of this approach compared to the HTTP ‘Basic’ authorization scheme is that the actual secret is never sent with the request. The secret is used to sign the request but it is not part of it, nor can it be extracted (when implemented correctly). Signatures are a safer way to accomplish the same functionality of sending the shared secret with the request over an unsecure channel.

6.9.4 Secrets Limitations

In OAuth, the shared secret depends on the signature method used. In the PLAINTEXT and HMAC-SHA1 methods, the shared secret is the combination of the client secret and token secret. In the RSA-SHA1 method, the client private key is used exclusively to sign requests and serves as the asymmetric shared secret. The way asymmetric key-pairs work, is that each side (the client and server) uses one key to sign the request and another key to verify the request.

The keys (private key for the client and public key for the server) must match, and only the right pair can successful sign and verify the request. The advantage of using asymmetric shared secrets is that even the server does not have access to the client’s private key which reduces the likelihood of the secret being leaked.

However, since the RSA-SHA1 method does not use the token secret (it does not use the client secret either but that is adequately replaced by the client private key), the private key is the only protection against attacks and if compromised, puts all tokens at risk. This is not the case with the other methods where one compromised token secret (or even client secret) does not allow access to other resources protected by other tokens (and their secrets).

When implementing OAuth, it is critical to understand the limitations of shared secrets, symmetric or asymmetric. The client secret (or private key) is used to verify the identity of the client by the server. In case of a web-based client such as web server, it is relatively easy to keep the client secret (or private key) confidential.

However, when the client is a desktop application, a mobile application, or any other client-side software such as browser applets (Flash, Java, Silverlight) and scripts (JavaScript), the client credentials must be included in each copy of the application. This means the client secret (or private key) must be distributed with the application, which inheritably compromises them.

This does not prevent using OAuth within such application, but it does limit the amount of trust the server can have in such public secrets. Since the secrets cannot be trusted, the server must treat such application as unknown entities and use the client identity only for activities that do not require any level of trust, such as collecting statistics about applications. Some servers may opt to ban such application or offer different protocols or extensions. However, at this point there is no known solution to this limitation.

It is important to note, that even though the client credentials are leaked in such application, the resource owner credentials (token and secret) are specific to each instance of the client which protects their security properties. This of course greatly depends on the client implementation and how it stores token information on the client side.

6.9.5 Timestamp and Nonce

The signature and shared secret provide some level of security but are still vulnerable to attacks. The signature protects the content of the request from changing while the shared secret ensures that requests can only be made (and signed) by an authorized client. What is missing is something to prevent requests intercepted by an unauthorized party, usually by sniffing the network, from being reused. This is known as a replay attack.

As long as the shared secrets remain protected, anyone listening in on the network will not be able to forge new requests as that will require using the shared secret. They will however, be able to make the same sign request over and over again. If the intercepted request provides access to sensitive protected data, it can be a significant security risk.

To prevent compromised requests from being used again (replayed), OAuth uses a **nonce** and **timestamp**. The term nonce means 'number used once' and is a unique and usually random string that is meant to uniquely identify each signed request. By having a unique identifier for each request, the Service Provider is able to prevent requests from being used more than once. This means the client generates a unique string for each request sent to the server, and the server keeps track of all the nonces used to prevent them from being used a second time. Since the nonce value is included in the signature, it cannot be changed by an attacker without knowing the shared secret.

Using nonces can be very costly for the server as they demand persistent storage of all nonce values received, ever. To make implementations easier, OAuth adds a timestamp value to each request which allows the server to only keep nonce values for a limited time. When a request comes in with a timestamp that is older than the retained time frame, it is rejected as the server no longer has nonces from that time period.

It is safe to assume that a request sent after the allowed time limit is a replay attack. From a security standpoint, the real nonce is the combination of the timestamp value and nonce string. Only together they provide a perpetual unique value that can never be used again by an attacker.

6.9.6 Signature Methods

OAuth defines 3 signature methods used to sign and verify requests: PLAINTEXT, HMAC-SHA1, and RSA-SHA1. PLAINTEXT is intended to work over HTTPS and in a similar fashion to how HTTP ‘Basic’ transmits the credentials unencrypted. Unlike ‘Basic’, PLAINTEXT supports delegation. The other two methods use the HMAC and RSA signature algorithm combined with the SHA1 hash method. Since these methods are too complex to explain in this guide, implementers are encouraged to read other guides specific to them, and not to write their own implementations, but instead use trusted open source solutions available for most languages.

When signing requests, it is necessary to specify which signature method has been used to allow the recipient to reproduce the signature for verification. The decision of which signature method to use depends on the security requirements of each application. Each method comes with its set of advantages and limitations.

PLAINTEXT is trivial to use and takes significantly less time to calculate, but can only be safe over HTTPS or similar secure channels. HMAC-SHA1 offers a simple and common algorithm that is available on most platforms but not on all legacy devices and uses a symmetric shared secret. RSA-SHA1 provides enhanced security using key-pairs but is more complex and requires key generation and a longer learning curve.

6.9.7 Signature Base String

As explained above, both sides must perform the signature process in an identical manner in order to produce the same result. Not only must they both use the same algorithm and share secret, but also they must sign the same content. This requires a consistent method for converting HTTP requests into a single string that is used as the signed content the Signature Base String.

7 TRUSTWORTHINESS OF SENSOR READINGS

In this section we introduce our algorithm to assessing trustworthiness of sensor readings. Spatial correlation has a meaning in the context of sensor data in a variety of monitoring applications, where a key characteristic is that nearby sensor nodes monitoring an environmental feature typically register similar values [9, 10]. This kind of data redundancy due to the spatial correlation between sensor observations is the corner-stone of the proposed algorithm for assessing trustworthiness of sensor readings. Thus, we use the spatial neighbourhood of a given sensor to compare its values with values of the neighbourhood to assess trustworthiness of the sensor.

The most related to our work is [7, 8], where a provenance and a game-theoretic approach for assessing trustworthiness in sensor networks were proposed. However, they did not consider spatio-temporal correlations between the sensor streams. In [11] a multi-stream join was proposed for mining spatio-temporal correlations between multiple streams.

7.1 Notation

We use superscript (i) to refer to the i -th stream. $A^{(i)} = \{a^{(i)}1 , a^{(i)}2 , . . . , a^{(i)}{}_{m(i)}\}$ is an alphabet in stream i . $S = \{s^{(1)}, s^{(2)}, . . . , s^{(|I|)}\}$ is a multi-stream defined as a set of input streams each of a possibly different length $n^{(i)}$ resulting from a different rate of generating symbols $s^{(i)}=[s^{(i)}1 , s^{(i)}2 , . . . , s^{(i)}n(i)]$ is the i -th stream (i -th attribute sequence). Every stream tuple (stream element) has three attributes: (I) timestamp $s^{(i)}$ t .timestamp = t , where t □ { 1, 2, . . .} ;. (II) stream identifier $s^{(i)}$ t .stream = i and (III) (relational tuple) denoted $s^{(i)}$ t .value , where $s^{(i)}$t .value □ $A^{(i)}$. For simplicity we just use $s^{(i)}$t to refer to $s^{(i)}$t .value. $X^{(k)}$t is the random variable corresponding to the value of the data point of sensor k at time t. $N^{(k)}$t be the set of neighbours (neighbourhood) of sensor k at time t. $E(N^{(k)}t)$ and $Var(N^{(k)}t)$ are the average and the variance of the data points of the neighbours at time t. T (k) = [$T^{(k)}$ ts , $T^{(k)}$ ts+1, . . . ,$T^{(k)}$ te] is the trust stream of the k-th sensor for the time window [ts , te], where ts is the start time and te is the end time. $T^{(k)}$ t is the trust score for sensor k at time t, where $T^{(k)}$t □ [0 , 1].

7.2 Problem definition

The problem of computing trustworthiness of sensors readings based on spatio-temporal correlation with neighbours can be defined as follows:

Given:

– an input collection of spatio-temporally correlated (temporary overlapping neighbouring streams) streams $S = \{s^{(1)}, s^{(2)}, . . . , s^{(|I|)}\}$, where

$s^{(i)} = [s^{(i)}{}_{t(i)s}, s^{(i)}{}_{t(i)s +1}, . . . , s^{(i)}{}_{t(i)e}]$ of possibly of different lengths.

– $x^{(i)}$ and $y^{(i)}$ are the coordinates of sensor stream $s^{(i)}$.

– t(i)s and t(i)e are the start and end timestamp of sensor stream s(i) respectively.

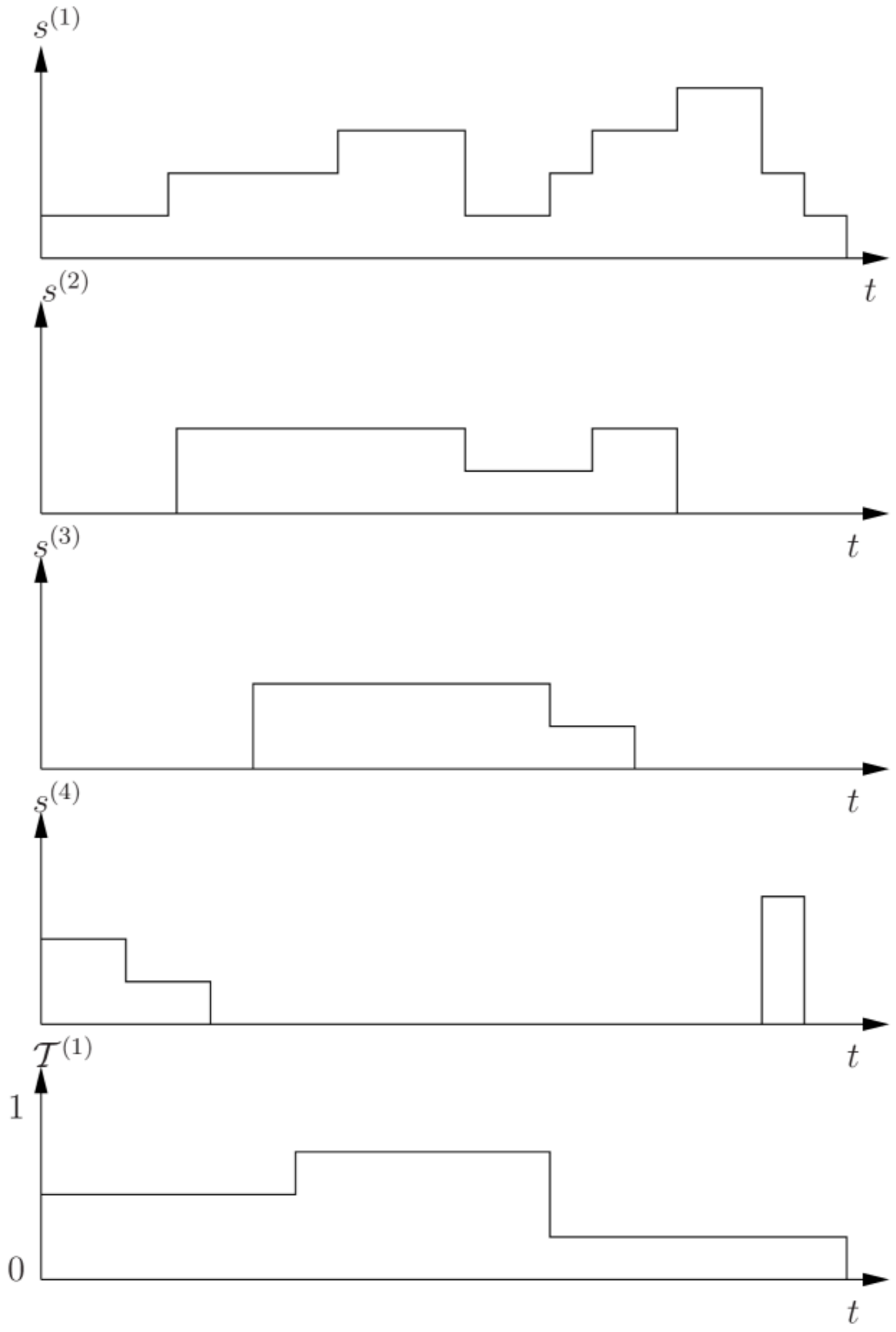
– sensor identifier k.

Task: compute the trust sequence $T^{(k)} = [T^{(k)}{}_{ts} , T^{(k)}{}_{ts+1}, . . . , T^{(k)}{}_{te}]$.

7.3 Overview of the method

Consider sensor streams $s^{(1)}$, $s^{(2)}$, $s^{(3)}$, $s^{(4)}$ in Illustration 5, and the task of assessing a corresponding trust stream for sensor $s^{(1)}$ called $T^{(1)}$ based on its spatio-temporal correlation with neighbouring streams. Thus, the general idea of our approach, to computing a trust stream for a given sensor node, is to compare its values with values of its neighbouring sensors. Then the more the streams of the neighbours are similar the higher the trust of the sensor. Note the neighbouring streams may not completely cover the time span of $s^{(1)}$ (presence of discontinuities) that makes the problem more difficult than simply correlating streams.

Thus, our method works as follows: (I) we assess the spatio-temporal correlations between the streams to form the neighbourhood for each sensor in a time period where we assume all streams are trustworthy and (II) given the neighbourhood we assess the trustworthiness of the sensors by comparing them to the centroids of the neighbourhood.



Trust of sensor stream $s^{(1)}$, $T^{(1)}$ computation using spatio-temporal correlation with neighbouring sensors $s^{(2)}$, $s^{(3)}$, $s^{(4)}$, where every sensor has different geographic coordinates

Figure 5. Trust of Sensor Stream Representations.

7.4 Algorithm

We associate a trust score with each data point that provides an indication about trustworthiness of the data point. The more trustworthy data a source provides the more trusted is the source. Thus, there is an interdependence between trust scores of data points and its sensor and vice-versa. Trust scores of data points are computed by taking into account data point values generated from sensor in a given neighbourhood of the given sensor. We use value similarity: the more data points referring to the same real-world event (neighbours) have similar values the higher the trust score of the data point. Trust score need to be continuously evaluated in the stream environment.

In particular, to express the trust score of sensor k at time t we use the Z-score as follows:

$$Z\left(X_t^{(k)}\right) = \sqrt{n}\frac{\left(X_t^{(k)} - E\left(N_t^{(k)}\right)\right)}{\left(\sqrt{Var\left(N_t^{(k)}\right)}\right)},$$

where

$$E\left(N_t^{(k)}\right) = \frac{1}{\left(N_t^{(k)}\right)}\sum_{s\in N_t^{(k)}} s.value$$

and

$$Var\left(N^{(k)}\right) = \frac{1}{\left(\left(N_t^{(k)}\right) - 1\right)}\sum_{s\in N_t^{(k)}}\left(s.value - E\left(N_t^{(k)}\right)\right)^2.$$

Then the can be expressed as the p-value

$$T_t^{(k)} = P\left(Z\, Z\left(X_t^{(k)}\right)\right).$$

Clearly, the trust values have the probability correspondence and are values in the interval [0, 1].

8 IMPLEMENTATION IN JAVA

The Java programming language and environment has many features that facilitate secure programming:

- No pointers, which means that a Java program cannot address arbitrary memory locations in the address space.
- A bytecode verifier, which operates after compilation on the .class files and checks for security issues before execution. For example, an attempt to access an array element beyond the array size will be rejected. Because buffer overflow attacks are responsible for most system breaches, this is an important security feature.
- Fine-grained control over resource access for both applets and applications. For example, applets can be restricted from reading or writing to disk space, or can be authorized to read from only a specific directory. This authorization can be based on who signed the code (see the concept of code signing) and the http address of the code source. These settings appear in a java.policy file.
- A large number of library functions for all the major cryptographic building blocks and SSL (the topic of this tutorial) and authentication and authorization (discussed in the second tutorial in this series). In addition, numerous third-party libraries are available for additional algorithms.

There are a number of programming styles and techniques available to help ensure a more secure application. Consider the following as two general examples:

- Storing/deleting passwords. If a password is stored in a Java String object, the password will stay in memory until it is either garbage collected or the process ends. If it is garbage collected, it will still exist in the free memory heap until the memory space is reused. The longer the password String stays in memory, the more vulnerable it is to snooping. Even worse, if real memory runs low, the operating system might page this password String to the disk's swap space, so it is vulnerable to disk block snooping. To minimize (but not eliminate) these exposures, you should store passwords in char arrays and zero them out after use. (Strings are immutable, so you cannot zero them out.)
- Smart serialization. When objects are serialized for storage or transmission any private fields are, by default, present in the stream. So, sensitive data is vulnerable to snooping. One can use the transient keyword to flag an attribute so it is skipped in the streaming.

The following packages are integrated into JDK:

- JCE (Java Cryptography Extension) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) (a short piece of information used to authenticate a message and to provide integrity and authenticity assurances on the message) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers.
- JSSE (Java Secure Sockets Extension) provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. JSSE provides functions for the secure passage of data between a client and a server running any application protocol, such as HTTP, Telnet, or FTP, over TCP/IP.
- JAAS (Java Authentication and Authorization Service) provides a framework and an API for the authentication and authorization of users.

We can enhance an already rich set of functions in the current Java language with third-party libraries.

9 THE PROTOTYPE

In this section we give an overview of the currently implemented prototype focused on OpenIoT architecture requirements. The documentation for how to use and configure the prototype can be found on the OpenIoT wiki[1]. Particularly CAS[2] for authentication and authorization is used. CAS is an open source multi-protocol SSO solution with a lot of flexibility in configuration. Particularly, it can integrate with several authentication methods such as Active Directory, JAAS, JDBC, LDAP, and so on. It can achieve high availability by providing support for storing client authentication state in distributed storage providers such as BerkleyDB, Ehcache, JDBC, Memcache, and so on. CAS can be configured to act as an OAuth2.0 server. Another OAuth provider and client library is Spring Security OAuth. However, we have opted for using CAS using OAuth wrapper because of its configuration flexibility and the ease of integration.

9.1 Trust-Module in the OpenIoT Architecture

The architecture of the Trust-Module that accommodates the specifics/requirements of the given OpenIoT architecture is described in this section. Thus, given the fact that sensor streams in OpenIoT are by default stored in the cloud database (LSM) for further processing, implies the following architecture of the trust module:

- Trust-Module is an independent module in OpenIoT.
- It obtains the sensor streams from LSM and outputs corresponding trust streams back to LSM.

There are the following ways of computing the trust stream given available sensor streams in LSM:

1. On-line (immediate) while storing the sensor streams to LSM. The disadvantage of this approach is a heavy overloading of the capabilities of the trust module, LSM and the communication link between them.
2. Off-line on demand: computed if the corresponding query arrives. This approach has the same disadvantage as the on-line approach, where in this case it is a blocking operation. The advantage of this approach is that LSM will not be populated with trust streams that may never be used.
3. Off-line periodically (deferred): computed periodically after the sensor streams have been stored in LSM.

[1] https://github.com/OpenIotOrg/openiot/wiki/Security

[2] http://www.jasig.org/cas/

We adopt the off-line solution combined with caching mechanism to optimize the computational and storage resources.

Figure 6 presents the view on the integration of the trust module in the OpenIoT architecture. Clearly, the follow of data is as follows:

- X-GSN provides sensor streams to LSM.
- The Trust-Module obtains the data from LSM and periodically outputs the corresponding trust streams back to LSM (stored in a separate entity that references the corresponding sensor stream).
- Trust-Module communicates with SCH and SDUM to process queries. In particular, SCH may trigger an on-demand computation of a trust stream if this is necessary for a given query (e.g., the query specifies a minimum trust threshold for sensor data), while SDUM will monitor the performance of TM and trigger periodic computation of trust streams.

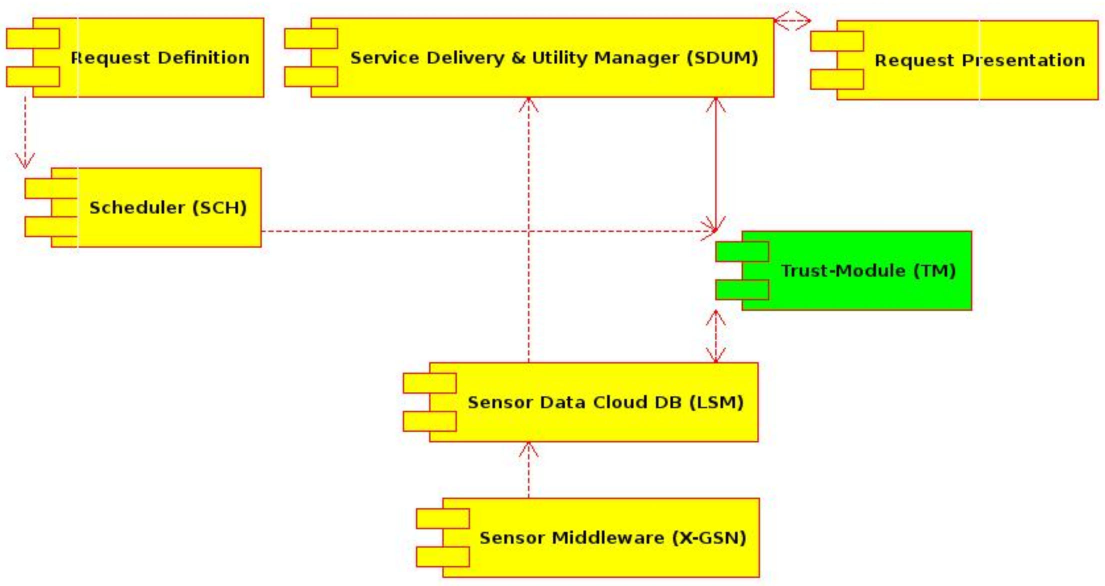

Request Definition
Service Delivery & Utility Manager (SDUM)
Request Presentation
Scheduler (SCH)
Trust-Module (TM)
Sensor Data Cloud DB (LSM)
Sensor Middleware (X-GSN)

Figure 6. Trust-Module in OpenIoT.

9.2 Trust-Module Implementation

In the prototype implementation, we have enabled OAuth2.0 server support of CAS 3.5.2 and the client authentication state data, which is stored in tickets, is configured to be placed in MySQL server. Also, authentication-using JDBC is enabled, which simply uses a MySQL table to verify username and password information.

On the client side, pac4j[3] library is used to provide support for authentication using OAuth2.0 protocol. This library targets all the protocols that support the following procedure for authentication and retrieving user profile:

1. From the client application, redirect the user to the "provider" for authentication (HTTP 302)
2. After successful authentication, redirect back the user from the "provider" to the client application (HTTP 302) and get the user credentials
3. With these credentials, get the profile of the authenticated user (direct call from the client application to the "provider").

OAuth2.0 being one of the protocols that follow the above mechanism is supported by pac4j.

Web clients, more specifically OpenIoT components, can use Apache Shiro[4] library for authentication and authorization. For enabling authentication through OAuth2.0, we use buji-pac4j[5] library, which is a web multi-protocols for Apache Shiro and supports CAS server using OAuth wrapper.

In summary, OpenIoT authentication and authorization prototype works as follows:

1. All the clients (e.g., OpenIoT components) first have to be registered in CAS server specifying their clientID, secret, and sevice URL.
2. Each client authenticates itself through OAuth2.0 providing the required credentials. After authentication the client obtains a ticket.
3. When a client wants to check the authorization of a user, it authenticates the user if necessary by redirecting the user to the CAS server. If the user is authenticated, it must have received a ticket from the CAS server. The client then contacts the CAS server providing its own ticket and clientID as well as the user's ticket to fetch the authorization information of the user.
4. If client A wants to use a service from client B, it has to forward the granted ticket of the concerning user to client B. Client B will then follow the same procedure as in the previous step for obtaining authorization information of the user.

[3] https://github.com/leleuj/pac4j

[4] http://shiro.apache.org/

[5] https://github.com/bujiio/buji-pac4j

10 CONCLUSIONS

In this document we presented the foundations of the security/privacy and trust mechanism in OpenIoT. The main feature of the security/privacy architecture is the use of the Central Authorization Server (CAS) for authorizing applications running on behalf a user by granting them an access token with a given time to live. This prevents any circulation of the credentials throughout OpenIoT components.

The main feature of the trust architecture is the use of an independent trust module that obtains sensors streams from LSM and generates corresponding trust streams that are stored in LSM.

In the implemented prototype we verified the applicability of the proposed architectures for the OpenIoT platform.

This document presents the first part of the security/privacy and trust specification (deliverable). In the second part, due in a year, we plan on refining/improving the presented architectures, leveraging their flexibility/modularity to accommodate external IoT infrastructures having differing characteristics (e.g., in term of security/privacy standards) and providing the final implementation.

11 REFERENCES

[1] RFC 4303, IP Encapsulating Security Payload (ESP).

[2] RFC 5246, The Transport Layer Security (TLS) Protocol.

[3] RFC 2818, HTTP Over TLS.

[4] RFC 5849, The OAuth 1.0 Protocol.

[5] RFC 6749, The OAuth 2.0 Authorization Framework.

[6] Ravi S. Sandhu, “Lattice-Based Access Control Models”, Journal Computer, Volume 26, Issue 11, November 1993, Page 9-19.

[7] Hyo-Sankg Lim, Yang-Sae Moon and Elisa Bertino, “Provenance-based Trustworthiness Assessment in Sensor Networks”, DMSN'10, September 13, 2010, Singapore.

[8] H.-S. Lim, G. Ghinita, E. Bertino, and M. Kantarcioglu. A game-theoretic approach for high-assurance of data trustworthiness in sensor networks. In 2012 IEEE 28th International Conference on Data Engineering, pages 1192–1203, Washington, DC, USA, 1-5 April 2012.

[9] Y. Ma, Y. Guo, X. Tian, and M. Ghanem. Distributed clustering-based aggregation algorithm for spatial correlated sensor networks. IEEE Sensors Journal, 11(3):641–648, 2011.

[10] M. C. Vuran, O. B. Akan, and I. F. Akyildiz. Spatio-temporal correlation: theory and applications for wireless sensor networks. Computer Networks Journal (Elsevier, 45:245–259, 2004.

[11] R. Gwadera. Multi-stream join answering from mining significant cross-stream correlations. Frontiers of Computer Science, 6(2):131–142, 2012.