Federated Identity and Access Management for the Internet of Things

Paul Fremantle*†, Benjamin Aziz*, Jacek Kopecký*, and Philip Scott*
*School of Computing, University of Portsmouth,Portsmouth, UK
{paul.fremantle, benjamin.aziz, jacek.kopecky, philip.scott}@port.ac.uk
†WSO2 Inc., Mountain View, California
paul@wso2.com

***Abstract*—We examine the use of Federated Identity and Access Management (FIAM) approaches for the Internet of Things (IoT). We look at specific challenges that devices, sensors and actuators have, and look for approaches to address them. OAuth is a widely deployed protocol – built on top of HTTP – for applying FIAM to Web systems. We explore the use of OAuth for IoT systems that instead use the lightweight MQTT 3.1 protocol. In order to evaluate this area, we built a prototype that uses OAuth 2.0 to enable access control to information distributed via MQTT. We evaluate the results of this prototyping activity, and assess the strengths and weaknesses of this approach, and the benefits of using the FIAM approaches with IoT and Machine to Machine (M2M) scenarios. Finally we outline areas for further research.**

***Keywords*-Internet of Things; Computer Security; Authentication;**

I. Introduction

The Internet of Things refers to the set of devices and systems that interconnect real world sensors and actuators to the Internet, which includes many different systems, such as connected cars [1], wearable devices including health and fitness monitoring devices [2], wireless sensor networks [3] and so on. The growth of the number and variety of devices that are collecting data is incredibly rapid. A study by Cisco [4] estimates that the number of Internet-connected devices overtook the human population in 2010, and that there will be 50 billion Internet-connected devices by 2020.

There are many security concerns about IoT devices [5], [6]. These include network as well as physical threats to devices. Many devices are based on low-power, inexpensive hardware including 8-bit controllers. These hardware devices are often understood to be unsuitable for high-strength encryption and signature algorithms, although recent research does show that in some cases they can support effective encryption using elliptic curves [7], [8].

Limor Fried is a well-known innovator in the IoT field, who has published a “Bill of Rights” for users of IoT devices [9]. One key motivator in this Bill of Rights for this work is the statement that: “Consumers, not companies, own the data collected by Internet of Things devices.” The approach that we explore directly enables this model where the user has control over who can access and modify that data.

One of the challenges that requires further research is the authorization and access control for data produced and consumed by IoT devices. This area is closely related to the concept of *privacy*. Many IoT devices are collecting personal data: including human activity, sleep patterns, health information, home automation usage etc. The result is that access to that data or ability to manipulate those devices may infringe on privacy. As a real example, in 2011, it was publicized that the sexual activity of users of the FitBit activity tracker was public by default [10].

The traditional approach to access control is based on the concept of roles, and is typically managed in a hierarchical, top-down approach [11]. This approach has distinct drawbacks for the Internet of Things. Firstly, it was designed without millions or billions of devices in mind. That would not be an issue, if every device has the same access requirements. However, a fundamental tenet of privacy is that users can decide (and understand) who can share their data. Consumers demand, for example, to allow a specific application access to only certain data. A user might allow their weight-loss club access to a rolling 7-day average but not to individual days weight values. This argues towards a highly controllable model where users can specify authorization for specific devices and/or applications. Cavoukian has promulgated the important concept of *Privacy by Design* and in [12] she states “Users must be empowered to execute effective controls over their personal information”.

The second concern with the traditional model is that it utilizes a centralized model of identity and authentication. When there are many devices manufactured by many different organizations and operating in many environments, this is an unrealistic requirement. There are also scaling issues with centralized identity models.

A third important concern that is not answered by traditional role-based security models is that of a mechanism for delegation of authority. In many IoT scenarios, there are machines that are operating on behalf of a user, and also scenarios where a device may operate on a third-party’s behalf for a specific period of time. For example, the owner of a smart lock may authorize a friend’s mobile phone to unlock that door for the next week so that the friend may feed the owner’s cat. This argues for a model where the user can *delegate* certain permissions to specific resources to a machine for a limited time.

A. OAuth

The OAuth 1.0 Protocol [13], and its successor, the OAuth 2.0 Framework [14], are protocols designed to solve the privacy and access control issues related to large-scale Internet-connected applications. OAuth allows users to delegate access to specified functions to third-party websites. It also allows users to share identification across websites without sharing their credentials across those websites. For example, the Twitter website uses OAuth 2.0 to allow third-party websites to "tweet" on your behalf.

A more detailed example of the problem which OAuth aims to solve is as follows: social networking websites often ask users for access to their email contact lists in order to bootstrap or extend the user's social network. In order to implement this, the social network would impersonate the user to their web-based email provider and access the contact list using the username and password of the user (which was previously asked for). This approach had significant concerns:

- There is no fine-grained access control: once the social network had access to the username and password they could do anything – for example, posting emails or spam.
- There is no time limitation: unless the user then changed the password then the social network could store this data and re-use it later, unless the user changed their password.
- There is a fundamental difference between machine-to-machine interactions, which increasingly use *tokens* for identification and access, and user-centric security, which uses usernames and passwords.

The OAuth 1.0 Protocol and OAuth 2.0 Authorization Framework are designed to solve these problems. Many Web-based systems nowadays have implemented the OAuth 1.0 and 2.0 protocols, including Google, Facebook, Twitter, Github and many others. The protocol is used to protect (and enable) API access and this is clear evidence that it can be used in highly scalable systems.

B. MQTT

The MQTT[1] protocol was originally devised as a protocol for telemetry over constrained networks [15]. Other messaging systems, such as IBM MQ Series [16], assumed high speed networks with low-costs per byte transmission. In many SCADA[2], IoT and Telemetry examples, the networks are constrained with high costs per byte (e.g. GPRS, satellite). Therefore the MQTT specification was designed to have a very low message overhead, with as little as two bytes extra per message. Another important aspect is power-usage, where using MQTT has been shown to use significantly less battery power than HTTP polling for similar scenarios [17]. The protocol is being standardized by the OASIS MQTT Technical Committee [18]. MQTT is used in many IoT scenarios, and there are libraries for microcontroller-based systems such as Arduino [19] that make it easy to utilize.

MQTT is based around a *publish and subscribe* model [20], often known as *pub/sub*. In this model, there are one or more central servers, known as *brokers*, that clients connect to. A client may publish information to the broker, subscribe to receive information, or both. The publishers are unaware of the subscribers, and vice-versa. Each publication and subscription is tied to a *topic*. The topic is a named virtual resource that is used to decouple publishers and subscribers. As well as decoupling publishers and subscribers from knowing about each other, the pub/sub model also decouples them in time, because all interactions are asynchronous.

MQTT has a very simple security model. For authentication, it currently only allows the use of a username and optionally password. For encryption and transport-level security, the Transport Layer Security (TLS) standard is recommended, although this is not always suitable for small devices. Some implementations also support the use of a Pre-Shared Key (PSK) with TLS which can be used for authentication as well as encryption. The specification does not describe or recommend any authorization models, but certain implementations support access control lists on specific topics.

We chose to utilize MQTT as the basis of this work because: MQTT is a de-facto standard protocol used by a variety of IoT and M2M systems, and is in process towards becoming a full International Standard; there are several open source implementations, making it easy to work with; the central broker model creates an easy place to implement authorization and access control measures; and the topic model is highly flexible and offers scope for authorization control.

C. Research Questions and Proposal

Our main research aim is to explore whether it is feasible and effective to use OAuth2 as part of the MQTT protocol flow and within an MQTT broker to make federated, user-directed access control decisions. This aim translates in terms of a number of research questions that include:

- Identifying whether there are any significant issues to using OAuth2 with MQTT
- Identifying parts of the OAuth2 specification that are amenable to be used with MQTT and whether there are any mismatches or areas where we need to modify existing specifications or create new ones to support the new model
- Understanding which message formats could work and how could we fit the OAuth2 tokens into the flow, what the overall flow itself would look like and generally whether there are improvements to the implementation systems that would help support our aim
- Understanding if there are any wider impacts or lessons to be learnt when applying Federated Identity and Access Management to IoT.

The rest of the paper is organised as follows. In Section II we discuss related work. In Section III, we provide an overview of our implementation of the proposed solution and in Section IV we discuss the outcome of our experiment. Finally, in Section V, we conclude the paper and discuss directions for future work.

[1] Originally named the Message Queue Telemetry Transport, but now just MQTT

[2] Supervisory Control and Data Acquisition

II. Related Work

The IETF has published a draft guidance on security considerations for IoT [6]. This draft does discuss both the bootstrapping of identity and the issues of privacy-aware identification. One key aspect is that of bootstrapping a secure conversation between the IoT device and other systems, which includes the challenge of setting-up an encrypted and/or authenticated channel such as those using TLS, HIP or Diet HIP. The Host Identity Protocol (HIP) [21] is a protocol designed to provide a cryptographically secured endpoint to replace the use of IP addresses, which solves a significant problem – IP-address spoofing – in the Internet. Diet HIP [22] is a lighter-weight rendition of the same model designed specifically for IoT and M2M interactions. While HIP and Diet HIP solve difficult problems, they have significant disadvantages to adoption. Firstly, they require low-level changes within the IP stack to implement. Secondly, as they replace traditional IP addressing they require a major change in many existing systems to work. In addition, neither HIP nor Diet HIP address the issues of federated authorization and delegation.

OAuth is a relatively new framework and there has, as yet, been little research into it. Pai et al [23] have utilized the Alloy Framework [24] to analyze the security constraints of the OAuth protocol. Similarly, MQTT is also relatively a new protocol. Perez [25] has modelled and analysed the performance of MQTT, and Lee et at [26] have analyzed message loss in MQTT networks and the correlation to the level of Quality of Service (QOS) requested. Mengusoglu and Pickering [27] have created an Autonomic Management system utilizing MQTT as the messaging protocol, whereas Stanford-Clark and Wightwick [28] have demonstrated the applicability of MQTT for environmental monitoring and control systems. Recently, a formal analysis of MQTT's quality of service semantics also has been undertaken by the second author in [29], which demonstrated some ambiguities related to those semantics.

There has been little research into the security of MQTT. The OASIS Technical Committee is considering security and in the latest working draft [30] there is a discussion of how to secure MQTT. The document explicitly mentions that OAuth could be used. As another example, Collina et al. [31] have built a system – QEST – that is both an MQTT broker and a RESTful HTTP server. They also suggest that adding OAuth-like support to their broker is worth exploring, but have not implemented this at the time of writing. On the other hand, Facebook Messenger [32] on mobile devices uses MQTT and Facebook is also a user of OAuth [33], but there is no published information on whether they utilize the OAuth tokens for MQTT authentication and authorization.

Other related works include the work of Augusto et al. [34] have built a secure mobile digital wallet by using OAuth together with the XMPP protocol [35]. While the OAuth protocol has mainly been focussed on Web- and HTTP-access control, there is a proposal [36] to support OAuth tokens with the SASL [37] authentication protocol, which demonstrates OAuth usage outside of HTTP. Finally, the use of OAuth with the Constrained Application Protocol (CoAP) – another IoT-focussed protocol – was suggested in passing at the IETF87 meeting [38], however, we are unaware of any work to implement this.

The related work clearly suggests that the use of OAuth together with IoT devices, and in particular with IoT specific protocols such as MQTT and CoAP is worth exploring. We did not find any evidence that this had been done before and so we looked to implement this in order to explore the concepts of federated, personal access control within IoT and M2M protocols.

III. Implementation

In order to implement the system, we used several existing open source projects that provided a set of building blocks. This allowed us to focus on the core concerns of authorization and security without spending too much time on implementation. There are some areas where the choice of existing technologies provided limitations, and we call out those areas in the results section below.

The overall system consists of four major components: the MQTT broker, the Authorization Server, the Web Authorization Tool and the device. The MQTT broker is based on the Mosquitto broker, including extensions we created to enable OAuth-based authentication and authorization. The Authorization Server is based on the open source *WSO2 Identity Server*. We made an assumption of a single broker and a single Authorization Server for this prototype. the Web Authorization Tool (WAT) allows a user or developer to create a token to authorize access to their personal data. In addition, we built and programmed an Arduino-based IoT device which publishes data to the MQTT broker. We provide more details about each of the components below.

Figure 1 shows the major components of the system and the major interactions between them.

A. Authorization Server

The WSO2 Identity Server [39] is an open-source identity and access management server that supports a wide variety of identity and access management protocols, including OAuth 2.0 and OpenID Connect amongst many others. The server offers an easy-to-use web-based console which allows administrators to provision users, configure OAuth applications and other tasks that were needed as part of this exploration.

We did not need to modify or extend the WSO2 Identity Server to implement the "Authorization Server" role defined in our architecture. However, the WSO2 Identity Server implements its own SOAP-based API for querying the validity and scopes of a token. This was not ideal to call from the message broker, and we preferred the using of the OAuth Token Introspection [40] API, which is a simpler RESTful API. We therefore created [41] a simple bridge between the two APIs using the WSO2 ESB [42], which meant that we could swap the WSO2 Identity Server out for any other OAuth2 server that implements this API. More generally, this extension means any system that use the Introspection API can now interact with the WSO2 Identity Server.

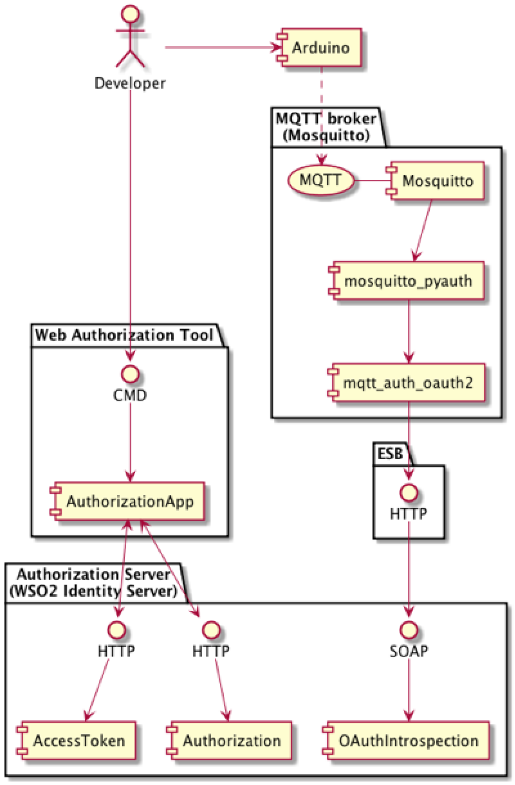

Fig. 1. Component diagram of the implemented system

In order to capture authorization scopes that a token is given access to, we created a JSON [43] encoding. The following example shows a scope that encodes a client who can write to any subtree of */topic/paul* and read/write to */scratch*:

```
[   {rw:"w", topic:"/topic/paul/#"},
    {rw:"rw", topic:"/scratch"} ]
```

This model uses the same syntax for wildcards as the MQTT specification, which is the logical and most effective approach. Because the scope list in OAuth2 is space-delimited, we could not use this JSON as-is to form the authorization scopes. To solve this, we used Base64-encoding to create simple strings that could be used as scopes. There are other solutions we could have taken, but this was expedient as it made parsing the scopes simple for the broker.

B. Web Authorization Tool (WAT)

We need to be able to instantiate the bearer and refresh tokens and associate the correct access control scopes with these tokens.

This process is described in greater detail in the OAuth 2.0 specification. In the OAuth terms, the MQTT device is the “client”. However, we are actually separating this into two parts: the bootstrap process, and the runtime process. The bootstrap process is a set of Web-based flows. Once this process is complete, we then take the bearer and refresh tokens and embed these into our device, which then uses them at runtime. In order to support these Web-based flows, we ideally needed to create a Web-application, because part of the flow is a redirect back to that application, which needs to be Web addressable.

However, we felt a command-line tool would be simpler and more appropriate. To solve this, we created a command-line tool in Python that ran a simple embedded Web-application. This command-line tool spawns a browser with the correct URL to request a token. This is important because the user must authenticate to the Authorization server, which then redirects that browser back to the local embedded Web server. The result is that there is no need to run a dedicated Web server and the command-line tool can be used to create tokens. This tool generates a textual version of the token, suitable for cutting and pasting into other systems; and generates C code for the Arduino, which can be cut and pasted into the Arduino tooling to update the Arduino. In a real environment as opposed to this prototype, we would have extended the tool to write this directly into the flash of a connected device.

In addition, we wanted to demonstrate the other end of the flow – a non-device-based application gaining access to the data published by the IoT device. Of course, this application must be authorized by the owner of the data (the Resource Owner) to be able to access this data. Ideally this would have been a proper Web application. However, we felt that this angle was the closest to the existing OAuth2 usage and so few insights would be gained from building this, so we emulated this using the default command-line clients provided by Mosquitto. We call this aspect the Message Viewer.

C. The Device

To create our sample IoT device, we utilized an 8-bit Arduino open hardware device [19], together with the Arduino Ethernet Shield, which is a daughterboard that provides Ethernet support. The Arduino toolkit has a library that supports the MQTT specification [44]. Our device utilizes a 9-axis inertial measurement unit (IMU) that provides acceleration, rotation and compass data – each in 3 dimensions, to track the movement of the device. Such devices, when attached to a person, provide significant data on the users position, activity and exercise levels. Such data exemplifies the problem space here: users wish to share this kind of data, but only in precisely controlled ways, and each user may have radically different approaches and concerns about both data sharing and privacy.

One important point to note is that we did not implement any TLS or encryption from the device to the MQTT broker. This was because of space requirements in the Arduino device. However, we consider this issue orthogonal to the other concerns as OAuth2 and MQTT layer cleanly sits over TLS.

D. MQTT Message Broker

Mosquitto is an open-source MQTT broker written in the C language [45]. It is highly portable. Mosquitto has a well-defined model for adding authentication/authorization plug-ins [46] and there are several third-party plugins but we could not find any example of an existing OAuth-based plugin for Mosquitto.

In this model, the MQTT IoT device connects to the broker, and instead of passing over a username and password, it sends an OAuth 2.0 *bearer token*. The bearer token is a specific type of token supported by the OAuth 2.0 specification that was designed to support very simple client software. Unlike the OAuth 1.0 specification, which required the client to perform signature operations in order to communicate, the bearer token is a token which can be stored and sent as proof of identity. This has its own security challenges which we discuss later.

To create our authorization plugin we utilized the mosquitto_pyauth [47] project. This plugin converts from the normal C-language based model for Mosquitto plugins into the Python language, allowing us to write the OAuth plugin for Mosquitto in Python instead of C.

Utilizing mosquitto_pyauth as a bridge into Python, we then wrote a Python-based plugin – *mosq-auth-oauth2* – that validates the OAuth2 bearer token which the client passes over. The plugin sends this OAuth token to the *OAuth Introspection Service*, that validates the token and returns a set of authorized scopes for this token. The authentication/authorization plugin then validates if the requested action (reading or writing to a topic) is valid against this scope.

A picture of the implemented architecture is shown in Figure 2 in Appendix A, where the prototype device is connected to a Mosquitto broker running on Raspberry Pi hardware. Additionally, Appendix A contains two sequence diagrams showing the system interactions: Figure 3 shows the sequence diagram of interactions during the bootstrap phase, and Figure 4 shows the sequence diagram of the interactions during the use of the device and the Message Viewer.

IV. Results

Overall the system worked as intended, and showed the following aspects:

- Both IoT and non-IoT clients were able to use OAuth tokens to authenticate to the broker.
- The broker was able to connect using a RESTful interface to the OAuth Authorization Server to validate tokens.
- The broker was able to introspect the token via the RESTful interface on the Authorization Server to retrieve a list of appropriate scopes.
- The broker was able to use the scopes to decide on whether to authorize a publish or subscribe operation.
- The Web Authorization Tool was able to implement the OAuth 2.0 bootstrap process to create Access Tokens, which were then embedded into the MQTT clients.

However, we did find several issues during implementation, which we will discuss next.

A. Refreshing the Token

Firstly, the use of HTTP as a protocol to implement the Refresh Token flow is not ideal. It requires the device (in our case the Arduino client) to implement two protocols (MQTT and HTTP), which is inconvenient. To give an example, our IMU and MQTT code took up 97% of the program space of the 8-bit Arduino on which we prototyped. The extra code for OAuth2 took us to 99%. Adding HTTP code for the refresh flow would have pushed us over the limit.

A fundamental issue is that the Refresh Token flow requires the device to produce the Client ID and Client Secret. This is a well known issue in the mobile space, and highlights a challenge in using the OAuth specification outside the Web application space. This is a limitation in the design of OAuth, which is effectively based around the concept of a Web Application. In the Web Application model, there is a single place where the Web Application is deployed and managed[3].

In an IoT (or for example the case of a mobile phone application), the client application is deployed in thousands or potentially millions of places. The task of securing the Client Secret is therefore much harder. Microcontrollers such as the Atmel Mega used in the Arduino do allow the code to be locked onto a device, but in any situation where the hardware is accessible, it is always possible for a determined hacker to break such security. This really depends on the value associated with compromising the security: if the reward is high enough, a determined hacker can utilize a Scanning Electron Microscope [48]. The alternative is to issue Access Tokens with very long expiry times, where the expiry time is expected to be longer than the lifetime of the device, or where there is an option to update the device with new firmware. This may be possible, but it goes against best practice for OAuth2.

A key question that arises is whether each device is considered in the OAuth2 world to be a separate Client or that the overall set of devices is considered to be a single Client. If each device is a unique client, then it is to code the Client ID and Client Secret into each device. If a single device is compromised, it only compromises that devices' secrets, and they can be revoked. However, it is (as-yet) untested whether OAuth implementations could cope with that many Clients (e.g. millions). In typical Web scenarios there are few Clients (say hundreds or maybe thousands at most).

It seems to us that it is closer to the current model to treat the set of similar devices as *a Client*. In the Web world, a single website acts as a Client and there are thousands or millions of users, each with a token. The analogy in the IoT world would be a single system with thousands or millions of devices attached. In this model, it would be a serious breach of security to embed the "global" Client ID and Client Secret into each device, as compromising one device would potentially compromise all of them.

To investigate this further, we built a second phase to the experiment, where we mapped the refresh flow into MQTT. Effectively, we created a second MQTT broker – the Authorization Broker (AB) – whose only job is to act as a proxy to the Authorization Server, and we used MQTT instead of HTTP to communicate from the device to this broker. During the implementation we discovered some interesting sub-issues specific to this refresh flow. Firstly, it is not trivial to send a message to a single device using MQTT. As the system is a pure pub/sub network there is no built-in way of targeting a single device. Obviously we did not desire to broadcast

[3] Of course a Web Application may be deployed in multiple datacenters for scalability and disaster recovery, but this is still logically a single place

the refresh token to all devices as that could be misused. To solve this, we implemented a special authentication rule and topic hierarchy on the AB that allowed us to securely send a message just to a single client.

Secondly, the OAuth specification optionally allows the Authorization Server to refresh the Refresh Token as well. This is a significant issue in a device-oriented world, because the HTTP call to the Authorization Server is unreliable. If the device ends up "out-of-sync" with the Authorization Server, then the device will need to be re-registered. In the Web world, this is easily sorted and happens frequently - it is just a case of the user re-authorizing. With devices, this may not be possible as there is often no Web Browser and UI – which is required in this case. To get round this, we worked with the WSO2 Identity Server development team to add an option to allow us to keep the refresh token constant. We would recommend any OAuth2 server that is being used for IoT devices would avoid changing the Refresh Token with every refresh. This applies to IoT devices using HTTP as well as MQTT or other protocols. Another option would be to use a reliable protocol to transmit the updated refresh token end-to-end from the Authorization Server to the device.

Another concern is whether this model (where any device can connect to the authorization broker and pass over a Refresh Token) is a security hole. Certainly we could protect it in some regard, but this may not improve matters. If we use a unique credential per device, we have created a circular issue (i.e. how do we manage that credential?). If we use a default credential that is the same for all devices, an attacker needs only to break one device to break this. On the other hand, without a refresh token, the attacker cannot do any real harm. They can cause denial-of-service by issuing many requests, but this could be prevented by standard DoS firewall techniques. On the whole, we felt this model was better than having a unique Client ID/Client Secret per device or having the same Client ID/Client Secret on all devices.

B. *Changing the scope*

The third issue that was identified during our work was that of updating the scope. In many cases, one might want to change the permissions associated with a device after the firmware has been loaded. In many Web systems, to change the scope of a token is done by re-issuing the token. As we have discussed, this is not appropriate for many embedded devices or those without a UI and browser.

A better approach is to update the scope of the existing token. This would remove the burden from the device and deal with it purely at the Authorization Server. However, as currently specified, the OAuth 2.0 specification has no explicit mechanism to allow this. That said, the specification does not prevent it either, but it would need to be an implementation specific approach and therefore would not be interoperable. We see this as an area where the challenges of the IoT should influence the development of OAuth or other similar protocols, as well as the capabilities of OAuth implementations.

V. Conclusions and Further Work

In this paper, we have shown that a standardized federated, dynamic, user-directed authentication and authorization model can be adapted from the Web for use in IoT devices. We have argued that this model is important for the concept of privacy with respect to IoT devices and the data that they generate and use. We have identified a number of issues, including both minor implementation issues as well as more fundamental issues where we propose further research. This was the first such implementation of OAuth2 with MQTT.

As a result of this work, there are several recommendations that we propose. These include the need for clear standardization of where to put the token as well as limiting the OAuth2 token size to some reasonable limit - at least for IoT use. Also there is a need to define a clear MQTT flow for refresh and avoid refreshing the refresh token. Finally, there is some concern over the AB's security and the need to be able to modify scopes. Two of these findings apply also to the use of OAuth2 with any protocol, including HTTP, when used with IoT devices: the need for the refresh token to be constant (or transmitted reliably), and the need to allow scopes to be modified for existing tokens.

While there were some issues with implementing FIAM for IoT using OAuth2 with MQTT, the benefits of building on existing widely implemented and deployed protocols are significant. Many years of work and review have gone into the security of OAuth2, and from this work we can state that it is possible to re-use this work with IoT devices and new protocols.

Several areas for further research emerge from this work. A formal model and proof of security attributes for the use of OAuth2 with MQTT and in particular, the refresh flow, would be beneficial. Similarly, it would be valuable to analyse the performance of this model and understand the extra cost of using OAuth2 compared with traditional security. One aspect we did not consider was allowing more than one OAuth2 Authorization Server to be used with a single broker, which would merit research.

We wish also to investigate the use of OAuth2 with CoAP and other protocols for IoT. During discussions of this work with device implementers, a common request has been to create an Arduino "shield" that embodies the MQTT, OAuth2 and TLS behaviour completely, which would make an interesting project. Finally, we believe that some investigation is merited into whether the OpenID Connect specification could add any benefits over OAuth2 in the IoT space.

References

[1] C. Evans-Pughe, "The connected car," *IEE Review*, vol. 51, no. 1, pp. 42–46, 2005.

[2] "Fitbit." [Online]. Available: http://www.fitbit.com/us/company#about

[3] F. L. Lewis, "Wireless sensor networks," *Smart environments: technologies, protocols, and applications*, pp. 11–46, 2004.

[4] D. Evans, "The internet of things," *How the Next Evolution of the Internet is Changing Everything, Whitepaper, Cisco Internet Business Solutions Group (IBSG)*, 2011.

[5] R. Roman, P. Najera, and J. Lopez, "Securing the internet of things," *Computer*, vol. 44, no. 9, pp. 51–58, 2011.

[6] e. a. O. Garcia-Morchon, "Security Considerations in the IP-based Internet of Things," IETF, Internet Draft, September 2013, available at http://tools.ietf.org/html/draft-garcia-core-security-06.

[7] M. Sethi, J. Arkko, and A. Keranen, "End-to-end security for sleepy smart object networks," in *Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on*. IEEE, 2012, pp. 964–972.

[8] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds. Springer Berlin Heidelberg, 2004, vol. 3156, pp. 119–132. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_9

[9] A bill of rights for the internet of things - room for debate - nytimes.com. (Visited on 11/13/2013). [Online]. Available: http://www.nytimes.com/roomfordebate/2013/09/08/privacy-and-the-internet-of-things/a-bill-of-rights-for-the-internet-of-things

[10] Fitbit users are unwittingly sharing details of their sex lives with the world. (Visited on 06/04/2013). [Online]. Available: http://thenextweb.com/insider/2011/07/03/fitbit-users-are-inadvertently-sharing-details-of-their-sex-lives-with-the-wo

[11] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

[12] A. Cavoukian, "Privacy in the clouds," *Identity in the Information Society*, vol. 1, no. 1, pp. 89–108, 2008.

[13] E. E. Hammer-Lahav, "The OAuth 1.0 Protocol," IETF, RFC 5849, April 2010, available at http://tools.ietf.org/html/rfc5849.

[14] D. H. (ed), "The OAuth 2.0 Authorization Framework," IETF, RFC 6749, October 2012, available at http://www.rfc-editor.org/rfc/rfc6749.txt.

[15] "Mqtt history." [Online]. Available: http://mqtt.org/wiki/doku.php/history

[16] L. Gilman and R. Schreiber, *Distributed computing with IBM MQSeries*. John Wiley & Sons, Inc., 1996.

[17] Power profiling: Https long polling vs. mqtt with ssl, on android. (Visited on 06/04/2013). [Online]. Available: http://stephendnicholas.com/archives/1217

[18] Oasis message queuing telemetry transport (mqtt) tc. [Online]. Available: https://www.oasis-open.org/committees/mqtt/

[19] Arduino, "Arduino." [Online]. Available: http://arduino.cc

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[21] R. Moskowitz, "Host identity protocol architecture," 2012.

[22] ——, "Hip diet exchange (dex)," 2012.

[23] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, "Formal verification of oauth 2.0 using alloy framework," in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*. IEEE, 2011, pp. 655–659.

[24] D. Jackson, "Alloy 3.0 reference manual," *Software Design Group*, 2004.

[25] J. Perez, "Mqtt performance analysis with omnet++," *Master's Thesis, September*, 2005.

[26] S. Lee, H. Kim, D.-k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *Information Networking (ICOIN), 2013 International Conference on*. IEEE, 2013, pp. 714–717.

[27] E. Mengusoglu and B. Pickering, "Automated management and service provisioning model for distributed devices," in *Proceedings of the 2007 workshop on Automating service quality: Held at the International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 38–41.

[28] A. J. Stanford-Clark and G. R. Wightwick, "The application of publish/subscribe messaging to environmental, monitoring, and control systems," *IBM Journal of Research and Development*, vol. 54, no. 4, pp. 1–7, 2010.

[29] B. Aziz, "A Formal Model and Analysis of the MQ Telemetry Transport Protocol," in *9th International Conference on Availability, Reliability and Security (ARES 2014)*. IEEE, 2014.

[30] Mqtt version 3.1.1 working draft 15. (Visited on 11/13/2013). [Online]. Available: https://www.oasis-open.org/committees/download.php/51356/mqtt-v3.1.1-wd15.pdf

[31] M. Collina, G. E. Corazza, and A. Vanelli-Coralli, "Introducing the qest broker: Scaling the iot by bridging mqtt and rest," in *Personal Indoor and Mobile Radio Communications (PIMRC), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 36–41.

[32] "Building facebook messenger." [Online]. Available: https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger

[33] "Facebook login." [Online]. Available: https://developers.facebook.com/docs/facebook-login/

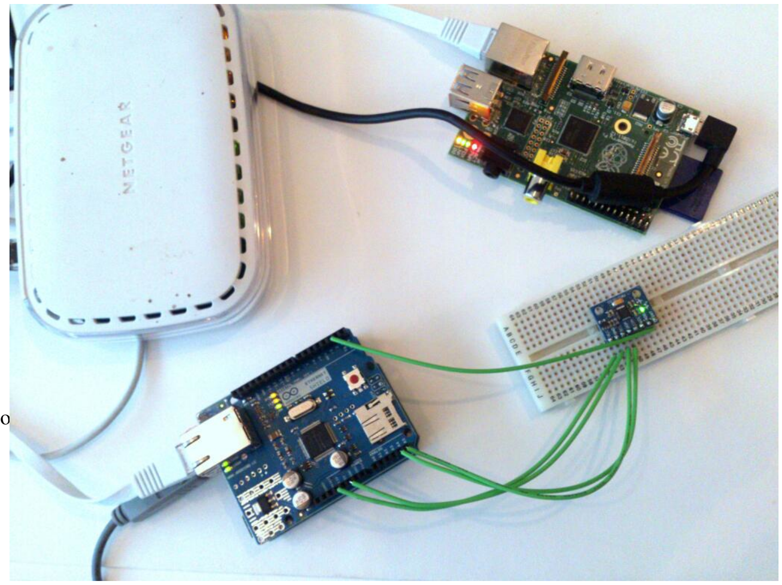

Fig. 2. Arduino Device prototype with 9-axis IMU

[34] A. B. Augusto and M. E. Correia, "An xmpp messaging infrastructure for a mobile held security identity wallet of personal and private dynamic identity attributes," *Proceedings of the XATA*, 2011.

[35] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," 2011.

[36] K. W. Mills, "A sasl and gss-api mechanism for oauth draft-mills-kitten-sasl-oauth-04. txt," 2011.

[37] J. G. Myers, "Simple authentication and security layer (sasl)," 1997.

[38] "Minutes of the core working group at ietf meeting 87." [Online]. Available: http://www.ietf.org/proceedings/87/minutes/minutes-87-core

[39] Wso2 identity server — wso2 inc. (Visited on 11/13/2013). [Online]. Available: http://wso2.com/products/identity-server/

[40] J. Richer, "Oauth token introspection," 2013.

[41] "Oauth2 introspection with wso2 esb and wso2 identity server." [Online]. Available: http://pzf.fremantle.org/2013/11/oauth2-introspection-with-wso2-esb-and.html

[42] Enterprise service bus — wso2 inc. (Visited on 11/13/2013). [Online]. Available: http://wso2.com/products/enterprise-service-bus/

[43] "The application/json Media Type for JavaScript Object Notation (JSON)," IETF, RFC 4627, July 2006, available at http://www.rfc-editor.org/rfc/rfc4627.txt.

[44] Arduino client for mqtt knolleary. (Visited on 11/13/2013). [Online]. Available: http://knolleary.net/arduino-client-for-mqtt/

[45] An open source mqtt v3.1 broker. (Visited on 11/13/2013). [Online]. Available: http://mosquitto.org/

[46] jpmens/mosquitto-auth-plug. (Visited on 11/13/2013). [Online]. Available: https://github.com/jpmens/mosquitto-auth-plug

[47] mbachry/mosquitto_pyauth. (Visited on 11/13/2013). [Online]. Available: https://github.com/mbachry/mosquitto_pyauth

[48] Microcontroller unlocking for code recovery- silicon investigations. (Visited on 11/15/2013). [Online]. Available: http://www.siliconinvestigations.com/Bchip/Code_ext.htm

Appendix

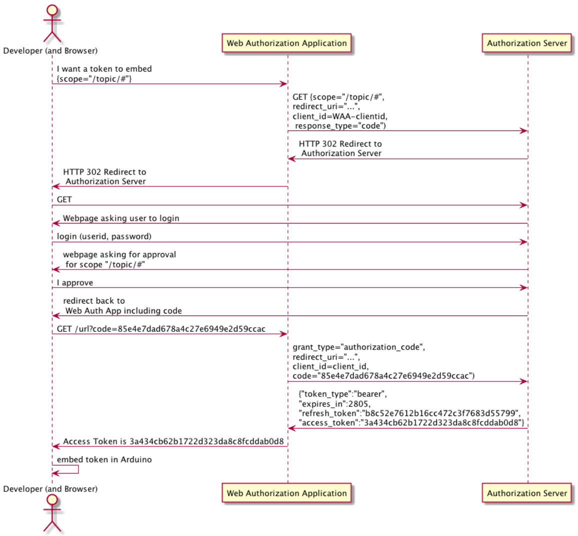

Fig. 3. UML Sequence Diagram demonstrating the bootstrap phase

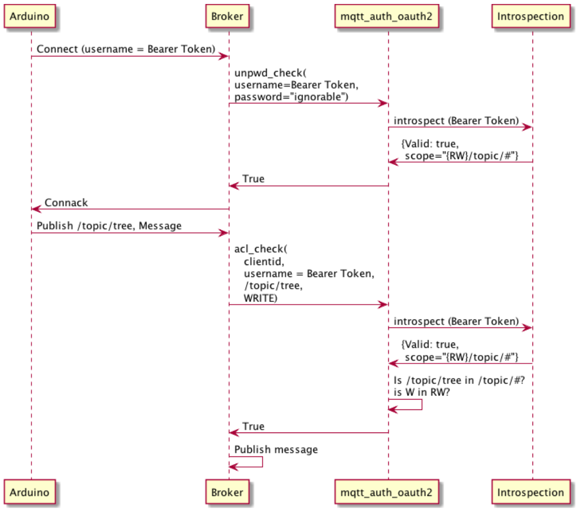

Fig. 4. UML Sequence Diagram demonstrating OAuth access control check

Acknowledgments

Thanks to the Mosquitto project and Eclipse Paho projects for the excellent code, to Prabath Sidiwardena for help with OAuth2 and the WSO2 Identity Server. Finally, thanks to Jane for proof-reading.