SRUP: The Secure Remote Update Protocol

Andrew John Poulter*, Steven J. Johnston†, Simon J. Cox‡
University of Southampton,
Faculty of Engineering and the Environment,
Southampton, United Kingdom
*Email: a.j.poulter@soton.ac.uk** *sjj698@zepler.org*† *s.j.cox@soton.ac.uk*‡

***Abstract*—Internet of Things, and other connected devices require secure mechanisms to facilitate the propagation of Command & Control messages to enable the remote management of the devices — including the ability to remotely perform software update on the devices. This paper introduces a new protocol designed to sit on top of the commonly-used MQTT protocol to provide a secure and confirmable mechanism to deliver such messages to remote Internet of Things devices.**

***Keywords*-MQTT; IoT; C2; Command & Control**

I. Introduction

This paper describes a new protocol which provides a secure and confirmed mechanism to facilitate the exchange of command-and-control messages to remote Internet of Things (IoT) devices.

Section II describes the utility of Command & Control messages within the context of the IoT, and discusses the requirements for remote software update for IoT devices; Section III describes Message Queuing Telemetry Transport (MQTT), a communication architecture commonly adopted in the IoT; and Section IV introduces the Secure Remote Update Protocol (SRUP) protocol, which is built on top of MQTT. Finally Section V describes the current state of an implementation of the protocol and planned future work.

II. Command & Control and Software Update for the Internet of Things

A. Command & Control for the Internet of Things

Whilst the IoT covers a multitude of different types of device, and different applications; many classes of IoT devices (for example, those used in environmental monitoring) are designed to operate remotely (potentially without convenient access to the device by a human operator) and semi-autonomously. As such these types of devices require a *system-level* Command & Control (C2) system to support their operation.

Such a C2 system enables the devices to receive and react to messages pertaining to the device itself (as opposed to the application software the device is running). This enables a human-operator or autonomous agent to interact with the device remotely.

Whilst the primary example of C2 messages within this paper are those to enable the software on the device to be updated; supporting many other types of C2 message will also be beneficial to the operation of a fleet of remote, semi-autonomous devices.

For example it may be desirable to be able to remotely trigger a reboot of the device, or to cause it to enter a low-power mode. Some applications may require a mechanism to turn off devices (noting that such an action would likely render the device inoperable, unless the device has a capability to turn itself back on); or even to cause the device to *self-destruct* — figuratively, or literally.

Examples of C2 messages that may be useful within the context of IoT devices may include commands to:

- Enter low power mode / wake to full power mode
- Turn off
- Self-destruct
- Reboot
- Self-Test
- Enter a specific operating mode (e.g. beacon mode, or safe mode)

B. Software Update in the Internet of Things

The ability to remotely apply software updates to IoT devices is critical. Software updates may be required to provide simple bug-fixes to address discovered vulnerabilities, or to add new functionality to the software. Updates of the application software can also deliver a significant reconfiguration of the operation of the device itself. Such reconfiguration can modify the device to provide new functionality — and can utilize previously dormant hardware within the device. This *adaptability* of the hardware can deliver *software defined hardware*.

Depending on the details of the implementation some aspects of this application software configuration can be achieved through the use of *data-driven software*: where the application's behaviour is determined by a combination of the software and the data. At its most extreme the data could be source code expressed in a general-purpose scripting language. More typically however the software will be a special purpose application whose operation will be determined by the data it is supplied. Using this type of approach can potentially simplify many routine software updates to the task of supplying a data file to the device — and causing it to be processed by the application software. This technique

978-1-5090-4130-5/16/$31.00 ©2016 Crown

also has advantages in the context of bandwidth-constrained environments: as a data file containing just the changes to an existing configuration data file on the device may be expected to have a very-much smaller file-size than a new binary executable software application file.

C. The challenge of remote software update

Unlike locally updating the software on a device there are unique difficulties that must be taken into account when considering a remote software update. IoT devices often have their primary user-interface provided by a network connection, and so even if device can be accessed physically it may still require *remote* update via the network interface. It must be impossible for the update process to cause the device to fail to a state in which it cannot be accessed via the network: as this would render the device unusable. As such the most important requirement is that any update is recoverable in the event of network or power-supply interruption: and that in the event of such an incident, the process can be either resumed or restarted.

There is also a requirement to be able to track whether a particular device has been updated; and know (with certainty) that the device has received, and is using, the updated software. In some situations the process also needs to be capable of delivering a unique software payload or update to each device to handle situations where bespoke software may be required by devices based on operational circumstances (e.g. all devices in a specific geographical area, or all devices belonging to a particular customer or operator).

D. Software Update Paradigms

There are essentially two ways to initiate a software update: either *pushing* the software to the device (establishing an inbound connection to the device — and uploading the data), or by triggering the device to fetch the software update itself, via an appropriate signal; and causing the device to download the software to itself. If pushing software to a device, an inbound network connection to the device is established and the data is simply uploaded. However, having the ability to open a port directly on a device increases the risk of that device being subject to compromise; and it also places a requirement on the device to be able to authenticate connections. As such it is extremely difficult to secure such a process — and it is not recommended for any application connected to the internet: especially applications where security or privacy are important.

The use of a signal to initiated software download by a device is analogous to the way that Operating System patches are often distributed: remote systems do not push the update to the computer — but rather the computer monitors a known *repository*, waiting for an indication that updates are available. This signalling may be implemented by either the device polling a suitable semaphore checking for a change, or by some other type of notification signal.

The SRUP protocol has been designed around the use of this approach.

The use of this technique also means that the device does not have to be directly addressable from the Internet: as may be the case if it is connected to a network which is protected by a Firewall or behind a router using Network Address Translation (NAT).

E. Cryptographic Security Considerations

Regardless of the process used to get the software onto the device, the process needs to be both secure (to prevent a malicious third-party from abusing the mechanism to cause devices to run malware), and confirmable: such that the *controller* can be certain that devices have been updated correctly.

This can be addressed by using cryptographic algorithms to sign the software and messages as authentically originating from an approved source. Furthermore by utilizing a private key within the device, it is possible to ensure that only messages, data or software updates intended for a specific device are usable by that device.

The identity of the URL where the data can be found, can be verified using Transport Layer Security (TLS) and Secure Hyper-Text Transfer Protocol (HTTPS); and their content checked for integrity using a hashing function such as those implemented in the Secure Hash Algorithm, version 2 (SHA2) standard [1].

III. A Publish / Subscribe Architecture for IoT Communications

There are a number of communications architectures used to facilitate communications with IoT devices; one of the most commonly used is a publish / subscribe paradigm. Within IoT devices perhaps the most commonly used protocol for this is MQTT.

A. MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight brokered publish / subscribe protocol, originally developed by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom) in 1999 to provide lightweight telemetry for the oil and gas industry. It became an OASIS standard in 2013. [2] All messages sent are routed via a broker; the broker is responsible for tracking all subscriptions, and sending data to subscribers when a publisher issues a message. MQTT runs over Internet Protocol Suite (TCP/IP) networks, using Transmission Control Protocol (TCP) [3] as the transport layer.

MQTT is by default an insecure protocol (all data is sent as plain text) — but it can be secured with the application of Transport Layer Security (TLS) [4] to the traffic.

There are a number of open-source brokers — this research has been carried out using the Mosquitto broker.

Mosquitto implements the MQTT protocol versions 3.1 and 3.1.1; and is an iot.eclipse.org project [5].

MQTT is a very widely supported protocol within the IoT. MQTT libraries exist for many common development platforms (including Arduino, and mbed); and it is supported by the vast majority of programming languages. There are also a large number of clients available — both command-line tools, and GUI applications; & there are a number of clients available for mobile devices too (both iOS and Android). MQTT is also the protocol used by Amazon Web Services (AWS) for their IoT platform. [6].

IV. The Secure Remote Update Protocol — SRUP

A. The concept of SRUP

SRUP is designed as a secure, efficient and straightforward protocol to carry C2 messages to IoT devices.

The SRUP protocol is built on top of MQTT — and provides an efficient binary message structure contained within the payload of an MQTT message. For example, a SRUP *update* message contains cryptographic signatures for the data to be retrieved; the source from which that data can be obtained; and other elements required to securely authenticate the message.

Although SRUP can be used in support of software updates on IoT devices, it is not a protocol for software updates. Rather SRUP securely conveys the C2 messages to the device: providing it with the information that it requires to carry out retrieval, and validation of the data; and a mechanism to provide confirmation of the process.

SRUP does not attempt to provide a protocol for the actual download of the data — this is handled over a conventional HTTPS connection to the server providing the software.

The advantage of SRUP over a more simple approach is that it makes it easy to target individual devices for specific updates; and provides a confirmable mechanism ensure that the update has been successfully and correctly received by the device. Whilst this may not be a general requirement today — it is possible to imagine that in the near-future such a practice may be more common-place (for example: to know whether devices have applied a security fix after a vulnerability was discovered). Moreover such a requirement does exist today for devices operating in certain Defence & Security environments; or other safety-critical operations where an auditable log may be required to positively identify the configuration state of devices.

The protocol also means that the process is recoverable in the event of an incomplete download. The protocol specifies confirmation and activation messages — meaning that the update would be applied on command, once the device has retrieved & checked the data, and successful retrieval has been signalled to the originator. This also means that a software update operation to be carried out on a fleet of IoT devices can be conducted in such a way as to minimize the impact on the functionality and availability of the fleet, that would otherwise occur if all devices were updated simultaneously.

SRUP requires that all devices within the scope of a given C2 server have a copy of that server's public key in order to authenticate messages from that server. The management and update of keys can be handled by using an approach similar to that shown for a software update. For messages sent from a device, the server will use that device's public key — with the device's identity being indicated via the MQTT subject. The server will receive the device's public key as a part of the process to register the device with the server. It is proposed that a class of C2 messages to support this device registration process could be introduced in a subsequent version of the protocol.

B. SRUP in action

To illustrate the use of SRUP for a software update command, consider a system composed of a number of IoT devices using MQTT and a suitable broker to communicate with a command-and-control server. A user-interface is provided via a web-application which communicates with the C2 server.

The exchange of routine data (such as sensor readings from the devices) can be communicated directly between the user and the devices via MQTT (over TLS where required); but C2 messages would be sent using SRUP on top of the (potentially encrypted) MQTT traffic.

To instigate a software update operation, the initial step would be for the user to provide the details to the C2 server, via the web interface. The C2 server would then send a SRUP `INITIATE` message to the device (or devices) in question.

A SRUP daemon running on the device receives the `INITIATE` message, and commences the download of the data over HTTPS. Having received the file, the daemon then calculates a hash value for the data retrieved (using the secure SHA-256 algorithm) and compares it with the value specified in the `INITIATE` message. If they match, the daemon would then send `RESPONSE` message to signify a successful completion of the retrieval; if they don't (or if any other errors occur) then the `RESPONSE` message would signify the failure of the retrieval & contain details of the error. Lastly, on receipt of the `RESPONSE` message, the C2 server sends an `ACTIVATE` message to request that the device *activates* the newly downloaded software or data.

The details of what occurs on receipt of the `ACTIVATE` message is highly dependent on the specifics of the device; but consider the most simple case, where there is just one piece of application software running on the device. In this situation the daemon must send a terminate signal to the currently running version of the application (which should ensure that any hardware under its control is placed into a safe state before exiting). Upon confirmation of successful termination of the application, the SRUP daemon should

then copy or move the software downloaded to a suitable location within the device's filesystem, and execute it. Typically it would be expected that for purposes of recovery, the previous version of the software would not be overwritten permitting reversion to a "known good state" in the event of issues with the newly obtained software.

C. Protocol Details

This section contains the technical details of the three message types associated with the software update command described above.

1) Initiate message: The `INITIATE` message must communicate five elements to the device.

- An identifier signalling the device for which the message is intended
- The URL at which the software can be retrieved by the device
- A cryptographic hash value for the software to be retrieved
- A unique token value to indicate the SRUP transaction
- A cryptographic signature — signing the data contained in the message

Using HTTPS ensures encryption of the communication to prevent eavesdropping, and authoritatively establishes the identity of the server to prevent man-in-the-middle style attacks.

The SRUP `INITIATE` message is sent as a compound binary message consisting of the various fields of the SRUP protocol, as shown in table I. In addition to the five items described previously — there is also a protocol version number (to easily permit future expansion of the protocol), and a message-type identifier to signal that this message is an `INITIATE` message.

Note that this scheme utilizes a number of variable length fields. This ensures that variable length elements such as the URL are sent as efficiently as possible (without padding) and without arbitrarily constraining the length of the URL that could be used.

In order for the receiving device to know how many of the arbitrary bytes compose each element of the message, in addition to the data shown in table I it is also necessary to send two additional bytes per element, which convey the length of the following element.

Although using two bytes for this adds slightly to the overall message length, it ensures that there is effectively no practical limit on the length of the elements. $(2^{16}-1) \equiv 65,535$ characters: more than long-enough for any currently conceivable URL or cryptosystem signature / hash value.

To avoid the potential for error when sending data between machines with different architectures (big-endian vs. little-endian) implementations of SRUP must include a custom marshalling / demarshalling routine for the lengths to ensuring that regardless of the local architecture, the *wire-format* for the lengths (little-endian) is correctly converted to a local two-byte `int`.

Element	Typical Length	Meaning
Version	1 byte	The version of SRUP being used — e.g. `0x01`
Message Type	1 byte	The type of SRUP message being sent: `SRUP_MESSAGE_TYPE_INITIATE` taking the value `0x01`
Signature	Protocol Dependent	The cryptographic signature of the message calculated from the control server's private key
UUID	Application Dependent but typically 16 bytes	The universally unique identifier of the device (or device group) for which the message is intended
Token	Application Dependent but typically 16 bytes	A token to uniquely identify this SRUP transaction
URL	Variable	The URL at which the software update can be retrieved
Digest	Protocol Dependent	A secure digest (Hash value) of the file to be retrieved

Table I
THE SRUP INITIATE MESSAGE TYPE

Therefore in order to send the example message shown in table II, the byte-stream to be sent as an MQTT message payload would be as shown in table III.

Note that to aid exposition — the SRUP fields shown here, take unrealistic example values.

Element	Value	Length
Version	`0x01`	1
Message Type	`0x01`	1
Signature	SIG_DATA	8
Target UUID	TARGET	6
Token	TOKEN	5
URL	https://www.example.com	23
Digest	DIGEST	6

Table II
THE ELEMENTS OF AN EXAMPLE SRUP INITIATE MESSAGE

2) Response message: The SRUP `RESPONSE` message also consists of a custom binary payload sent in an MQTT message. The `RESPONSE` payload consists of:

- The transaction token supplied in the corresponding initiate message (so that the server could track the message that the device was responding to)
- A status message indicating whether or not the process had been successful
- A cryptographic signature to ensure that the message originated from the real intended target device

Hex Data	Meaning
01	Version
01	Message Type
00 08	Length of Signature (8)
53 49 47 5F 44 41 54 41	ASCII String: SIG_DATA
00 06	Length of Target (6)
54 41 52 47 45 54	ASCII String: TARGET
00 05	Length of Token (5)
54 4F 4B 45 4E	ASCII String: TOKEN
00 17	Length of URL (23)
68 74 74 70 73 3A 2F 2F 77 77 77 2E 65 78 61 6D 70 6C 65 2E 63 6F 6D	ASCII String: https://www.example.com
00 06	Length of Digest (6)
44 49 47 45 53 54	ASCII String: DIGEST

Table III
AN EXAMPLE OF THE RAW BYTES OF A SRUP INITIATE MESSAGE IN HEXADECIMAL NOTATION

The elements of the response message are shown in table IV.

Element	Typical Length	Meaning
Version	1 byte	The version of SRUP being used — e.g. 0x01
Message Type	1 byte	The type of SRUP message being sent — SRUP_MESSAGE_TYPE_RESPONSE taking the value 0x02
Signature	Protocol Dependent	The cryptographic signature of the message calculated from the device's private key
Token	Application Dependent but typically 16 bytes	The token specified in the previous SRUP initiate message
Status	1 byte	A value indicating the success of the update — or conveying the reason for the failure

Table IV
THE ELEMENTS OF THE SRUP RESPONSE MESSAGE

The values which can be taken by the status byte are shown in table V.

As with the initiate message — the length of the token and signature will be variable, and thus will be preceded by the length of each, in bytes.

3) Activate message: Once the server has received the response from the device signifying successful completion of the retrieval, it can then send a message instructing the device to apply the new software or configuration. Again this

Identifier	Value	Meaning
SRUP_UPDATE_SUCCESS	0x00	Update data successfully received
SRUP_UPDATE_FAIL_SERVER	0xFD	Update unsuccessful — HTTPS server did not respond
SRUP_UPDATE_FAIL_FILE	0xFE	Update unsuccessful — the specified file could not be retrieved from the server
SRUP_UPDATE_FAIL_DIGEST	0xFF	Update unsuccessful — hash value of the retrieved file did not match

Table V
VALUES FOR THE SRUP RESPONSE STATUS BYTE

is sent as a custom message payload using similar principles to the other message types illustrated in this example. The activate message requires the simplest payload: as it only needs to send the transaction token, and a signature. The elements of the message are shown in table VI.

Element	Typical Length	Meaning
Version	1 byte	The version of SRUP being used — e.g. 0x01
Message Type	1 byte	The type of SRUP message being sent — SRUP_MESSAGE_TYPE_ACTIVATE taking the value 0x03
Signature	Protocol Dependent	The cryptographic signature of the message calculated from the control server's private key
Token	Application Dependent but typically 16 bytes	The token specified in the previous SRUP initiate & response messages

Table VI
THE ELEMENTS OF THE SRUP ACTIVATE MESSAGE

Depending on the requirements of the implementation, the device could indicate that it has received and acted upon the message by sending second response message. Additional RESPONSE statuses could be added to the standard to represent this.

D. Additional SRUP capabilities

Although the basic implementation of SRUP is as described, further work is planned to expand the protocol.

1) SRUP for other types of C2 message: In addition to using SRUP for software updates — the same idea can be used to send other types of system-level Command & Control message.

Using SRUP in this way makes it extremely hard for a malicious party to spoof the C2 messages: as only validly signed messages would be acted upon — with the protocol

enforcing the checking of the integrity and veracity of the retrieved data.

Any of the types of C2 message described in section II could be incorporated into SRUP by simply adding additional message types to be used to send these instructions — together with cryptographic signatures and (where applicable) digests to provide a means to verify the authenticity of the messages.

2) SRUP over HTTPS: For devices which would spend much of their time asleep — it would also be possible to implement SRUP over HTTPS: having the device poll a web-service to retrieve messages at a suitable frequency, and entering a very-low power *sleep* mode in between.

Whist reception of this message could initiate an MQTT connection to a specified broker; for such occasional traffic it may be preferable to use HTTPS and to reimplement the underlying SRUP protocol using JavaScript Object Notation (JSON) to transport the elements of the message.

V. Conclusion

In order to demonstrate the potential utility of SRUP a C++ library has been written to implement the protocol, and this has been used together with the Mosquitto broker; and an example device-side daemon and C2 server. Both the daemon & server have been implemented using C++ components, and Python scripts: with the interprocess communication handled using Apache Thrift [7]. Initial testing has shown the protocol to be effective; and given the security afforded by the use of cryptographic signatures within the SRUP messages, the use of an encrypted MQTT connection is only required for situations where C2 information needs to be secured against eavesdropping. Further work is in progress to extend the protocol to cover some of the other types of C2 message described in section II.

The C++ source code for a library implementing the SRUP protocol is available, and is released under the terms of the MIT open licence [8].

The initial release of the source-code for the software library can be found at:

doi:10.5258/SOTON/401895

The source code for the latest version of the library and an example implementation of the device-side daemon, and C2 server can be found at:

https://github.com/dstl/SRUP/

Acknowledgments

The lead author is a Principal Computer Scientist with the UK Ministry of Defence Science & Technology Laboratory (Dstl) who have funded this research. The authors would like to thank Dstl for their co-operation & support with this research.

References

[1] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," Gaithersburg, MD, Aug. 2015. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[2] A. Banks and R. Gupta, "MQTT Version 3.1.1," Oct. 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[3] J. Postel, "Transmission Control Protocol," Defense Advanced Research Projects Agency, Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[4] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," Internet Engineering Task Force, Tech. Rep. RFC5246, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5246

[5] Mosquitto.org. (2016) Mosquitto – An Open Source MQTT v3.1/v3.1.1 Broker. [Online]. Available: http://mosquitto.org

[6] Amazon Web Services. What Is AWS IoT? - AWS IoT. [Online]. Available: http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[7] M. Slee, A. Argawal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Report, 2007. [Online]. Available: https://thrift.apache.org/static/files/thrift-20070401.pdf

[8] MIT Open Licence. [Online]. Available: https://opensource.org/licenses/MIT