Building Hardened Internet-of-Things Clients with Language-theoretic Security

Prashant Anantharaman*, Michael Locasto†, Gabriela F. Ciocarlie†, Ulf Lindqvist†

*Dartmouth College, Hanover, New Hampshire
prashant.anantharaman.gr@dartmouth.edu
†Internet of Things Security and Privacy Center, SRI International
{michael.locasto, gabriela.ciocarlie, ulf.lindqvist}@sri.com

***Abstract*—Unprincipled input handling has caused many of the most prevalent and severe vulnerabilities in the Internet era, and this trend appears to continue in the emerging Internet of Things (IoT). In this paper, we present a methodology to build secure input-handling functionality for application-layer IoT protocols by applying the Language-theoretic Security (LangSec) philosophy. We have built working implementations for the XMPP and MQTT protocols and demonstrated that our clients, which consist of less than a hundred lines of code, correctly recognize all valid messages in our tests. With respect to CPU time, our clients compare well against the most widely deployed implementations of these two protocols.**

I. Introduction

Current estimates show that the *Internet of Things (IoT)* will soon have billions of deployed devices. Approximately 50 million smart meters are in households in the USA at the moment. Several taxi companies, natural gas pipelines and industrial control systems make use of some of the popular IoT protocols. Some of these applications use personally identifiable information and any attack could lead to privacy concerns. Moreover, the widespread deployment of these protocols increases the probability of an attack as several recent attacks like the Mirai botnet [1] indicated. In the rush to deploy these IoT services, IoT vendors have limited or no focus on the required security mechanisms for their architectures.

Parser bugs in the recent years have been infamous – Heartbleed, Android Master Key, Apple's Goto. Parser bugs have been haunting the Internet for the past several years, and the same trend is expected to continue in the IoT. Our current *Internet* works on a "penetrate and patch" paradigm and, if the same paradigm is applied to the IoT, it would continue to lead to more parser-based vulnerabilities in the future [2]. Hence, we propose using *Language-theoretic security* to build hardened IoT end-devices.

Language-theoretic security comes from the idea that many of the input recognition vulnerabilities could be avoided by treating all input as a *language* and completely recognizing it before processing it. One possible approach would be to use a parser generator like lex and $yacc$, which requires a considerable amount of programming effort and makes it very difficult to audit the code.

*Prashant Anantharaman was at SRI International, Menlo Park when this work was performed.

Hammer [3] is a parser combinator toolkit that has been designed to aid developers in building code that can enforce input validation rules. This paper discusses how *hammer* could be used along with protocol state machines to build hardened implementations of IoT clients.

Summary of Contributions

- Our implementation demonstrates the efficacy of LangSec for Internet-of-Things protocols. Our technique could be deployed on IoT devices in the future.
- We show that the effort required to implement our technique is very reasonable given the security benefits offered to the IoT setting.

The structure of the paper is as follows. We first summarize the XMPP and MQTT protocols and provide a brief overview of bugs found in these protocol implementations. We then describe how our state-machine and parsers work together and discuss the limitations of our approach. Finally, we provide a comparison between our clients and existing relevant ones and take a glimpse at future work.

II. Background and Related work

The LangSec approach to security focuses on recognizing and handling all input safely. This functionality can be enforced by using a parser combinator toolkit like *Hammer*, which has bindings for several languages like C, C++, Python, Ruby and Java. To perform the task of building these hardened clients, we need to have a deep understanding of the target protocols.

A. Understanding application layer IoT protocols

XMPP. XMPP is a popular chat protocol that runs on the TCP/IP stack, and supports and enforces the use of TLS [4]. All clients connect to central servers.

XMPP messages are XML streams that need XML parsers. A stream is negotiated between a client and a server. The server supports some optional features, namely, *binding*, TLS, and *SASL authentication*.

A trimmed version of XMPP could be a perfect fit for the IoT world [5]. Service infrastructures for the IoT based on XMPP have been proposed in the past [6]. XMPP maintainers have agreed to make use of TLS v2, disable the support

DOI 10.1109/SPW.2017.36

for SSL v2, and make use of STARTTLS [7]. Despite the support for TLS, crafted messages could lead to exploits due to the lack of a clear parser and recognition boundary (shotgun parsers). Several secure architectures for XMPP have been proposed in the past [8], [9], [10], but none of them addresses the problem of shotgun parsers and untrusted input handling in XMPP implementations, which could have devastating repercussions in the IoT.

Some examples of XMPP messages are presented below.

```
<message type="chat" id="purpleb4dbd712"
to="arthur@sri.lit/host-134"
from="alice@sri.lit/XYZ567">
<active xmlns=
"http://jabber.org/protocol/chatstates"/>
<body>This is a sample XMPP message.
</body>
</message>

<success
xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

The $< message >$ is used to send a message from one client to another after the stream negotiation is complete. The $< success >$ message is received by a client from a server if the SASL authentication was successful. The other relevant messages are $< stream >$, $< stream - features >$, $< proceed >$, $< auth >$ and $< bind >$.

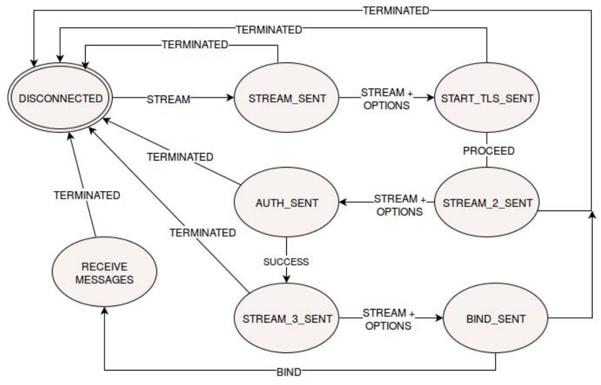

Fig. 1: XMPP protocol state machine.

MQTT. MQTT is an asynchronous publish-subscribe architecture that works via a broker and was designed to consume less battery and bandwidth [11]. MQTT supports a limited number of messages - *Connect, Connect Ack, Subscribe Request, Subscribe Ack, Ping Request, Ping Response* and *Publish Message*. The message type is specified in the first 4 bits of the MQTT packet. The format of the MQTT packet is described in Figure 2. To initiate a connection with a broker, an MQTT client only needs to send one *Connect* message, in contrast to the large number of messages in XMPP. The client subscribes to channels using the *subscribe* message. The server sends messages *Connack, Suback, Pingresponse*. Figure 3 shows the state machine of the MQTT protocol.

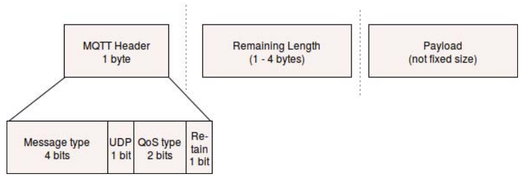

Fig. 2: MQTT packet structure.

Hunkeler et al. proposed modifications to MQTT to be used in wireless sensor networks. MQTT has come under a lot of scrutiny for not being implemented correctly, with plenty of production systems not enforcing TLS or password-based authentication [12].

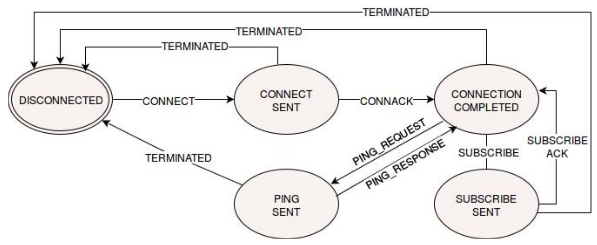

Fig. 3: MQTT protocol state machine. The ovals represent states and the arrows represent MQTT messages.

We have seen past work on designing authorization mechanisms and making use of TLS to secure MQTT [13] [14]. Attribute-based encryption has been used in MQTT to encrypt MQTT packets for different channels [15]. Again, none of this work addresses the issue of handling untrusted input.

B. LangSec and Protocol State Machines

Shotgun parsers. Shotgun parsers perform data checking, handling, and processing, interspersed with each other. The shotgun parser pattern has led to several recent vulnerabilities as context-sensitive data formats can be dangerous and not easy to handle and recognize [16]. Instances of untrusted data propagation due to shotgun parsers were also found in Android applications in the past [17]. Shotgun parsers are only one of the possible *Langsec*-related weaknesses [18] [19].

Parsing errors encountered in XMPP and MQTT. Parsing errors could lead to memory corruption and logic errors, which could in turn lead to severe security vulnerabilities. We studied the CVE database [19] for the ”MQTT” and ”XMPP” search strings and analyzed the type of vulnerabilities detected between 2013 – 2016. We found at least 14 vulnerabilities in XMPP implementations, out of which only two were cryptography-related (i.e., improper implementation of STARTTLS). Almost all other vulnerabilities included crafted XMPP input. The search string ”MQTT” also yielded similar, but fewer results. MQTT had a higher percentage of crafted input vulnerabilities as well.

Table I summarizes a comparison of parser-related errors and cryptographic vulnerabilities in popular MQTT and XMPP implementations from the CVE database.

Client	Protocol	Vulnerabilities from Parsing Errors	Cryptographic Vulnerabilities
Prosody IM	XMPP	4	0
Cisco Jabber	XMPP	4	1
Pidgin	XMPP	3	0
Smack IM	XMPP	1	1
IBM Message Sight	MQTT	2	0
WebSphere MQ	MQTT	1	0

TABLE I: Classification of vulnerabilities found in popular application-layer IoT protocols from 2013 – 2016 on Common Vulnerabilities and Exposures (CVE).

Protocol State Machines. In this paper, we also make use of protocol state machines to define contextual parsers. In the past, there has been work in using protocol state machines for hardening protocols. Poll et al. [20] describe the difference in the state machines in various implementations of openssl. Graham et al. [21] state that finite state machines are sufficiently expressive for Internet protocols and are sufficiently performing for high-throughput applications.

Our work combines the concept of *protocol state machines* and the use of *parser combinators* to harden the implementations of IoT clients by making them less susceptible to input-processing vulnerabilities.

III. Methodology

In this section, we provide a comprehensive tutorial of how the state machine and the parsers hook together, and take a look at the limitations of this approach.

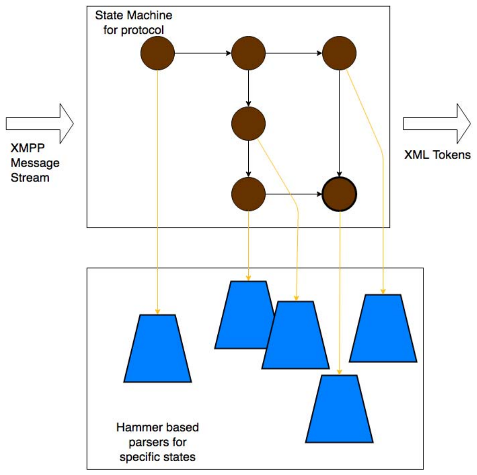
State Machine for protocol
XMPP Message Stream
XML Tokens
Hammer based parsers for specific states

Fig. 4: Overall architecture of a client: Depending on the current state, a different parser is called to recognize the message.

Our implementation is in **ruby**, and makes use of the *hammer ruby bindings* [3] and the *state_machines gem* [22], and hence assumes the correctness of both these implementations.

Figure 4 gives an overview of the architecture of our IoT client. We implement the state machine defined in Figure 1. There are separate input recognizers for each of the states, which get called when the client receives a message at a particular state. Only once fully recognized, this input is further processed.

Our process of building a client involves four steps:

- Construct a simplified state machine from protocol specifications.
- Build the state machine using the $state\_machine$ gem.
- Identify the input language for each state.
- Define parsers for each of the receiving states of the state machine.

Constructing the protocol state machine. A protocol state machine is constructed from the specifications. To simplify the finite state machine, we need to enforce a few cases. For example, in our implementation of XMPP and MQTT, we enforce TLS and password based authentication because we think it is the right approach to follow as per the guidelines. The states and transitions that need to be handled in the case of not enforcing TLS and passwords can now be ignored. We make use of the *state-machine ruby gem* [22] to this effect. The gem includes a set of test helpers that could be used to aid in development as well. We implement the state machines in Figures 1 and 3.

The state machine gem could be used in this fashion with *before_transition* and *after_transition* methods used to trigger methods.

```
state_machine :state, initial: :start do
  before_transition on: :stream_1_sending,
    do: :stream_1_sent_call
  before_transition on: :start_tls_sending,
    do: :start_tls_sent_call
  before_transition on: :ssl_negotiation,
    do: :ssl_negotiation_call
  before_transition on: :auth_sending,
    do: :auth_sent_call
  before_transition on: :bind_sending,
    do: :bind_sent_call
  after_transition on: :quitting,
    do: :quit_call
  event :stream_1_sending do
     transition [:start] =>
        :stream_1_sent
  end
  event :stream_1_received do
     transition [:stream_1_sent] =>
        :stream_1_received
  end
end
```

The above code snippet shows a part of the implementation of the state machine. The event methods show transitions between states. At each state that is receiving a message, we need to fully recognize the message received with the help of a parser.

Defining the receiving state parsers. We need to define a separate grammar for the messages we are expecting for each receiving state. For example, in the implementation of the XMPP protocol, if we sent the *stream* stanza to initiate an XMPP connection, we should be receiving a *stream* stanza back from the server along with some options. We need to use a parser to make sure this message is completely recognized. We need to define the language that is to be accepted at every state. Let us first look at the grammar for simple $< stream >$ messages.

$$
\begin{aligned}
&whitespace \rightarrow \backslash x20\\
&doublequote \rightarrow \backslash x22\\
&newline \rightarrow \backslash x5c\ \backslash x6e\\
&tab \rightarrow \backslash x5c\ \backslash x74\\
&gaps \rightarrow whitespace \mid tab \mid newline\\
&many\_gaps \rightarrow gaps \mid gaps\ gaps\\
&zero\_gaps \rightarrow \mathcal{E} \mid gaps \mid gaps\ gaps\\
&starttag \rightarrow zero\_gaps\ \backslash x3c\ zero\_gaps\\
&start\_closetag \rightarrow zero\_gaps\ \backslash x3c\\
&zero\_gaps\ \backslash x2f\ zero\_gaps\\
&endtag \rightarrow zero\_gaps\ \backslash x3e\ zero\_gaps\\
&closetag \rightarrow zero\_gaps\ \backslash x2f \backslash x3e\ zero\_gaps\\
&stream\_word \rightarrow \backslash x73\ \backslash x74\ \backslash x72\ \backslash x65\ \backslash x61\ \backslash x6d\\
&stream\_open\_tag \rightarrow starttag\ stream\_word\ endtag\\
&stream\_close\_tag \rightarrow start\_closetag\ stream\_word\ endtag
\end{aligned}
$$

```
@hammer = Hammer::Parser
whitespace = @hammer.ch('\x20')
doublequote = @hammer.ch('\x22')
newline = @hammer.token("\x5c\x6e")
tab = @hammer.token("\x5c\x74")
gaps = @hammer.choice(whitespace, tab, newline)
many_gaps = @hammer.many1(gaps)
zero_gaps = @hammer.many(gaps)
starttag = @hammer.sequence(zero_gaps,
        @hammer.ch('\x3c'), zero_gaps) # "<"
start_closetag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x3c'), zero_gaps,
    @hammer.ch('\x3e'), zero_gaps # "</"
endtag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x3e'),
    zero_gaps) # ">"
closetag = @hammer.sequence(zero_gaps,
    @hammer.ch('\x2f'), @hammer.ch('\x3e'),
    zero_gaps) # "/>"
stream_word = @hammer.token("\x73\x74\x72
        \x65\x61\x6d") # "stream"
stream_open_tag = @hammer.sequence(starttag,
                  stream_word,
                  endtag) # <stream>
stream_close_tag = @hammer.sequence(start_closetag,
                   stream_word,
                   endtag) # </stream>
```

The grammar defines whitespaces, double-quotes and all the other special characters as separate productions to help reusage of the combinators. The $stream\_word$ parser defines the word *stream* in the hexadecimal notation. The $starttag$ production has the character $<$ and the $endtag$ has the character $>$. These productions can be reused while defining productions for the XML tags. All the tags allow zero or more spaces or special characters surrounding the tag symbols. This grammar was converted to hammer parser-combinator code in *ruby*, and is shown in the code above.

It is important to understand the combinators provided by hammer to represent grammar. The *many* combinator is used to apply the combinator given as an argument zero or more times. We use the *many1* combinator if it has to be used one or more times. The *sequence* combinator is used to group these combinators together in a sequence. The *ch* combinator is used to represent a single character and the *token* combinator is used to represent a string. We make use of these basic building blocks to convert the grammar to the hammer parser combinator code.

Validating input. We will now look at how we use the *hammer parsers* we have already defined to recognize a message given as an argument. The hammer parser provides an object of class *HParseResult* if the parsing was successful, and a *nil* if the parsing was not successful. Upon successful recognition, we need to fire events to perform certain actions. In the example below, we are recognizing a message with the $< stream >$ message, and fire an event if the parsing was successful.

```
def validate_stream(args)
    if !stream_open_tag.parse(args).nil?
        true
    else
        false
    end
end

def stream_1_received(message)
    if validate_stream(message) and
          state == "stream_1_sent"
        fire_events(:starttls_sending)
    end
end
```

The above method would fire the *starttls_sending* transition, if the message was recognized fully and correctly.

File organization. As per the LangSec philosophy, all the input validators are placed in a separate class. These validators return a boolean if given a string to parse. These validators are called from all other files as and when input recognition is needed. Placing the validators in a separate class serves two purposes. Firstly, it helps in re-usability of parser-combinator code. Secondly, it could aid developers and people performing code audits to easily identify mistakes in input recognition.

Limitations

- We can only define context-free grammars and regular expressions using parser-combinator toolkits. Hence, if the language to be recognized by our parser is either a context-sensitive grammar, or a Turing-complete language, we would have to simplify it down to a context-free grammar or a regular expression resulting in loss of functionality. However, we argue that this is essential since Turing-completeness or context-sensitivity of a language only results in more input processing errors.
- Since our implementations were lightweight, we see that our implementations were faster than a few other widely used open-source clients. In general, a layer to recognize input completely adds some overhead time. However, this cost is reasonable considering the benefits of this approach.

IV. Evaluation and Discussion

To evaluate our implementation of the *MQTT* and *XMPP* protocols, we run timing experiments to determine how our implementation performed in comparison to other implementations. We also perform unit and preliminary fuzz testing, and describe our methodology.

A. Experiments

We compare the performance of our implementation and other open source implementations of XMPP and MQTT available. We analyze SleekXMPP [23] for python, Smack [24] for Java, and Xrc [25] for ruby. We compare the total time to reach a connected state. These experiments were performed on a Raspberry Pi 2. For the server side of the XMPP protocol, we make use of *Vines - An XMPP chat server* [26]. In a similar setup, for the MQTT protocol, we use *ruby-mqtt*, *pyMQTT* and the *Mosquitto broker*.

Client	CPU Time
Our implementation	0.42 s
Xrc	0.59 s
Smack	0.30 s
QXMPP	0.41 s
SleekXMPP	0.90 s

TABLE II: Comparison of average time to connect to the XMPP server.

Client	CPU Time
Our implementation	218 μs
ruby-mqtt	2.3 ms
pyMQTT	1.2 ms

TABLE III: Comparison of average time to connect to the MQTT broker.

In Tables II and III, we observe that our clients run in comparable time to most other XMPP and MQTT clients. Our clients are comparatively light weight and need fewer features than the traditional clients; hence, the similar CPU time values.

Message	CPU Time	Number of lines in hammer
Stream 1	17 ms	39
Proceed	18 ms	20
Stream 2	27 ms	39
Success	3 ms	20
Stream 3	24 ms	39
Bind	11 ms	25

TABLE IV: Comparison of average time to recognize various XMPP message inputs and the human effort in terms of lines of code.

Message	CPU Time	Number of lines in hammer
Connect Ack	39 μs	6
Subscribe Ack	126 μs	7
Ping received	97 μs	6

TABLE V: Comparison of average time to recognize various MQTT message inputs and the human effort in terms of lines of code. MQTT messages for receiving acknowledgements from the server generally contain just 2 bytes and are easy to parse.

In Tables IV and V, we show that the amount of time used to recognize various inputs is minimal (i.e., in the order of tens of milliseconds).

We also observe that, with just under 250 lines of code, we can implement the input recognizers for a simplified version of XMPP, which indicates that the human effort required for such an implementation is not significant. Writing a separate layer of a parser and defining the protocol state machine explicitly are steps that are necessary to be taken and improve the process of code auditing.

B. Unit testing

We evaluate the correctness of the validation and state machine components using unit testing. We provide results of our unit testing in Table VI. We wrote unit tests for most of the parsers required for the two implementations, achieving a 100% code coverage for the state machine implementations and over 80% code coverage for the other parser implementations.

Protocol	Component	Coverage
XMPP	State machine	100%
XMPP	Parser validation	86.55%
MQTT	State machine	100%
MQTT	Parser validation	81%

TABLE VI: Unit test coverage of our XMPP and MQTT implementations via *coverage gem*.

C. Preliminary fuzzing

To assess the trustworthiness of our input recognizers, we build a simple generational fuzzer to check what input is accepted by our recognizers. The process of generating messages using our fuzzer is defined below:

- We provide the parser-combinator object as an input to our fuzzer generator.
- Our fuzzer generator generates a parse tree from the parser-combinator object.

- Our fuzzer generator generates messages by traversing the parse tree and emitting all possible strings.
- These sequences of messages are sent to our client and tested for correctness.

We use this technique to build a dataset of possible messages at each state of the protocol state machine. Our client was fed a large series of these sequential messages and it did not crash for any of these sequences; hence, showing some resilience. However, we acknowledge that such fuzzers need to be targeted towards dependencies between fields and protocol boundaries. We leave this investigation as future work.

V. Lessons Learned

We set out with the goal of demonstrating the efficacy of using LangSec in IoT protocols. We make use of the *hammer parser-combinator toolkit* to this end. It proved very easy for us to convert a context-free grammar to a parser-combinator implementation. The hammer library includes bindings for several programming languages and we made use of the ruby bindings for our implementations.

The specifications of protocols are in plain-text and are lengthy. Given the LangSec methodology, we have to read through these specifications and design a language compliant with the protocol specifications. Specifications could be more exact if they specified the protocol state machine and the input language for each receiving state. This would prevent parser differentials.

We found that context-free grammars for each receiving protocol state were sufficient to recognize XMPP and MQTT messages. However, if a language is context-sensitive or Turing-complete, it has to be simplified and constrained to be a context-free language, such that it can be fully recognized. Shotgun parsers are very prevalent and are one of the major causes of zero days. Thus, we recommend that commercial IoT applications make use of the LangSec design to avoid a number of zero days.

The hammer parser-combinator toolkit fared reasonably well in a constrained environment, showing that the overhead due to the recognizer was very negligible. The programmer does not have to write a lot of additional lines of code. The generation of the state machine could be automated with a tool to further reduce programmer effort. We simplified the language of the protocols to make it more strict, by enforcing rules that were otherwise optional in the protocol. This would reduce functionality of the protocol, but, at the same time, improve the security of the protocol.

VI. Conclusion

In this paper, we address the widespread problem of unprincipled input handling in IoT clients. We explore some of the popular application-layer IoT protocols and then discuss some of the recently known vulnerabilities in IoT protocols. We then describe the methodology used to build LangSec-compliant IoT clients. We analyze our implementation and show that the performance cost and the human effort in terms of lines of code added due to the hammer-based parser are a very reasonable price to pay for security. We generated large datasets of valid sequential XMPP and MQTT messages to see for what input our clients broke, and found no input that our implementation could not handle.

Our future work will include a parser-combinator-based smart fuzzer, which will generate targeted datasets exploiting the boundaries of the protocol, and building formally verified LangSec parsers for various protocols.

Acknowledgement

We thank Dr. Martin Schäf and Dr. Dejan Jovanovic for their initial thoughts regarding formal verification. This work was supported by SRI International's strategic independent R&D initiative on IoT security and privacy.

References

[1] B. Herzberg, D. Bekerman, and I. Zeifman, "Breaking Down Mirai: An IoT DDoS Botnet Analysis," 2016. [Online]. Available: https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html

[2] K. Palani, E. Holt, and S. Smith, "Invisible and forgotten: Zero-day blooms in the IoT," in *IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, March 2016, pp. 1–6.

[3] M. L. Patterson, "Hammer: Parser Combinators in C," 2016. [Online]. Available: https://github.com/UpstandingHackers/hammer

[4] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Address Format," 2015.

[5] M. Kirsche and R. Klauck, "Unify to bridge gaps: Bringing XMPP into the internet of things," in *IEEE International Conference on Pervasive Computing and Communications (PerCom) Workshops*. IEEE, 2012, pp. 455–458.

[6] S. Bendel, T. Springer, D. Schuster, A. Schill, R. Ackermann, and M. Ameling, "A service infrastructure for the internet of things based on xmpp," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2013, pp. 385–388.

[7] P. Saint-Andre, "A Public Statement Regarding Ubiquitous Encryption on the XMPP Network," 2014. [Online]. Available: https://github.com/stpeter/manifesto/blob/master/manifesto.txt

[8] S. N. Foley and W. M. Adams, "Trust management of xmpp federation," in *IFIP/IEEE International Symposium on Integrated Network Management*, May 2011, pp. 1192–1195.

[9] D. Conzon, T. Bolognesi, P. Brizzi, A. Lotito, R. Tomasi, and M. A. Spirito, "The VIRTUS Middleware: An XMPP Based Architecture for Secure IoT Communications," in *International Conference on Computer Communications and Networks (ICCCN)*, July 2012, pp. 1–6.

[10] A. Celesti, M. Fazio, and M. Villari, "Se clever: A secure message oriented middleware for cloud federation," in *IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2013, pp. 35–40.

[11] IBM, "Mqtt version 3.1." *Standard, OASIS*, 2014.

[12] L. Lundgren, "Light-Weight Protocol! Serious Equipment! Critical Implications!" *DEFCON*, 2016.

[13] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya, "Authorization mechanism for mqtt-based internet of things," in *IEEE International Conference on Communications Workshops (ICC)*. IEEE, 2016, pp. 290–295.

[14] Y. Upadhyay, A. Borole, and D. Dileepan, "Mqtt based secured home automation system," in *Symposium on Colossal Data Analysis and Networking (CDAN)*, March 2016, pp. 1–4.

[15] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar, "Secure mqtt for internet of things (iot)," in *International Conference on Communication Systems and Network Technologies (CSNT)*, April 2015, pp. 746–751.

[16] S. Bratus, M. L. Patterson, and D. Hirsch, "From shotgun parsers to more secure stacks," *Shmoocon, Nov*, 2013.

[17] K. Underwood and M. E. Locasto, "In Search of Shotgun Parsers in Android Applications," in *IEEE Security and Privacy Workshops (SPW)*, May 2016, pp. 140–155.

[18] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *IEEE Cybersecurity Development Conference (IEEE SecDev)*, November 2016.

[19] "CVE – Common Vulnerabilities and Exposures." [Online]. Available: https://cve.mitre.org/

[20] E. Poll, J. D. Ruiter, and A. Schubert, "Protocol State Machines and Session Languages: Specification, implementation, and Security Flaws," in *IEEE Security and Privacy Workshops*, May 2015, pp. 125–133.

[21] R. D. Graham and P. C. Johnson, "Finite State Machine Parsing for Internet Protocols: Faster Than You Think," in *IEEE Security and Privacy Workshops*, May 2014, pp. 185–190.

[22] A. Boudih, "Creating state machines for attributes on any Ruby class," 2016. [Online]. Available: https://github.com/state-machines/state$_machines$

[23] N. Fritz, "SleekXMPP Python Jabber/XMPP implementation," 2016. [Online]. Available: https://github.com/fritzy/SleekXMPP

[24] T. Evans, D. Olajide, F. Schmaus, J. Sipula, and G. der Kinderen, "A highly modular and portable open source XMPP client library written in Java," 2015. [Online]. Available: http://www.igniterealtime.org/projects/smack/

[25] R. Nakamura, "Xrc - XMPP Ruby Client," 2014. [Online]. Available: https://github.com/r7kamura/xrc

[26] D. Graham, "Vines - XMPP/Jabber chat server for Ruby," 2014. [Online]. Available: http://www.getvines.org