Robustness Testing of CoAP Server-side Implementations through Black-box Fuzzing Techniques

Bruno da S. Melo[1], **Paulo Lício de Geus**[1]

[1]Instituto de Computação (IC) – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einstein, 1251 – CEP 13083-970 – Campinas – SP – Brazil

{brunom,paulo}@lasca.ic.unicamp.br

Abstract. *This paper presents the current status of our research on the robustness of CoAP server-side implementations. We discuss the importance of the CoAP protocol as an enabler of the Internet of Things (IoT) vision, and also the current state of CoAP implementations available out there. Then, we proceed to test those implementations using fuzzing techniques previously used in the literature in areas such as Web Service and Network Protocol security testing, namely Random, Mutational and Generational Fuzzing, both "dumb" and "smart". Finally, we provide preliminary results and analysis regarding i) how robust the CoAP implementations studied are and ii) how the different fuzzing techniques used compare to each other.*

1. Introduction and Motivation

The promise of the Internet of Things (IoT) is to increase the efficiency of our lives by providing new, valued-added services through the integration of several technologies and communications solutions, so our society could benefit from a wide range of applications in areas such as agriculture, manufacturing, city infrastructure (e.g. mobility & transportation, energy & water distribution, environment monitoring), retail, logistics, healthcare, home & building, and many others [1]. This vision leads to a very heterogeneous environment, in terms of both hardware and software [Atzori et al. 2010]. Also, considering the complex, heterogeneous scenarios IoT networks may lead us to, there are security and privacy concerns that need to be addressed before this new paradigm can be widely accepted by the public [Granjal et al. 2015].

Even so, the amount of connected devices is already growing, with current estimates ranging from 20.8 to 30.7 billions for 2020 [2]. Regardless of the exact figure and despite the fact that the "*real* IoT" vision—with uniquely identifiable edge nodes running IPv6, reachable through a Service-Oriented or Resource-Oriented Architecture (ROA or SOA)—is not realized yet, we can find reportings of vulnerable devices in the "ad-hoc, *fake* IoT"—with devices mostly running HTTP over IPv4 behind NAT or proprietary stacks—both found in a lab [Patton et al. 2014] as well as in the wild [3] [4], the biggest and most famous one probably being the Mirai botnet [5].

[1]http://www.libelium.com/libelium-smart-world-infographic-smart-cities-internet-of-things/

[2]http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated

[3]http://www.crn.com/slide-shows/internet-of-things/300084663/8-ddos-attacks-that-made-enterprises-rethink-iot-security.htm

[4]http://www.checkpoint.com/press/2014/media-alert-check-point-researchers-discover-isp-vulnerabilities-hackers-use-take-millions-consumer-internet-wi-fi-devices/

[5]https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/

Although a number of standards for IoT communications have been proposed in the recent years, and none of them have become a *de facto* standard yet, the protocol stack composed by IPv6 over Low-Power Wireless Personal Area Networks (**6LoWPAN**s), IPv6 Routing Protocol for Low-Power and Lossy Networks (**RPL**), and the Constrained Application Protocol (**CoAP**) [Shelby et al. 2014] running over **IEEE 802.15.4** has been thoroughly explored by the research community, due to its focus on enabling direct end-to-end integration of edge devices—and initially isolated Wireless Sensor Networks (WSNs)—to the Internet [Palattella et al. 2013, Granjal et al. 2015]. A parallel between this protocol stack and the current Internet stack can be seen in Figure 1.

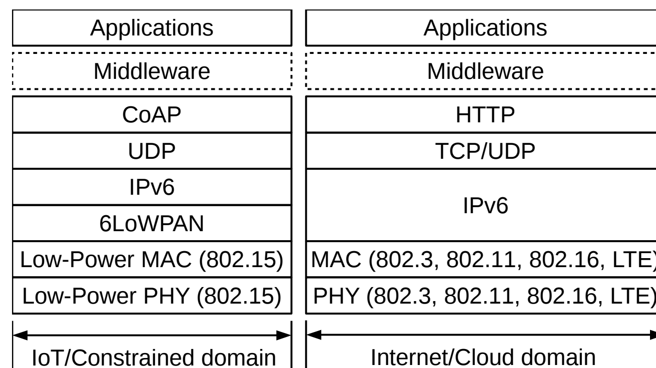

Figure 1. 6LoWPAN-based stack for IoT in comparison with the traditional protocol stack used in the Internet.

Running at the application layer, CoAP is a RESTful protocol featuring a well-defined mapping to HTTP and its methods (`GET`, `POST`, `PUT` and `DELETE`), and supporting multicast, Universal Resource Identifiers (URIs), content-type identification, resource discovery, response codes and simple subscription and caching mechanisms. Aiming for simplicity and low overhead, it requires a fixed 4-byte header. Since the messages are transported over UDP, CoAP implements simple reliability and message deduplication mechanisms as well. Additionally to the base specification, several extensions to the protocol were already standardized as well, the most popular ones being Observe [6], Block [7] and CoRE Link-Formats [8].

As IoT devices are exposed to the Internet to allow resource sharing and ubiquitous services provisioning, the CoAP protocol becomes a possible attack vector, which may be exploited by a myriad of vulnerabilities, including protocol parsing, URI processing, proxying and caching, risk of amplification, IP address spoofing, etc. System-wide, these threats include (D)DoS, corrupted nodes, fraudulent packet injection, and the possibility of a malicious entity compromising a node to perform lateral movements on a private network [Mohsen Nia and Jha 2016].

To the best of our knowledge, there is little to no work on the practical security of this "*real* IoT" vision, and most of them focus either on cryptographic approaches [Raza et al. 2013], or attack scenarios performed by previously compromised CoAP nodes in a constrained network [Martins et al. 2016]. Our work, on the other hand,

[6] RFC7641 - Observing Resources in the Constrained Application Protocol (CoAP)
[7] RFC7959 - Block-wise Transfers in the Constrained Application Protocol (CoAP)
[8] RFC6690 - Constrained RESTful Environments (CoRE) Link Format

focus on understanding and finding out the ways in which a CoAP node could be compromised in the first place. To that end, we leverage fuzzing techniques to discover vulnerabilities in available CoAP implementations.

2. Background and Related Work

As previously stated, we found no work on CoAP Robustness or Security Testing. There are, however, a few projects targeting the functional test of CoAP implementations, which we use as a basis for our testing system, especially as an input for the Mutation-based Fuzzer (see PCAP Conversation in Figure 2).

The European Telecommunications Standards Institute (**ETSI**) promotes Plugtest events, in which different organizations can test their own implementations of a given standard, focusing on the interoperability of products and services. Together with other industry members, ETSI has organized four IoT CoAP Plugtests so far, the latest one in March, 2014 [9]. The **F-Interop** project aims to develop and provide online testing tools to perform Conformance, Interoperability, and other non-functional testing, although it does not cover Robustness nor Security testing. A video demonstration of a Proof-of-Concept (PoC) CoAP interoperability testing tool is currently available at the project website [10]. The test specifications used are the ones from ETSI CoAP#4 Plugtest. We note this is an ongoing research project, so further development can be expected. **Peach Fuzzer** is a commercial fuzzer with CoAP support [11]. The tool has an open source Community Version available as well, but CoAP is not supported in that distribution. Codenomicon's **Defensics** is another commercial fuzzer with support for CoAP [12]. This tool has no open source or evaluation version. Table 1 shows a comparison between these testing solutions.

Table 1. Comparison of available CoAP Testing Solutions.

Project	CoRE Reference Specification				Tool Support / Automated Testing	Security Testing
	Base	Observe	Block	Link-Format		
ETSI Plugtests	RFC7252	draft-12	draft-14	RFC6690	- Offline pcap analyzer - Automatic stimuli	-
F-Interop	RFC7252	draft-12	draft-14	RFC6690	- Real-time pcap analyzer - Manual stimuli	-
PEACH Fuzzer	RFC7252	draft-16	draft-17	-	- Fully automated	Mutational Fuzzing
Defensics	RFC7252	-	-	-	- Fully automated	Generational Fuzzing

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. Regarding fuzzing techniques in general, [Oehlert 2005] describes two ways of obtaining testing data for fuzzing—data generation and data mutation—; differentiates between intelligent fuzzers (those that leverage some knowledge of the target format) and unintelligent fuzzers (those that, for instance, just randomly changes bits); discuss common fuzzing problems, such as the care to be taken regarding target formats using hashes or

[9] http://www.etsi.org/news-events/events/741-plugtests-2014-coap4
[10] http://www.f-interop.eu/index.php/tools-experiments
[11] http://www.peachfuzzer.com/wp-content/uploads/CoAP.pdf
[12] http://www.codenomicon.com/products/defensics/datasheets/coap-server.html

checksums; and the difficulties to check if a target application behaves correctly or not—suggesting the use of source code instrumentation and monitoring parameters from the System Under Test (SUT) execution, such as memory and CPU usage. The present study will provide a comparison between those techniques.

Finally, our efforts are guided by work on fuzzing techniques previously applied in Web Service Security testing, such as data perturbation for generating test cases, by [Offutt and Xu 2004], guidelines to build penetration testing tools by [Antunes and Vieira 2016] and work on using Boundary Value Analysis and dictionaries of peculiar strings to test Network Protocols as done by the PROTOS project [13].

3. Experimental Setup

The architecture of our testing system can be seen in Figure 2. Fuzzing engines (responsible for generating fuzzed data or scrambling—mutating—valid data into fuzzed data) and the workload executor (responsible for sending and receiving packets) were based on the Scapy [14] framework. The steps of an experiment instance are as follows:

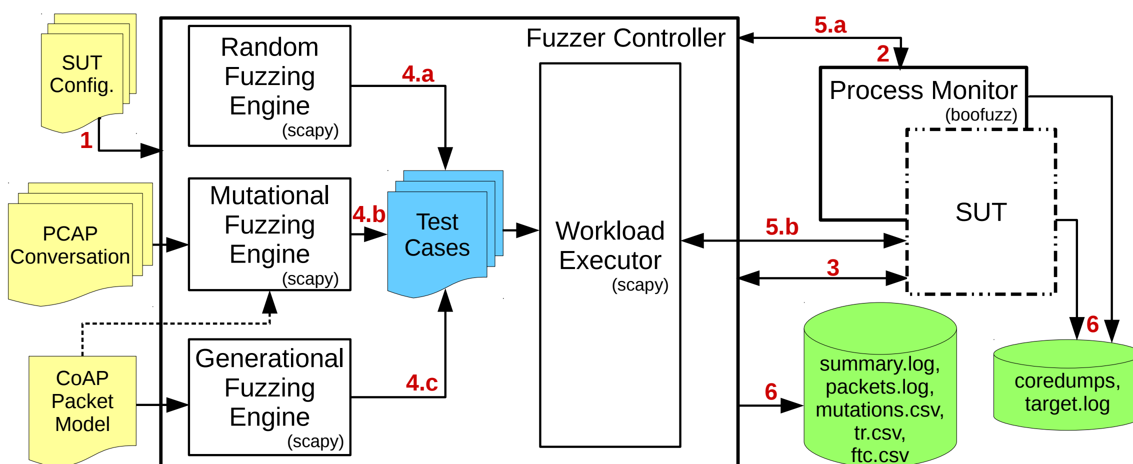

Figure 2. Architecture of the testing system.

1 We fill a configuration file for the given System Under Test (SUT), holding an unique name, the command line to start and stop it, any required environment variables it may need, and parameters regarding how long we should wait for the SUT to (re)start, as well as a target path to be used as heartbeat. Then, we can start our Fuzzer.

2 Our Fuzzer communicates with the Process Monitor [15] (which runs at the target machine—which, in turn, could be the same as the fuzzer machine—listening on a given TCP port), through Remote Procedure Calls (RPC), sending the parameters from the SUT configuration file to it, so the Process Monitor is able to properly start the SUT.

3 Now that the SUT is already running, the first step of our Fuzzer is to send a `GET .well-known/core` CoAP request to the SUT, in order to obtain a list of available links (or paths) the SUT exposes, as well as other relevant information about those links, e.g. if a given link supports the observe feature; the resource type it returns; etc.

[13]https://www.ee.oulu.fi/research/ouspg/Protos
[14]https://github.com/secdev/scapy
[15]This entity was adapted from the Boofuzz tool, available at https://github.com/jtpereyda/boofuzz

4.a If we are experimenting with **Random Fuzzer**, a set of completely random Test Cases (TCs) is generated. We also run experiments with a slightly modified version that we call **Informed Random Fuzzer**, which adds `Uri-Path` options—obtained from **step 3**—to the TCs generated, providing some guidance to the testing process.

4.b Else, if we are running a **Mutational Fuzzer**, a set of TCs is generated from mutations performed on previously captured (through `tcpdump` or Wireshark) CoAP conversations. We also run experiments with a slightly modified version that we call **Smart Mutational Fuzzer**, which uses information from the CoAP Packet Model to perform smarter and/or more complex mutations, as well as the target links from **step 3**. Complex mutations requiring knowledge of the CoAP Packet Model includes option duplication, option removal or any mutation to specific protocol fields.

4.c Otherwise, if we are running a **Generational Fuzzer**, we use techniques such as Boundary Value Analysis, Equivalence Partitioning and dictionaries of carefully-crafted strings to fill out CoAP packet templates based on the CoAP Packet Model, generating smarter TCs to be sent to the SUT. In this mode, we are free to explore any possible CoAP packet, given the inherent knowledge of protocol structure.

5 Workload Execution, sending the generated Test Cases to the SUT. Before actually sending it (**5.b**), the Fuzzer informs the Process Monitor of a new incoming TC using the `pre_send()` method through RPC (**5.a**). After the fuzzed packet is sent, and a possible response packet is received (**5.b**), the Fuzzer performs some checks. If the *packet is not answered*, it is appended to a list, together with its TC number; if this list reaches a threshold (we currently use 5), the Fuzzer will check the SUT's health both through an RPC polling method (`post_send()`, which will contact the Process Monitor to check for process's signals or coredump files related to the SUT), as well as through a heartbeat, which is basically a request known to always work on that SUT (the most common one is just a `GET .well-known/core` request). If the fuzzer establishes that the SUT has crashed, relevant information (coredump file, exit status and signal, last five packets sent, etc.) is saved (see **step 6**) and the SUT is restarted, so the process can continue from **step 5**; else, it just continues from **step 5** again. If, however, the *packet is answered*, the Fuzzer can just continue from **step 5**.

6 Relevant information is always saved throughout the fuzzing process. This includes a summary and a report for each packet template (Generational Fuzzer), mutated value per TC, execution times, etc. Especially useful information relevant to the offline crash analysis described in Section 5 includes the SUT log and coredump files. Additionally, the list of "last sent packets" related to each failed TC can be used to reproduce the failures in a more controlled way, assisting in manual debug for root cause analysis.

4. Target Implementations

Since CoAP is a rather new protocol, we have faced some difficulties trying to find relevant samples to be used as SUTs. We ran an extensive and systematic search against public source code and/or software repositories as well as general search engines and other sources, from which results can be seen on Table 2, together with the number of SUT candidates found and their current status regarding availability to our research. The definition for each status is described in the following paragraphs. We note that these numbers are roughly preliminary, since this study about the current use, status and availability of CoAP implementations is expected to constitute one of our research's contributions.

Table 2. Number of SUT candidates found per Code/Software Repository and/or Search Engines. Only the keyword `'coap'` applies, unless noted.

Repository/Search Engine	SUT Candidates				
	Irrelevant	Hard	Pending	OK	Total
https://github.com[a]	1099	33	122	16	1270
https://bitbucket.org	16	2			18
https://gitlab.com	61	1	3		65
http://coap.technology	24		2		26
https://en.wikipedia.org/wiki/Constrained_Application_Protocol	26		2		28
https://code.google.com/archive	6		2		8
https://sourceforge.net	6	2			8
https://www.openhub.net	13				13
http://www.krugle.org	2				2
https://www.codeplex.com	6				6
http://www.grepcode.com	8				8
https://google.com[b]		2	3		N/A

[a] Includes keywords `'lwm2m'` and `'om2m'` (two Device Management protocols based on CoAP).
[b] Mostly ongoing. For each of the relevant keywords, we intend to check the first 5 pages, i.e. 50 results.

After we find an SUT candidate, we perform a preliminary manual inspection of any available source code and documentation to assess the suitability of that candidate towards our research goals. This assessment includes, but is not limited to i) look for any executable which will be listening on an UDP port expecting CoAP packets (including example applications that ships together with programming libraries for CoAP, demo applications ranging from personal projects to industry or academia prototypes, etc.); ii) check if we have available infrastructure to run such an executable (hardware architecture, supported operating system, etc.); and iii) build, install and run the candidate to test for basic CoAP communication.

Table 3. Short list of SUT candidates.

SUT Candidate	Status	CoAP Library	Lang.	SUT Candidate	Status	CoAP Library	Lang.
aiocoap-rd	OK	aiocoap	Python	mbed-client	Hard	mbed SDK	C++
californium-pt	OK	californium	Java	microcoap-server	OK	microcoap	C
californium-proxy	OK	californium	Java	molex-transcend-gw[a]	Hard	?	?
go-coap-server	OK	go-coap	Go	riot-gcoap-server	OK	gcoap + nanocoap	C
ikea-tradfri-gw[a]	Hard	?	?	rtos-wot-server	Hard	Erbium	C
iotivity-fridge-server	Pending	libcoap	C, C++	swiftcoap-server	Hard	SwiftCoAP	Objective-C

[a] As far as we know, these are the only two commercial products using CoAP out there. IKEA Tradfri[16]is a smart lighting + gateway ecosystem and Molex Transcend[17]is a Power over Ethernet (PoE) gateway.

From that assessment, the SUT candidate is classified into one of the following four statuses, due to—but not limited to—the following reasons: a) Irrelevant - duplicate or search result already found in a previous repository, very minimal (or old) support to CoAP RFCs, no actual executable to listen for packets; b) Hard - missing infrastructure, not possible to follow documentation in order to run it due to a language barrier; c) Pending - looks like a good candidate, only pending the build-install-run step, which can take some time since we may have to deal with different programming languages, building

[16]http://www.ikea.com/gb/en/products/lighting/smart-lighting/
[17]http://www.transcendled.com/webfoo/wp-content/uploads/AI-POE-Gateway.pdf

platforms or other unpredicted adversity; and d) OK - we are able to run it, and the SUT is not a candidate anymore, being ready to be tested. Table 3 shows a few of our relevant SUT candidates so far, including their statuses, CoAP libraries they use and programming languages they are made in. The application environment diversity should be noted, as it is one of our biggest obstacles.

5. Preliminary Experimental Results

In order to find details about the crashes reported by the Fuzzer, we need to investigate the output artifacts generated between steps 5 and 6 from the process described in Section 3.

For TC-related and time-related metrics, we have files holding TC/sec, TC/crash and execution times, to name a few, which can be drilled down by packet template (Generational Fuzzer only) or CoAP option name, supporting different possible types of analysis, e.g. which options cause more crashes throughout all SUTs, etc. We did not perform this kind of analysis yet.

For crash-related information, we can use both the logfile produced by the SUT itself, which is properly marked as part of the process described in Section 3, so we can relate a given piece of output to a known TC number; as well as the coredump files generated. This task is currently automated, but there is still room for improvements, as it is very dependent on the SUT itself, since our parser for the SUT logfile has to deal with information provided by the specific SUT or specific SUT's programming language, for instance. Nevertheless, we show preliminary crash-related results in Table 4.

Table 4. Experimental results for some SUTs.

SUT	Generational Fuzzing						Random Fuzzing[a]				Informed Random Fuzzing[a]			
	Target Paths	Gen. TCs	% TC Exec.	Exec. Time	Crashes Total	Crashes Uniq.	% TC Exec.	Exec. Time[b]	Crashes Total	Crashes Uniq.	% TC Exec.	Exec. Time[b]	Crashes Total	Crashes Uniq.
californium-pt	29	3086634	100%	90h	9	0	100%	30	0	0	100%	34	0	0
coapthon-server	13	1255058	20,6%	22h	20725	7	5,2%	12	100	4	83,7%	35	50	3
ibm-coap-proxy	1	103442	95,2%	3h	209	1	56%	16	100	1	8,1%	6	200	1
jcoap-pt	4	385262	46,4%	7h	4217	14	100%	29	3	1	100%	41	50	3
libcoap-server	3	299746	90,8%	4h30	693	1	100%	30	0	0	100%	33	0	0
libnyoci-pt	7	670358	99,7%	12h30	308	6	100%	31	0	0	100%	37	0	0

[a] 40000 TCs were generated for Random and Informed Random Fuzzing.
[b] Execution Time in **minutes** for Random and Informed Random Fuzzing.

We can see from Table 4 that, although the number of generated TCs is bigger for Generational Fuzzing than for both types of Random Fuzzing—it is proportional to the number of packet templates used, which in turn is proportional to the number of paths exposed by the SUT—, incurring in larger execution times, this technique is also able to detect more crashes, for most cases. We also observe the number of recurrent crashes. We are currently using a simple heuristics in which if a given parameter, such as an specific packet template or CoAP option causes a threshold number of crashes, we skip to the next template/option, trying not to lose much time on potential duplicated crashes; this is what causes the round numbers in total crashes for both types of Random Fuzzing.

6. Final Considerations

Preliminary results shows the ability of fuzzing techniques in uncovering crashes during CoAP servers' execution and testing, which in turn represents potential security vulnerabilities in those servers (observing that a crash already threatens the availability property

of security anyway). It also shows that a more informed, elaborate technique, such as generational fuzzing, is able to expose crashes not exposed by random fuzzing. Even so, our research currently have limitations regarding the relevance and number of our samples (the SUTs), due to no widespread use of CoAP so far. Additionally, although the Mutational Fuzzing Engine module is not ready yet, it is a work in progress, and the efficiency of this technique will be compared to the ones presented here in Section 5 as well. Finally, we expect to obtain more relevant and detailed results, such as performing root cause analysis for discovered crashes, as well as to apply these results, as our research progresses.

References

Antunes, N. and Vieira, M. (2016). Designing vulnerability testing tools for web services: approach, components, and tools. *International Journal of Information Security*, pages 1–23.

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

Granjal, J., Monteiro, E., and Sa Silva, J. (2015). Security for the internet of things: A survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312.

Martins, R. d. J., Schaurich, V. G., Knob, L. A. D., Wickboldt, J. A., Filho, A. S., Granville, L. Z., and Pias, M. (2016). Performance analysis of 6lowpan and coap for secure communications in smart homes. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 1027–1034.

Mohsen Nia, A. and Jha, N. K. (2016). A comprehensive study of security of internet-of-things. *IEEE Transactions on Emerging Topics in Computing*, 6750(c):1–1.

Oehlert, P. (2005). Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine*, 3(2):58–62.

Offutt, J. and Xu, W. (2004). Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1.

Palattella, M. R., Accettura, N., Vilajosana, X., Watteyne, T., Grieco, L. A., Boggia, G., and Dohler, M. (2013). Standardized protocol stack for the internet of (important) things. *IEEE Communications Surveys & Tutorials*, 15(3):1389–1406.

Patton, M., Gross, E., Chinn, R., Forbis, S., Walker, L., and Chen, H. (2014). Uninvited connections: A study of vulnerable devices on the internet of things (iot). *2014 IEEE Joint Intelligence and Security Informatics Conference*, pages 232–235.

Raza, S., Shafagh, H., Hewage, K., Hummen, R., and Voigt, T. (2013). Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720.

Shelby, Z., Hartke, K., and Bormann, C. (2014). Rfc7252 - the constrained application protocol (coap).