A Testing Framework for Discovering Vulnerabilities in 6LoWPAN Networks

Abdelkader Lahmadi, César Bernardini, Olivier Festor

A Testing Framework for Discovering Vulnerabilities in 6LoWPAN Networks

Abdelkader Lahmadi
Université de Lorraine, LORIA
Vandoeuvre-lès-Nancy, F-54506,France
Email: abdelkader.lahmadi@loria.fr

César Bernardini, Olivier Festor
Inria
Villers-lès-Nancy, F-54600, France
Email: cesar.bernardini@inria.fr, olivier.festor@inria.fr

***Abstract*—In this work, we present the process of identifying potential vulnerabilities in 6LoWPAN enabled networks through fuzzing. The 6LowPAN protocol has been designed by the IETF as an adaptation layer of IPv6 for Low power and lossy networks. The fuzzing process is build upon the Scapy packets manipulation library. It provides different mutation algorithms to be applied on 6LoWPAN protocol messages to assess its implementations security and robustness. The protocol behaviors are described using an XML format to define different testing scenarios.**

***Keywords*-vulnerability discovery; fuzzing; 6LoWPAN; wireless sensor networks;**

I. Introduction

The future Internet is an IPv6 network interconnecting traditional computers and a large number of smart objects. Smart objects are generally added to the Internet using IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) [1]. These networks are formed by devices that are compatible with the IEEE 802.15.4 [2] standard as its communication protocol for Medium Access Control (MAC) layer and Physical (PHY) layer. IEEE 802.15.4 devices are characterized by low computational power, scarce memory capacity, lower bit rate, short range, and low cost. LoWPAN have devices that work together and connect the physical working environment to real-world applications like sensors with wireless application. As node density in sensor networks increases and these networks required connection with other networks via internet, then Internet Engineering Task Force (IETF) defines IPv6 over LoWPAN as techniques to implement the TCP/IP protocol in WSNs. 6LoWPAN provides a WSN node with IP communication capabilities by putting an adaptation layer above the IEEE 802.15.4 link layer for the packet fragmentation and reassembly purpose. These networks have begun to take an important place in our life, and they take control over important and private information in intelligent houses and critical industrial facilities. Therefore, among many other issues, the challenge of assessing the security in the IoT must be addressed. In traditional Internet network-based fuzz testing has become an effective way to ensure the security and reliability of communication protocol systems. It is therefore reasonable to explore this technique for IoT protocols [3].

Fuzzing is a highly automated technique [4] that covers numerous boundary cases using invalid, unexpected or random data as application input to better ensure the absence of exploitable vulnerabilities. A fuzzer is able to perform hundred of thousands or millions of testing iterations, covering a significant number of interesting iterations permutations for which it would be difficult to write individual test cases. The application itself will react to these inputs reporting exceptions, crashes, failing in the normal behavior or keeping the normal flow. The technique goal is find out unexpected scenarios that lead to situations that escape from the normal flow scenario and produce an unexpected behavior. Fuzzing approach has been widely applied on several traditional network protocols, services and document formats including HTTP, FTP, SIP, PDF,XML, etc [5], [6]. It is a valuable technique for automatic testing of protocol implementations to improve their security and reliability. Identifying such flaws is extremely important since they might be exploited by malicious parties to launch attacks.

This paper presents a testing suite designed to enabled 6LoWPAN compliant device developers to test the security and robustness of their protocol stack through fuzzing. The fuzzer engine itself is built around the Scapy [7] packet manipulation framework, to which we have added a module that supports the assembly, disassembly and fuzzing of the 6LoWPAN protocol. It incorporates features for replicate common interactions between devices, called scenarios, represented in a simple XML Language. Each scenario, defines the sequence of packets to be sent, and how the fuzzed device should reply to these packets. The fuzzer also supports the application of different mutation methods over created scenarios. These mutation methods can be either predefined where they are delivered as a library, or implemented directly by the framework user. The fuzzer relies on an embedded *IEEE 802.15.4* driver for linux, that we have developed to inject messages into the target network. In our current case, the target is either the protocol itself or the application that resides on a remote device. In this case, we are sending messages through the network to test the target device's behavior.

The structure of this paper is as follows. Section II details our fuzzing methodology of 6LoWPAN protocol using an XML based format to describe its messages and

behaviors. Section III presents our fuzzing tool design and implementation. Finally, Section IV concludes the paper and presents our future work.

II. Fuzzing Methodology

Network-based fuzzing requires the construction of interaction scenarios between the fuzzer and the target device to select the best combination of packets which will most probably generate a crash into the target device. The goal is to replicate interactions between devices to apply fuzzing algorithms in specific places in the communication process. Thus, we are able to find several flaws hidden deeply in the device implementation. We specified an XML-based language to represent messages in an interaction process for a testing scenario. The fuzzer is driven by these XML scenarios that define the sequence of packets to be sent, and how the fuzzed system should reply to these packets. Using such approach, The fuzzer supports stateless and stateful modes. In a stateless mode, only messages to be sent are specified without taking into account their responses. However, in a stateful mode, the scenario describes the entire or a part of the protocol interaction including messages to be sent and their respective responses. As depicted in Figure 1, the developer interacting with the fuzzer can then use the provided mutation policies, or create new ones, and define how to apply them to the selected scenario, and start fuzzing the device.

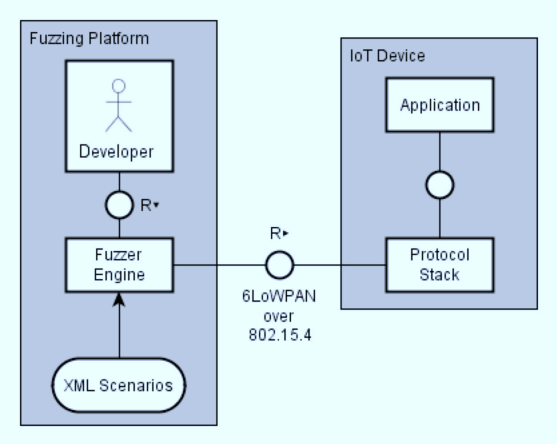

Figure 1. The 6LoWPAN fuzzing framework.

A. Message specification

A message is a target protocol packet with a certain predefined name and a set of parameters to be mutated. It is a binary sequence hat should be sent through some network interface to test certain behavior (modify a parameter, wait for a specific response, etc...) in a device. In our approach, the messages are usually generated with the scapy library but they could be created from any binary sequence. As we will explain in section III, we have extended scapy with a *6LoWPAN* layer to create all the messages required by an interaction scenario. The messages will be contained in a *Python* file that could be created manually or imported from the wireshark tool. Every message should have a name representable in a *Python* string. For example: *message1*, *1*, *124124* are valid possibilities. Snippet 1 depicts a basic 6LoWPAN message that we have named *message ping*. It has some special fields set and the unset parameters are filled by scapy with default values.

Snippet 1 An example of building a 6LoWPAN message using Scapy.

```
ping_msg = IPv6(dst="aaaa::11:22ff:fe33:4455", src
    ="aaaa::1")/ICMPv6EchoRequest()
packet = LoWPAN_IPHC(tf=0x0, _hopLimit=64)/
    ping_msg
messages['message ping'] = packet
```

We have shown a typical case of packet construction, but there are other scapy features that we could take advantage of them in the packet building process. One of these feature is fuzzing specific fields. As depicted in Snippet 2, we are able to set the *sam* field in the message ping as a random value.

Snippet 2 An example of building a 6LoWPAN message with a random value.

```
ping_msg = IPv6(dst="aaaa::11:22ff:fe33:4455", src
    ="aaaa::1")/ICMPv6EchoRequest()
packet = LoWPAN_IPHC(tf=0x0, _hopLimit=64)/
    ping_msg
packet.setfieldval('sam', packet.getfield_and_val
    ('sam')[0].randval())
messages['message ping'] = packet
```

B. Scenario format

A scenario is an ordered sequence of packets that specify clearly which message should be sent and what are the expected responses. It specifies what should be done with a packet: sent or wait for an answer through a given interface (i.e. usb0, eth1, wlan15). The scenario itself reproduces a communication between devices, specifying the sent messages and the most important or all properties of the expected responses. Also, the scenarios language allow us to test some properties, to make basic packet manipulation on-the-fly to be able to interact with the tested device in unknown situations, or context depending situations. A potential scenario could represent a ping interaction between two devices, testing that the value sent in the data field of the ICMP message is the same in the received message. As well as we could modify the destination address on-the-fly to make a more reusable scenario.

An XML structure has been defined to describe interactions based on the *6LoWPAN Protocol* behavior. We

have defined 3 different tags for this structure: *scenario*, *recv* and *send* for the scenario itself, message sending and message receiving respectively. Every tag should has its own attributes and subtags. The *scenario* tag defines the main properties, parameters and it contains the message's sequence. It includes the scenario's name and the set of messages to send and the expected messages to receive. The *send* tag specifies a message that should be sent, the message could be based on a previously created message with Scapy or it could be created on-the-fly with some hexadecimal code. The XML Format also offers a feature to decide what type of packet should be sent: *6LoWPAN*, *IPv6* or *IEEE 802.15.4*. In addition, the *send* tag could contain some subtags to modify a field on-the-fly by providing a value previously calculated or a certain value set by the user. The send tag includes the attributes: the name of the message, the identifier of the message available in python messages file, the packet type (6loWPAN, IPv6, 802.15.4), the packet payload encoded in hexadecimal format. We have to note that we only need to specify either the identifier of the message or the payload. Each send tag specifies one or multiple *field* subtag to modify field values of the packet. The field tag defines the scapy layer where the field is defined, its name, it value and its type.

The *recv* tag specifies an expected message to be received. It mainly includes a timeout attribute to define the timeout for receiving the message and one or multiple *test* subtags to check specific received field's values. Each *test* subtag includes as attributes: the layer of the field, its name, its value and type, an operation name to be applied on the field value. We have defined several operations to check, to modify or to assign a value of the field. Snippet 3 depicts a basic scenario for sending a simple 6LoWPAN packet. In this scenario, we modify on the fly the values of the fields tf and dac. The message named *basic_6lowpan* has been defined in the messages dictionary (a python data structure) as follows:

```
messages['basic_6lowpan'] = LoWPAN_IPHC()/IPv6()
```

Snippet 3 An example of a basic scenario for sending a single 6LoWPAN packet.

```
<?xml version="1.0" encoding="utf-8"?>
<scenario name="basic_6lowpan">
    <send name="1" message="basic_6lowpan">
        <field name="tf" layer="LoWPAN_IPHC" value
            ="1" type="int" />
        <field name="dac" layer="LoWPAN_IPHC"
            value="1" type="int" />
    </send>
</scenario>
```

A more advanced scenario is depicted in Snippet 4 to describe a ping interaction between a host and a 6LoWPAN node. In this scenario, we used a message named $basic\_ping\_echo$ defined as a python variable in the messages dictionary.

Snippet 4 A ping interaction using Echo requests and replies over IPv6.

```
<?xml version="1.0" encoding="utf-8"?>
<scenario name="basic_ping">
    <send name="1" message="basic_ping_echo">
        <field name="id" layer="ICMPv6EchoRequest"
             value="90" type="int" />
        <field name="seq" layer="ICMPv6EchoRequest
            " value="152" type="int" />
        <field name="data" layer="
            ICMPv6EchoRequest" value="hola, hola,
            hola" type="string" />
    </send>
    <recv>
        <test layer="ICMPv6EchoReply" field="id"
            compare="equal" value="90" type="int"
            />
        <test layer="ICMPv6EchoReply" field="seq"
            compare="equal" value="152" type="int"
             />
        <test layer="ICMPv6EchoReply" field="data"
             compare="equal" value="hola, hola,
            hola" type="string" />
    </recv>
</scenario>
```

In [8], we have detailed more advanced scenarios and usage examples.

III. 6LoWPAN Fuzzer design and implementation

A. System overview

Our fuzzing tool is implemented as a plugin-in to Scapy, a framework for manipulating network packets. In particular, it is based upon functionality from a decoding layer of 6LoWPAN packets. The fuzzer communicates with IoT devices, through a 802.15.4 network interface connected the to fuzzing platform which must be in range of the tested devices, and must be capable of relaying raw Link Layer frames. To be able to inject crafted 6LoWPAN packets into the network, we used an Atmel RZUSBstick, a 802.15.4 USB dongle, running a modified version of the Contiki OS, that simply relays packets without altering them. As we explained in section II, the fuzzer engine is then driven by a scenario written in XML that defines the sequence of sent and received packets. It also uses a set of callback functions written in python that define which messages and message fields to alter, and how. The fuzzer engine provides the following functionalities:

- Forge and decode IEEE 802.15.4 and 6LoWPAN packets using an added module to the Scapy packet manipulation framework.
- Apply different mutation algorithms on message bits and fields.
- Define request/response based network scenarios in XML.

- Execute XML-based interaction scenarios which define fuzzed packets and fields, to allow 6LoWPAN application or device developers to assess the robustness of their implementations.

The overall architecture of the fuzzing engine is depicted in Figure 2.

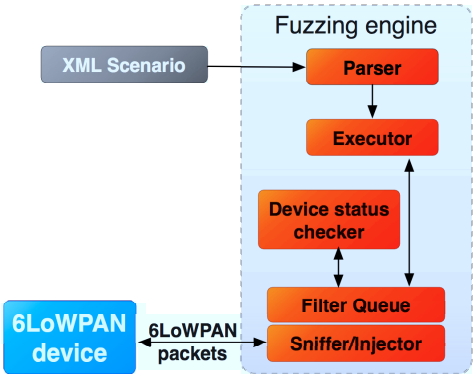

Figure 2. The fuzzing engine architecture.

The engine mainly relies on different modules which are responsible for executing interaction scenario, applying mutation algorithms over 6LoWPAN messages and checking the status of the tested device. If the device is not responding, it means that an existing vulnerability has been reached and exploited. The executor module parses the sniffed packets obtained through the USB dongle running a customized contiki code. These packets are then queued to be analyzed, their respective answers are generated and mutation are applied according to the executed scenario.

B. 6LoWPAN messages decoding and injection

6LoWPAN [9], [10] has been defined as an encapsulation and header compression mechanism that defines a set of compression/decompression rules taking advantage of the most common messages sent through a typical wireless sensor network, and exploiting some features of the underlaying layer, *IEEE 802.15.4*, and the upper layer, *IPv6*. Even more, all the information that should be repeated is avoided allowing the protocol to send really short messages and in this way, helping the devices to save energy. The protocol also has defined some special rules to fragment *IPv6* long messages. Being able in this way, to send the *minimal length IPv6 message*.

1) 6LoWPAN message decoding: We have extended the *scapy* library with a module to support assembly/disassembly of the protocol 6LoWPAN. The developed module is available in the Scapy Community Repository. Several classes has been developed to create the whole structure of 6LoWPAN packets format. All the classes are inherited from the *scapy* packet module except the class *SixLoWPANAddrField* which represents 6LoWPAN addresses. In 6LoWPAN, there are several types of packets: Mesh, Broadcast, IPHC, Uncompressed IPv6 packets and Fragmentations packets. The main difficulty was to handle fragmentation when 6LoWPAN splits long packets in several successive packets. When fragment packets appears, we have used two different handlers for representing them according to their position. If the packet is the first part of a big packet it is handled by the *LoWPANFragmentationFirst* handler and the successor is handled by the *LoWPANFragmentationSubsequent* handler.

As 6LoWPAN protocol has a particular encoding system, there are several fields that has a variable length depending on other parameters. The module provides different functions to provide fields length value. For example: *source_addr_mode, destination_addr_mode, pad_trafficclass*, etc...

In Snippet 5, we provide an example of decoding a 6LoWPAN message. In this example, the packet is provided in an hexadecimal format assigned to the variable *lowpan_frag_second*. To decode the packet, we need to call the constructor *SixLoWPAN* with the variable containing the packet data as a parameter.

Snippet 5 An example of decoding a 6LoWPAN message in Scapy.

```
>>> from scapy.layers.sixlowpan import *
>>> lowpan_frag_second = "\xe3\x42\x00\x23\x10\x3a
    \x2f\x2f\x77\x77\x77\x2e\x77\x33\x2e\x6f\x72\
    x67\x2f\x54\x52\x2f\x68\x74\x6d\x6c\x34\x2f\
    x6c\x6f\x6f\x73\x65\x2e\x64\x74\x64\x22\x3e\
    x0a\x3c\x68\x74\x6d\x6c\x3e\x3c\x68\x65\x61\
    x64\x3e\x3c\x74\x69\x74\x6c\x65\x3e\x57\x65\
    x6c\x63\x6f\x6d\x65\x20\x74\x6f\x20\x74\x68\
    x65\x20\x43\x6f\x6e\x74\x69\x6b\x69\x2d\x64\
    x65\x6d\x6f\x20\x73\x65\x72\x76\x65\x72\x21\
    x3c\x2f\x74\x69\x74\x6c\x65"
>>> lowpan_frag_sec_packet = SixLoWPAN(
    lowpan_frag_second)
>>> lowpan_frag_sec_packet.show2()
###[ SixLoWPAN(Packet) ]###
###[ 6LoWPAN Subsequent Fragmentation Packet ]###
     __datagramSize= 834
     __datagramTag= 0x23
     __datagramOffset= 16
###[ Raw ]###
        load      = '://www.w3.org/TR/html4/loose.
            dtd">\n<html><head><title>Welcome to
            the Contiki-demo server!</title'
```

2) Messages injection: The decoding module is only able to generate or modify 6LoWPAN packets on the fuzzing platform running on a host. However, we need a way to inject packets over 802.15.4 wireless links to be processed by a target device. We have thus modified Contiki [11] sources to generate a firmware to be run on an AVR Razor USB stick. This stick will act as a bridge between the fuzzing platform and the 6LoWPAN network. It receives packets generated by the 6LoWPAN module and injects them over the 802.15.4 wireless interface. The modified version of

Contiki is able to send native 6LoWPAN packets without any processing, or IPv6 packets that will be then processed and transformed into 6LoWPAN packets. The injection steps of a packet through the 802.15.4 bridge are depicted in Figure 3.

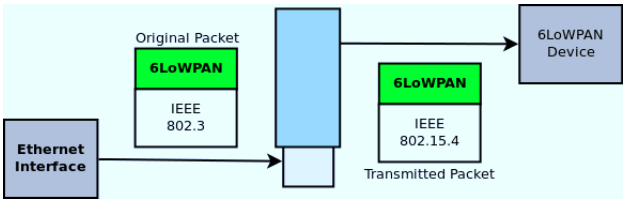

Figure 3. The injection of a packet through the 802.15.4 bridge towards a 6LoWPAN device.

C. Mutation algorithms

Network-fuzz testing implies applying mutations over messages to corrupt one or multiple values of a specific message. Our fuzzing engine supports different mutation algorithms to be applied at different levels.

1) Random bit mutation: This method is the simplest and trivial one. It takes a pre-built message and applies corruption using an *xor* operation to random bits. The corruption ratio is defined as a parameter provided by the user. It denotes the percentage of number of bits to be modified from the total number of bits of a message. The corruption operation is applied before computing the checksum control. Otherwise, it is useless and the packet will be rejected without being processed by the target device.

2) Random field mutation: This method takes as input the fields of 6LoWPAN headers, selects some of them and inserts random values. Let's take the example of a 6LoWPAN message with a compressed header. This message has 23 fields. Using a corruption ratio of 0.5, the mutation function replaces 12 fields with random values using the function *RandVal* available in Scapy.

3) Custom mutation: The fuzzing engine provides also a python interface to write user-defined mutation functions to be applied on the set of messages defined in the XML interaction scenario. It is only required to define a python function which will be called to apply the mutation over one or more messages. Snippet 6 shows an example of a user-defined mutation function to be applied on the message identified as msg_to_fuzz in the interaction scenario. The corruption ratio is obtained from the $metadata$ parameter.

Snippet 6 An example of a user-defined mutation function to be applied on a specific message of an interaction scenario.

```
from scapy.utils import corrupt_bits

def no_op_algorithm(scapy_message, options,
    metadata, variables):
    if options['name'] == "msg_to_fuzz":
        corrupt_bits(str(scapy_message), p=
            metadata['corrupt_ratio'])
    return str(scapy_message)
```

IV. Conclusion and Future Work

Our fuzzing tool suite, built on top of the Scapy framework using a decoding layer of 6LoWPAN messages, should be applied by anyone who regularly performs security auditing or vulnerability discovery in 6LoWPAN-enabled networks. An XML based specification has been defined to describe fuzzing scenarios against a target device. The XML scenario contains the set of message to be sent to the device and their respective responses. The tool provides mechanisms to inject 6LoWPAN packets through a 802.15.4 embedded driver to arbitrary modify that packets using random bits or fields mutations. Even, the tool is still at an early stage, it has been tested to discover vulnerabilities in a 6LoWPAN implementation provided by the Contiki project. The fuzzing software is available for public use and enhancement.

There are many future directions to explore with our fuzzing tool. Most, immediately, the bridge code should be extended to support the injection of 802.15.4 messages without further processing. Removing such limitation, we will be able to inject modified 802.15.4 frames. We also have to widely test our fuzzer against multiple 6LoWPAN implementations. Currently, we have only tested the Contiki implementation. Other, more challenging, work is possible. One example is the addition of more complex algorithms which track and mutate messages over multiple devices to discover more vulnerabilities. Additional work automating the fuzzing process is another valuable direction. It is possible to automate the system through the integration of solvers for bit-vector constraints [12] to drive the fuzzing strategy.

A. Availability

This entire tool suite is publicly available licensed under the LGPL. It can be downloaded at http://http://sicslowfuzzer.gforge.inria.fr/. Contributions are encouraged.

Acknowledgment

This work was partially supported by the FP7 PPP FiWARE project.

References

[1] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*. John Wiley & Sons, Chichester, UK, 2009, 2010.

[2] T. I. . W. Groups, "Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (lr-wpans)," 2006. [Online]. Available: http://www.ieee802.org/15/pub/TG4.html

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[4] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, pp. 58–62, Mar. 2005.

[5] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[6] H. Abdelnur, O. Festor, and R. State, "KiF: A stateful SIP Fuzzer," in *1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, Columbia University. New York, États-Unis: ACM SIGCOMM, Jul. 2007. [Online]. Available: http://hal.inria.fr/inria-00166947

[7] B. Burns, E. Markham, C. Iezzoni, P. Biondi, J. Stisa Granick, S. Manzuik, P. Guersch, D. Killion, N. Beauchesne, E. Moret, J. Sobrier, and M. Lynn, *Security power tools*, 1st ed. O'Reilly, 2007.

[8] C. Brandini, A. Lahmadi, and O. Festor, "Development of a fuzzing tool for the 6LoWPAN protocol," INRIA, Technical Report RR-7817, Nov. 2011. [Online]. Available: http://hal.inria.fr/hal-00645948

[9] I. W. Group, "Rfc4944: Transmission of ipv6 packets over ieee 802.15.4 networks." [Online]. Available: http://tools.ietf.org/html/draft-ietf-6lowpan-hc-15

[10] ——, "Compression format for ipv6 datagrams in low power and lossy networks (6lowpan)," 2011. [Online]. Available: http://tools.ietf.org/html/draft-ietf-6lowpan-hc-15

[11] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th international conference on Computer aided verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770351.1770421