Bbuzz: A Bit-aware Fuzzing Framework for Network Protocol Systematic Reverse Engineering and Analysis

Bernhards Blumbergs and Risto Vaarandi

Bbuzz: A Bit-aware Fuzzing Framework for Network Protocol Systematic Reverse Engineering and Analysis

Bernhards Blumbergs
NATO Cooperative Cyber Defence Centre of Excellence
IMCS UL, CERT.LV Laboratory
name.surname[a]cert.lv

Risto Vaarandi
Centre for Digital Forensics and Cyber Security
Tallinn University of Technology
name.surname[a]ttu.ee

Abstract—**Fuzzing is a critical part of secure software development life-cycle, for finding vulnerabilities, developing exploits, and reverse engineering. This relies on appropriate approaches, tools and frameworks. File and protocol fuzzing is well covered, multiple approaches and implementations exist. Unfortunately, assessed tools do not posses the required capabilities for working with protocols, where constructing bit groups are not byte aligned. In this paper, a systematic approach is proposed and tool prototype developed for the cyber red teaming purposes. In a case study, the developed Bbuzz tool is used to reverse engineer a proprietary NATO Link-1 network protocol allowing to inject rogue airplane tracks into air operations command and control system.**

Index Terms—**computer network operations, remote fuzzing, network security, network protocol reverse engineering, cyber red teaming**

I. Introduction

When performing a thorough unknown target information system vulnerability test, i.e., a red teaming engagement [1], there are cases when critical network communication protocols need to be targeted and assessed in a limited period of time. In this paper, the term *binary protocol* means any computer network communication protocol consisting of multiple bit groups (i.e., bit-fields) of various length, not always being aligned to the size of one byte. When testing either a known protocol, such as IPv6, or a proprietary one, such as Link-1 [2], proper tools are required to allow testing or reverse engineering with minimum effort.

Fuzzing or fuzz-testing is an integral part of secure software development life-cycle (SDL) [3], creating a process to allow developers build more secure software. Most common open-source fuzzing tools and frameworks used by penetration testers are bundled in such GNU/Linux penetration testing distributions as Kali Linux [4] and Black Arch Linux [5]. Majority of tools are web application oriented, with some being general purpose frameworks, and only few of them developed exclusively for network protocol fuzzing. Out of those, most tools aim at application layer protocols (e.g., FTP, HTTP, SSH), but are not concerned with the underlying transport and Internet layers, which are critical to ensure correct communications. Fuzzing of lower layer protocols would address the critical security issues of the TCP/IP protocol stack implementation in operating system kernel. In assessed tools, a typical test-case definition expects one byte as a minimum amount of data and are not concerned about individual bits and non-byte aligned groups of bits. Unfortunately, a number of widely used protocols support the field sizes of various bit lengths. For example, IPv6 header Flow Label field is 20 bits long and cannot be byte-aligned to maintain the protocol specification. Furthermore, these tools are not applicable for unknown or proprietary network protocols. Proposed *Bbuzz* tool uses one bit as the smallest unit and can operate starting from Layer-2 network frames. Thus, providing features for flexible test-case creation, whilst striving to ensure a balanced simplicity of use. The implemented functionality allows to use Bbuzz for quick assessment of any protocol, its features, information carried within, automatic creation of the initial test-case, and conduct fully automated fuzzing.

This paper addresses the identified drawbacks in open-source tools for network protocol fuzzing and reverse engineering, and provides the following contributions:

1) simple and systematic approach for protocol reverse engineering (described in Section III); implemented in
2) Bbuzz tool – open-source bit-aware network protocol fuzzing framework.

In this paper the Bbuzz tool is used for analyzing network protocols and real case study is presented, Section II gives an overview of related work; Section III explains the systematic approach and describes Bbuzz core concepts; Section IV covers a case study of Link-1 protocol reverse engineering; Section V concludes this paper.

II. Related Work

Closed source proprietary solutions (e.g., Codenomicon Defensics) tend to be expensive, and in most cases not feasible for short black-box penetration testing or ad-hoc red teaming engagements. In such cases, open-source solutions are explored and adapted to meet the testing requirements [6]. No closed source tools were assessed since no trial versions were made available or provided upon authors request.

Tools, such as SNOOZE by Banks et.al. [7], AutoFuzz by Gorbunov et.al. [8], RFSM by Zhao et.al. [9], and T-Fuzz by Johansson et.al. [10] are either designed for application layer network protocol stateful fuzzing or applicable for specific protocols (e.g., LTE, ZigBee), and no public source code was identified. The research paper of T-Fuzz tool hints, that it supports variable bit length test-case specification, however, it is specifically designed as an extension to a proprietary framework to perform telecommunication protocol testing.

SPIKE fuzzing framework [11] is a discontinued solution developed by ImmunitySec. Even though forked SPIKE instances exist, it was disregarded because of lacking support, required steep learning curve and significant time investment to start developing SPIKE-based test-cases.

Sulley [12] is a fuzzing framework written in Python, and implements the core concepts of SPIKE. It provides rich features in data generation, monitoring of target system network, process and state activity, and tracking the detected faults. Despite being feature rich, it is not well maintained and has majority of bugs. For network protocol fuzzing, the test-case specifications are automatically byte-aligned by prepending zeroes to reach one byte. Such forced byte-alignment does not allow a required depth of granularity for network protocol testing.

Boofuzz [13] is a fork and successor of Sulley. Besides having an active development, feature addition and code maintenance, it still inherits the same core development principles of Sulley and automatically performs byte alignment of bit fields. This issue [14] was submitted by us and acknowledged by the developer for feature addition. However, up to this date, it is still open and not implemented.

Peach Community Edition [15] is a general purpose fuzzing framework providing diverse feature set and systematic approach to describing and performing testing. Peach allows to define the data and state model, implements monitoring agents and various testing engine strategies and configurations. Due to these features, it is well acknowledged and used by the software vulnerability researcher community [16]. Data and state models are described in XML formatted configuration files (i.e., Pit files). Defining them is a tedious task involving deep in advance knowledge of the target under test. Such knowledge, in most cases, is not available when starting with unknown network protocol analysis, testing or reverse engineering. Furthermore, community edition has lengthy release cycles and infrequent bug fixes, and is lacking features which are implemented exclusively in the commercial edition. When evaluating Peach applicability to IPv6 testing, it was identified, that Pit files are already available for that [17], however, configuration was limited only to connectivity information (e.g., IPv6 and MAC addresses, ports and network interface). Besides that, it was not identified that Peach would support the variable bit-field test-case specification. Peach, being a general purpose solution, tries to cover file-based application as well as network protocol testing.

Other network protocol testing tools, bundled in Kali Linux and Black Arch penetration testing GNU/Linux distributions, such as Taof [18] and Zzuf [19], exist, however, they are limited to known application layer protocols and do not allow bit-field based protocol specification and testing. Also, file-based fuzzing solutions, such as American fuzzy lop [20], were considered to generate network packet mutations to be wrapped for delivery over network. This is not feasible when further payload mutations are required based on the received replies form the system under test.

To address the identified drawbacks of assessed fuzzing solutions, Bbuzz: uses one bit as the minimum size of data insetad of one byte; is protocol independent and can be used to test any network protocol at any layer instead of being limited to a particular protocol or layer; grants flexibility to configure all communications options, if deemed necessary, instead of few fixed ones; provides functionality to aid fast protocol initial assessment and test-case definition without requiring huge time investment; has a clear and strightforward test-case syntax without the need to create complex ones; and is publicly available on GitHub.

III. Core Concepts and Implementation

Developed Bbuzz tool prototype is written in Python 3, using standard libraries, and is released publicly under the MIT license at *https://github.com/lockout/bbuzz*.

Bbuzz aims to bring a simple and systematic work-flow in subsequent steps: obtaining communication sample, defining the base connection layer, describing the payload, performing the payload mutations, executing the fuzzing, and conducting logging, monitoring and tracking the state of the system under test. Systematic work-flow, implemented in Bbuzz object classes, is described in the following sub-sections. The analyst can interact with all separate object classes either via a Python script, Python interactive sell, or a system shell (e.g., iPython).

A. Sample Acquisition

Protocol analysis starts either by referring to specification (e.g., IETF RFC) or by capturing traffic samples with a network packet sniffer (e.g., tcpdump or Wireshark). Packet capture gives the required starting information, such as TCP/IP protocol stack layer, payload properties, and delivery method (e.g., unicast, multicast, broadcast). For known protocols, this provides the information about the use and implementation, and in case of unknown – raw data for further analysis. Depending on the complexity of the protocol, the analyst can perform visual analysis of the sample, use the analysis functionality of the network packet capture tools, and use the Bbuzz built-in analysis functionality.

Bbuzz helper tool, written in Python Scapy framework, is used to listen on the network traffic and capture the packets matching the defined filter criteria. Captured packets are converted into their binary representation and written to a file. Bbuzz function *analyze_payload()* parses this capture file and identifies the bit-field patterns which are static (i.e., immutable) among all, and the ones which change (i.e., mutable). This allows to quickly perform pattern matching and identify a possible initial test-case. With this step, the usage and principal properties of the protocol are identified to continue further analysis. For example, suppose a capture file contains the following data:

```
111001011010111110011010111100110
111001011010111111001000100000110
111001011010111010011011011100110
111001011010111000100110100000110
```

Bbuzz payload analysis generates the following output, representing mutable and immutable bit groups to quickly assess the communication protocol and identify the starting test case:

```
('111001011010111', immutable, 15)
('110011010111', mutable, 12)
('00110', immutable, 5)
```

To speed up the analysis and reverse engineering process of an unknown protocol, the Bbuzz tool calculates Shannon entropy for the whole payload and per each of the identified bit-fields. This unique functionality helps to quickly pinpoint the properties of the payload and its fields, revealing the likely type of information contained therein. For example, ASCII characters having the entropy of around 4.6 and compressed or encrypted data – 7.9 Shannon.

The approach in the case study uses strict filtering of captured packets based on criteria, such as source IP address, destination port, transport protocol, and payload size. This creates uniform packet capture without any noise. For Link-1 communications it is possible to use explicit filtering. There are cases where such filtering is not possible and resulting data-set would contain noise. For mining strong patterns from data-sets with noise, frequent itemset mining (FIM) algorithms have been often suggested [21] [22]. Nevertheless, the use of well-known FIM algorithms, such as Apriori, for packet payload data is complicated for several reasons. Firstly, these algorithms detect unordered groups of items, while reporting the offset of bits in packet payloads is essential for meaningful bit patterns (i.e., reporting a frequent itemset $\{0,1\}$ is not helpful for the end user). Secondly, bit patterns can be quite long (e.g., consisting of hundreds of bits), but mining long patterns from large data sets is known to be computationally expensive [23]. A FIM algorithm called LogHound [23] has been designed for encoding positional information into items (bits) and mining long patterns from larger data sets. During the experiments, we have used LogHound with higher support thresholds (e.g., 80-95%) for filtering out occasional noise and detecting bit patterns which are present in majority of packets.

B. Establishing Basic Network Connectivity

Basic network connectivity is handled by a *Bbuzz Protocol* class to describe and create the base protocol layer, which is required to send data over the network and meet the requirements of the test. The specified initial base protocol layer allows the testing of further upper layers, those treated as payload to be described as a test-case of a Bbuzz Payload class. Bbuzz Protocol class accepts the base layer to be specified either as "raw2" for Layer-2 connections (e.g., IPv4 or IPv6 testing), "raw3" for Layer-3 connections (e.g., TCP or UDP testing), and "raw4" to specify the Layer-4 connections (e.g., FTP and Link-1 testing). For example, base connection layer "raw2" establishes a Layer-2 connectivity allowing testing of upper layer protocols, such as IPv4 or IPv6. This flexibility allows to create and run test cases without the need to define complex protocol classes and test-cases.

To define the base protocol layer, Bbuzz uses the syntax as presented in example Fig.1, and takes two parameters. The first parameter specifies the initial base layer connection, and the second one is a Python dictionary providing the configuration parameters for that specific layer. Layer-2 connection options require the destination MAC address (DESTINATION_MAC) and Ethernet frame type (ETHER_TYPE). Optionally, source MAC address (SOURCE_MAC) can be specified. The ETHER_TYPE option can be used to provide additional information besides such types as IPv6 (0x86DD) or IPv4 (0x0800), but also, for example, to include IEEE 802.1Q VLAN tagging data (e.g., VLAN tag 0x810000010800). Layer-3 connection options require the destination IP address (DESTINATION_IP) and IP version (IP_VERSION). Optionally, source IP address (SOURCE_IP), SOURCE_MAC and DESTINATION_MAC values are accepted. Layer-4 connection options should include the required ones of the "raw3" accompanied by protocol number (PROTO_NUMBER), for example TCP (SOCK_STREAM = 0x01) or UDP (SOCK_DGRAM = 0x02), and destination port (DESTINATION_PORT). Additionally, source port (SOURCE_PORT) and connection specific parameters can be set, such as configuring the network interface as broadcast by specifying the BROADCAST option. Finally, the network interface identifier (e.g., eth0, enp0s25) should be provided for binding and sending data.

Fig. 1. Example of Link-1 broadcast base Layer-4 creation

```
interface = "enp0s25"
proto = bbuzz.protocol.Protocol(
      'raw4',
      {
         "SOURCE_IP": "10.78.2.169",
         "DESTINATION_IP": "10.78.2.255",
         "PROTO": 0x02,
         "DESTINATION_PORT" : 1229,
         "BROADCAST": True
         }
      )
proto.create(interface)
```

C. Describing Protocol Fields

Description of various protocol fields is handled by the *Bbuzz Payload* class to provide a structured approach to payload description for a test-case. Bbuzz Protocol class accepts values in different formats to ease the definition of test-cases according to data format. Payload field specifications can be added or loaded from a file to an instantiated Payload class object, therefore providing clear structure and flexibility for test-case definition.

To define the payload, Bbuzz uses syntax as presented in example Fig.2, and accepts two parameters. The first parameter represents the initial value of the data field, and the second one is a Python dictionary describing the data field and providing specific settings for mutation strategies. The value or values, in case of set of fixed known values, of the data field, can either be selected from the captured network traffic, specified according to the protocol specification, or any arbitrary value if it is chosen by the analyst. Payload field options require the following attributes: data format (FORMAT), field type (TYPE), field length in bits (LENGTH), data group (GROUP), and field mutation state (FUZZABLE). FORMAT specifies in what format the data is represented, such as binary, hexadecimal, decimal, octal, string, or bytes values. This eases the test-case definition with the values either from the protocol specification or from packet capture. Based on the format specified, the appropriate data conversion to bits will be chosen by the engine. TYPE indicates what data this field represents, and supports the following types for binary, numeric, string, delimiter, or static data. Based on the data type, the field mutation strategy will be chosen to generate sets of mutated values. LENGTH is the length of field in bits and, when performing the mutations, the size will be aligned to the defined one by the mutation engine. LENGTH set to '-1' represents a variable length field for which mutations of variable length would be produced. This is beneficial for tests of memory buffer overflow conditions. If no length is specified, it is calculated by the Payload class based on the length and type of the presented data. GROUP is a boolean value and when set to True, designates that the data value is a tuple, i.e., a set of comma separated fixed values. This can be used in case of known set of immutable values applicable for a particular field of the payload, for example, a set of Next Header values for IPv6 header. Specifying groups of values minimizes the time needed for testing, and ensures a better code coverage. If not provided, default GROUP value is set to False. FUZZABLE is a boolean parameter to represent either the data is mutable (True) or immutable (False). Additionally, Payload class calculates and assigns an unique identifier (HASH) to every field specified. This is used by some Bbuzz mutation functions to keep track of the mutation state for that particular field.

Fig. 2. Example of IPv6 header Flow Label assignment

```
load = bbuzz.payload.Payload()
load.add('0',
       {
          "FORMAT": "bin",
          "TYPE": "binary",
          "LENGTH": 20,
          "FUZZABLE": True
          }
       )
```

D. Payload Mutation Engine

Described payload mutations are handled by *Bbuzz Mutate* class to generate mutations for the particular payload field based on specified options. This class converts all the specified data values to binary, orchestrates the mutation process, selects appropriate mutation engine for the particular data type, and produces a mutated payload instance for network transmission.

Bbuzz mutation class is instantiated by *mutagen = bbuzz.mutate.Mutate(load)* and accepts two parameters. The first mandatory parameter presents the defined Payload class object, and a second optional parameter presents Python dictionary to configure the mutation process. Currently, Mutation class provides two main strategies – generating known bad values including ones introduced by coding mistakes for the particular data type, and pseudo random value generation. A third – genetic mutation strategy, will be implemented in upcoming release, in order to precede or obsolete random generation. This approach would produce further mutations for the generated valid payload instances able to solicit a reply from the system under test. The mutation strategies have the goal to attempt to minimize the required time to get testing results and generate a more sensible test-cases. It is not feasible to brute-force, i.e., produce all of the possible mutations for data fields, since that would yield too large set of mutations, be slow and consume huge amounts of computing resources.

Mutation concepts and common coding mistakes (e.g., off-by-one), allowing to trigger software exceptions and vulnerabilities, were identified from [16] [24] [13] to be further adapted and implemented. The mutation engine generates finite amount of mutations out of known bad values according to the specified data type. Binary mutation engine performs bit-wise operations (e.g., inversions, endianess change), bit-shifting (e.g., binary shift right with prepending '1' from left), and pattern insertion. For example, few sample mutations for a bit-field with value of '00110' are *bit-flip: (11001)*, *endianess swap: (01100)* and *bit-shift-right values: (10011, 11001, 11100, 11110, 11111)*. Numeric mutation engine creates mutations based on mathematical coding mistakes, such as turning the value into a two's complement negative, deducting or adding 1 to the value to trigger off-by-one errors. For example, few mutations for initial value of '00110' are *two's complement negative: (11010)*, *addition of '1': (00111)*, and *substracting '1': (00101)*. Delimiter mutation engine performs substitution with other known delimiter values or commonly misplaced characters. For example, replacing delimiter ';' (111011) with characters like ',' (101100) or ':' (111010). String mutation engine performs string encoding modifications, such as encoding the string into UTF-8, UTF-7, or, if enabled – increasing length to trigger buffer overflows. Static defined data fields do not produce any mutations and are treated as immutable. Random mutation engine is started, unless explicitly disabled, by the Mutation class once the generated known value mutation instances have depleted. For all mutation engines, where random values are generated, the seed is constant, unless changed, to ensure that test-cases are reproducible.

Instead of immediately generating the mutation stack containing all possible mutations, a Python generator is used to produce the next mutation instance upon request, in a fast and memory efficient manner. Mutation generator produces n-fold Cartesian product of all available payload field mutation sets. Meaning, that all possible payload combinations from individual field mutation sets are generated in order to grant the most complete test-case set. In essence, if *F1* is the protocol first field with all generated mutation set *A*, and *F2* is the second field with all generated mutation set *B*, then the Cartesian product would be calculated as follows:

$$A \times B = \{(a,b) \mid a \in A, b \in B\}$$

$$if : A = \{000, 010, 110, 100\}, B = \{11111, 00000\}$$

$$then : A \times B = \{(000, 11111), (000, 00000), (010, 11111),$$

$$(010, 00000), (110, 11111), (110, 00000), (100, 11111),$$

$$(100, 00000)\}$$

E. Target Monitoring and Test Execution Logging

Monitoring the system under test and performing the logging of the fuzzing execution is handled by *Bbuzz Monitoring* class. Logging component provides the back-trace of the fuzzing process to identify which packet or a sequence of packets triggered unexpected behavior or a crash condition. Monitoring component performs tracking of the state of the target system, gathering status information (e.g., system alive state or crash dumps). As well as receiving and tracking the responses from the target system, where applicable, to allow a more intelligent approach to fuzzing process and state machine development.

Components of this class already implemented is the time-stamped logging of the sent packets and a rudimentary approach to tracking the alive status of the target system via ICMP. Further developments are described in section V. To instantiate Monitoring class, it takes an IP address of the system under test – *mon = bbuzz.monitor.Monitor(ip="10.0.244.191")*.

F. Fuzzing Management

The management of the fuzzing is performed by *Bbuzz Fuzz* class, which involves actions such as next mutation retrieval from mutation engine, mutated instance delivery over the established network socket, control the timing of the packet sending, logging the sent test-cases with the response from the target system, and managing the process based on the target monitoring metrics. To instantiate the Fuzzer class, it takes the defined Protocol, Mutate and Monitor objects – *fuzzer = bbuzz.fuzz.Fuzz(proto, mutagen, mon)*.

The fuzzing process is started with *fuzzer.start(timing=0.5)*. This instantiates the base protocol layer for communication establishment, takes next payload mutation, and sends over the network socket. While the fuzzing is ongoing, the Fuzzer class uses Monitoring class functionality to record the sent mutations, their time-stamp, and if possible – the response and state of the target system.

IV. Case Study

Case study describes a red teaming engagement at NATO's largest technical cyber-defense exercise Locked Shields 2017 [25]. The scope of engagement was to verify the integrity of air command and control (C2) system communications. The desired effect was to present a large amount of fake unidentified aircraft tracks (i.e., volatile data representing an airplane instance at a particular moment in time), and have an impact on the situation awareness and decision making process.

The exercise network, consisting of multiple sub-nets, deployed the NATO air C2 systems in one of the subnets, consisting of ICC (Integrated C2) workstations and NIRIS (Networked Interoperable Real-time Information Services) radar instance. NIRIS is a MS Windows Server 2008 R2 with related software and services required to receive the information from the radar and broadcast this information on the local sub-net for engaged systems, such as ICC client. MS Windows 7 client with ICC client software listens to network broadcast traffic on a defined port and represents the received air tracks and details on the world map. These UDP-based network broadcast communications use Link-1 protocol wrapped in particular NIRIS format. Link-1 protocol, according to its specification, is a proprietary protocol containing groups of variable length bit fields, where each field represents a particular property of an air track (e.g., coordinates, altitude, bearing, flight number). Exercise deployment of air C2 systems were implemented as close as possible to represent real instances, and the cyber defense of these systems was assumed by the exercise participants (i.e., blue teams).

To complete this objective, with not too much information available, red team (RT) approached it as a black-box engagement. Before the start of the exercise, RT was granted access to the exercise network and air C2 component instances therein to allow attack approach assessment and development. The radar system was broadcasting legitimate air tracks on the network, received and displayed on the map by the ICC client instances. RT was able to capture this traffic to start the initial assessment, perform protocol reverse engineering and attempt to broadcast fake air tracks to be plotted on map. The Bbuzz protocol assessment capabilities allowed to quickly identify mutable and immutable bit fields of the Link-1 protocol, which allowed to start the test-case creation. It was assumed, that such properties as aircraft identification number and IFF (Identify Friend or Foe) status should not be changing for the same aircraft, therefore being treated as immutable, and properties, such as coordinates, velocity and altitude should be changing and therefore assumed as mutable. With this limited knowledge the fuzzing case was created and protocol fields mutated in order to produce observable results, which allowed to identify usage of a particular bit fields in Link-1 protocol. The aim was, in the limited amount of time, to identify only appropriate fields which are relevant for the attack, those being, aircraft number, coordinates, bearing, velocity and IFF status.

With the minimum Link-1 implementation information available, the red team used Bbuzz tool to successfuly identify critical parts of protocol in one day and further reverse engineer them. This enabled broadcasting fake air tracks to be displayed on the radar screens of the defending blue teams. In addition, for this attack to be successful, RT had to gain local area network access via other means such as spear-phishing campaign (e.g., e-mails containing malicious attachments or URL links), taking control over the firewalls (e.g., misconfiguration), and exploiting vulnerabilities in the publicly facing services (e.g., e-mail and web servers). Such attack represents a significant impact to real situation awareness, air operation decision making and overloading the air traffic control for handling and routing the fake traffic.

V. Conclusions and Future Work

In this paper a systematic approach to network protocol fuzzing and reverse engineering is discussed, with a Bbuzz tool prototype implementation described and its applicability in real use cases for red team engagements.

Planned Bbuzz updates involve research and development for system under test state information gathering, crash log collection, genetic payload mutation based on received responses, and target system management. For target system monitoring, approaches, not impacting the state of the system and able to extract information (e.g., core dumps, daemon or service logs, network traces), should be considered. Methods, such as remote probing, agent deployment, or virtual machine (VM) hypervisor management are applicable with their benefits (e.g., hypervisor level data collection) and implicit drawbacks (e.g., using TCP/IP stack under test also for monitoring purposes). Virtualized environments provide scalability and effectiveness of the target system deployment and management (e.g., snapshots, crash state recovery). However, for fuzzing purposes, possibility to interact with the target system via a wrapper or an API should be available. Virtualization platforms, such as VMWare Workstation or SUN VirtualBox, can be considered, both having their strengths and limitations (e.g., limited to VM deployment and management, not allowing interaction with the system itself). Kernel-based virtualization (e.g., KVM, QEMU) are well integrated with GNU/Linux and have API libraries (e.g., libvirt) for most programming languages. However, the extent of the interaction with the target system is not yet researched by the authors. Micro-virtualization solutions (e.g., Docker) allow running core required components, instead of allocating resources to fully featured VM. Nevertheless, this approach requires a deeper understanding if it is enough to conduct a reliable test, and the level of interaction provides valuable information of the target system state.

Future work includes research on the Bbuzz development, and usage for finding vulnerabilities in TCP/IP protocol stack implementation in embedded operating system kernels.

VI. Acknowledgments

The authors thank Olaf M. Maennel (TUT) and Raimo Peterson (NATO CCD CoE) for supporting this work.

References

[1] P. Brangetto, E. Çalişkan, and H. Rõigas, "Cyber Red Teaming - Organisational, technical and legal implications in a military context," tech. rep., NATO CCD CoE, 2015.

[2] NATO Standartization Agency, "STANAG 5501, Tactical Data Exchange - Link 1 (Point-to-Point), Ed.7," standartization agreement, North Atlantic Treaty Organization, 2015.

[3] "Microsoft SDL." https://www.microsoft.com/en-us/sdl/. Accessed: 27/03/2017.

[4] "Kali Linux Tools." http://tools.kali.org/tools-listing. Accessed: 28/03/2017.

[5] "Black Arch Linux Fuzzers." https://blackarch.org/fuzzer.html. Accessed: 28/03/2017.

[6] S. D. Zhang and L. Y. Zhang, "Vulnerability mining for network protocols based on fuzzing," in *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pp. 644–648, November 2014.

[7] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, *SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr*, pp. 343–358. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[8] S. Gorbunov and A. Rosenbloom, "AutoFuzz: Automated Network Protocol Fuzzing Framework," *nternational Journal of Computer Science and Network Security*, vol. 10, pp. 239–245, August 2010.

[9] J. Zhao, S. Chen, S. Liang, B. Cui, and X. Song, "Rfsm-fuzzing a smart fuzzing algorithm based on regression fsm," in *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pp. 380–386, October 2013.

[10] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 323–332, March 2014.

[11] "The Advantages of Block-Based Protocol Analysis for Security Testing." http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html. Accessed: 30/03/2017.

[12] "Sulley fuzzing framework." https://github.com/OpenRCE/sulley. Accessed: 29/03/2017.

[13] "boofuzz: Network Protocol Fuzzing for Humans." https://github.com/jtpereyda/boofuzz. Accessed: 29/03/2017.

[14] "Fuzz a bitfield of various fixed bit sizes #88." https://github.com/jtpereyda/boofuzz/issues/88. Accessed: 29/03/2017.

[15] "Peach Community Edition." http://www.peachfuzzer.com/resources/peachcommunity/. Accessed: 28/03/2017.

[16] D. egaldo et.al., *Gray Hat Hacking: The Ethical Hacker's Handbook*, ch. 5, pp. 117 – 142. McGraw-Hill Education, 4 ed., January 2015.

[17] "IPv6 Peach Pit Data Sheet." http://www.peachfuzzer.com/wp-content/uploads/IPv6.pdf. Accessed: 29/03/2017.

[18] "Taof - The art of fuzzing." https://sourceforge.net/projects/taof/. Accessed: 29/03/2017.

[19] "Zzuf - Multi-Purpose Fuzzer." http://caca.zoy.org/wiki/zzuf. Accessed: 29/03/2017.

[20] "American fuzzy lop." http://lcamtuf.coredump.cx/afl/. Accessed: 29/03/2017.

[21] Q. Zheng, K. Xu, W. Lv, and S. Ma, "Intelligent Search of Correlated Alarms from Database Containing Noise Data," in *2002 IEEE/IFIP Network Operations and Management Symposium*, pp. 405–419, 2002.

[22] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly Extraction in Backbone Networks using Association Rules," in *2009 ACM SIGCOMM Internet Measurement Conference*, pp. 28–34, 2009.

[23] R. Vaarandi, "A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs," in *2004 IFIP International Conference on Intelligence in Communication Systems*, vol. 3283, pp. 293–308, LNCS, 2004.

[24] N. Rathaus and G. Evron, *Open Source Fuzzing Tools*. Syngress Publishing, December 2007.

[25] "Locked Shields 2017." https://ccdcoe.org/largest-international-technical-cyber-defence-exercise-world-takes-place-next-week.html. Accessed: 20/04/2017.