Detecting applications vulnerabilities using remote procedure calls

Lukas Jokubauskas [1], Jevgenijus Toldinas [1] and Borisas Lozinskis [1]

[1] *Kaunas University of Technology, Studentų street 50, LT-51368 Kaunas, Lithuania*

Abstract

Computer software often comprises multiple components, such as a frontend application and a backend database, which need to exchange information. Many modern desktop applications also follow the design of web software and have separate frontend and backend processes. Inter-process communication mechanisms or third-party frameworks provided by the operating system are used for communication between processes. Improperly implemented remote procedure calls can lead to code vulnerabilities that can be exploited for malicious purposes. In this paper, we present a novel method for detecting application vulnerabilities using the remote procedure call approach, namely Detecting Applications Vulnerabilities using Google Remote Procedure Call (DAVuGRPC) that aims to utilize statically created taint and its dynamic fuzzification during the execution of the application.

Keywords

Vulnerability detection, dynamic analysis, taint dataset, RPC, gRPC

1. Introduction

A software vulnerability can be defined as a defect, weakness, or simply an error in an application that can be exploited by an attacker to change the system's regular behavior [1]. Because the quantity of software systems and applications is growing, so is the number of vulnerabilities. There are various application vulnerabilities: injection, cross-site scripting, broken authentication and session management, format string, insecure direct object reference, and many others [2]. In the software industry, vulnerability identification and remediation have been a core and vital operation. Hackers can take advantage of undetected flaws and wreak significant damage to people [3]. While program analysis tools exist, they often only discover a small subset of probable errors based on predefined rules. With the widespread availability of open-source repositories, data-driven methodologies for discovering vulnerability trends have become possible [4].

The techniques for finding application vulnerabilities are classified into two main categories: static analysis and dynamic analysis [5]. Static application analysis entails methods for inspecting source code or compiled binary without running it. Dynamic analysis is studying an application while it is running, with the use of a debugger or other techniques, such as [1]:

- Fault injection is a testing approach that introduces problems to an application to test its behavior. To generate the possible faults, some knowledge of the application is required.
- Fuzzing testing involves feeding the application with random data to see if it can handle it correctly.
- Dynamic taint during the execution of the application, the tainted data is monitored to determine its appropriate validation before accessing sensitive functions.
- Sanitization is a method of avoiding vulnerabilities caused by using user-supplied data by implementing newly included functions or custom routines

IVUS 2022: 27th International Conference on Information Technology, May 12, 2022, Kaunas, Lithuania

whose main objective is to evaluate or sanitize any input from users before using it inside an application.

Most of the time, cyber security specialists do not have access to the source code of the applications they are testing. As a result, cyber security specialists aim to automate some tasks using dynamic methodologies. The power of these strategies resides in the fact that the number of false positives is low, and the precision is extremely high [6].

The methods offered by operating systems that allow processes to handle shared data or interact are referred to as inter-process communication (*IPC*) [7]. *IPC* is a set of methods for communicating with two processes that may or may not be on the same machine. Remote procedure call (*RPC*) methods are widely used in systems because they lower system complexity and development costs. The primary purpose of an *RPC* is to make remote procedure calls transparent to users, allowing them to make remote procedure calls in the same way that they would make local procedure calls [9].

In this paper, we present a novel method for detecting application vulnerabilities using the remote procedure call approach, namely Detecting Applications Vulnerabilities using Google Remote Procedure Call (*DAVuGRPC*) that aims to utilize statically created taint and its dynamic use during the execution of the application. For that purpose, we employ the fuzzification technique for the tainted dataset.

The rest of the paper is organized as follows. The second section discusses the related works. The third section overviews application programming interfaces. The fourth section describes the *gRPC* payload. The fifth section presents the proposed application's vulnerabilities detection method using *gRPC*. The evaluation framework and experimental setup are presented in section six. The seventh section presents experimental results. The last section concludes the paper with a discussion of future work.

2. Related work

Fuzzing is a popular and successful method for detecting security flaws in the software when a system is tested by processing test cases generated by another program in a continuous loop. Simultaneously, the system monitored for any errors that may have been disclosed as a result of processing this data. *Fuzzing* strategies are classified into three groups based on the role: sample generation techniques, dynamic analysis approaches, and static analysis techniques [11]. Random mutation, grammatical representation, and scheduling algorithms are three types of sample generation approach that are used to choose and mutate seeds as well as restrict and generate new samples. To assist in the generation of the new sample, dynamic analysis techniques are employed to acquire dynamic information on the running application. Symbolic expressions, the executed path, taint information on the sample, and codes are all included in this data. Control flow analysis and data flow slices are examples of static analysis. Although static analysis frequently yields false-positive results, it can be used in conjunction with other methods to get useful pretreatment data.

In [12] proposed a system that combines machine learning and bandit-based optimization with state-of-the-art grey-box fuzzing approaches. Authors show significant improvements over numerous state-of-the-art grey-box *fuzzers*, such as *AFL*, *FidgetyAFL*, and the recently released *FairFuzz*. Thompson Sampling was used to learn adaptive distributions over mutation operators. The first *concolic* execution-based smart fuzzing method for detecting heap-based buffer overflow in executables was provided in [13]. The suggested *fuzzer* runs the binary program and determines the path and vulnerability restrictions for the executed path symbolically. It combines the constraints to generate test data that traverses the execution path and detects any flaws. The *fuzzer* removes each path constraint one at a time and solves the resulting constraints to generate test data that follows novel execution paths. The suggested approach propagates the tainted data through direct assignment and arithmetic operations.

In [7] authors proposed a new fuzzing solution to discover inter-process communication bugs without source code, by combining static analysis and dynamic analysis. Static analysis is used to recognize format checks and help construct inter-process communication messages of valid formats. Dynamic analysis is used to infer the constraints between inter-process communication messages and model the stateful logic with a probability matrix. This lets to generate high-quality inter-process communication messages to test services and discover deep and complex bugs.

In [8] authors presented the first grey box *fuzzer* for protocol implementations. Unlike the existing protocol *fuzzers*, the solution takes a

mutational approach and uses state feedback to guide the fuzzing process. It acts as a client and replays variations of the original sequence of messages sent to the server and retains those variations that were effective at increasing the coverage of the code or state space. A significant performance boost was demonstrated over the state-of-the-art.

Another similar solution [11] was suggested to perform a stateful communication protocol fuzzing. The approach contains a state switching engine with a multi-state fork server to consistently and flexibly fuzz different states of a compiler-instrumented protocol program. The solution was implemented by using a state-of-the-art grey-box *AFL fuzzer*. Experimental results showed that the solution achieved two times more unique crashes when compared to only fuzzing the first packet during the protocol communication.

Inter-Process Communication (*IPC*) refers to a variety of approaches for one-way or two-way data transmission between threads in one or more processes that can run on a single computer or multiple computers connected by a network [14], [15]. Message passing, synchronization, shared memory, and remote procedure calls (*RPC*) are some of the *IPC* approaches that can be divided into groups based on how they communicate shared memory and message passing [16]. The authors in [17] introduced direct *IPC* (*dIPC*) to marry the isolation of processes with the performance of synchronous function calls because *IPC* imposes overheads on a variety of different environments. Threads in one process can call a function on another process, offering the same performance as if the two processes were a single composite application, but without jeopardizing their isolation.

3. Application programming interfaces

Application Programming Interfaces (*APIs*) are software intermediaries that define certain rules and determinations for applications to interact and communicate with one another. An *API* is in charge of delivering a user's response to a system, which is then returned to the user by the system. Representational State Transfer (*REST*), *RPC*, and query language for *APIs* (*GraphQL*) are the three basic models for creating *APIs* [18]. The response from the back-end data is delivered to the clients (or users) through the *JSON* or *XML* communications format when using *REST APIs*. The *HTTP* protocol is commonly used in this architectural style.

The acronym *gRPC* [20] stands for Google Remote Procedure Call, and it is an *RPC*-based variation. This technology is based on an *HTTP 2.0 RPC API* implementation, but *HTTP* is not presented to the *API* developer or the server. As a result, there's no need to worry about how *RPC* principles are mapped to *HTTP*, which simplifies things. The goal of *gRPC* is to speed up data transmission between micro services. It is based on the concept of selecting a service, then establishing methods and parameters to allow for remote calling and return types. It also describes the *RPC* API paradigm in an interface description language (*IDL*), which makes determining remote operations easier. Protocol Buffers (*Protobuf*) are used by default in the *IDL* to describe the service interface as well as the structure of payload messages. *gRPC* can handle four types of interactions:

- Unary – when the client makes a single request and gets a single answer.
- Server streaming – in response to a client's request, the server sends a stream of messages. When all of the data has been transmitted, the server sends a status message to conclude the operation.
- Client streaming – the client delivers a stream of messages to the server, which responds with a single message.
- Bidirectional streaming – the client and server streams are autonomous, which means they can send messages in any sequence. Bidirectional streaming is started and stopped by the client.

gRPC is a great choice for multi-language systems, real-time streaming, and IoT systems that require light-weight message transfer, such as serialized *Protobuf* messages. Furthermore, *gRPC* should be considered for mobile apps because it does not require the use of a browser and can profit from fewer messages, preserving the speed of mobile processors [19].

4. *gRPC* payload data structure

By default, *gRPC* serializes payload data using *Protobuf*. Protocol buffers are a language-independent, platform-independent, and flexible framework for serializing structured data in a forward and backward compatible manner. It's similar to *JSON* but smaller and faster, plus it

creates native language bindings. Protocol buffers are made up of the definition language (in *.proto* files), the code generated by the proto compiler to interact with data, language-specific runtime libraries, and the serialization format for data written to a file (or sent across a network connection) [21].

Protocol buffer messages and services are described by engineer-authored .proto files. You can define whether a field is optional, repeated (*proto2* and *proto3*), or single when defining .proto files (*proto3*). Setting a field to required is not an option in proto3, and it is strongly discouraged in proto2 [22].

5. Detecting application vulnerabilities using *gRPC*

The stages of processing and interpreting network traffic packets are depicted in Figure 1. A general framework for detecting application vulnerabilities using *gRPC* is shown in Figure 2. There are two basic messaging strategies: changing the values of one field or all fields in one loop. There is also the situation where a message field's value is fixed and cannot be modified.

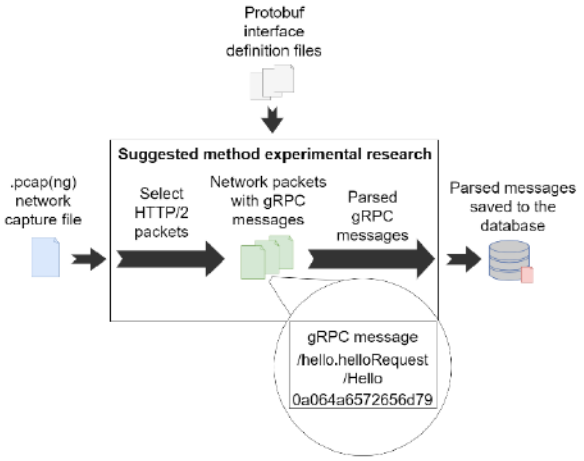

Figure 1: The stages of processing and interpreting network traffic packets

Both preceding solutions can be used in this scenario, but only if the required fields are left intact (see Figure 2). In the settings, you can define the messaging technique you want to employ.

The premise remains the same for both change techniques when it comes to fields modifications. The numeric message fields are modified by altering the values in the message using fuzzy logic.

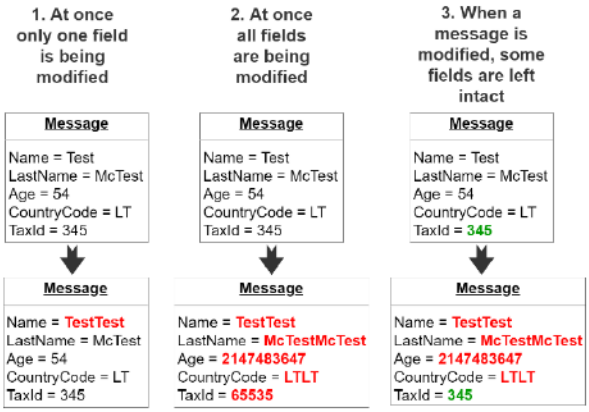

Figure 2: A general framework for detecting application vulnerabilities using *gRPC*

The range of substituted values for numeric fields is divided into value types and ranges (see Table 1).

Table 1
The range of substituted values for numeric fields

Value type	Value range	
	The smallest possible value	The largest possible value
bool	0	1
string	min length = null	max length = 2^{32}
int32, sint32, sfixed32	-2147483648	2147483647
uint32, fixed32	0	4294967295
int64, sint64, sfixed64	-9223372 036854775808	92233720 36854775807
uint64, fixed64	0	184467440 73709551615
float	1.175494351 E - 38	3.402823466 E + 38
double	2.225073 8585072014 E - 308	1.79769 31348623158 E + 308

The method for detecting vulnerabilities in applications using *gRPC* starts with scanning the initial remote procedure messages (see Figure 3). The proposed method will accept data that can be retrieved using the *Tcpdump* or *Wireshark* network packet analyzer from *.pcap* or *.pcapng* files. The proposed method accepts *Protobuf* files

.proto, which are used to filter out unnecessary messages and send messages to the application under test. Because *protobuf* messages are utilized in the *gRPC* remote procedure call framework, which is based on the *HTTP/2* protocol [23], *protobuf* messages must be requested in all *HTTP/2* protocol requests. After reviewing the contents of the *HTTP/2* request, it is determined whether this message is intended for at least one of the services described in the *.proto* files of the tested software. The data is saved if the message has a service match. If no match is detected, the algorithm repeats the process with a new *HTTP/2* request. *Protobuf* messages in binary format are extracted from these queries, which were constructed using the protocol buffer's interface description language [22].

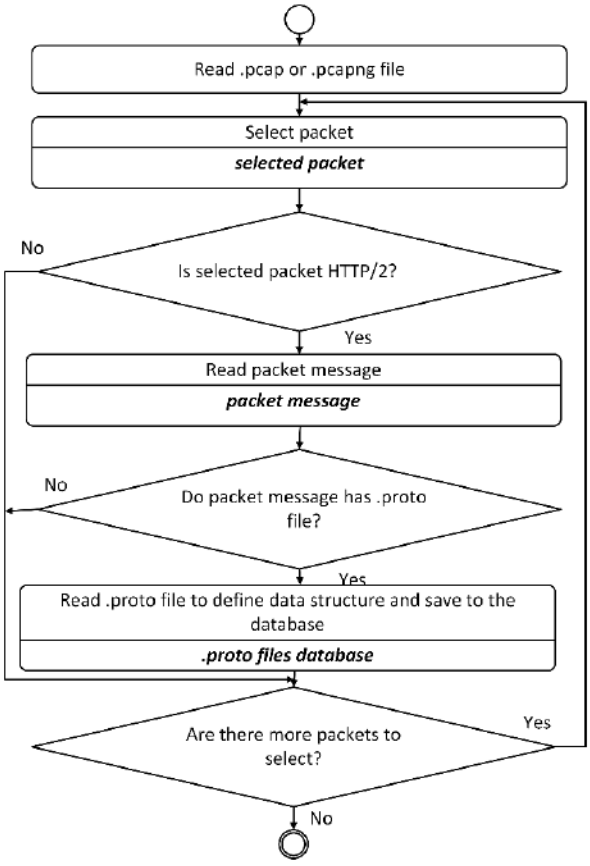

Figure 3: Packet scanning process for extracting remote procedure messages

After all remote procedure calls, message structures, and data types are saved to the database, the process of detecting application vulnerabilities using *gRPC* starts. The process of the proposed method is depicted in Figure 4. Starting vulnerability detection, .proto file, messages structure, and data types uploaded from the database. The execution monitoring procedure and the application under test are both started. The *gRPC* message creator using *fuzzy* logic changes the values of the message data accordingly to the types and possible values given in Table 1.

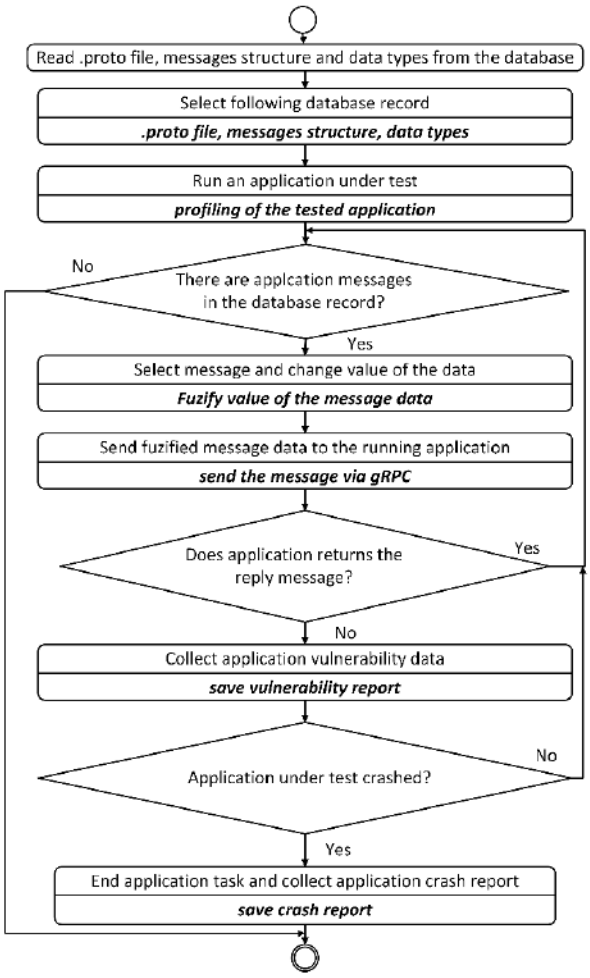

Figure 4: The process of proposed method for detecting application vulnerabilities using *gRPC*

The message with the highest expected number of message change cycles is chosen in the first iteration and changed values of the message data are constructed based on it. The messages are created in subsequent cycles depending on the execution progress and the tested application replies to the *gRPC* sent messages. The received response is sent for further analysis. The application is being tested if it is still running or if no reply is received. Verification of the tested application progress is sent to the report generating procedure. A new test iteration is started after the *gRPC* message generating process receives the execution status and response data from the application under test. The application activity monitoring process detects the tested application fault (no response) the crash

report process collects all relevant fault data and saves the application crash report.

6. Evaluation framework and experimental setup

A general framework for evaluation of the proposed method for detecting application vulnerabilities using *gRPC* is depicted in Figure 5.

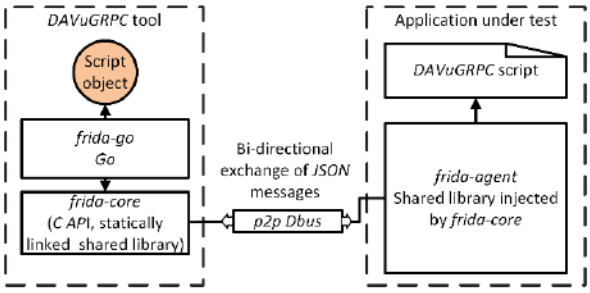

Figure 5: A general framework for evaluation of the proposed method

The Frida dynamic analysis library is used to track the application under test execution. To use the library programming interface in the Go programming language, we use the frida-go library, which allows us to use the Frida library's needed functions. The Frida library inserts additional code during execution that permits JavaScript to be performed after enabling the application under test execution. These scripts have full access to the application under test memory and can also change how functions are executed.

When a method in the application under test is called in the *DAVuGRPC* tool, the script begins to capture blocks of executed method instructions. The *Frida* Library's Interceptor and Stalker development *API* were used to do this. The completed instruction blocks are transmitted to the *DAVuGRPC* tool at the end of the application under the test method. In addition to this information, the application of the under test method's execution time is recorded. Data from the application under the test is sent using the *Frida* library's *P2P Dbus* communication channel, which allows data to be exchanged between the *DAVuGRPC* tool and the application under the test script code. This *P2P Dbus* channel is also used when JavaScript scripting methods are invoked. The structure of *DAVuGRPC* tool is represented in Figure 6.

The user can see the terminal interface after configuring and running the *DAVuGRPC* tool, which displays three main blocks: information on the time and duration of the test process, the overall results of the test process, and the current progress of the test process.

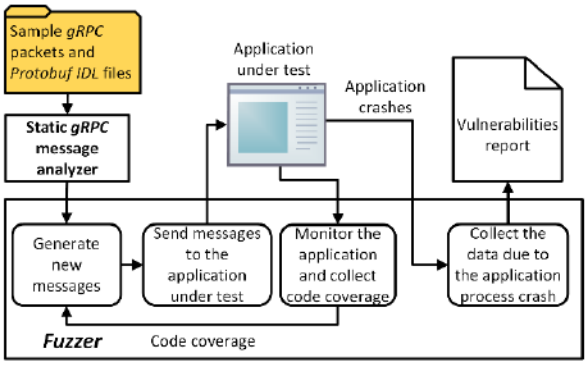

Figure 6: The structure of *DAVuGRPC* tool

7. Experimental results

Our experiments are performed using AMD Ryzen 5 2600 processor with six physical and twelve logical cores @ 3.40GHz; 16 GB RAM; Windows 10 Pro 64 bits OS.

For the experimental investigation, a testing platform was created with applications written in the *C++* that uses *gRPC*. There have been twenty-three remote procedures implemented: ten procedures (Proc0 – Proc9) have various types of buffer overflow and null-pointer dereference vulnerabilities and thirteen without any vulnerability. The proposed method was compared with the *proto-fuzzer* and *WinAFL* with *libprotobuf-mutator* library solutions (Table 2).

Table 2
Comparison of the *DAVuGRPC* tool

Procedures	Results (No. of sent messages / Detection time in *sec*)		
	DAVuGRPC	*proto-fuzzer*	*WinAFL*
Proc0	5/5	10/2	665000/-
Proc1	4/6	3/2	3737/28
Proc2	2/4	7/3	1357/5
Proc3	3/5	8/2	5043/78
Proc4	5/4	4/2	4983/11
Proc5	6/5	10/2	649000/-
Proc6	7/5	2/2	962/3
Proc7	7/5	34/3	10900/67
Proc8	16/6	-/-	-/-
Proc9	-/-	-/-	-/-

Based on the results we can evaluate that the proposed method detects stack-based, heap-based, and null-pointer dereference vulnerabilities in the short time sending a small number of *gRPC* messages.

8. Conclusion

The goal of *gRPC* is to speed up data transmission between micro services. It also describes the *RPC* API paradigm in an interface description language (*IDL*), which makes determining remote operations easier. The main results of this paper are as follows:

- *gRPC* could be successfully used in applications vulnerabilities detection.
- Dynamic application testing outperforms static methods because of a low number of false positives and extremely high precision.
- Fuzzing is the most acceptable method since it is the most universal and combines the best sides of static and dynamic testing.
- To increase efficiency, the method uses code-coverage feedback to prioritize complex remote procedure messages. This is achieved by using Frida dynamic analysis library.
- Proposed applications vulnerabilities method using remote procedure calls and realized *DAVuGRPC* tool shows acceptable results for stack-based, heap-based buffer overflow and null-pointer dereference vulnerabilities with the short time whereas the small number of *gRPC* messages has been sent.
- The proposed method found 11 out of 12 vulnerabilities. The method has lower performance than the *proto-fuzzer* solution; however, it sends fewer messages over the testing process.

Future work will be as follows:

- Add nested messages value fuzzing.
- Implement complex fuzzification logic with recognition dependencies between the same values in the messages.
- Add additional dynamic instrumentation framework support since the current Frida implementation is unstable.
- Add compressed *gRPC* messages support.

9. References

[1] W. Jimenez , A. Mammar and A. Cavalli, "Software Vulnerabilities, Prevention and Detection Methods: A Review", July 2010. PenTest Magazine [Online]. URL: http://www-lor.int-evry.fr/~anna/files/sec-mda09.pdf

[2] S. Garg, R.K. Singh and A.K. Mohapatra "Analysis of software vulnerability classification based on different technical parameters" 2019. Information Security Journal: A Global Perspective, 28:1-2, pp. 1-19. doi:10.1080/19393555.2019.1628325

[3] J. Fan, Yi. Li, S. Wang, and T. N. Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries" 2020. Proceedings of the 17th International Conference on Mining Software Repositories. Association for Computing Machinery, New York, NY, USA, pp. 508–512. doi:10.1145/3379597.3387501

[4] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood and M. W. McConley. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning." 2018. 17th IEEE International Conference on Machine Learning and Applications (ICMLA'2018) pp. 757-762. doi:10.1109/ICMLA.2018.00120

[5] J. Fell, "A Review of Fuzzing Tools and Methods", March 10, 2017. PenTest Magazine [Online]. URL: https://wcventure.github.io/FuzzingPaper/Paper/2017_review.pdf

[6] O. Zaazaa and H. El Bakkali, "Dynamic vulnerability detection approaches and tools: State of the Art," 2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS), 2020, pp. 1-6. doi:10.1109/ICDS50568.2020.9268686

[7] K. Yang, H. Zhao, C. Zhang, J. Zhuge and H. Duan, "Fuzzing *IPC* with Knowledge Inference," 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, 11-1109. doi:10.1109/SRDS47363.2019.00012

[8] V.-T. Pham, M. Bohme, and A. Roychoudhury, "AFLNET: A Greybox *Fuzzer* for Network Protocols," in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, Oct.

2020, p. 460–465. doi:10.1109/ICST46399.2020.00062

[9] H. Bagci, and A. Kara, "A Lightweight and High Performance Remote Procedure Call Framework for Cross Platform Communication", 2016. In Proceedings of the 11th International Joint Conference on Software Technologies - ICSOFT-EA, (ICSOFT 2016) ISBN 978-989-758-194-6, p. 117-124. doi:10.5220/0005931201170124

[10] T. Hussain, S. Satyaveer, and M. Seth, "A Comparative Study of Software Testing Techniques Viz. White Box Testing Black Box Testing and Grey Box Testing." IJAPRR International Peer Reviewed Refereed Journal, Vol. II, Issue V, 2015.

[11] C. Chen, C. Baojiang, M. Jinxin, W. Runpu, G. Jianchao and L. Wenqian, "A systematic review of fuzzing techniques" 2018 Computers & Security, Volume 75, pp. 118-137, ISSN 0167-4048. doi:10.1016/j.cose.2018.02.002

[12] S. Karamcheti, G. Mann and D. Rosenberg, "Adaptive Grey-Box Fuzz-Testing with Thompson Sampling" 2018 In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security (AISec '18). Association for Computing Machinery, New York, NY, USA, pp. 37–47. doi:10.1145/3270101.3270108

[13] M. Mouzarani, B. Sadeghiyan and M. Zolfaghari, "A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes," 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), 2015, pp. 42-49. doi:10.1109/PRDC.2015.10

[14] Z. Spasov, D. Bogdanova, and M. Skopje, "Inter-Process Communication, Analysis, Guidelines And Its Impact On Computer Security" 2010 The 7th International Conference for Informatics and Information Technology (CIIT 2010). Institute of Informatics. URL: http://ciit.finki.ukim.mk/data/papers/7CiiT/7CiiT-11.pdf

[15] N. C. Will, T. Heinrich, A. B. Viescinski and C. A. Maziero, "Trusted Inter-Process Communication Using Hardware Enclaves," 2021 IEEE International Systems Conference (SysCon), 2021, pp. 1-7. doi:10.1109/SysCon48628.2021.9447066

[16] D. Hamed, "Inter-Process Communication (*IPC*) in Distributed Environments: An Investigation and Performance Analysis of Some Middleware Technologies" 2020. International Journal of Modern Education & Computer Science. Vol. 12 Issue 2, pp. 36-52. doi:10.5815/ijmecs.2020.02.05

[17] L.Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero. "Direct Inter-Process Communication (d*IPC*): Repurposing the CODOMs Architecture to Accelerate *IPC*" 2017. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17). Association for Computing Machinery, New York, NY, USA, pp. 16–31. doi:10.1145/3064176.3064197

[18] N. Koutroumpouchos, G.Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis. "ObjectMap: detecting insecure object deserialization" 2019. In Proceedings of the 23rd Pan-Hellenic Conference on Informatics (PCI '19). Association for Computing Machinery, New York, NY, USA, pp. 67–72. doi:10.1145/3368640.3368680

[19] M. Berga, A. Santos, "*gRPC* vs REST: comparing APIs architectural styles" June 03, 2021. Imaginary Cloud [Online]. URL: https://www.imaginarycloud.com/blog/*gRPC*-vs-rest/

[20] *gRPC* a high performance, open source universal *RPC* framework. [Online]. URL: https://*gRPC*.io/

[21] Protocol Buffers Overview. [Online]. URL: https://developers.google.com/protocol-buffers/docs/overview

[22] Protocol Buffers Language Guide. [Online]. URL: https://developers.google.com/protocol-buffers/docs/proto#specifying-rules

[23] Internet Engineering Task Force, Hypertext Transfer Protocol Version2 (HTTP/2). [Online]. URL: https://datatracker.ietf.org/doc/html/rfc7540