PULSAR: Stateful Black-Box Fuzzing of Proprietary Network Protocols

Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi,
Daniel Arp, and Konrad Rieck

Computer Security Group
University of Göttingen
{hgascon,christian.wressnegger,fabian.yamaguchi,
darp,konrad.rieck}@uni-goettingen.de

Abstract. The security of network services and their protocols critically depends on minimizing their attack surface. A single flaw in an implementation can suffice to compromise a service and expose sensitive data to an attacker. The discovery of vulnerabilities in protocol implementations, however, is a challenging task: While for standard protocols this process can be conducted with regular techniques for auditing, the situation becomes difficult for proprietary protocols if neither the program code nor the specification of the protocol are easily accessible. As a result, vulnerabilities in closed-source implementations can often remain undiscovered for a longer period of time. In this paper, we present PULSAR, a method for stateful black-box fuzzing of proprietary network protocols. Our method combines concepts from fuzz testing with techniques for automatic protocol reverse engineering and simulation. It proceeds by observing the traffic of a proprietary protocol and inferring a generative model for message formats and protocol states that can not only analyze but also simulate communication. During fuzzing this simulation can effectively explore the protocol state space and thereby enables uncovering vulnerabilities deep inside the protocol implementation. We demonstrate the efficacy of PULSAR in two case studies, where it identifies known as well as unknown vulnerabilities.

Key words: Model-based Fuzzing, Vulnerability Discovery, Protocol Reverse Engineering

1 Introduction

A myriad of network services and protocols is employed in today's computer networks, ranging from classic protocols of the Internet suite to proprietary binary protocols implemented only by particular vendors. While these network services steadily expand their capabilities, securing their functionality still remains a challenging task: A single vulnerability in the implementation of a protocol can suffice to undermine the security of a network service and expose sensitive data to an attacker. For example, a flaw in the implementation of the universal plug-and-play protocol rendered roughly 23 million routers vulnerable to attacks from the Internet [27].

Several methods for locating and eliminating vulnerabilities in protocol implementations have been proposed in the last years, each addressing different aspects of the problem. For example, if the implementation of the protocol is easily accessible, different techniques from program analysis can be applied for hunting down security flaws, such as white-box fuzzing [e.g., 13, 15], dynamic taint tracking [e.g., 9, 34], symbolic execution [e.g., 7, 31] and static code analysis [e.g., 18, 25, 36, 37]. The situation, however, changes fundamentally if neither the code nor the specification of the protocol are directly accessible. While in some cases there are means for retrieving the implementation of a protocol, for example by reading out a firmware image or reverse-engineering a binary package, the complexity of this effort may still impede a sufficient security analysis.

Only few approaches exist [14, 17] that can help spotting vulnerabilities in settings where code and specifications are hard to obtain. These approaches provide first means for automatically inferring fuzzers for proprietary protocols if a program analysis is not possible or difficult to carry out. Due to the lack of insights in the protocol code; however, these approaches are not capable of guiding the fuzzing process through the implementation. As a consequence, flaws that are linked to deep states in the protocol implementation are hard to reach efficiently.

In this paper, we present PULSAR, a method for stateful black-box fuzzing of proprietary network protocols. Our method combines concepts from fuzz testing with techniques for automatic protocol reverse engineering and simulation. It proceeds by observing the network traffic of an unknown protocol and inferring a generative model for message formats and protocol states that can not only analyze but also simulate communication. In contrast to previous approaches, this model enables effectively exploring the protocol state space during fuzzing and directing the analysis to states which are particularly suitable for fuzz testing. This *guided fuzzing* allows for uncovering vulnerabilities deep inside the protocol implementation. Moreover, by being part of the communication, PULSAR can increase the coverage of the state space, resulting in less but more effective testing iterations.

We empirically evaluate the capabilities of PULSAR in two case studies. First, we analyze the standard text-based protocol FTP as an illustrative example and then proceed to applying PULSAR to the proprietary binary protocol OSCAR, implemented in many instant messengers. To demonstrate the efficacy of simulating network communication, we direct our fuzzer against clients of the respective protocols, as these are harder to test with regular fuzzers due to their active role in the communication. In both case studies, PULSAR is able to spot known flaws in these clients, but also hints us to previously unknown vulnerabilities.

The rest of the paper is organized as follows: we introduce our method for stateful fuzzing of proprietary protocols in Section 2 and evaluate its efficacy in Section 3. Limitations and related work are discussed in Section 4 and 5, respectively. Section 6 concludes the paper.

2 Methodology

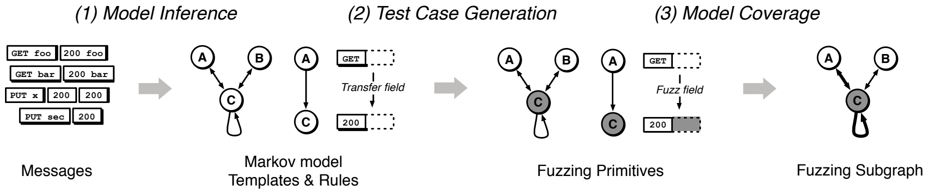


Fig. 1: Overview of PULSAR and the different analysis steps.

The goal of PULSAR is to be able to effectively fuzz the implementation of proprietary protocols for which no specification exits and the underlying code is hard to analyze. In order to achieve this, our method starts by inferring a model of the protocol including its state machine and the format of the messages. The combination of both elements allow us to actively control the communication in order to guide the fuzzing process and to build faulty inputs that are sent to the network service. As explained in Figure 1, PULSAR proceeds in the following steps:

1. *Model inference.* A sample of network traces from the protocol under test is captured and a model is inferred from its messages. This include a Markov model representing the state machine of the protocol, templates that identify the format of the messages and rules that track the data flow between messages during communication.
2. *Test case generation.* The extracted templates and rules enable defining a set of fuzzing primitives that can be applied to message fields at specific stages of the communication. Using these primitives, test cases for black-box fuzzing are automatically generated.
3. *Model Coverage.* To increase the coverage of the security analysis, protocol states that are particularly suitable for fuzzing are selected. To this end, the fuzzer is guided to subgraphs in the state machine that are rarely visited and contain the largest number of messages with variable input fields.

PULSAR is implemented as an open-source tool[1] that once placed in the network can operate as a service or client and simulate communication with the corresponding party. In the following we describe the three steps conducted by PULSAR in more detail.

2.1 Model Inference

While model-based fuzz testing outperforms brute force fuzzing [30], it also does rely heavily on the quality of the specification used for the generation of the test cases. In the case of fuzzers whose goal is to identify errors in the implementation of well-known protocols, these models can be built on the basis of existing RFCs or proper documentation.

[1] `https://github.com/hgascon/pulsar`

On the contrary, poorly documented or totally closed proprietary protocols represent a tough challenge for such methods.

To address this problem, our method builds on the techniques introduced by Krueger et al. with PRISMA [19], a probabilistic approach to model both the message content and the state machine of an unknown protocol solely relying on standard captures of network traffic. The quality of these models surpasses that of previous works targeting the problem of reverse engineering network protocols without the need to access the binary implementation. As detailed in Section 3, the inferred model allows our method not only to generate relevant security test cases but to simulate the inputs and outputs of a real entity within the environment of the system under test.

Data Acquisition. In a real scenario, a software application usually communicates with different entities in the network, establishing several connections based on different protocols. As a fuzzing session of PULSAR targets an individual service, we start by capturing all traffic transmitted and received by an application between a unique combination of source and destination IPs and PORTs. Then, we re-assemble the captured packages and feed the complete streams into a session extractor. A session identifier is assigned to each one of the streams. If no packet is received for a selected time interval, a session will be marked as terminated, so that a new packet within the same connection will belong to a new session. The interval can be provided as a parameter and tuned to suit the rate of new connections established by the application under test.

We need to note here that a model learned from network traces alone may naturally lack parts of the functionality of the protocol if this functionality has not been observed during the training phase. Therefore, the analyst can generate specific interactions with the test application to model the inputs and outputs of the system that need to be audited.

Message Clustering. After traffic recording and session identification, we model each message as a sequence of bytes. To infer common structures among the series of messages we begin by mapping these sequences of bytes into a finite-dimensional vector space for clustering by the following two strategies.

For text-based protocols, where messages are typically formed by string tokens separated by pre-defined characters, each dimension is associated with an individual token in the feature vector. Thus, each dimension indicates the occurrences of a specific token within a message. In the case of binary protocols, we follow a similar approach where each individual n-gram (i.e., series of bytes of a specific length) within a message is mapped to the correspondent dimension in the feature vector. As the goal of this analysis phase is to model the different types of messages of the protocol, we proceed with a dimensionality reduction phase that allows the clustering algorithm to focus on the most discriminative characteristics from each message. Following the design of PRISMA [19], we use a simple statistical test [16] to remove volatile features, such as cookies and random strings, and constant elements that occur in almost every message.

Once that each message is represented as a vector, we use the Euclidean distance as similarity metric to apply the clustering algorithm. This allow us to extract common message structures which typically occur during a certain stage of the modeled protocol. Since most protocols are assembled from parts, we apply the non-negative matrix factorization algorithm (NMF) for part-based clustering [21]. NMF is an effective and well-known clustering algorithm that represents given data as a factorization of the data matrix (features × traces). After elimination of duplicated entries, the solution to the optimization problem let us identify clusters of messages that share similar structure and therefore belong to the same type.

Protocol State Machine. Network protocols are inherently defined by their state machine. As the exact state machine can only be inferred from the actual implementation of the protocol, PULSAR approximates the state machine from observed network traces. To this end, we annotate each message indicating if it has been generated by the client or the server. For this annotated version, a sliding window of size two links each message to previously observed traces. By computing the probabilities over these linked messages, we finally arrive at a second order Markov model that provides a probabilistic approximation of the real state machine.

Next, we minimize this Markov model into a deterministic finite automaton (DFA). To this end, we keep transitions with probabilities larger than zero and their associated states and at each transition we modify the DFA to accept the event of the second state. The DFA minimization algorithm introduced by Moore [26] let us generate an equivalent DFA that accepts the same language but with a smaller number of states, which allows the security analyst to manually inspect the model if required.

Message Format. In the clustering step we identify common tokens in the recorded messages. The position where these tokens occur in a session during the communication can be linked to a correspondent transition in the state machine. This enables us to correlate tokens with the state of the service. By analyzing the tokens of messages which are observable at the same state, we can improve the initial clustering stage and extract generic format definitions for these messages that we call *templates*.

In particular, after tokenizing each message according to the type of protocol (i.e. text-based or binary) and the embedding used (i.e. token or byte n-gram), we assign each message of a session to the corresponding state of the Markov model. For each one of the states we generate a unique group for all messages with the same number of tokens. If all messages within a group contain the same token at a specific position, this token is fixed as a constant. On the contrary, we consider tokens that differ even if only once as variables and its position is defined as a *field*. As a result, each state of the Markov model is associated with a series of templates that represent the generic type of messages that may be observed at such state of the communication.

	State A_S	State B_C	State C_S
Session 1	`ftp 3.14`	`USER anon`	`331 User anon ok`
Session 2	`ftp 3.12`	`USER ren`	`331 User ren ok`
	⋮	⋮	⋮
Session n	`ftp 2.0`	`USER liz`	`331 User liz ok`
Template	`ftp` □	`USER` □	`331 User` □ `ok`

Fig. 2: Example of template generation for a simplified FTP communication.

Figure 2 presents a generic example of the process based on a series of FTP messages from different sessions.

Data Flow. Once the session information, the Markov model and the message templates are defined, we infer a set of *rules* to characterize the flow of information between different messages during a session. More specifically, we establish dependencies so that data found in a preceding message can be used to fill the different fields in a subsequent message.

In particular, we consider each possible combination of template occurrences for the horizon of length $k = 2$, i.e. (t_{-2}, t_{-1}, t_0) and find all messages assigned to these k templates which are sent in a session in this exact order. For each field f in such templates, we look for a rule that let us fill f with data content of a different field from previous messages. If no rule matches, the tokens are recorded and a new data rule is defined, indicating how to fill f with a random choice over previously seen data.

Table 1 describes the different type of rules we have implemented in our system. For instance, in the example from Figure 2 the field associated with the state C can be filled with the field of the previous message in all cases.

Rule	Description
Copy	Exact copy of the content of one field to another.
Seq.	Copy of a numerical field incremented by d.
Add	Copy the content of a field and add data d to the front or back.
Part	Copy the front or back part of a field split by separator s
Data	Fill the field by randomly picking data d which we have seen before.

Table 1: Rules checked during model building. Parameters like d and s are automatically inferred from the training data.

2.2 Test Case Generation

Up to this point, PULSAR is able to simulate both ends of the communication with high accuracy. Furthermore, the templates and fields in our model give us the opportunity to feed the other side of the connection with faulty inputs at a certain point in a session. By applying fuzzing primitives to the data provided by the rules, we can send an ill formatted message when the service expects to parse a variable data field controlled by the remote side.

In particular, the system proceeds as follows: When a message from the other end of the communication is received, it is matched to one of the templates of the states for which a valid transition exist. As the state machine of the protocol is defined as a Markov model of second order, a valid transition is represented by the new matched template and the two previously matched templates in the form of a chain *A:B:C*. This means that if templates A and B have been observed, our system will try to match a received message to the template C that allow this transition. The set of rules for this transition is used by the system to build the next message in the case that a response is required.

In some cases, the received message at a certain stage of the communication may differ from that observed in the training data. As a result, some tokens or bytes may not allow for an exact template match even if the semantics of the message are expected by the model. Thus, to trigger a transition we use the Levenshtein string distance to measure the similarity between the received message and all reachable templates and select the most similar template as a match. This type of *semi-valid* transition has two effects. In the first place, the probability of reaching a "fuzzable" state is increased and second, if the semantics from the similarity matched template are too far from the semantics of the correct message, the response can be understood as a faulty input in itself. From the fuzzing perspective this is equivalent to a jump to an erroneous state in the real state model of the protocol. This situation may also led to errors in implementations where the network service is not able to handle a wrong sequence of messages during a session or a message from a different session.

After selecting a template D , we use the rules describing the transition B:C:D in combination with a fuzzing primitive to build the next message. Possible primitives to select during testing include: *invalid UTF-8 byte sequence*, *constant string overflow* or *random string overflow* with or without a percentage of non-alphanumeric characters. A modular architecture allows for new fuzzing primitives to be added by the community to our open-source tool independently of the fuzzer implementation.

2.3 Model Coverage

A classic problem shared by random and more advanced model-based fuzzers is that of achieving a high coverage of the testing space. In the case of PULSAR, the system is able to fuzz the communication but also to be an active part of it as a network service. This allow us to guide the interaction between both ends and can be exploited in order to reach in less time those states where messages can be fuzzed.

After a message has been received and matched to a template, we must select a valid response template. For the purpose of simulating traffic as closed as possible to the real protocol the response template can be chosen according to the probability observed for each transition in the training data. However, when a fuzzing session is active, we define the *fuzzing subgraph* (FS) algorithm to effectively select the next response.

The FS algorithm controls the progress of the fuzzer across consecutive iterations and along the different states of the model, that is, new connections initiated by the application under test when a session is terminated. Its ultimate purpose is not only to increase the exploration of the model but to reach fuzzable states faster.

The algorithm proceeds as follows:

1. When a new fuzzing process is started, a *fuzzing mask* is assigned to each one of the templates. A fuzzing mask is a binary array of size equal to the number of fields in a template and indicates what fields are to be fuzzed the next time this template is selected to build a message. If a template has N fields, there exist 2^N possible fuzzing masks for each one of the templates. Initially, each mask is set to 2^N.

2. A *subgraph* is defined by a root state and all the states that can be reached in D transitions. The fuzzing weight of the subgraph is defined as the sum of the weights of its states. The weight of a state is computed as the sum of the fuzzing masks of its templates at a certain point in time.
3. When a message is received and matched, the state with the highest *subgraph* weight is selected from all states that represent a valid transition. The response template is chosen from this state according to the probability of occurrence in the training data.
4. The communication continues until a fuzzable state is reached. When a template is selected for fuzzing its fuzzing mask is decreased by one.

Modifying the fuzzing mask changes what fields of a template are fuzzed the next time the template is selected. Moreover, it also decreases the fuzzing weight of its state and previous states' subgraphs. As a result, the paths in the model with more fuzzing opportunities at early stages will be walked first. As the fuzzing masks of these templates decrease, the weight of the *subgraph* will also decrease, allowing for the exploration of adjacent paths in the model. If all states reachable from the current state through a valid transition have the same *subgraph* weight, we select the next state randomly.

3 Case Studies

We proceed to demonstrate the capabilities of PULSAR in two case studies with real-world protocols. In particular, we evaluate our method's ability to derive stateful fuzzers for the well-known protocol FTP (Section 3.1) as well as for the proprietary protocol OSCAR as used by different instant messengers (Section 3.2).

3.1 Core FTP Client

At first, we evaluate our method's ability to automatically discover vulnerabilities in implementations of a classic text-based protocol. To this end, we employ PULSAR to identify flaws in the Core FTP Client[2], a commercial, closed-source FTP client. This program has been found to contain several buffer overflow vulnerabilities, providing us with up-to-date ground truth for our analysis.

In June 2014, Gabor Seljan reported several heap-based buffer overflows in the Core FTP Client that can possibly be exploited by attackers to run arbitrary code in the context of the FTP client (`CVE-2014-4643`). These buffer overflows can be triggered by sending overly long responses to client requests in various stages of the communication. Clearly, to trigger these vulnerabilities the client needs to transition into the vulnerable state. Hence, suitable responses must be returned by the server and thus Seljan manually prepared a sequence of server responses in his proof-of-concept exploit.

In order to automatically identify these vulnerabilities in the FTP client, we record 987 traces from usual interaction between the client and the server running vsftpd[3]. Based on these traces PULSAR automatically generates the state machine depicted in

[2] http://www.coreftp.com

[3] http://vsftpd.beasts.org

Figure 3 as well as the corresponding message templates and rules. States containing templates with variable fields are shaded for both ends of the communication.

Every state in the state machine is labeled according to the terminology defined by the Markov model: Namely, the observed event that triggers the transition to that state and the event that is generated from this state. For instance, an event labeled `X.UAC, Y.UAS` indicates that a message from the client `UAC` has been observed and a response from the server `UAS` is required at this stage and vice versa. `X` and `Y` indicate the cluster identifier of the messages and the templates associated with that state. In case that templates *without* fixed tokens are assigned to that state the identifier is set to `*`. This also implies that the template is formed only by fields split by separators.

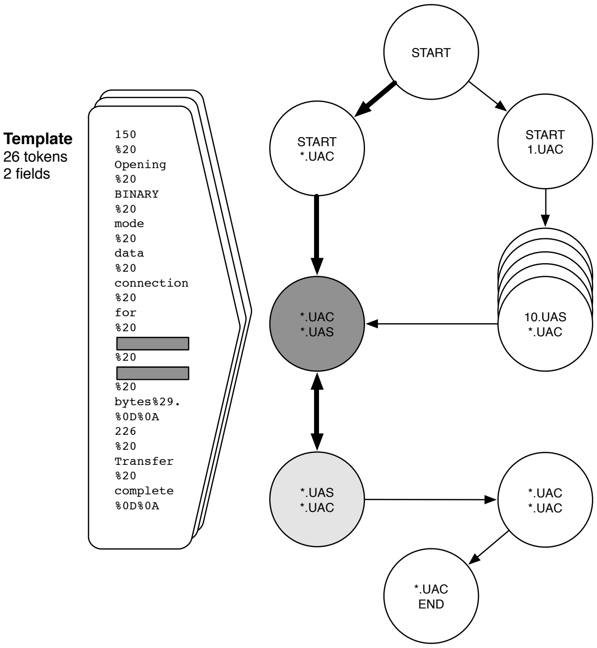
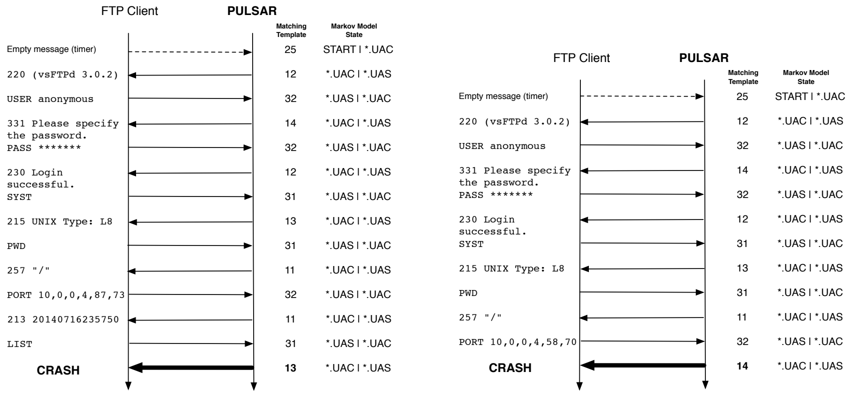


Fig. 3: State machine and example of template generated from FTP traces. The template contain 26 tokens and 2 of them are identified as variable fields.

By using the state machine generated by PULSAR we are able to trigger all of the 6 vulnerabilities reported by Seljan in the scope of `CVE-2014-4643` and two previously unknown buffer overflows vulnerabilities. Note that our approach does not require any prior knowledge of the FTP protocol or programming to trigger these bugs. Instead, it merely requires an independently learned state machine in order to impersonate a FTP server and have the client connect to it.

Figure 4 shows the two message sequences exchanged between the client and the fake FTP server—mimicked by PULSAR—resulting in the discovery of the two buffer overflows. In both sequences PULSAR first imitates the login procedure, allowing the



(a) Messages triggering a crash after a LIST command.

(b) Messages triggering a crash after a PORT command.

Fig. 4: Sequences of messages sent and received by the Core FTP client and PULSAR which lead to the termination of the client as a result of buffer overflows when the responses to the LIST and PORT commands are parsed.

client to authenticate itself by issuing a `USER` followed by a `PASS` command. The client then issues a `PWD` command in order to determine the current working directory to which the fake server responds with a seemingly valid directory. Next the client attempts to enter *active mode* by sending the `PORT` command to the server.

At this point in the communication the message sequences of Figure 4(b) and 4(a) diverge. While in Figure 4(b) PULSAR immediately responds with an overly long string causing the client to crash, in Figure 4(a) a valid response is sent back to the client and the dialog is kept alive. Subsequently the client issues the `LIST` command and crashes as result of an overly long response. Note that the client only crashes in response to the `LIST` command after entering active mode while remaining operational in passive mode. This highlights the necessity of stateful fuzzing to identify vulnerabilities located at deeper levels of the state machine.

3.2 Pidgin ICQ/AIM

In our second experiment, PULSAR is employed to learn a state machine for the *Open System for Communication in Realtime* (OSCAR) protocol, a lesser known binary protocol used by the AOL Instant Messenger and ICQ. OSCAR is an exceptionally complex protocol with a login procedure that comprises four stages and involves two independent servers, the *authorization server* and the *BOS server*. The authorization server has the responsibility to verify user credentials, generate an authorization cookie and redirect to a BOS server for all further processing.

In the past, several vulnerabilities in processing of BOS server messages have been identified in the popular instant messengers Pidgin and Adium. In particularly, several remotely triggerable crashes are known, which result from insufficient validation of UTF-8 strings sent to the client by the BOS server (`CVE-2011-4601`). We explore

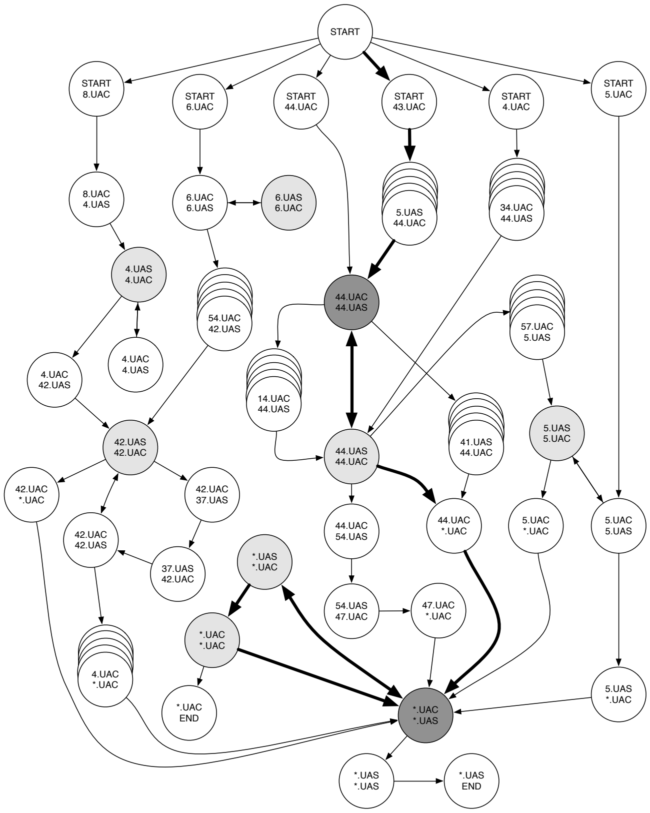


Fig. 5: Markov Model from OSCAR traces.

whether PULSAR is capable of automatically triggering these bugs, by generating a state machine for the BOS server from 512 network traces. To ensure that our BOS server is contacted, a firewall rule for `netfilter` is used to redirect all traffic sent to the real BOS IP address to our server, thus allowing the client to perform the first login stage with a real authorization server but effectively redirecting to our system all further requests issued from the client to the real BOS server.

Figure 5 shows the state machine learned for the communication between the ICQ/AIM client and the BOS server on port 5190. For clarity, large paths without fuzzable states are shown piled. The path through the model from the beginning of the communication to the state where the fuzzed message triggers the error in the client is highlighted.

Figure 6 shows in more detail the sequence of messages exchanged between Pidgin and the fake BOS server simulated by PULSAR. In combination with the Markov model it can be seen how the system is able to correctly complete the protocol negotiation phase with the client. After this phase, the client considers itself completely authenticated and the user can start interacting with the application. When the user requests to add a buddy

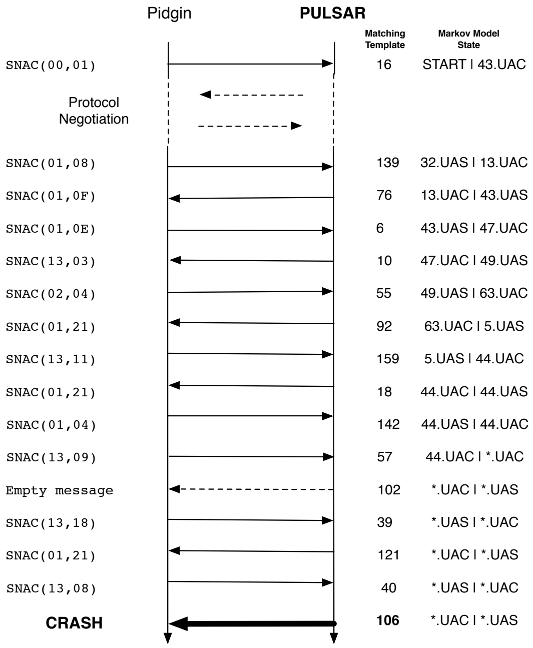

Pidgin
PULSAR
Matching Template
Markov Model State
SNAC(00,01) 16 START | 43.UAC
Protocol Negotiation
SNAC(01,08) 139 32.UAS | 13.UAC
SNAC(01,0F) 76 13.UAC | 43.UAS
SNAC(01,0E) 6 43.UAS | 47.UAC
SNAC(13,03) 10 47.UAC | 49.UAS
SNAC(02,04) 55 49.UAS | 63.UAC
SNAC(01,21) 92 63.UAC | 5.UAS
SNAC(13,11) 159 5.UAS | 44.UAC
SNAC(01,21) 18 44.UAC | 44.UAS
SNAC(01,04) 142 44.UAS | 44.UAC
SNAC(13,09) 57 44.UAC | *.UAC
Empty message 102 *.UAC | *.UAS
SNAC(13,18) 39 *.UAS | *.UAC
SNAC(01,21) 121 *.UAC | *.UAS
SNAC(13,08) 40 *.UAS | *.UAC
CRASH 106 *.UAC | *.UAS

Fig. 6: Sequence of messages sent and received by the ICQ/AIM client and PULSAR to produce a crash as a result of a missed format verification when parsing a negative response to a buddy list request.

to the list, our system fuzzes the response with an invalid UTF-8 sequence that triggers the crash of the client.

In summary, this experiment shows that PULSAR is capable of learning even complex and unusual binary protocols and trigger vulnerabilities deep within the state machine.

4 Limitations

Our experiments show that PULSAR is capable of identifying security flaws inside protocol implementations. Since the discovery of vulnerabilities, however, cannot be fully automated in the generic case due to Rice theorem [29], our method naturally has certain limitations. In this section, we examine these limitations and discuss possible improvements.

Our system strongly relies on the comprehensiveness and completeness of the observed network traffic and is thus unable to model protocol paths which do not occur in this traffic. This is a common problem of automatic inference approaches and can not be completely solved. Yet, it can at least be alleviated by incorporating knowledge about the protocol under test. For example, the analyst can specifically induce and record protocol functionality that is security sensitive or might be prone to vulnerabilities. Moreover, depending on the particular environment, the analyst may change relevant parameters of the protocol, such as addresses and usernames, to help constructing corresponding templates and rules in the model.

Similarly, our approach does not reconstruct type information of fields which can be help to significantly reduce the range of tested values, thus improving the efficiency of fuzzing. As a remedy, the analyst might manually assign types to certain fields. However, the presented results show that our approach is already capable to identify vulnerabilities without this information—thereby compensating the lack of type information.

As most network monitoring approaches, PULSAR is unable to deal with encrypted network traffic. Although this problem can not be solved in general, it might in some cases be possible to inspect traffic through a proxy that acts as man in the middle. The model can then be learned from the collected traffic prior to forwarding it to the destination. Similarly, the final fuzzing can be conducted by transmitting fuzzed messages directly through the proxy.

5 Related Work

PULSAR unites two research areas from computer security in the scope of network protocols: First, we reverse engineer the protocol by automatic inference and second, based on the learned specification fuzzy testing is applied to the communications parties in order to reveal security vulnerabilities in the implementations. In the following we attempt to provide an overview of work conducted in these two vivid fields of research.

Protocol Re-engineering. Originally, the task of reverse engineering a network protocol has been a time-consuming, demanding and above all, manual task. Over almost a decade of research, however, the community has significantly advanced this field by proposing numerous techniques for automating the task of protocol re-engineering.

Nowadays, state-of-the-art methods can be divided into two orthogonal strains of research: On the one hand, methods that utilize and instrument an existing implementation based on, for instance, dynamic taint-analysis [6, 9, 11, 24, 28, 34] and on the other hand, those that attempt to derive the protocol specification from recorded network data only [8, 10, 19, 20, 22, 23, 33]. The task of deriving a protocol model is especially challenging in case the analyst does not have access to a concrete implementation showcasing the protocol interaction, but network recordings only. This exactly is the specific field of operation PULSAR acts in and therefore, we subsequently discuss this line of research in more detail. Another key distinction can be made between stateless [5, 10, 20] and stateful protocol inference [8, 19, 23, 33]. Common to all approaches on reversing engineering network protocols is the need to differentiate variable from constant segments in the transferred data. In this respect many methods are based on or influenced by early work from Beddoe [4] and the *Protocol Informatics Project* [3] where sequence alignment algorithms from the field of bioinformatics were used to break up the protocol's messages into their individual components.

Roleplayer [10], for instance, extends this by certain heuristics for identifying IP addresses and domain names. In essence the method does not respect temporal states but already addresses the need for inter-field relations. Leita et al. [23] present a system (ScriptGen) that also makes use of sequence alignment algorithms but splits up its application over two phases of different granularity. A later extension of ScriptGen [22] is more relevant in our context. The authors enhance the approach such that it is able to address intra- as well as inter-protocol dependencies of variable fields and contents. This is particularly important for keeping alive recreated dialogs in a meaningful way. PRISMA [19]—the protocol inference framework we chose to build our method on—is able to accomplish this as well. Similarly, the authors of [33] make use of a Markov model and a layered application of the sequence analysis proposed by Beddoe just as ScriptGen does. Unfortunately, this approach is not able to relate variable fields over temporal states.

Protocol Fuzzing. Using fuzz testing it is possible to uncover security flaws in software by strategically generating input in an automated fashion [see 32]. Two levels of abstraction can be discriminated here: (a) *black-box fuzzing* [35] where a tester observes the software from the "outside" only seeing what in- and output is passed in or out respectively, and (b) *white-box fuzzing* [13] that allows the tester to inspect the code (either binary or source code) and for instance, make use of symbolic execution and constraint solving.

This separation obviously applies to protocol fuzzing as well. In this context however, it is crucial to differentiate between stateless and stateful systems. Fuzzing multi-party communication in a completely random fashion is foredoomed to fail. Only with the knowledge of the protocol's states and semantics at hand it is possible to navigate the fuzzer through the communication. This lead to stateful network fuzzers like KiF [1], SNOOZE [2] or Peachfuzzer [9, 12], whereby one differentiates special purpose [1], specification-based [e.g., 12] and model-based [e.g., 9, 14, 17] fuzzers. The latter kind is usually powered by protocol inference as discussed in the previous paragraph and as implemented by PULSAR. Our approach differs from this work in that it operates in absence of the code and the specification for a protocol and thus comes handy in cases where proprietary protocols are used, for example, in embedded systems.

Closest to PULSAR are the approaches AutoFuzz [14] and the system described by Hsu et al. [17], which both also infer the protocol state machine and message formats from network traffic alone. Although these approaches share the same practical setting with PULSAR, they do not make use of the inferred information for fully simulating communication, likely due to the absence of dependence rules that enable us to let data flow between protocol states.

6 Conclusion

Finding vulnerabilities in the implementations of proprietary protocols is a challenging problem of computer security. In this paper, we present a novel method for black-box fuzzing that can help to spot vulnerabilities in protocol implementations, even if neither the code nor the specification of the protocol are available. To this end, our method PULSAR builds on concepts of protocol reverse engineering and simulation that enable us to automatically infer and guide fuzzers for proprietary protocols. Our evaluation demonstrates the utility of such fuzzers, where we identify vulnerabilities in the implementations of a text-based and a binary protocol.

While we have applied PULSAR against rather common network protocols, the method is also suitable for searching bugs in unusual implementations, such as in embedded devices inside cars and industrial control systems. Due to the capability of operating without code and specification, a collection of network traces is sufficient for PULSAR to infer a first fuzzer for an unknown protocol. Moreover, the simple design of the generative model inferred by PULSAR also enables a practitioner to inspect and manually refine the model which provides a bridge to regular fuzzing with manually crafted protocol grammars.

Acknowledgements

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the project INDI (6KIS0154K) and the German Research Foundation (DFG) under the project DEVIL (RI 2469/1-1).

References

[1] H. J. Abdelnur, R. State, and O. Festor. KiF: A stateful SIP fuzzer. In *Proc. of International Conference on Principles, Systems and Applications of IP Telecommunications (IPTCOMM)*, pages 47–56, 2007.

[2] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *Proc. of Information Security Conference (ISC)*, pages 343–358, 2006.

[3] M. Beddoe. The protocol informatics project. `http://www.4tphi.net/~awalters/PI/PI.html`, visited July, 2015.

[4] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. Technical report, McAfee Inc., 2005.

[5] G. Bossert, F. Guihry, and G. Hiet. Towards automated protocol reverse engineering using semantic information. In *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2007.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. of IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[8] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2010.

[9] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Proc. of IEEE Symposium on Security and Privacy*, 2009.

[10] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2006.

[11] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2008.

[12] Deja vu Security. Peachfuzzer. , visited July, 2015.

[13] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[14] S. Gorbunov and A. Rosenbloom. AutoFuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security (IJCSNS)*, 10(8):239–245, 2010.

[15] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proc. of USENIX Security Symposium*, pages 49–64, 2013.

[16] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: data mining, inference and prediction*. Springer series in statistics. Springer, New York, N.Y., 2001.

[17] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proc. of IEEE International Conference on Network Protocols (ICNP)*, pages 114–123, 2008.

[18] J. Jang, A. Agrawal, , and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[19] T. Krueger, H. Gascon, N. Kraemer, and K. Rieck. Learning stateful models for network honeypots. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 37–48, Oct. 2012.

[20] T. Krueger, N. Kraemer, and K. Rieck. ASAP: Automatic semantics-aware analysis of network payloads. In *Proc. of ECML Workshop on Privacy and Security Issues in Machine Learning*, pages 50–63, Sept. 2010.

[21] D. Lee and H. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, 1999.

[22] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *Recent Adances in Intrusion Detection (RAID)*, Sept. 2006.

[23] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: An automated script generation tool for honeyd. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2005.

[24] Z. Lin, X. Jiang, and D. Xu. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2008.

[25] B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proc. of USENIX Security Symposium*, 2005.

[26] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34:129–153, 1956.

[27] H. Moore. Security flaws in universal plug and play: Unplug. don't play. Technical report, Rapid 7, 2013.

[28] J. Newsome, D. Brumley, and J. Franklin. Replayer: Automatic protocol replay by binary analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2006.

[29] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[30] I. Schieferdecker, J. Grossmann, and M. Schneider. Model-based security testing. *Electronic Proceedings in Theoretical Computer Science*, 80:1–12, Feb. 2012.

[31] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

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

[33] S. Whalen, M. Bishop, and J. P. Crutchfield. Hidden markov models for automated protocol learning. In *Proc. of International Conference on Security and Privacy in Communication Networks (SECURECOMM)*, pages 415–428, 2010.

[34] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2008.

[35] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling blackbox mutational fuzzing. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[36] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[37] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 499–510, Nov. 2013.