Radboud Universiteit Nijmegen

Faculty of Science

Master in Kerckhoffs Computer Security
Master Thesis

Protocol state fuzzing of an OpenVPN

Author:
Tomas Novickis
tomas.novickis@student.ru.nl

Supervisors:
Erik Poll
erikpoll@cs.ru.nl
Kadir Altan
KAltan@deloitte.nl

May 25, 2016

Contents

List of Figures

List of Tables

Nomenclature

AES	Advanced Encryption Standard
AH	Authentication Header
API	Application Programming Interface
DES	Data Encryption Standard
DFA	Deterministic Finite Automaton
DNS	Domain Name System
DoS	Denial of Service
ESP	Encapsulating Security Payload
GRE	Genetic Routing Encapsulation
HTML	HyperText Markup Language
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IKEv2	Internet Key Exchange
IP	Internet Protocol
IPSec	Internet Protocol Security
IPv6	Internet Protocol version 6
IV	Initialization Vector
L2TP	Layer 2 Tunneling Protocol
MAC	Message Authentication Code
NAT	Network Address Translation
OSI	Open Systems Interconnection model
PPP	Point-to-Point Protocol
PPTP	Point-to-Point Tunneling Protocol
PSK	Pre-shared Static Key
RFC	Request for Comments
SA	Security Association
SCP	Secure Copy
SFTP	Secure File Transfer Protocol
SSL	Secure Sockets Layer
SSTP	Secure Socket Tunneling Protocol
SUT	System Under Test
TCP	Transmission Control Protocol
TCP/IP	Internet Protocol Suite
TLS	Transport Layer Security
TUN	Network Tunnel
UDP	User Datagram Protocol
VM	Virtual Machine
VPN	Virtual Private Network

Abstract

Virtual Private Network is a widely used technology for secure data transmission. The purpose of a VPN is to provide a secure of way of transferring sensitive data between two or more parties over an insecure channel. Flaws in the implementations of security protocols are some of the most serious security problems. This paper describes a popular VPN solution, OpenVPN, as well as methodology used to infer state machines from a security protocol, using largely automated fuzzing techniques. If a vulnerability is found, an attacker may remotely exploit vulnerable systems over the Internet. State machines can be used to specify possible sequences of sent and received messages in different states of protocol. Learning techniques allow the automatic inference of the behavior of a protocol implementation as a state machine. Additionally, fuzzing is a well known and effective testing method which allows discovering different flaws within the implementations. Combining automatic state machine inference and protocol fuzzing, it is possible to produce a universal state machine which is a good representation of the implemented protocol structure. Manually inspecting these state machines allows for a straightforward way to possibly find bugs, inaccuracies or vulnerabilities in the implementation.

Keywords: virtual private networks, OpenVPN, state machines, protocol state fuzzing, vulnerability detection

1 Introduction

A Virtual Private Network (VPN) is a technology that provides a method of securing information transmitted over an insecure channel. By allowing users to establish a virtual private tunnel, it is possible to enable a secure access to the private or internal company resources, data, and communication services over the Internet. There are numerous VPN protocols: Point-to-Point Tunneling Protocol (PPTP), Layer 2 Tunneling Protocol (L2TP), Internet Protocol Security (IPsec), OpenVPN, to name the most popular ones. Some of these VPN solutions are largely complex and include multiple separate protocols with each of them having its own security and availability challenges. For instance, the implementation of IPSec protocol may use protocols like L2TP, Authentication Header and Encapsulating Security Payload (security protocols), IKE (key management) , and a variety of cryptographic algorithms for authentication and encryption. Considering the substantial number of VPN implementations and their variations available, it is reasonable to assume that at least some of them may have undiscovered vulnerabilities. Testing and possibly finding such bugs and vulnerabilities is the main goal of this research paper. The focus of this research paper is well known SSL VPN based protocol, called OpenVPN.

A communication protocol is a set of rules and conventions that describe how information is to be exchanged between two or more entities [4]. It defines message types that are comprised of a sequence of fields organized with specific rules. These messages are then being sent from one party to another which leads to protocol achieving distinctive unique states within itself. These states describe all the stored protocol information at a given instant of time. Given what is known about communication protocols, it is reasonable to assume that any given communication protocol can be modeled as a list of transitions (messages) and states.

Protocol state machines describe the message sequences that occur in the communication protocol during the session. In this research, the state machines will be used to learn how the protocol is implemented and how it behaves under the different conditions. A proven tool which is able to test communication protocols and provide a state machine based on the results is called LearnLib [5]. Using automated techniques, implemented in the LearnLib library, it is possible to automatically infer these the machines from the protocols. The more details about LearnLib library and its working will be explained in Chapter 3.2.1 and Chapter 3.3.

To use LearnLib, input alphabet of messages that are sent to the System Under Test (SUT) must be prepared. Likewise, an output alphabet of all possible received responses must be also be prepared.A special test harness will be constructed in order to act as an intermediate later between the LearnLib and the SUT. Then it will be run under varying conditions in order to monitor its behavior and inputs. Since the selected OpenVPN protocol has neither an official specification nor an RFC, it must be well analyzed and explored to be successfully implemented within the learning framework.

This particular testing technique used in this research is called fuzzing, It is commonly used to test systems for implementation flaws, such as bugs, vulnerabilities or stability issues. The idea of fuzzer is to send the target an unusual inputs with an intention to possible produce unexpected behavior. Fuzzing, as well as different fuzzing software will be explained in more detail in Chapter 3.3.

This goal of this research is to take a known security protocol, in particular, OpenVPN implementation, and use aforementioned fuzzing techniques, combined with state machine learning to derive an accurate state machine of the protocol. This state machine is supposed to give a clear representation of the protocol implementation. Next, manual analysis of the state machine can be done to observe if the implementation strictly follows and complies with the official specification. Furthermore, the state machine is tested for whether there is more than one path leading to a successful exchange of application data as well as checked for unusual or unnecessary behavior, possible bugs, inaccuracies or vulnerabilities in the implementation.

Organization. The thesis is organized as follows. Chapter 2 gives an overview on different research works related to learning state machines from network protocols. Chapter 3 presents an overview about different kinds of Virtual Private Networks and their implementations, automata theory and protocol fuzzers. Chapter 4 focuses on detailed explanation of OpenVPN protocol and its inner workings. Chapter 5 introduces test harness concepts and explains its purpose in this project. Chapter 6 presents practical work performed in this research which involve programming a test harness (custom OpenVPN client) and attaching it to the LearnLib fuzzing framework in order to test OpenVPN server. Next, in the chapters 7 and 8 the findings are presented and suggestions for improvement and future work are given. Finally, Chapter 9 contains results and conclusions.

2 Related Work

This chapter will give a quick overview on the realated work performed in the field of learning state machines from the security protocols.

Security communication protocols have not been explicitly studied using automatic inference and state machines. Nonetheless, there was some important research performed on few security protocols. J. de Ruiter performed protocol state fuzzing in eight TLS implementations [6] in order to find implementation flaws. To infer state machines from implementations J. de Ruiter used Angluin's L* algorithm's Java implementation called LearnLib. The following TLS implementations were found to have flaws, in particular, GnuTLS, Java Secure Socket Extension, and OpenSSL.

- *GnuTLS (3.3.8).* Sending HeartbeatRequest message during handshake results in an alert being returned in response to the Finished message. The handshake protocol would proceed but result in fatal error thus allowing to go to a corresponding state via so-called "shadow" path. Due to this bug, no integrity is provided for any previously received message.
- *Java Secure Socket Extension (1.8.0_20).* During regular TLS protocol run the two paths are possible leading to exchange application data. One of those paths is missing ChangeCipherSpec message from the client. Nevertheless, the server will respond with a ChangeCipherSpec message to a plaintext finished message from the client. Since the server can be tricked into accepting plaintext data, it invalidates any assumption of data integrity or confidentiality of data sent to the server [6].
- *OpenSSL (1.0.1i).* In version 1.0.1 is it possible to hijack a session by sending an early ChangeCihperSpec message to both the client and the server. This can be done because session keys are computed even when no ChangeCipherSpec is received.

F. Aarts et al. [7] presented an approach to infer models of entities in communication protocols, which also handles message parameters. They used abstraction, as used in formal verification, to the black-box inference setting. Authors successfully inferred state machines of fragments of Session Initiation Protocol (SIP) and TCP. Furthermore, F. Aarts et al. used the similar regular inference and abstraction approach to infer a model of Biometric passport [8] that describes how the passport responds to certain input sequences. Even though Biometric passport has quite a simple machine, it has several nuances. Also, the application sometimes exhibits non-deterministic behavior. Since LearnLib is restricted to infer behavior or deterministic Mealy machines, it cannot cope with non-deterministic behavior.

Merten et al. [9] presented a straightforward method on how to manually create application-specific data-aware test drivers for LearnLib. Finite state machines as protocol models were used and studied an active black-box checking algorithm and passive trace minimization algorithm on MSN instant messaging

protocol [10]. They found many previously unknown bugs in both MSN clients.

Fides Aarts, et al. presented an approach to infer models of entities in communication protocols, which also handles message parameters [7]. Their framework adapts regular inference to include data parameters in messages and states for generating components with large or infinite message alphabets.

The aforementioned research papers acted as an important source of material for this research. In particular, J. de Ruiter's [6] and F. Aarts et al. [7][8] papers were particularly helpful when examining testing possibilities and constructing a test harness.

3 Background

In the following chapter the overview of the virtual private networks, in particular, IPsec, SSL and PPTP VPN solutions (3.1), their common implementations (3.1.4), introduction to state machines (3.2), automata learning (3.2.1) and most commonly used fuzzers (3.3) is presented.

3.1 Virtual Private Networks

A VPN is a virtual network, built on top of existing physical networks, which can provide a secure communications mechanism for data and IP information transmitted between networks [11]. Its main purpose is to facilitate the secure transfer of sensitive data across insecure channels (public networks). There are three major families of VPN implementations in wide usage today: Secure Sockets layer (SSL), IPsec and PPTP-based VPNs. They are the most widely applied by big corporations thus requires the highest level of reliability and security. The main attributes of the VPN include [11]:)

- **Confidentiality** means that no one can read the captured traffic at the packet level. All of the information going through the insecure (public) channel is encrypted and leaks no information about actually sent data. This is achieved using encryption primitives.
- **Authentication** means that only the specifically assigned users can communicate using a secure VPN network. This prevents the unauthorized or impersonating users to access the data in the VPN network.
- **Integrity** assures that none of the data is being transmitted was being tampered between the authenticated sender and receiver.

Before going into details of VPNs, it is important to understand the structure of networking. Each network protocol resides on a specific layer, defined in several different conceptual models, like OSI (Open Systems Interconnection) model [1] or TCP/IP (Internet Protocol Suite) [2] model (Tab.1).

OSI model has seven network layers - Physical, Data link, Network, Transport, Session, Presentation, Application, while TCP/IP has four network layers - Data Link, Network, Transport, and Application. Each layer adds more information to the data packet and higher level layers are not aware of lower level layer functions. Thus a security control at a higher layer cannot provide full protection for lower layers. Different VPNs provide security at different network layers and this is important to consider when choosing a VPN solution as it may determine the easiness to use it in the company. The three most commonly used VPN technologies are SSL, IPsec, and PPTP. Each of them resides on different network layers. Nevertheless, the layering of VPNs is not very straightforward as they usually use multiple protocols which are present in different layers.

For instance, in the OSI model, TLS protocol resides between Transport layer (TCP protocol) and Application layer. However, as OSI is just a conceptual

Model / Layer	OSI	TCP/IP
1	Application	Application
2	Presentation	Transport
3	Session	Internet
4	Transport	Link
5	Network	
6	Data Link	
7	Physical	

Tab. 1: OSI [1] and TCP/IP [2] computer networking models

model and protocols are often very complicated, it is tough to assign a concrete layer for each protocol.

As mentioned previously, in order to perform a successful fuzzing of a security protocol the client implementation of OpenVPN will be made to communicate with the OpenVPN server. To achieve this, it is important to at least grasp the network fundamentals, some of which were mentioned in this chapter. During this research a lot of time was spent working on low level components of the OpenVPN implementation. Since OpenVPN operates inside the User Datagram Protocol (UDP) protocol (explained in Ch. 4), most of the work have been done working on the Transport layer (OSI model), which carries UDP protocol. Additionally, during the construction of the test harness, there were some choices made in regards on how to best implement network layers using Python sockets and Scapy abstraction layers. This will be explained in more detail in Ch. 6.

3.1.1 IPsec VPNs

This section gives a brief summary on fundamentals of one the most commonly used VPN protocol - IPsec.

IPsec is a framework of open standards for ensuring private communications over IP networks which has become the most commonly used network layer security control [11]. IPsec is based on securing Network layer of TCP/IP model. In many environments securing Network layer is a better solution than securing higher Transport or Application layers. It makes a way for network administrators to enforce certain security policies, and also provides a more flexible way in protecting IP information for each packet [11]. Depending on the implementation IPsec can provide a combination of following security measures: confidentiality, integrity, peer authentication, replay protection, traffic analysis protection and access control [11]. The three commonly used architecture types used in IPsec, are [11]:

1. *Gateway-to-Gateway architecture.* VPN gateway may be a dedicated device or just a part of network device, such a router or firewall [11]. VPN

gateway essentially separates internal network from anything over other side of gateway.

2. *Host-to-Gateway architecture.* Primarily used to provide secure remote access. The host (IPsec client) uses VPN tunnel to connect to a VPN gateway. Whenever a host wishes to create a VPN connection to a server, he must authenticate and establish a connection with a gateway, which then manages host's connection to a VPN server.

3. *Host-to-Host architecture.* Least common architecture type, mostly used by system administrators to perform remote management. In this scenario direct connection is established between two separate hosts - one host acts as a VPN client and another as a VPN server.

In all three architecture types, in order to connect to the host, typically must authenticate itself. This is usually done either by gateway itself of by consulting a dedicated authentication server.

As noted earlier, IPsec uses multiple additional protocols to establish a secure connection [11]:

Authentication Header (AH) , defined in RFC 4302 [12], provides integrity protection for all packet headers (except few IP header fields) and user authentication. Optionally it can provide replay and access protection. AH is not able to encrypt data.

Encapsulating Security Payload (ESP) , defined in RFC 4303 [13], has two modes: tunnel and transport. Tunnel mode can provide encryption and integrity protection for an encapsulated IP packet as well as authentication for ESP header, while transport mode can provide encryption and integrity protection for the payload of an IP packet and integrity protection for the ESP header.

Internet Key Exchange (IKE) is used to negotiate, create and manage Security Associations (SA) [11]. SA is a set of rules needed to define the features and security mechanisms for the establishing a IPsec connection. It can be defined manually, however it does not scale well with large-scale VPNs. A more common method is using one of the five possible IKE exchange modes - main, aggressive, quick, informational or group. The modes differ in speed and the usage of their cryptographic primitives for establishing a secure connection. IKEv2 is the newest version of IKE, and it improves the protocol in the following areas: clearly defined RFC (RFC 5996 [14]), simplicity, reliable message delivery, protection against Denial of Service (DoS) attacks, and improved usage of IKE with Network Address Translation (NAT) gateways [11].

3.1.2 SSL VPNs

Originally known as Secure Socket Layer (SSL), and later renamed to Transport Layer Security (TLS), TLS is a protocol designed to provide a secure connection over an insecure network. TLS achieves the following goals: confidentiality, integrity of data, authentication of server and authentication of the client [15]. It

uses X.509 certificates with asymmetric cryptography to authenticate the counterparty and negotiate a symmetric session key. To establish a secure connection, TLS uses *Handshake*, *ChangeCipherSpec* and *Alert* subprotocols. Multiple different implementations of TLS were fuzzed and tested by J. de. Ruiter [6] and therefore, this research is not concerned about the testing security of TLS itself.

An SSL VPN can tunnel an entire network's traffic or secure an individual connection. One advantage of SSL VPNs over an IPsec VPNs is that SSL VPNs can connect to more restricted environments where NAT (Network Address Translation) or strict firewall rules are used. The reason for this is because most organizations do not filter the traffic on the TCP port 443, as it is usually used for employees to securely access Internet. For instance, OpenVPN, SSL-based VPN, uses UDP Port 1194 for secure data transmission. However, in the case that port is filtered, OpenVPN can also make use TCP port 443. The advantages of using UDP as a transport protocol are discussed at the beginning of the Chapter 4.

There are two primary types of SSL VPNs, namely SSL portal VPNs and SSL tunnel VPNs [15].

- SSL portal VPN works over a single network port, namely TCP 443 and acts as a Transport layer VPN. It allows users to connect with most web browsers to access web related content.
- SSL tunnel VPN is used to access multiple network services through a tunnel that is running under SSL. The main difference from portal SSL VPNs is that tunnel SSL VPNs allow accessing multiple network services, including applications and protocols that are not web-based. The requirement for SSL tunnel VPN is that it must be able to handle different active content like Java, JavaScript, Flash, ActiveX [15]. Being able to use more services, this tunnel has more capabilities compared to a portal type VPNs, but it may prevent some users from being able to connect to VPN. The SSL VPN tunnels are created in SSL, but just like in IPsec tunnels, IP traffic is fully protected by the tunnel [15].

Note: in this research the abbreviations SSL and TLS are used interchangeably. In general, newer version (TLS) is preferred, however some systems and protocols are very well known for their name which includes SSL, e.g. SSL VPN.

3.1.3 PPTP VPNs

Point-to-Point Tunneling Protocol (PPTP) is a virtual private network implementation method which uses TCP control channel and a Generic Routing Encapsulation (GRE) tunnel to encapsulate Point-to-Point Protocol (PPP) [16] packets and send them over TCP/IP links. The protocol was developed by a vendor consortium and documented in RFC 2637 [17]. PPTP encapsulated virtual network packets inside the PPP packets, which are then encapsulated

inside the GRE packets and these encapsulated inside the TCP control channel. Everything is then sent over IP network on TCP port 1723.

3.1.4 VPN implementations

There are number of VPN implementations available both closed and open source as well as for multiple platforms. Several known VPN packages are listed below.

- StrongSwan is an open source IPsec implementation for the Linux operating system [18]. Maintained by Andreas Steffen, strongSwan supports features, such as IPv6, Android 4+, X.509 public key certificates, certificate revocation lists, RSA private key storage on smartcards, ability to interoperate with various MS Windows and Mac OS X VPN clients, full implementation of IKEv2 protocol, and much more.
- OpenVPN is an SSL VPN implementation which implements OSI layer 2 or 3 secure network extension using the industry standard TLS protocol [19]. Specifics of OpenVPN are explained in detail in Chapter 4.
- SoftEther [1] is an open source multi-protocol VPN software. It supports a variety of VPN protocols: OpenVPN, L2TP/IPsec, L2TPv3/IPsec, EtherIP, Microsoft Secure Socket Tunneling Protocol (SSTP), VPN over HTTPS (SSL VPN), VPN over Domain Name System (DNS), VPN over Internet Control Message Protocol (ICMP).

Both IPsec and SSL-based VPNs were considered for this research. Seeing as IPsec is usually very complex, comprising of multiple protocols, the OpenVPN was chosen to perform the security tests. It would require considerable time to modify or create an IPsec client which establishes a full IPsec connection using multiple complex protocols. Nevertheless, the advantage IPsec has over the OpenVPN, is that there exist an RFC specification document for each of IPsec security protocols (IPsec [20], AH [12], ESP [13], IKEv2 [14]), while the OpenVPN does not have one.

3.2 State Machines

This chapter will provide a brief summary on state machines, and how they will be applied in this research.

A Mealy machine [21] is a finite state machine where every transition involves an input and a resulting output. Its output is determined both by its current state and current outputs. A *Mealy machine* is a tuple $A = \langle I, O, Q, q_0, \delta, \lambda, \rangle$, where I, O and Q are nonempty sets of input symbols, output symbols, and states, respectively; $q_0 \in Q$ is the initial state; $\delta : Q \times I \to Q$ is the transition function; and $\lambda : Q \times I \to O$ is the output function. Elements of I^* and O^*

[1] www.softether.org

are input and output strings respectively. In this research *Mealy machines* are used to model a complete structure of an OpenVPN protocol. Next chapter describes how state machines can be used to learn network protocols.

3.2.1 Automata Learning

Automata (State Machine) learning, also known as regular inference, can be used to create formal models of different real-time systems. In recent years it was applied to learn models for electronic passports [8], telephony systems [22], and communication protocol entities [23]. The main challenge to infer the correct state machine lies in correctly preparing application-specific learning setup. This includes determining a suitable form of abstraction and finding ways to manage concrete runtime data that influences the behavior of the target system [9].

Communication protocols can be modeled with simple state machines (Deterministic Finite Automaton (DFA), Mealy) or mode advanced Extended Finite State Machine (EFSM) [23].

To infer state machines LearnLib library uses L* algorithm, which was first described by Dana Angluin in 1987 [24]. Niese extended the L* algorithm to support Mealy machines which are implemented in LearnLib. It is an active query algorithm which puts queries consisting of an input string to a certain teacher and uses the returned output to model a Mealy machine.

A so-called *Learner* (i.e. LearnLib library), who initially has no knowledge about the Mealy machine $\mathcal{M}$, can ask queries to a *Teacher*, who knows the automaton [25]. There are two possible queries:
– A *membership query* consists of sending an input string to a *Teacher* and receiving a response back.
– An *equivalence query* consists of asking whether a hypothesized machine $\mathcal{H}$ is equivalent to $\mathcal{M}$. the *Teacher* will answer *yes* if $\mathcal{H}$ is correct or else provides a counterexample.

Normally *Learner* keeps asking a sequence of membership queries until a solid hypothesis $\mathcal{H}$ can be built from responses. Next equivalence query is made to find out whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. If equivalent, then *Learner* has succeeded, or else the returned counterexample is used to perform subsequent membership queries until hypothesized automaton is equivalent to $\mathcal{M}$ [25].

In the experiments performed, the *Learner* is the LearnLib instance and the Teacher is the OpenVPN server. However, the *Learner* cannot simply send the queries to the *Teacher* and expect it to understand. OpenVPN server works by sending and receiving specific packets which include different set of dynamic components. An abstraction layer must be created to translate LearnLib Learner queries to the packet format understandable to Teacher (OpenVPN server).

An *Oracle* for $\mathcal{M}$ is a device which accepts and inclusion query of hypothesis $\mathcal{H}$ as input, where $\mathcal{H}$ is a Mealy machine with inputs I [7].

The two possible challenges when implementing *Learner*-*Teacher* setup are [9]:

- *Authentication.* Security protocols usually require valid authentication tokens or valid login credentials to proceed to next communication step.
- *Dependences on substructures:* Often returned data structure cannot be directly used as a parameter value for different methods and some understanding of the application's data structures is needed in order to follow protocol.

3.3 Protocol Fuzzers

Fuzzing is a testing technique used to find flaws and unexpected behavior in the software [10]. During the fuzzing process, the system under test is provided with random, unexpected or invalid data, which essentially targets the corner cases, and is not considered during software development. Below some of the more popular fuzzing frameworks are presented. Since often the fuzzed system is complex, it is not possible to have a single tool to test them all. These fuzzing frameworks allow creating a specific fuzzing solution for a particular system. Some of the possible attributes of a fuzzing framework are:

- Support for abstract with the goal to minimize the number of tedious tasks.
- Support for intelligent fault injection - a known input, previously known to cause system problems. An example would be different SQL, string injections, format string, and directory traversal.
- Possibility to interact with a debugger attached to the target system to see more clearly how the system reacts.
- Support for a wide range of formatted data to be included in the fuzzing script.
- Support for code to reuse and saves a lot of time when building fuzzing components for different software products in the future.

The following chapter will present and give a short summary on the five popular fuzzing frameworks - Snooze, Peach, Sulley, Netzob and LearnLib.

3.3.1 Snooze

Snooze is a tool used to construct stateful network protocol fuzzers [26]. It includes Fault Injector, the Traffic Generator, the Protocol Specification Parser, the Interpreter, the State Machine Engine, and the Monitor.

1. The Fault Injector tries to manipulate and fuzz messages in such way, which may trigger a fault in the target system.
2. The Traffic Generator transforms user received messages into network packets and sends them to the target system. It may update certain fields like checksums or content length fields.

3. The Parser is used to parse the protocols specification so other parts of the tool can understand the data.

4. The Interceptor is the main component which runs the fuzzer and must be provided with protocol specifications, user scenarios and a module implementing scenario primitives.

5. The State Machine Engine keeps track of the state of transmitted and received messages. It can check if messages trigger the state change from the current state of a new state.

6. The Monitor performs the behavior and the traffic data of the target system. It is able to find faults, abnormal behavior and unexpected output from the fuzzed target [26].

3.3.2 Peach

Peach [2] is a fuzzer, supporting a variety of features, e.g. multiprotocol support, stateful (context-aware) fuzzing, multithreaded/distributed fuzzing, ability to parse responses, distinguishing between data types and vulnerabilities. Main basic components of the Peach are:

1. The Generator generates input data which is sent to the target system. This can range from simple strings to complex data types.

2. The Transformer changes the data in a specific way. This is usually an encoding (base64, HyperText Markup Language (HTML), and gzip).

3. The Publisher implements a form of transport for generated data through a protocol to a target system.

4. The Group contains one or more generators.

3.3.3 Sulley

Sulley is a Python-based fuzzing framework [27]. A few notably included features of Sulley are packet capture, VMware automation, crash reporting, ability to restart the target application in the event of a crash and continue fuzzing. Just like Spike, it uses block-based fuzzing. Sulley components include primitives, blocks, groups, encoders, dependencies, block helpers, and legos. Sulley works with several agents that are described below.

- The network monitor agent monitors network communications (using pcapy) and logs data to PCAP files.
- The process monitor agent is responsible for detecting faults. When the fault is triggered, it is both logged and transmitted back to the Sulley session.

[2] www.peachfuzzer.com

- The VMware control agent provides an Application Programming Interface (API) for interacting with a Virtual Machine (VM). It can make snapshots of the current VM state and if needed it can revert back to them.
- The web monitoring agent provides a graphical user interface for interaction with a fuzzer as well as graphical representation of fuzzing progress [27].

3.3.4 Netzob

Netzob [3] is an open source tool for reverse engineering, traffic generation and fuzzing of communication protocols. It allows inferring the message format and the state machine of a protocol through passive and active processes. Netzob uses specific algorithms to learn and represent vocabulary of the protocol. The tool can also semi-automatically learn state machine (grammar) of a protocol using specific algorithms. It also supports protocol simulation which is based on inferred protocol.

3.3.5 LearnLib

LearnLib is a free, open source Java library for automata learning algorithms [5]. LearnLib provides Java implementation of Angluin's L^* algorithm is able to produce finite automata and Mealy machines. LearnLib consists of three main interfaces: LearningAlgorithm, MembershipOracle, and EquivalenceOracle.

1. LearningAlgorithm encapsulates implementations of learning algorithms and offers three methods: *startLearning*, *getHypothesisModel* and *refineHypothesis*.
2. MembershipOracle encapsulates any structure that can answer membership queries. For Mealy machine learning, this Oracle provides system output for every single input symbol. This interface has a single method called *processQueries*. This method has to be provided with a collection of query objects that have to be processed by the membership Oracle.
3. EquivalenceOracle is used to analyze counterexamples. Since conjectures created by learning algorithms are not fully complete after the initial learning round, the mechanism is needed to refine conjectures by analyzing counterexamples that reveal mismatches between conjecture and SUT [5].

This research is focused on the LearnLib as a learning-fuzzing framework. The main reason for latter is that several related research projects were performed using LearnLib which yielded positive results. Additionally, most other fuzzing frameworks do not provide a way to infer a state machine based on observed client-server interactions.

[3] www.netzob.org

4 OpenVPN

In this chapter the OpenVPN and its inner workings are further examined. Before selection of the OpenVPN protocol, another VPN protocol - IPsec was considered. IPSec, however, provides a full low-level specification which is important to anyone trying to implement VPN themselves. The OpenVPN, on the other hand, does not have an official specification, instead, a quick low-level summary on its security, as well as documents for configuring the VPN itself. Additionally, the OpenVPN has quite a lot of useful comments in the source code, however they do not always match the observed behavior or a low-level summary provided on the official website. Due to this reason, it was decided to document the OpenVPN protocol in a low-detail understandable fashion, which could be used by anyone trying to write a custom implementation of the OpenVPN. To achieve this, multiple sources of information were studied and analyzed. Notably, low-level summary on the official website, source code, captured the OpenVPN packets in Wireshark, VPN related books, official support forums. Following chapter will provide a low-level details on the OpenVPN protocol based on the collected and summarized data.

4.1 Introduction

The OpenVPN is a SSL VPN which implements the OSI layer 2 and 3 secure network extension [19]. It allows any IP subnetwork being tunneled over a single UDP or TCP port and completely relies on the security of OpenSSL. The high-level client-server communication is shown in Fig. 1. Just like other VPNs, the OpenVPN provides the essential security services, such as authentication, encryption, integrity protection, and access control.

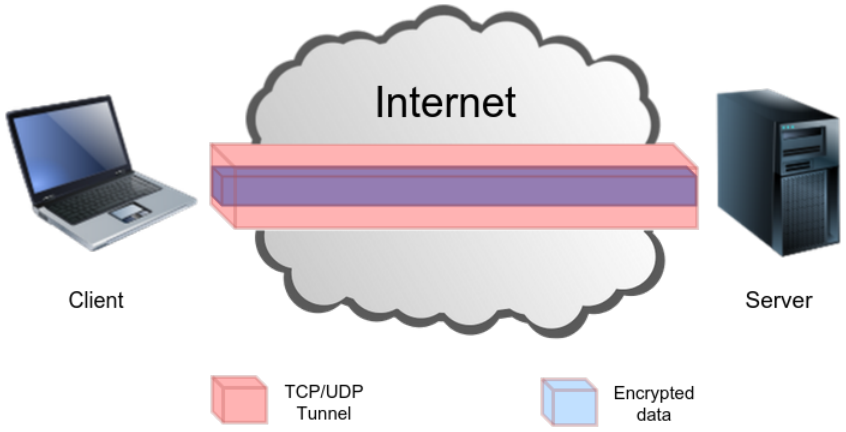

Fig. 1: OpenVPN tunnel

The OpenVPN utilizes the OpenSSL encryption library which itself implements TLS. Originally known as Secure Socket Layer (SSL), TLS is a protocol designed to provide a secure connection over an insecure network. TLS uses X.509 digital certificates[4] with asymmetric cryptography to authenticate counterparty and negotiate a symmetric tunnel session key. TLS is most widely known for being used over Hypertext Transfer Protocol (HTTP) to provide encryption and authentication, which led to the naming of HTTPS, which stands for HTTP Secure. To establish a secure connection, TLS uses Handshake, ChangeCipherSpec, and Alert subprotocols. Multiple different implementations of TLS, including OpenSSL, were fuzzed and tested by de. Ruiter [6] and therefore this paper is not concerned about testing security of TLS itself.

Unlike a vanilla TLS, the OpenVPN gives the user an opportunity to use a static key (or pre-shared passphrase) to generate a what is known as *HMAC firewall*, which authenticates the whole TLS handshake sequence. The details of *HMAC firewall*, as well as the security features it provides are explained in Chapter 4.5. The OpenVPN multiplexes the TLS session used for authentication and key exchange (Control Channel) with the actual encryption tunnel data stream (Data Channel)(Fig. 2) [3]. As UDP is connectionless protocol, the encrypted and signed IP packets are tunneled over UDP without any reliability guarantee. The reliability needed for secure authentication is provided by the TLS protocol which uses TCP as its reliability layer. It is important to note that control and data channels are explained separately, however, they are inside the same UDP (or TCP) tunnel hence on the same network layer.

The general structure of the OpenVPN packet and ensuing encapsulation can be seen in Fig. 3. It is explained in detail further in section 4.1.1. As shown, the OpenVPN data is encapsulated inside the UDP (or less commonly TCP) layer. The structure shown in Fig. 3 applies to all OpenVPN packets, however, different packets will have different OpenVPN payloads. For instance, `P_CONTROL_V1` packets will have TLS protocol data encapsulated inside the OpenVPN payload.

Default configuration

The OpenVPN provides a lot of different options for configuring and establishing the VPN connection. This chapter (Ch. 4) in detail describes the OpenVPN protocol mechanics and structure without going into the details of different configuration options the implementation itself may provide.

It is important to note that due to various ways of configuring the OpenVPN and selecting different options, it is unfeasible to provide a clear structure for every type of packet possible. Different options enable various functions, and thus, the packet format and structure will differ. Below the packet structure of the default OpenVPN configuration is shown.

[4] Digital certificates are used to ensure that the public key belongs to someone who claims to be the owner of that public key. The certificate contains the details (name, serial number, expiration dates, a digital signature of Certificate Authority (CA)) of the person or organization as well as public key [28].

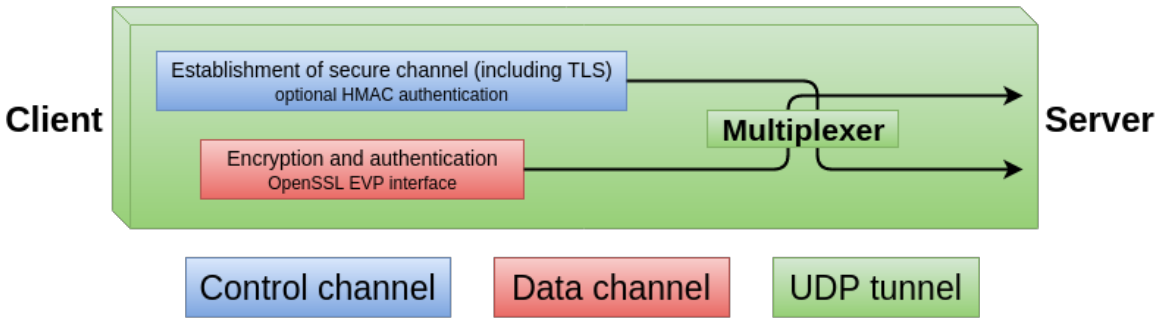

Fig. 2: OpenVPN using multiplexer

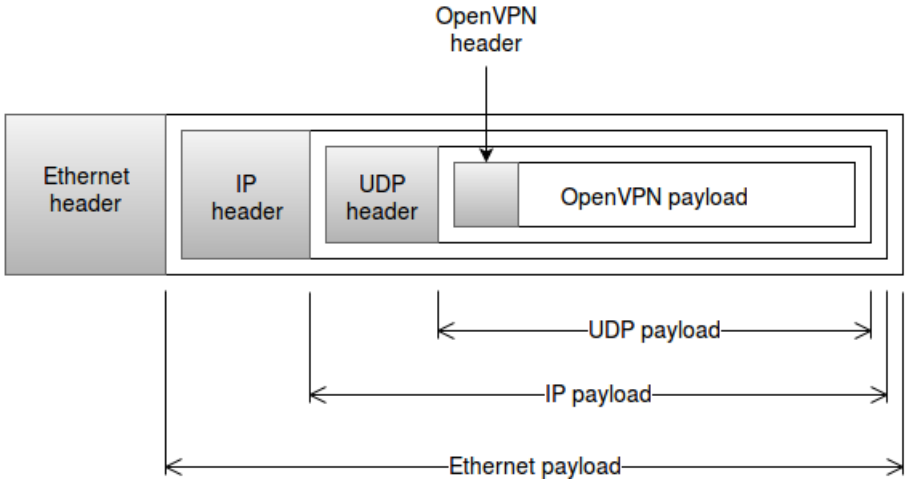

Fig. 3: OpenVPN packet encapsulation

- TLS mode (as opposed to pre-shared static key mode) - uses SSL/TLS protocol + certificates for authentication and key exchange (explained in Chapter 4.2.2).
- HMAC authentication of packets which are the part of TLS mode authentication sequence. It is enabled using `--tls-auth` option in the configuration, and protects against buffer overflows (explained in Chapter 4.5).
- TLS mode key method #2. Defined by `--key-method 2` option (explained in Chapter 4.2.2).

4.2 Authentication modes

The OpenVPN provides two different authentication modes. The main mode uses TLS with certificates and/or user's credentials to provide security. Another

mode uses pre-shared static keys, which by default provides authentication and encryption. There are multiple differences and security considerations for using each of the modes. The high-level comparison can be seen in Tab. 2.

The aforementioned comparison table compares even slight differences the two modes may have, which may not be relevant in real world scenario. For instance, TLS mode is described as having the slower speed, however, that entirely depends on how often the tunnel session key renegotiation is performed. Granted that the OpenVPN tunnel session keys are not often renewed, the difference becomes marginal as the tunnel encryption itself is performed using symmetric keys in both pre-shared static key and TLS modes. Public-key cryptography is only used to establish secure channel so exchange of symmetric key can take place. This is explained in more detail in Chapter 4.2.2.

Both the pre-shared static key mode and the TLS mode provide equivalent security assuming they are implemented correctly and use identical cryptographic primitives. That said, pre-shared static key mode does not provide the key renewal mechanism and thus perfect forward secrecy[5] is not provided. This could be regarded as a security concern. Since pre-shared static key mode uses symmetric-key algorithm[6], it is simpler to implement and has improved performance when it comes to cryptographic operations. However, pre-shared static key mode has very limited scalability as no automatic key-exchange is possible. In addition, symmetric key must be present on each OpenVPN peer in a plaintext form which is an additional security challenge. Peer authentication in the pre-shared static key mode is provided only by the ownership of the shared symmetric key owned by each of the peers.

Both authentication modes are explained in more detail in the next two chapters.

4.2.1 Pre-shared static key mode

The first authentication mode of the OpenVPN is pre-shared static key mode. In pre-shared static key mode, the encryption and authentication are performed solely using a static key, generated and shared between the OpenVPN peers before the tunnel is started. This static key contains four independent keys: HMAC send, HMAC receive, encrypt and decrypt. By default in static key mode, both hosts will use the same HMAC key and the same encrypt/decrypt key. In the official OpenVPN implementation, the key is derived using random bits obtained from the RAND_bytes OpenSSL function. Authentication in this mode is provided solely by the ownership of pre-shared static key.

The main security challenge of a pre-shared static key mode is that static key must be manually shared between all involved parties. Ideally, the most secure way to transfer a symmetric key would be to physically deliver it to

[5] Perfect forward secrecy ensures that compromise of long-term keys does not compromise the tunnel session keys used in the past.

[6] Symmetric-key algorithm is a cryptographic algorithm which uses the same key for both data encryption and decryption.

the VPN peer, however, this method certainly does not scale well, as the key renewal procedure would be greatly inefficient. An alternative method would be automatically distributing keys via SSH on a daily basis, so any compromise of the key would only allow an attacker to decrypt traffic captured on that particular day. This method, however, would largely rely on the security of an SSH implementation. Common ways of doing that would be using SSH based protocols - Secure Copy (SCP) or Secure File Transfer Protocol (SFTP) to securely transfer keys over the Internet.

The aforementioned static key has 2048 bit length, and consists of 512 bit encrypt, decrypt, HMAC send, and HMAC receive keys. Despite the long key, the default implementation only uses 128 bits (Blowfish) for cipher and 160 bits for HMAC (SHA1), which is used for tampering protection. The reason behind such a large key is to accommodate future ciphers and HMAC hashes which may require longer keys[3].

Static key mode by definition does not provide Perfect Forward Secrecy, so if the symmetric key is stolen, all the communications previously encrypted by that specific key are compromised. This is an important security consideration, which should be well thought over by those using the OpenVPN.

It is good to understand how the OpenVPN handles pre-shared static key authentication and encryption, however this mode does not scale well and usually most organizations prefer to use TLS mode.

4.2.2 TLS mode

Second, and the most popular OpenVPN authentication mode is TLS mode. In contrast to pre-share static key mode, TLS mode uses TLS protocol to authenticate, establish secure channel and exchange the symmetric tunnel session key between peers. Just like in pre-shared static key mode, session key is used to encrypt the data tunnel, however, the authentication and symmetric key exchange take place using TLS protocol. This not only provides an automatic and secure way of distributing symmetric keys, but also a way to renew the symmetric key at any point during the communication. The aforementioned aspect of the TLS mode provides the Perfect Forward Secrecy, which is not present in pre-shared static key mode.

The structure of the tunnel session key derivation TLS packet, as shown in Wireshark, can be seen in Fig. 4.

The transfer of tunnel session keys are encrypted and carried inside the TLS Record layer, so it cannot be decrypted without the proper TLS certificates. The two main steps in this protocol are shown below.

1. Negotiation of the TLS connection. Both sides of the connection are authenticated by exchanging certificates and verifying the certificate of the opposing side. If the authentication is successful, the protocol proceeds wit the step two. Otherwise, the connection is terminated.

2. Tunnel session keys are negotiated over the already established secure TLS

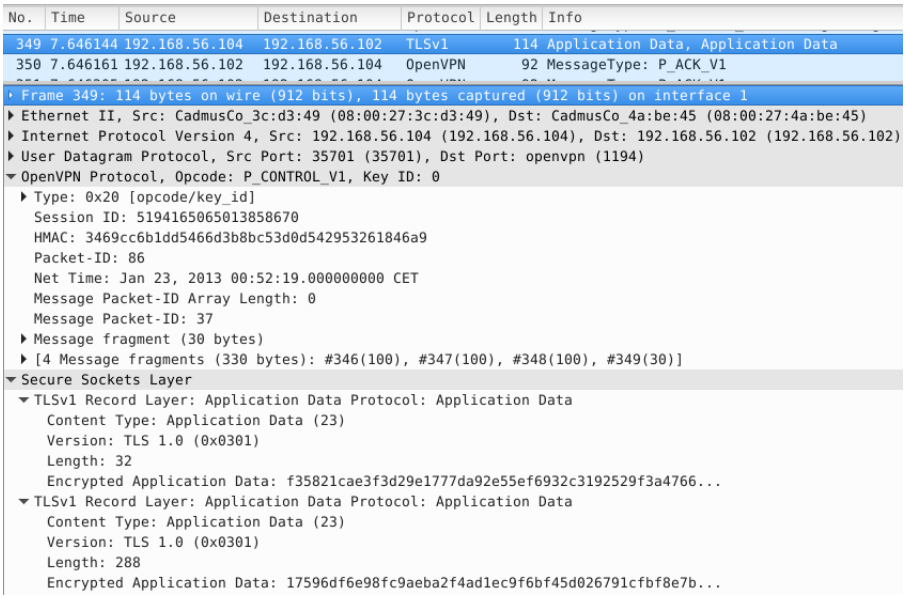

Fig. 4: Session key derivation TLS packet as seen in Wireshark

channel. The tunnel session key derivation TLS packet structure depends on the OpenVPN key method being used. TLS mode supports two key methods, which are described below.

(a) If the first key method is used, then the tunnel session keys are derived from OpenSSL cryptographic library *RAND_bytes* function. `P_CONTROL_HARD_RESET_CLIENT_V1` and `P_CONTROL_HARD_RESET_SERVER_V1` messages are used to initiate key derival procedure. The tunnel session key derivation TLS packet structure is shown in Tab. 3.

(b) If the second key method is used, (default in the OpenVPN 2.0+), then the tunnel session keys are derived from the *RAND_bytes* function passed through the TLS pseudo-random function (TLS PRF). TLS PRF mixes random key material using entropy from both sides of the connection. `P_CONTROL_HARD_RESET_CLIENT_V2` and `P_CONTROL_HARD_RESET_SERVER_V2` messages are used to initiate key derival procedure. Tab. 4 shows how the TLS plaintext packet is assembled.

In order to successfully construct a OpenVPN client, it is important to understand the key differences explained in this chapter between the TLS modes, key methods and their respective packet structures.

	Pre-shared static key mode	**TLS mode**
Crypto mode	Symmetric	Asymmetric & Symmetric
Handshake	No	Yes
Implementation	Simpler	More complex
Speed	Faster	Slower
CPU consumption	Smaller	Higher
Scalability	Limited	Very good
Key exchange	No	Yes
Encryption keys renewal	No	Yes
Peer authentication	Yes	Yes
Perfect forward secrecy	No	Yes

Tab. 2: Comparison of pre-shared static key mode and TLS mode

No.	**Type**	**Note**
1	Cipher key	n bytes
2	HMAC key length in bytes	1 byte
3	HMAC key	n bytes
4	Options string	n bytes, null terminated, client/server options string should match

Tab. 3: The tunnel session key derivation TLS packet using first key method [3]

No.	**Type**	**Note**
1	Literal 0	4 bytes
2	Key method type	1 byte
3	key_source structure	
4	options_string_length, including null	2 bytes
5	Options string	n bytes, null terminated, client/server options string must match
6	username_string_length, including null	2 bytes
7	Username string	n bytes, null terminated
8	password_string_length including null	2 bytes
9	Password string	n bytes, null terminated

Tab. 4: The tunnel session key derivation TLS packet using second key method (items 6 to 9 are optional) [3]

Value	Op code	Message type	Explanation
1	0x01	P_CONTROL_HARD_RESET_CLIENT_V1	Initiate TLS mode handshake using first key method
2	0x02	P_CONTROL_HARD_RESET_SERVER_V1	Reply to TLS mode handshake initiation using second key method
3	0x07	P_CONTROL_HARD_RESET_CLIENT_V2	Initiate TLS mode handshake using second key method
4	0x08	P_CONTROL_HARD_RESET_SERVER_V2	Reply to TLS mode handshake initiation using second key method
5	0x03	P_CONTROL_SOFT_RESET_V1	Force tunnel session key renegotiation
6	0x04	P_CONTROL_V1	Control channel packet (TLS data)
7	0x05	P_ACK_V1	Acknowledgment
8	0x06	P_DATA_V1	Data channel packet (encrypted)

Tab. 5: Combined list of message types for TLS mode (both key methods)

Value	Message type	Comments
1	P_CONTROL_HARD_RESET_CLIENT_V2	Key method 2 from client
2	P_CONTROL_HARD_RESET_CLIENT_V2	Key method 2 from server
3	P_CONTROL_SOFT_RESET_V1	Forcing tunnel session key renegotiation
4	P_CONTROL_V1	Control channel packet (TLS data)
5	P_ACK_V1	Acknowledgement
6	P_DATA_V1	Data channel packet (encrypted)

Tab. 6: Message types for default OpenVPN configuration (4.1)

4.3 Messages And Client-Server Communication in TLS mode

This section only concerns the TLS mode messages and authentication using the default OpenVPN configuration as described in Chapter 4.1. The pre-shared static key mode does not have the key exchange and authentication mechanisms, thus client-server communication of later mode is not discussed.

The OpenVPN TLS mode has eight different message types used to establish (Control channel) and manage (Data channel) the VPN tunnel. Tab. 5 shows all possible message types in the OpenVPN protocol using TLS mode. All of them are not used at the same time, and the message types depend on the selected key method (explained in Chapter 4.2.2). The default OpenVPN configuration (Ch. 4.1) only uses five message types (Tab. 6).

All of the OpenVPN message types can be seen in the sequence diagram of the OpenVPN TLS mode session establishment between a client and a server (Fig. 5). In the TLS mode, every packet is recognized based on its opcode. Opcode uses high 5 bits of the first byte of the OpenVPN layer header and is

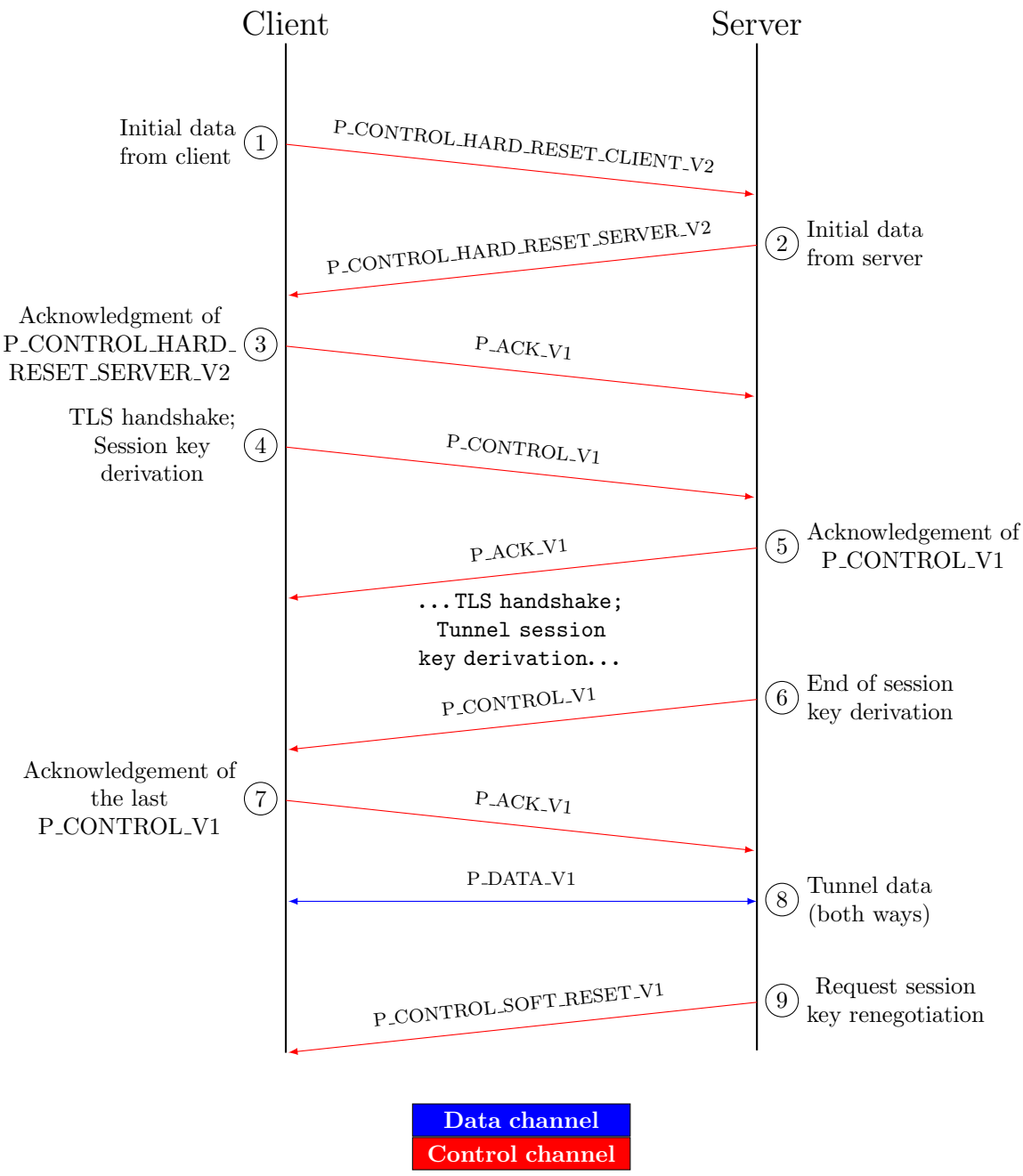

Fig. 5: Regular OpenVPN session (TLS mode, default configuration (4.1)

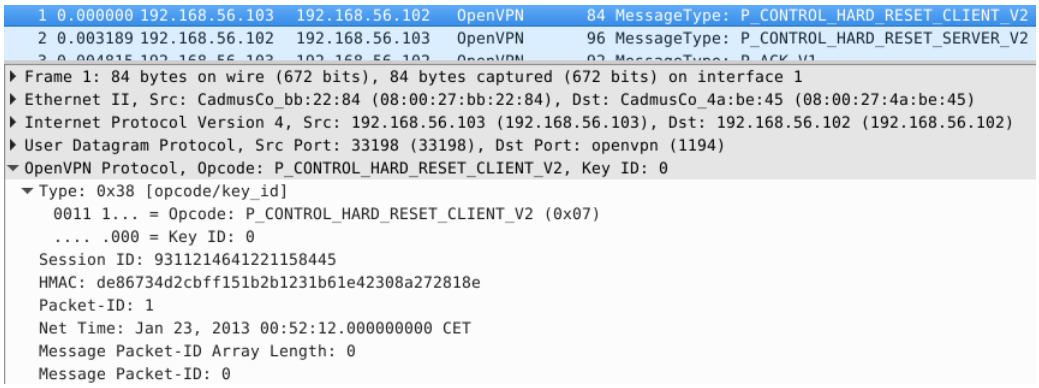

Fig. 6: Initial OpenVPN packet as seen in Wireshark

included in every single OpenVPN packet. The example of the initial OpenVPN packet (message type `P_CONTROL_HARD_RESET_CLIENT_V2`) can be seen in Fig. 6.

As previously mentioned, first OpenVPN Control channel packet is `P_CONTROL_HARD_RESET_CLIENT_V2`, which is inspected by checking its one-byte header, containing the packet's opcode and key ID. The opcode defines if the packet is processed as a Control channel or Data channel packet. In the client-server communication diagram (Fig. 5) Control, the channel messages are indicated in red and Data channel messages in blue. All the `P_DATA_V1` and `P_CONTROL_V1` packets have TLS protocol data inside, however, their OpenVPN layer structure differs. This will be elaborated in the following chapters.

As mentioned previously, the OpenVPN uses two separate channels - Control channel for authentication and establishing a tunnel, as well as Data channel, which contains the OpenVPN tunneled data when the tunnel has already been established. The following chapters will provide details about Control and Data channels.

4.3.1 Control Channel

The Control channel initiates the secure data tunnel establishment procedure, performs TLS handshake, including session starting, key exchange and cipher suite agreement. After the Data channel is established, the Control channel is also responsible for the key renegotiation procedure. The Control channel is depicted in the OpenVPN client-server model in Fig. 5. There are nine distinguishable steps in any OpenVPN TLS mode session. Steps 1 to 7 and 9 represent the Control channel, while step 8 represents the Data channel. The following chapter, Chapter 4.3.2, will particularly go into detail on the Data channel. As seen in Fig. 5, the Control channel can be separated into following parts:

- Step 1, 2 and 3: session initialization packets sent between client and server along with their corresponding acknowledgment packets;

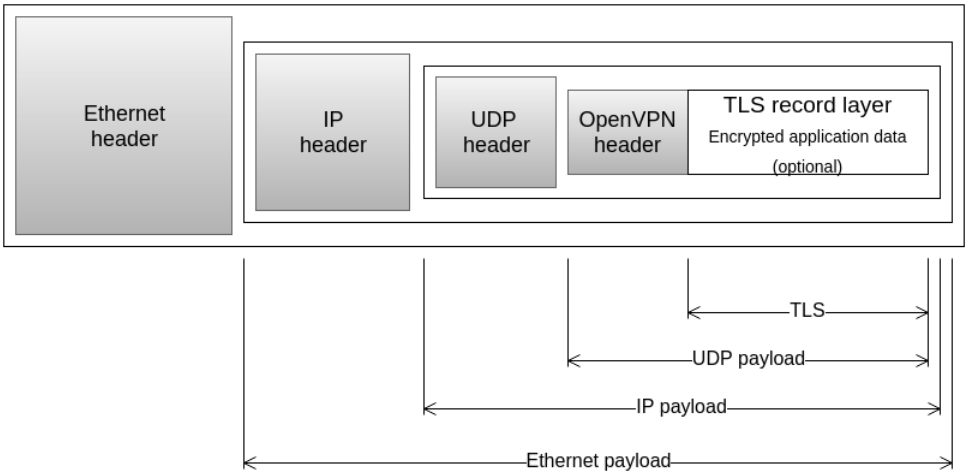

Fig. 7: Network layer encapsulation within the P_CONTROL_V1 packet

- Step 4, 5, 6 and 7: TLS handshake encapsulated inside the OpenVPN protocol layer. Following the successful TLS handshake, tunnel session key derivation is performed.
- Step 9: request for tunnel session key renegotiation.

The Control packet's plaintext header contains opcode and session ID, which determine if the packet is destined for an active TLS session or whether a new TLS session should be started. Each side of communication has its own 64-bit session ID, which is randomly generated and used to identify the OpenVPN session. Data channel tunnel session key renegotiation can be requested based on the message type `P_CONTROL_SOFT_RESET_V1`. Renegotiation of the tunnel session key can be performed periodically, based on three parameters:

- Renegotiate tunnel session key after N seconds (`reneg-sec N`);
- Renegotiate tunnel session key after N bytes (`reneg-bytes N`);
- Renegotiate tunnel session key after N packets (`reneg-pkts N`).

Control channel packets are `P_CONTROL_HARD_RESET_CLIENT_V2`, `P_CONTROL_HARD_RESET_SERVER_V2`, `P_CONTROL_V1`and `P_ACK`. `P_CONTROL` reset messages are used to initiate the OpenVPN protocol and include initial protocol data. The structure of `P_CONTROL` reset message type packets is shown in Tab 7. `P_CONTROL_V1` type indicates control packet which has TLS protocol data inside. Once the TLS session has been initialized and authenticated, the TLS channel is being used to derive and exchange secret keys. TLS control channel is also used to renew key material after a certain period, as defined by `--regen-sec`, `--regen-bytes` or `--regen-pkts` options. The structure of `P_CONTROL_V1` message packet type is the same as for initial reset packets (Tab. 7), except that

No.	Type	Size	Note
1	Opcode / key ID	1 byte	
2	Session_ID	64 bit	Randomly generated, used to identify TLS session
3	HMAC	16 or 20 bytes	Used only if `--tls-auth` (see note below) is specified
4	Packet ID	4 or 8 bytes	Includes sequence number and optional `time_t` timestamp
5	Network time	4 bytes	Local network time
6	P_ACK packet_ID array length	1 byte	
7	P_ACK packet ID array		Used if length > 0
8	P_ACK remote session_id		Used if length > 0
9	Message packet ID	4 bytes	

Tab. 7: P_CONTROL initial reset packet structure

it has the TLS protocol layer as its payload (Fig. 7). The networking layer encapsulation of `P_CONTROL_V1` can be seen in Fig. 7.

For every `P_CONTROL_V1` message `P_ACK_V1` message is being sent to acknowledge the received `P_CONTROL_V1` packet. Acknowledgment messages can be either be encoded in the dedicated `P_ACK` packet, or they can be prepended to a `P_CONTROL` message. Note that a packet structure may differ a lot depending on the different OpenVPN options used in the configuration. default configuration (as explained in Ch. 4.1).

After capturing the OpenVPN traffic, it can be noticed, that there are more `P_CONTROL_V1` messages than `P_ACK_V1` messages. This means that not all `P_CONTROL_V1` messages are being delivered and thus acknowledged. This prompts for retransmission of some `P_CONTROL_V1` messages. The structure of `P_ACK_V1` packet type is depicted in Tab. 8.

No.	Type	Size	Note
1	Opcode / key_id	1 byte	
2	Local session ID	64 bit	Randomly generated, used to identify TLS session
3	HMAC	16 or 20 bytes	used only if `--tls-auth` (see note below) is specified
4	packet-id	4 or 8 bytes	includes sequence number and optional `time_t` timestamp
5	Net time	4 bytes	
6	P_ACK packet_id array length	1 byte	
7	P_ACK packet-id array		used if length > 0
8	Remote session ID	64 bit	

Tab. 8: P_ACK_V1 packet structure

4.3.2 Data channel

The Data channel starts after the symmetric key is exchanged using already established TLS connection. It uses a connectionless UDP tunnel where there is no retransmission and ACK mechanism. The relation between Control and Data channels can be seen in Fig. 5. The packet structure of `P_DATA_V1`, as well as network layer encapsulation can be seen in Wireshark trace in Fig. 8. As can be seen, the packet header consists only of opcode and key ID. Just like in Control channel packets, the opcode is used to identify the packet type while the key ID is used to identify the associated local TLS state. State activity, authentication and the peer's source are checked for correctness, and in the case of success they are saved to be used for later authentication and decryption.

`P_DATA_V1` packets (VPN payload, used in an unreliable channel) represent encrypted, authenticated, and encapsulated OpenVPN tunnel packets. The full packet structure is shown in Tab. 9. The Data channel header is split into two separate parts - packet header and data payload header. It can be seen that both encrypted and authenticated parts of data extends into the data payload header.

Packet header holds the opcode, which identifies the packet type, and keying material. Data payload header consists of HMAC, Initialization Vector (IV) and packet ID. HMAC is computed by combining IV and ciphertext. To compute HMAC, OpenSSL EVP `evp_md` function is used together with the key. `Evp_md` can be `EVP_sha1()`, `EVP_ripemd160()` or other digest function. The OpenVPN uses BlowFish and SHA1 as default cipher and message digest algorithms.

Once each peer has its set of symmetric tunnel keys, the tunnel is started and data can flow between both OpenVPN connections. The plaintext of the encrypted envelope consists of a 64 bit sequence number and payload data (IP

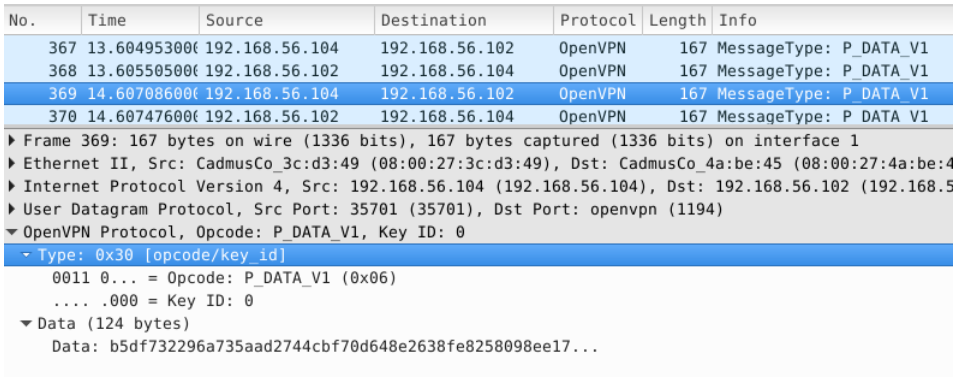

Fig. 8: Data channel `P_DATA_V1` packet as seen in Wireshark

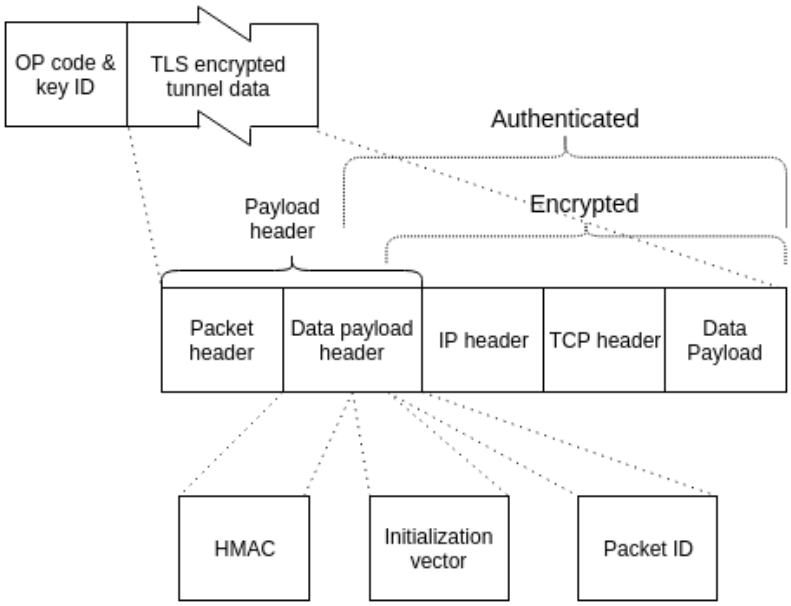

Fig. 9: The OpenVPN data packet (`P_DATA_V1`) structure

packet or Ethernet frame). Encryption, decryption, and HMAC functions are provided by the OpenSSL EVP interface. It allows to select arbitrary ciphers and digest algorithm. IV size is cipher-dependent, and it usually equals to cipher's block size, i.e. Data Encryption Standard (DES) is 64 bit block, and Advanced Encryption Standard (AES) is 128 bits block [3].

4.4 State Machine of the OpenVPN

In this chapter, the attained knowledge of the OpenVPN protocol was combined with the automata theory in order to manually create a presumed OpenVPN protocol state machine with a respectable degree of precision.

Before drawing the state machine, it is necessary to first understand and visualize the OpenVPN protocol. This could be achieved by manually drawing a state machine based on the all known specifications and observations of the protocol. Having a clear picture of the expected outcome might help in finding the strange and unusual activities in the protocol. The presumed protocol state machine can be seen in Fig. 10. Transitions $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and $7 \rightarrow 8 \rightarrow 4$ refer to the Control channel (4.3.1) which is used to establish the secure VPN channel between two parties. Control channel includes the TLS handshake which takes place on following transitions: $4 \rightarrow 5 \leftrightarrow 6$. Data transfer in the secure tunnel (4.3.2) operates in as follows: $6 \leftrightarrow 7$. The number of repetitive transitions depends on the packet size as well as the amount of data being transmitted. Alert/ConnectionRefused transition represents an OpenVPN error message which corresponds to the terminated connection.

Finally, once the real state machine is derived during fuzzing the OpenVPN server with LearnLib, a comparison could be made between presumed state machine and the true state machine. Any differences should be manually inspected for unusual behavior. The discrepancy between the presumed state machine and learned state machine may signify the presence of bugs, inaccuracies or vulnerabilities within the implementation.

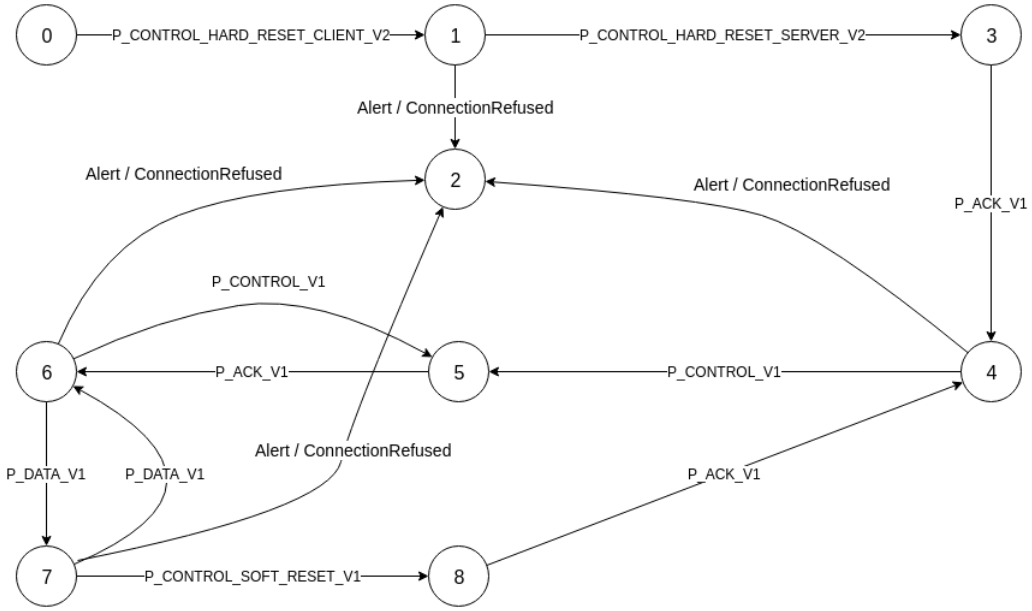

Fig. 10: Presumed OpenVPN state machine

4.5 Security Enhancing Options

The OpenVPN protocol implementation provides a great deal of options for configuring a VPN solution depending on particular requirements or needs. The more interesting options are specifically designed against malicious activities, and may have a substantial impact on the security. Below the three most relevant security enhancing options are presented.

Known as *HMAC firewall*, the OpenVPN has a `--tls-auth` option which uses a pre-shared static key (PSK) to add an HMAC signature to all TLS handshake packets for integrity verification. This option provides defense against:

- Buffer overflow vulnerabilities. This option forces all incoming packets to authenticate via HMAC before they are passed on to the TLS code in OpenSSL. This essentially acts as a Message Authentication Code (MAC) firewall which prevents the initiation of TLS handshake by an unauthorized user who does not have a symmetric key to authenticate the packets.
- DoS attacks or port flooding. Any packets without the correct HMAC will be immediately dropped so it is not possible to perform a DoS attack by initiating a TLS handshake.

The OpenVPN documentation recommends using *HMAC firewall* if the server is listening for packets from possible multiple unknown IP addresses. In the configuration file it is specified as `--tls-auth ta.key 0` and `--tls-auth ta.key 1` for server and client respectively. Here `ta.key` is a pre-shared static key, which

can be generated by using the OpenVPN itself (`openvpn --genkey --secrect ta.key`). It is important to note that `tls-auth` does not improve TLS authentication in any way, because it is used neither for authentication nor for encryption. It can, however, act as an additional defense in case the flaw or vulnerability in TLS cipher-suite is discovered. Since it adds additional complexity to the protocol, to perform initial testing, it was decided to omit this option. The generation and distribution of the pre-shared key adds an additional layer of complexity.

Second security option is designed to prevent privilege escalation in the event of session compromise and/or command execution on the system running OpenVPN. This is achieved by using `--user nobody` and `--group nobody` parameters. These parameters ensure that even if some kind of remote buffer overflow exploit were discovered, the exploit would with minimal privileges [29].

Last option is used to prevent replay attacks. This form of network attack repeats the previously sent valid data packets in order to impersonate a legitimate user [30]. As a replay protection, the OpenVPN uses a sliding-window mechanism (similar to IPsec [20]). Additionally, there is a defense mechanism against replay protection which is applied using `--replay-persist file` parameter. This option will log the current replay protection state (most recent timestamp and sequence number) and in the event of session termination or restart it will compare incoming timestamp and sequence number with logged one. If a replay is detected it will reject the packets [29].

4.6 OpenVPN Routing vs. Bridging

As noted in Chapter 3.1, understanding and selecting the correct networking components largely important. Therefore, in this next chapter, two different OpenVPN networking possibilities are presented.

The OpenVPN supports two networking modes for connecting networks, namely routing and bridging [31].

- Routing is essentially an interconnection of independent subnets. The router is a Level 3 (OSI model) device which forwards the packet according to a specified IP address.
- Bridging is a much simpler infrastructure because it operates only on the same local network (subnet). Bridge is a Layer 2 (OSI model) device which forwards packets based on the physical MAC address instead of IP address.

To perform testing, the bridged mode (tap virtual interface) was used, since the OpenVPN was running between two separate virtual machines on the same physical machine. The OpenVPN uses TUN (Network Tunneling Protocol) and TAP (Network TAP) virtual network kernel devices. These devices are entirely supported by software and provide packet reception and transmission of user space programs. The network interface can be configured as a Layer 2 device

(TAP) or as a Layer 3 device (TUN). For non-IP traffic or bridging, TUN devices cannot be used thus TAP is the only option. Also TAP must be used on the OpenVPN versions 2.2 or older when using Internet Protocol version 6 (IPv6) traffic.

Depending on which networking model is chosen, the test harness 5 would have to be constructed accordingly.

5 Test Harness

This chapter will describe the purpose and functionality of intermediate abstraction layer called test harness. The three different systems that are used in the setup of this research, are:

1. Learner - an implementation of LearnLib interface which uses an abstract input alphabet to learn the automaton of the protocol.
2. Teacher - System Under Test (SUT), an unmodified OpenVPN server, which yields an output based on Learner's input.
3. Test harness - an intermediate abstraction layer between the Learner and the Teacher. This is a custom made OpenVPN client which was being constructed during this research.

The purpose of the test harness is to allow the communication between two different parties - the Learner and the Teacher. Since LearnLib uses alphabets to provide input, it cannot be directly connected to the Teacher, in this case, an OpenVPN server. Its input is in an abstract language, and it expects the same abstract language as an output. To connect LearnLib to a real system, it is necessary to create a special abstraction mapper (Fig.11) that abstracts the alphabet of the Teacher into specific format, understandable to the Learner. This abstraction mapper is called the test harness and it translates input words into sequences of method calls which enables the Learner to communicate with the Teacher. The job of the test harness is to encompass both concrete and abstract sets of input, and output symbols, a set of states and a transition function which describes how the occurrence of a concrete symbol affects the state and an abstraction function which maps concrete symbols to abstract symbols [32].

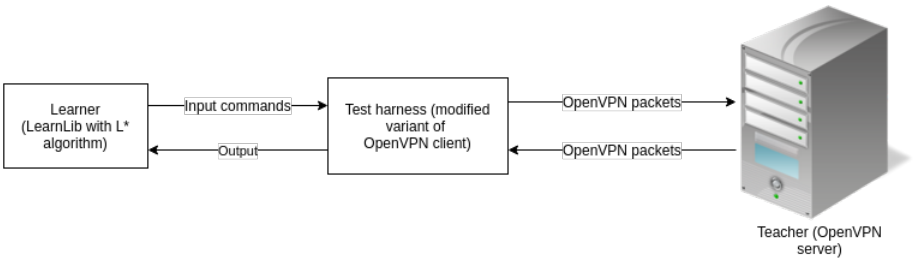

Fig. 11: Learner-Test harness-Teacher setup

The alphabet can be constructed in the following way [9]:

- Methods can be translated into abstract symbols of the learning alphabet.

Input alphabet	Output alphabet
P_CONTROL_HARD_RESET_CLIENT_V1	P_CONTROL_HARD_RESET_SERVER_V1
P_CONTROL_HARD_RESET_CLIENT_V2	P_CONTROL_HARD_RESET_SERVER_V2
P_ACK_V1	P_ACK_V1
	P_CONTROL_SOFT_RESET_V1

Tab. 9: Input and output alphabets used to establish OpenVPN tunnel

- API parameters can be handled by parameterizing the abstract learning symbols of parameterized interface methods.
- Return values can be abstracted according to the return type.

The input and output alphabets of establishing an OpenVPN connection are displayed in Tab. 9. The `P_CONTROL_SOFT_RESET_V1` message is not actually an output because it does not require any input from the Learner. It is just an OpenVPN server initiating the session key renewal. Nevertheless, it is interesting to test the key renewal process by forcing the server to initiate the new session key exchange.

The OpenVPN also uses the `P_CONTROL_V1` message type, which contains a fragmented TLS handshake data (explained in Chapter 4.3.1). The TLS packets are encapsulated inside this `P_CONTROL_V1` message type and are sent back and forth between client and server. However, since this is the TLS data, it is not considered as an element of input or output alphabet for the test harness. The fact that TLS handshake is wrapped inside the `P_CONTROL_V1` message type, made it considerably harder to build an OpenVPN client.

6 Experiments

In the previous chapters, theories were discussed that were necessary to examine before performing the experiments for this research. This chapter will explain the thorough process of the conducted experiments as well as giving a detailed description of the main ideas behind the steps that were taken when performing the experiments for this research.

In this research, the test harness was an OpenVPN client which could translate LearnLib messages into correct packets understandable to OpenVPN server. The test harness was written in Python using the Scapy library (6.2). The OpenVPN implementation is written in C, and Angluin's L^* algorithm in Learnlib is written in Java, however, this does not provide any difficulties because the client and the server can simply communicate using network sockets. Therefore, the Python client uses network sockets to interact with an OpenVPN server written in C. While OpenVPN is an open source, both the client and the server implementations are merged in the same thousand-line file which makes it exceptionally difficult to reuse original source code in a test harness.

Additionally, before writing even a simple VPN client, it is important to understand what message types does the server expects to receive. This is done mostly by analyzing the OpenVPN source code as well as inspecting the captured OpenVPN connection packets using Wireshark or other packet capture software. Knowing this, a needed message type can be send in order to receive a desirable response back from the server.

The next chapters give an introduction to the tools used, then follows the explanation of test harness construction process and finally there will be an overview of an issues faces during the construction process.

6.1 Tools

The experiments for this research were carried out using several different open source tools. In particular:

- OpenVPN - official open source OpenVPN implementation, containing many configuration options and capabilities in configuring VPNs.
- Python - high-level programming language, used for construction of a test harness.
- Scapy - Python based packet manipulation tool used to efficiently create, capture, and send network packets.
- Wireshark - open source network packet analyzer which is widely used to capture, dissect, and inspect network packets.
- LearnLib - open source Java library for fuzzing and active automata learning.

While creating the test harness, the only choice that needed to be made regarding tools was the programming language being used. Choices for other software, like OpenVPN and LearnLib, were made at the start of the research as they are the key components of this research.

6.2 Constructing a test harness

This chapter will talk about the approach chosen to construct an OpenVPN client (test harness), and well as explain the details of network programming with Scapy.

The test harness, which is essentially the simple OpenVPN client, is coded in the Python programming language. For crafting and inspecting packets, the packet manipulation tool called Scapy [7] was used. Scapy provides a Python interface into libpcap (packet capture interface) and allows for a fast creation of packets with working default values. It allows the user to forge easily and decode packets of a wide number of protocols, capture packets, sniff traffic, match requests/replies, fuzz and more. Scapy is very flexible and the user can easily construct packets in a variety of ways as well as send packets on different network layers. For instance, `send()` function will send packets at Layer 3 and handle routing as well as Layer 2. Meanwhile, `sendp()` will send packets at Layer 2 which means right interface and link layer protocol must be chosen.

Two different ways of assembling packets were considered:

1. Creating a UDP Socket with python and only creating OpenVPN layer with Scapy, then sending OpenVPN data inside the UDP socket.

2. Creating all the layers (IP, UDP) with Scapy without using sockets.

For this research the first method was chosen. It provides more flexibility, because a separate UDP socket is open, and can later be used to send TLS packets. Python provides libraries for crafting TLS packets which can be put on top of the created OpenVPN layer. Scapy assembles packets based on separate layers which can be stacked on top of each other. For instance, an example below showcases a UDP packet carrying some raw data.

```
scapyPacket = Ether()/IP(dst='192.168.138.129')/UDP()/Raw(load='some raw data')
```

In this example there are three different layers defined on top of each other: `(Ether)`, `(IP)`, `(UDP)`. Then raw data follows, which is being put inside the UDP layer. Each of the layers has multiple parameters which can be manually defined, just like `dst='192.168.138.129'` in the IP layer showcased above. The great thing about Scapy is that it is well designed to work with default parameters. That is, the user can omit them completely in the layer definition, and Scapy automatically uses pre-defined correct values.

At first, a simple Python client skeleton using socket programming to send a `P_CONTROL_HARD_RESET_CLIENT_V2` message and receive a response from the

[7] www.secdev.org

server was created. Scapy provides an easy way to define custom protocols and later simply use them as a separate layer when forming a packet. Defining an OpenVPN protocol (Layer):

```
class openvpn1(Packet):
     name = "P_CONTROL_HARD_RESET_CLIENT_V2"
     fields_desc = [XByteField("opcode", 0x38),
                    LongField("Session_ID", 0),
                    XByteField("Message_Packet_ID_Array_Length", 0),
                    IntField("Message_Packet_ID", 0)]
```

Selecting the correct values for each parameter is important as otherwise the protocol will not function. `Message_Packet_ID` parameter in the `P_ACK_V1` when replying to `P_CONTROL_V1` is meant against replay protection, and its value corresponds to the sequential number of the packet itself. If the value is 0 or not equal to the `P_CONTROL_V1` number value, then the server will not respond and will keep repeating the last message.

The code snipped below displays the creation of TLS Client Hello message using Scapy.

```
def createClientHello(:
print(‘‘Crafting TLS Client Hello\ldots’’),
tlsClientHello = openvpn3(opcode=0x20, Session_ID=initSesId,
                     Message_Packet_ID_Array_Length=0,
                     Message_Packet_ID=0001)/TLSRecord(/TLSHandshake
                     (/TLSClientHello(compression_methods=range(0xff)
                     [\dots-1], cipher_suites=range(0xff))
                     (/TLSClientHello(
                     print(‘‘done.’’)
                     return tlsClientHello))))))
```

Due to multiple difficulties during the programming of TLS handshake, the test harness was not finished. Scapy provides a layer of abstraction which is very convenient for fast construction of packets, however for OpenVPN client to fully operate, it needs to be created very precisely based on what the OpenVPN server expects. For that, a low-level networking programming methodology might be more useful. Nevertheless, the main blocker during the programming phase was the lack of low-level specification. A lot of time was spent analysing existing Wireshark packet traces and trying to figure out how the protocol operates.

6.3 Observations

When conducting the experiments, several issues and problems were faced. These issues made the research somewhat slower than expected. A lot of difficulties came from the fact that OpenVPN does not provide a low-level specification on its workings. The majority of issues are documented below.

1. To assemble initial OpenVPN messages, it is crucial to precisely understand the structure of each packet sent and received. Initially, it was tried to send `P_CONTROL_HARD_RESET_CLIENT_V2` message with previously captured OpenVPN payload, however the following errors were received:

```
TLS Error: incoming packet
    authentication failed.
```

   and

```
Authenticate/Decrypt packet error:
    packet HMAC authentication
    failed.
```

   These error signals that there is a problem with authentication and probably incorrectly generated HMAC. To avoid wasting time on solving incorrect authentication it was decided to temporary disable HMAC authentication (`--tls-auth`) and proceed without it.

2. Often network protocols are not able to handle big data chunks (i.e. network packet size limitation) and thus, those data chunks are being split over multiple packets. When observing captured packets with Wireshark, usually multiple `P_CONTROL_V1` messages are displayed in a row which is an example of data size being too big to fit in a single packet. After the data has arrived, Wireshark reassembles the packets according to the specified boundaries it detected.

3. It is important to note that different packet capture programs interpret captured network according to their own rules so it may appear differently. It is not wise to always trust the interpretation of the packet capture software. On the other hand, it is always good to examine and try to interpret the packet bytes themselves because they will always be correct. For instance, during initial testing the Wireshark results were compared on two different machines - Ubuntu 14.04.01 and Kali Linux (based on Debian 3.18.6-1). For that, the newest (1.12.7) versions of Wireshark were compiled and built on each machine separately, however the Kali machine could not distinguish OpenVPN traffic by default and showed 'malformed packet' error. After manually specifying to Wireshark that traffic is indeed OpenVPN, the error was no longer shown. This is due enabling Wireshark built in dissector, which usually automatically recognizes the OpenVPN protocol based on the UDP port 1194. The naming and boundaries of an Wireshark output may differ compared to an official OpenVPN website or other packet analysis software. For instance, Scapy was misinterpreting captured OpenVPN packets as some other protocol. The OpenVPN data in Scapy was sometimes interpreted as a DNS data.

4. For testing purposes while creating an OpenVPN client, it was very helpful to observe the client and server logs for any errors reported while

establishing communication. OpenVPN provides a verbose option (`verb n`) which can range from 0 (no logging) to 11 (most explicit). Highest possible option was used to extract as much information as possible and not miss any relevant details.

5. During the experiment it was noticed that, the current time in the *Random* field in *ClientHello* and *ServerHello* record layers is defined only by 4 bytes even though the full representation is shown as `Nov 20, 2015 15:16:53.000000000 CET`. The way TLS makes this long time format fit into 4 bytes is by instead using a number of seconds which passed since January 1st, 1970, 00:00 UTC to current local time. This was quite interesting considering that there would not be any other way to fit full date into 4 bytes.

7 General findings about OpenVPN & using Scapy in combination with Wireshark

During the OpenVPN protocol analysis, and the construction of a test harness, several interesting observations were made. This chapter contains a list of general findings regarding the OpenVPN protocol and the tools that were used for carrying out the experiments.

1. Some contradicting information was found in the security overview [3] of the official OpenVPN website, compared to the comments in the official open source implementation. In particular:

   (a) The official OpenVPN security overview [3] states that HMAC packet authentication (`--tls-auth` mode) is only used to keep the packet integrity and does not improve the peer authentication.

   > "This feature by itself does not improve the TLS auth in any way, although it offers a 2nd line of defense if a future flaw is discovered in a particular TLS cipher-suite or implementation"

   As the HMAC packet authentication prevents any unauthorized initiations from other parties, it does indeed act as an aditional authentication mechanism, considering there is a different pre-shared static key used for a HMAC authentication. Moreover, the comment in the OpenVPN source code states:

   > "Without –tls-auth, we leave authentication entirely up to TLS."

   The above quote implies, that `--tls-auth` mode improves authentication.

   (b) The official OpenVPN security overview [3] states the following:

   > "`P_CONTROL_HARD_RESET_CLIENT_V1` – Key method 1, initial key from client, forget previous state."

   The above suggests, that the `P_CONTROL_HARD_RESET_CLIENT_V1` message type includes the initial key from the client. However, this is not the case and the data sent in this initial packet is shown in Fig. 7. Furthermore, it would not be feasible to send a secret key in plaintext before the completion of the TLS handshake.

2. Another interesting finding is related to the OpenVPN replay protection mechanism which is enabled by `--replay-persist file` parameter. The quote from the official OpenVPN website [29] states: "It should only be used when replay protection is enabled with `--tls-auth` or `--secret` modes." Considering that OpenVPN only has these two authentication modes, it begs the question in which instances replay protection mechanism should not be used.

3. Finally, even though Scapy and Wireshark use different sockets to capture traffic, Scapy interferes with Wireshark packet capture capabilities. If Scapy is not running, then Wireshark correctly captures packets, however if the `sniff` function is used in Scapy then sometimes Wireshark does not capture correctly. To solve this issue, a dummy virtual interface was created using *iproute2* toolkit and the data was sent on this interface while capturing with Wireshark on default interface.

8 Future Work

Since the planned fuzzing component of this research was not performed, the correct continuation would be to finish the testing and evaluation of the security of an OpenVPN protocol. To continue with the project, it is important to evaluate issues and shortcomings faced in this research. Defining a clear scope and a precise plan would more likely yield positive results. As it is often the case with programming projects, certain issues may consume an extensive amount of time and the outcome will not be as expected. Therefore, it is important to evaluate personal skills and competence in the area of performed research. Performing a sophisticated project may be very tempting and undoubtedly interesting, yet it requires a certain set of skills to successfully finish it in an outlined time frame. It is advised to be quite comfortable with low-level network programming before starting similar research.

The packet manipulation Python library Scapy was very useful for quickly creating the desired packets, however that was irrelevant, considering, that the main problem was the lack of low-level knowledge of the OpenVPN protocol. It is recommended to fully grasp the protocol inner workings and then the tools like Scapy will be very efficient.

Fuzzing is certainly an interesting type of testing as it simulates real-world situations very well. Therefore, maintaining the planned approach to perform a LearnLib framework based fuzzing remains the preferred and suggested method. Additionally, and given enough time, it might be valuable to perform a more sophisticated fuzzing. Specifically, implementing the fuzzing environment which would not only fuzz the message types but the message parameters as well would likely lead to a better tested system. In general, fuzzing based testing has a broad spectrum of options and techniques, which mostly depend on the capabilities of the used hardware and the designated time frame.

9 Conclusions

In this research several VPN protocols and fuzzers were presented. An in-depth analysis of the widely used open source VPN protocol - OpenVPN - was studied and performed. Considering the popularity of OpenVPN protocol, it is surprising that it does not have a low-level specification document for its operation. It took a lot of time to fully understand the inner workings of the protocol and incorporate that knowledge while constructing the test harness. Moreover, the lack of detailed specification led to many unexpected stops during the programming process of the test harness, which consumed a large amount of time and subsequently the planned fuzzing component of the research was not performed.

This research paper provides a low-level specification of the OpenVPN protocol, based on the data collected from multiple sources, including official security overview, VPN books, support pages, captured network traces, and analysis of the source code. Additionally, an OpenVPN implementation was evaluated based on the known data and observations, which led to drawing of a presumed OpenVPN protocol state machine (Fig. 10 on page 33), which could be used in the future research. Finally, the complications faced during the research, inconsistencies in the OpenVPN documentation, as well as other interesting findings are all documented in this research paper.

References

[1] G. Bora, S. Bora, S. Singh, and S. M. Arsalan. OSI refernce model: An overview. *IJCTT*, 7(4):214–218, 2014.

[2] D. E. Comer. Internetworking with TCP/IP (2nd ed.), vol. i. *IJCTT*, pages ISBN 0–13–468505–9, 1991.

[3] Stijn Huyghe. Security overview. *Retrieved April 16, 2016, from https://openvpn.net/papers/openvpn-101.pd://openvpn.net/index.php/open-source/documentation/security-overview.html*, 2004.

[4] Transport Interfaces Programming Guide. Retrieved april 16, 2016, from http://libguides.gwumc.edu/c.php?g=27779&p=170342. Oracle. 2010.

[5] H. Raffelt, B. Steffen, , and T. Berg. Learnlib: a library for automata learning and experimentation. *9th International Conference, FASE*, 3922:377–380, 2005.

[6] J. de Ruiter. Lessons learned in the analysis of the EMV and TLS security protocols. *Ph.D dissertation, Radboud University Nijmegen*, 2015.

[7] F. Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *22nd IFIP WG 6.1 International Conference, ICTSS*, 6435:188–204, 2010.

[8] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. *4th International Symposium on Leveraging Applications, ISoLA*, 6415:673–686, 2010.

[9] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria. Automated learning setups in automata learning. *ISoLA*, 7609:591–607, 2012.

[10] G. Shu, Y. Hsu, and D. Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. *28th IFIP WG 6.1 International Conference*, 5048:299–304, 2008.

[11] S. Frankel, K. Kent, R. Lewkowski, A. D. Orebaugh, R. W. Ritchey, and S. R. Sharma. Guide to IPsec VPNs. 2005.

[12] S. Kent. IP authentication header. RFC 4302. *IETF*, 2005.

[13] S. Kent. IP encapsulating security payload (ESP). RFC 4303. *IETF*, 2005.

[14] C Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet key exchange protocol version 2 (IKEv2). *IETF*, 2010.

[15] S. Frankel, P. Hoffman, A. D. Orebaugh, and R. Park. Guide to SSL VPNs. 2008.

[16] W. Simpson. The point-to-point protocol (PPP). *IETF*, 1994.

[17] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-point tunneling protocol (PPTP). *IETF*, 1999.

[18] Andreas Steffen. The new ikev2 vpn solution. *Institute for Internet Technologies and Applications*, 2007.

[19] Stijn Huyghe. Openvpn 101: introduction to openvpn. *Retrieved April 16, 2016, from https://openvpn.net/papers/openvpn-101.pdf*, 2004.

[20] S. Kent and K. Seo. Security architecture for the internet protocol. RFC 4301. *IETF*, 2008.

[21] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal 34*, pages 1045–1079, 1955.

[22] A. Hagerer, H. Hungar, T. Margaria, O. Niese, B. Steffen, and H-D. Ide. Demonstration of an operational procedure for the model-based testing of CTI systems. *5th International Conference, FASE*, 2306:336–339, 2002.

[23] T. Bohlin, B. Jonsson, and S. Soleimanifard. Inferring compact models of communication protocol entities. *4th International Symposium on Leveraging Applications, ISoLA*, 6415:658–672, 2010.

[24] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[25] F. Aarts. Inference and abstraction of communication protocols. *Master thesis*, 2009.

[26] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. *ISC*, 4176:343–358, 2006.

[27] R. Sears P. Amini, A. Portnoy. Sulley: Fuzzing framework. *Retrieved April 16, 2016, from http://www.fuzzing.org/wp-content/SulleyManual.pdf.*

[28] Official Journal L013. Directive 1999/93/EC of the European Parliament and of the Council of 13 December 1999 on a Community framework for electronic signatures. *Retrieved April 16, 2016, from http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31999L0093:EN:HTML*, 1999.

[29] Manuals. *Retrieved April 16, 2016, from https://openvpn.net/index.php/open-source/documentation/manuals/69-openvpn-23.html*, 2008.

[30] J. Ning, I. Broustis, K. Pelechrinis, S. V. Krishnamurthy, and M. Faloutsos. Coping with packet replay attacks in wireless networks. *2011 8th Annual IEEE Communications Society Conference on*, pages 368–376, 2011.

[31] Community Wiki and Tracker. Bridgingandrouting. *Retrieved April 16, 2016, from https://community.openvpn.net/openvpn/wiki/BridgingAndRouting.*

[32] D. Giannakopoulou and D. MERY. Fm 2012: Formal methods. *18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, page 15, 2012.

11 Appendix

The Wireshark packet trace below displays complete establishment of an OpenVPN protocol tunnel using TLS mode (`P_CONTROL` and `P_ACK`) along with the tunnel data `P_DATA_V1`.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.103	192.168.56.102	OpenVPN	84	MessageType: P_CONTROL_HARD_RESET_CLIENT_V2
2	0.003189000	192.168.56.102	192.168.56.103	OpenVPN	96	MessageType: P_CONTROL_HARD_RESET_SERVER_V2
3	0.004815000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
4	0.005863000	192.168.56.103	192.168.56.102	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 1)
5	0.005873000	192.168.56.103	192.168.56.102	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 2)
6	0.005880000	192.168.56.103	192.168.56.102	TLSv1	110	Client Hello
7	0.006143000	192.168.56.102	192.168.56.103	OpenVPN	92	MessageType: P_ACK_V1
8	0.007420000	192.168.56.102	192.168.56.103	OpenVPN	92	MessageType: P_ACK_V1
9	0.048018000	192.168.56.102	192.168.56.103	OpenVPN	196	MessageType: P_CONTROL_V1 (Message fragment 1)
10	0.048038000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 2)
11	0.048045000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 3)
12	0.048050000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 4)
13	0.048525000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
14	0.048609000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
15	0.048693000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
16	0.048777000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
17	0.049764000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 5)
18	0.049774000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 6)
19	0.049780000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 7)
20	0.049923000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 8)
21	0.050052000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
22	0.050136000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
23	0.050221000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
24	0.050305000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
25	0.050799000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 9)
26	0.050872000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 10)
27	0.050976000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 11)
28	0.051122000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
29	0.051126000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 12)
30	0.051346000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
31	0.051354000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
32	0.051424000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 13)
33	0.051437000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
34	0.051577000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
35	0.051925000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 14)
36	0.051933000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 15)
37	0.051940000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 16)
38	0.051946000	192.168.56.102	192.168.56.103	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 17)
39	0.052180000	192.168.56.103	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1

No.	Time	Source	Destination	Protocol	Length	Info
340	7.644286000	192.168.56.102	192.168.56.104	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 48)
341	7.644302000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
342	7.644437000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
343	7.644521000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
344	7.644889000	192.168.56.102	192.168.56.104	TLSv1	118	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
345	7.645498000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
346	7.645846000	192.168.56.104	192.168.56.102	OpenVPN	196	MessageType: P_CONTROL_V1 (Message fragment 34)
347	7.645967000	192.168.56.104	192.168.56.102	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 35)
348	7.646136000	192.168.56.104	192.168.56.102	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 36)
349	7.646144000	192.168.56.104	192.168.56.102	TLSv1	114	Application Data, Application Data
350	7.646161000	192.168.56.102	192.168.56.104	OpenVPN	92	MessageType: P_ACK_V1
351	7.646305000	192.168.56.102	192.168.56.104	OpenVPN	92	MessageType: P_ACK_V1
352	7.646380000	192.168.56.102	192.168.56.104	OpenVPN	92	MessageType: P_ACK_V1
353	7.647089000	192.168.56.102	192.168.56.104	OpenVPN	196	MessageType: P_CONTROL_V1 (Message fragment 50)
354	7.647170000	192.168.56.102	192.168.56.104	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 51)
355	7.647311000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
356	7.647344000	192.168.56.102	192.168.56.104	TLSv1	166	Application Data, Application Data
357	7.647516000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
358	7.648357000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
359	10.095233000	192.168.56.104	192.168.56.102	TLSv1	174	Application Data, Application Data
360	10.097239000	192.168.56.102	192.168.56.104	OpenVPN	92	MessageType: P_ACK_V1
361	10.097471000	192.168.56.102	192.168.56.104	OpenVPN	184	MessageType: P_CONTROL_V1 (Message fragment 53)
362	10.097727000	192.168.56.102	192.168.56.104	TLSv1	138	Application Data, Application Data
363	10.098535000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
364	10.147384000	192.168.56.104	192.168.56.102	OpenVPN	92	MessageType: P_ACK_V1
365	11.991671000	192.168.56.103	192.168.56.102	OpenVPN	95	MessageType: P_DATA_V1
366	13.140540000	192.168.56.102	192.168.56.103	OpenVPN	95	MessageType: P_DATA_V1
367	13.604953000	192.168.56.104	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1
368	13.605505000	192.168.56.102	192.168.56.104	OpenVPN	167	MessageType: P_DATA_V1
369	14.607086000	192.168.56.104	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1
370	14.607476000	192.168.56.102	192.168.56.104	OpenVPN	167	MessageType: P_DATA_V1
371	15.608272000	192.168.56.104	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1
372	15.608723000	192.168.56.102	192.168.56.104	OpenVPN	167	MessageType: P_DATA_V1
373	16.611942000	192.168.56.104	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1
374	16.612544000	192.168.56.102	192.168.56.104	OpenVPN	167	MessageType: P_DATA_V1
375	19.059757000	192.168.56.104	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1
376	19.060157000	192.168.56.102	192.168.56.103	OpenVPN	167	MessageType: P_DATA_V1
377	19.060762000	192.168.56.103	192.168.56.102	OpenVPN	167	MessageType: P_DATA_V1

12 Source code

```
#!/usr/bin/env python
# Tomas Novickis, Radboud University Nijmegen
from scapy.all import *
from scapy.layers.ssl_tls import *
from M2Crypto.EVP import HMAC
import base64, random, uuid, M2Crypto, OpenSSL, binascii, ssl,
    time, pprint, os, sys, datetime
from time import strftime, gmtime
from socket import socket, AF_INET, SOCK_DGRAM, SOCK_STREAM
from dtls import do_patch

import sys, os
try:
   import scapy.all as scapy
except ImportError:
   import scapy
try:
   basedir = os.path.abspath(os.path.join(os.path.dirname(
        __file__),"../"))
   sys.path.append(basedir)
   from scapy_ssl_tls.ssl_tls import *
except ImportError:
       from scapy.layers.ssl_tls import *

import socket

class openvpn1(Packet):
       name = "P_CONTROL_HARD_RESET_CLIENT_V2"
       fields_desc = [XByteField("opcode", 0x38), LongField("
            Session_ID", 0), XByteField("
            Message_Packet_ID_Array_Length", 0), IntField("
            Message_Packet_ID", 0)]
class openvpn2(Packet):
       name = "P_ACK_V1"
       fields_desc = [XByteField("opcode", 0x28), LongField("
            Session_ID", 0), XByteField("
            Message_Packet_ID_Array_Length", 0), IntField("
            Message_Packet_ID", 0), LongField("Remote_Session_ID
            ", 0)]
class openvpn3(Packet):
       name = "P_CONTROL_V1"
       fields_desc = [XByteField("opcode", 0x20), LongField("
            Session_ID", 0), XByteField("
            Message_Packet_ID_Array_Length", 0), IntField("
            Message_Packet_ID", 0)]

if __name__=="__main__":
```

```
mysocket=socket.socket(socket.AF_INET, socket.SOCK_DGRAM
    )
#do_patch()
#tlssocket=ssl.wrap_socket(socket(AF_INET, SOCK_STREAM))
mysocket.connect(('192.168.138.129', 1194))
#tlssocket.connect(('192.168.138.129', 1194))
print("connect done")
mystream=StreamSocket(mysocket)

#ascapypacket=Ether()/IP(dst="192.168.138.129")/UDP(
    dport=1194, len=50)/Raw(load="\x38\x81\x38\x14\x62\
    x1d\x67\x46\x2d\xde\x86\x73\x4d\x2c\xbf\xf1\x51\xb2\
    xb1\x23\x1b\x61\xe4\x23\x08\xa2\x72\x81\x8e\x00\x00\
    x00\x01\x50\xff\x26\x2c\x00\x00\x00\x00\x00")
#ascapypacket=Ether(dst='00:0c:29:37:c5:6c')/Raw(load="\
    x38\x81\x38\x14\x62\x1d\x67\x46\x2d\xde\x86\x73\x4d\
    x2c\xbf\xf1\x51\xb2\xb1\x23\x1b\x61\xe4\x23\x08\xa2\
    x72\x81\x8e\x00\x00\x00\x01\x50\xff\x26\x2c\x00\x00\
    x00\x00\x00")
#initPayl = "\x38\xc2\xba\x6b\x7d\xee\x12\xf9\xc8\x00\
    x00\x00\x00\x00\x00\x00\x00\x00"
ses = 14031645777635506632
print("ses type:")
print type(ses)
initp = openvpn1(opcode=0x38, Session_ID=ses,
    Message_Packet_ID_Array_Length=0, Message_Packet_ID
    =0000)
print("packet created")
initp.show()
#initp = Raw(load=hi)
mystream.send(initp)
print("init sent")

captured = []
content = ''
ses_id = ''
first = 1
def sho(packet):
       print("sho started")
       print("s id")
       print ses_id
       global first, ses_id
       if first == 1:
              global content
       captured.append(packet)
       #print("len: %d" % len(packet))
       #print captured
       if first == 1:
```

```
                content = str(packet).encode("HEX") #
                    safe packet as string and display in
                    hex
                print("content:")
                print(content)
                ses_id = content[-50:-34]
                print("ses_id")
                print(ses_id)
                first = 0
print("Sniffing...")
sniff(iface="eth0:0", prn=sho, filter="host
    192.168.138.129 and udp port 1194", timeout=4, store
    =0)
print("Sniffing finished")
print("ses id after sniffing")
print ses_id

print("ses id type")
print type(ses_id)
print ses_id
print("remote session id:")
remote_session_id = long(ses_id,16)
print type(remote_session_id)
print remote_session_id

print("ses type:")
print type(ses)
print ses

ackp = openvpn2(opcode=0x28, Session_ID=ses,
    Message_Packet_ID_Array_Length=0x01,
    Message_Packet_ID=0000, Remote_Session_ID=
    remote_session_id)

time.sleep(1)
print("sending ack")
mystream.send(ackp)
print("ack sent")
#-------------------------------------------------------------------------------


myTime = strftime("%b %d %H:%M:%S %Y")
myTime = myTime + " GMT"
print ("myTime:")
print myTime
timestamp = ssl.cert_time_to_seconds(myTime) # get
    current time in seconds in float
finalTime = hex(int(timestamp))
finalTime = finalTime[2::] # remove 0x
```

```
currentTime = binascii.unhexlify(finalTime) # prepare to
     be sent on wire in bytes

# create TLS Handhsake / Client Hello packet
print("creating TLS handshake")
#p = openvpn3(opcode=0x20, Session_ID=ses,
    Message_Packet_ID_Array_Length=0, Message_Packet_ID
    =0000, Remote_Session_ID=remote_session_id)/
    TLSRecord()/TLSHandshake()/TLSClientHello(
    compression_methods=range(0xff)[::-1], cipher_suites
    =range(0xff))
p = openvpn3(opcode=0x20, Session_ID=ses,
    Message_Packet_ID_Array_Length=0, Message_Packet_ID
    =0000)/TLSRecord()/TLSHandshake()/TLSClientHello(
    compression_methods=range(0xff)[::-1], cipher_suites
    =range(0xff))
#p = Raw(load=helloPayl)/TLSRecord()/TLSHandshake()/
    TLSClientHello()
print("record created")

pp = openvpn3(opcode=0x20, Session_ID=ses,
    Message_Packet_ID_Array_Length=0, Message_Packet_ID
    =0001)/Raw(load="\x16\x03\x01\x00\xc3\x01\x00\x00\
    xbf\x03\x01"+currentTime+"\xc7\x88\xb2\x29\xcb\x59\
    x4b\xc6\x98\xef\x29\x3f\xdf\x45\x1c\x24\x3c\x34\x3b\
    x01\x95\x26\xc4\x7b\x61\x9f\xe3\x14\x00\x00\x50\xc0\
    x14\xc0\x0a\x00\x39\x00\x38\x00\x88\x00\x87\xc0\x0f\
    xc0\x05\x00\x35\x00\x84\xc0\x12\xc0\x08\x00\x16\x00\
    x13\xc0\x0d\xc0\x03\x00\x0a\xc0\x13\xc0\x09\x00\x33\
    x00\x32\x00\x9a\x00\x99\x00\x45\x00\x44\xc0\x0e\xc0\
    x04\x00\x2f\x00\x96\x00\x41\xc0\x11\xc0\x07\xc0\x0c\
    xc0\x02\x00\x05\x00\x04\x00\x15\x00\x12\x00\x09\x00\
    xff\x02\x01\x00\x00\x45\x00\x0b\x00\x04\x03\x00\x01\
    x02\x00\x0a\x00\x34\x00\x32\x00\x0e\x00\x0d\x00\x19\
    x00\x0b\x00\x0c\x00\x18\x00\x09\x00\x0a\x00\x16\x00\
    x17\x00\x08\x00\x06\x00\x07\x00\x14\x00\x15\x00\x04\
    x00\x05\x00\x12\x00\x13\x00\x01\x00\x02\x00\x03\x00\
    x0f\x00\x10\x00\x11\x00\x0f\x00\x01\x01")

print "sending TLS payload"
mystream.send(pp)
print ("TLS payload sent")
time.sleep(10)
#s.sendall(str(p))
#resp = s.recv(1024*8)
#print "received, %s"%repr(resp)
#SSL(resp).show()

#s.close()
```