Master thesis
Computer Science

Radboud University

Fuzzing OpenVPN

Author:
Sven van Valburg
s4042123

Supervisor
Erik Poll
`erikpoll@cs.ru.nl`

Second Assessor
Joeri de Ruiter
`joeri@cs.ru.nl`

May 31, 2018

Contents

Abstract

A commonly used technology to securely transmit data over a network is a Virtual Private Network (VPN). Using a VPN a user can create a secure encrypted tunnel which can be used to safely transmit sensitive data. Naturally, flaws in the implementation of a VPN's security protocol can potentially expose this data to the outside world. This master thesis will look into one of the more popular VPN solutions, OpenVPN, with two goals in mind. One, paint a clear picture of the inner workings and message structure of the OpenVPN protocol. And two, create a fuzzer to attempt to find flaws in the OpenVPN protocol.

Chapter 1

Introduction

Virtual Private Networks (VPN) are a technology to facilitate secure communication over an insecure network (such as the internet). VPN solutions can be categorized under several types, each with their own approach to security, upsides and downsides and reliance on different combinations of protocols and standards. The three major types are IPSec [10], PPTP [17] and TLS [9] based VPN solutions. Given the complexity of VPN solutions and the fact that various different implementations of these types exist, it is not unthinkable that some of these implementations have undiscovered security vulnerabilities. This paper will focus on testing one of these implementations, the TLS based VPN called OpenVPN.

The testing method used will be fuzzing. This means that we will be sending garbled protocol messages to an OpenVPN server to see how it responds. By monitoring the server's behaviour and the network traffic between client and server and placing this information beside the fuzzed messages that caused this behaviour it is possible to find flaws in the OpenVPN implementation. These flaws can then potentially lead to concrete security vulnerabilities which can be exploited by an attacker sending a packet with the same mutations as the fuzzed messages.

To set up such a fuzzer, clear knowledge of the protocol in question is needed. As one will need to be able to construct valid protocol messages before one can mutate their contents. Clear knowledge of the protocol also helps determine what fields to corrupt and how to corrupt them. Therefore the goal of this paper is twofold. First off, the OpenVPN protocol must be clearly mapped out. This includes how the protocol will behave under normal operation, the different security options that may be present and the structure of the different message packets themselves so they can be manually constructed. Secondly, a fuzzer must be build which can automatically carry out the actual fuzzing attempts.

The organization of this paper will be as follows. Chapter 2 will give an overview of past vulnerabilities already found at the time of writing this paper. Chapter 3 will describe the background topics like Virtual Private Networks and fuzzing in more detail. Chapter 4 will delve into the OpenVPN protocol and lay out its security options, a high level representation of a regular session and the different message types and their contents. Chapter 5 will explain the implementation process and decisions made while building the fuzzing tool, as well as detailing the results and any interesting findings along the way. Chapter 6 will present the results of the experiments performed using the fuzzing tool. And finally, Chapter 7 and Chapter 8 will have suggestions on future work on this topic and summarize the results and conclusions.

1.1 The scope of the thesis' research

When making the decision to fuzz a program or protocol it is important to decide what exactly to fuzz. OpenVPN makes use of external TLS libraries and sets up a standard TLS connection as part of its protocol. The decision was made to explicitly not fuzz this connection, as we want to solely focus on OpenVPN's protocol and its implementation and not the specific TLS implementation used by the current version of OpenVPN. Furthermore, if we want to fuzz the TLS library it would be a lot easier to do this separate from OpenVPN. Leaving TLS out of the picture also means we can use an existingTLS implementation for the fuzzer and have that implementation generate the needed messages for us, rather than having to self-generate these messages.

This approach has its up and downsides. It narrows the scope to purely the OpenVPN protocol and thus excludes the fuzzing off and any bugs found in the external TLS libraries used. But in doing so it might also overlook any bugs or vulnerabilities in the OpenVPN specific code that uses said libraries.

Chapter 2

Security analysis of OpenVPN

This chapter will give an overview of already performed work relating to OpenVPN's security and its individual components. The conclusions from this work can then be used to help guide my own research by finding interesting points to attack or identifying parts of OpenVPN that will not be included in the scope of the paper. Vulnerabilities already found in earlier versions of OpenVPN can also be used to test my own methods, as we can use these to test if the created fuzzer can find these vulnerabilities as well.

2.1 The types of vulnerabilities

When looking at OpenVPN and the potential security problems it may have, one can divide these problems into four categories.

1. TLS vulnerabilities: as OpenVPN uses an existing TLS implementation, any vulnerability in this implementation can also end up being a vulnerability in OpenVPN.

2. OpenVPN vulnerabilities: any internal OpenVPN vulnerabilities, purely related to the OpenVPN protocol. Examples include a faulty state transition or crashes caused by modifying the OpenVPN headers of messages in the protocol.

3. Interaction vulnerabilities: OpenVPN has several helper functions to hook into the TLS implementation. Any vulnerabilities caused by these functions fall under this category.

4. User errors: Try as we might, it's impossible to remove the human factor from security vulnerabilities. In this case a user might configure their OpenVPN implementation in such a way that a connection becomes easy to attack.

Depending on your fuzzing approach you might not be able to touch upon one or more of these categories. As already stated in Section 1.1, the fuzzer in this paper will leave the TLS handshake messages unaltered while fuzzing and can thus not find any TLS vulnerabilities, but might still be able to run into OpenVPN or Interaction vulnerabilities.

2.2 CVE analysis of OpenVPN

This section will briefly cover CVEs related to OpenVPN. However, of the 23 CVEs, most have to do with non-default settings opening up (sometimes platform specific) attack angles, timing attacks, cryptographic vulnerabilities or other angles and are therefore not relevant to this paper, as just fuzzing would not uncover these. Below is a list of the CVEs that can potentially be found through packet fuzzing. and their typing according to Section 2.1.

- CVE-2014-8104, OpenVPN 2.x before 2.0.11, 2.1.x, 2.2.x before 2.2.3, and 2.3.x before 2.3.6 allows remote authenticated users to cause a denial of service (server crash) via a small control channel packet. This is a type 2 vulnerability.
- CVE-2017-7478, OpenVPN version 2.3.12 and newer is vulnerable to unauthenticated Denial of Service of server via received large control packet. Note that this issue is fixed in 2.3.15 and 2.4.2. This is a type 2 vulnerability.
- CVE-2017-7521, OpenVPN versions before 2.4.3 and before 2.3.17 are vulnerable to remote denial-of-service due to memory exhaustion caused by memory leaks and double-free issue in extract_x509_extension(). This is a type 3 vulnerability.
- CVE-2017-7522, OpenVPN versions before 2.4.3 and before 2.3.17 are vulnerable to denial-of-service by authenticated remote attacker via sending a certificate with an embedded NULL character. While the function that reads the certificate sends back an error when the NULL character is found. OpenVPN did not halt execution and can later crash. This is a type 3 vulnerability.
- CVE-2017-7508, OpenVPN versions before 2.4.3 and before 2.3.17 are vulnerable to remote denial-of-service when receiving malformed IPv6 packet. This is a type 2 vulnerability.

The first two CVEs listed will be revisited lated as these could be replicated through packet fuzzing. The later two however fall outside the scope of this paper (Section 1.1) as these involve bugs in the OpenVPN code that calls upon the TLS libraries. These bugs can only be found by fuzzing the TLS traffic which, as stated in the previous section, will not be done.

It is worth noting that some of these CVEs were found in early 2017 when Guido Vranken also fuzzed OpenVPN using libFuzzer, AddressSanitizer, UndefinedBehaviorSanitizer and MemorySanitizer [4]. However due to OpenVPN's complex nature he has had to manually edit the OpenVPN code to allow for this type of fuzzing. In particular he had to block OpenVPN's ability to run external programs (as he didn't want to fuzz the system programs OpenVPN called upon, just OpenVPN itself), cut off direct resource access (as to not write random files or send data to random IP's) and remove certain ASSERT blocks so the code would not simply abort immediately on failing the assert. His efforts have uncovered several vulnerabilities:

- The earlier mentioned CVEs 7521, 7522 and 7508.
- A possible remote client crash, data leak and stack buffer corruption for users who are connecting to an Windows NT Lan Manager v2 proxy (CVE-2017-7520). This is a type 2 vulnerability.
- And a vulnerability that can result in a stack buffer overflow if the config file contains an excessively long -tls-cipher option (no CVE). This is a type 2 vulnerability.

2.3 TLS

OpenVPN relies heavily on external TLS libraries to function (namely OpenSSL or mbedTLS) (more on this in section 3 and 4). This makes it impossible to mention the security of OpenVPN without at least mentioning the security of TLS in general and any research and major past vulnerabilities that have popped up. I would however like to reiterate that this paper will not attempt to fuzz the TLS libraries used in OpenVPN as this would greatly inflate the scope. The widespread use of TLS outside of OpenVPN also means that research into and testing of TLS implementations is already constantly being done and I would like this paper to focus on OpenVPN itself, rather than the widely used libraries it depends on.

There have been many problems and bugs relating to TLS in the past [14], both general problems with the protocol as well as specific implementation problems. But to name a few very noteworthy ones in particular:

The Heartbleed [8] bug was a major and critical security vulnerability found in the OpenSSL library, a widely used implementation of the TLS protocol. The bug exploited the Heartbeat functionality of the TLS protocol, a test message any user could send over an established TLS connection to test the connection. A normal Heartbeat exchange consists of a user sending a string to the server and the server replying with the exact same message.

An attacker could however modify tamper with the length field of this test string, tricking the server into sending back much more data than just this string. This essentially forced the server to perform a memory dump, as it would pad the returned string with whatever it had stored in memory at the time. The returned data could vary from singular messages from other connections to the private keys needed to decrypt entire encrypted message sequences from other users.

POODLE was a downgrade attack in which a man in the middle could trick a client and server into establishing a connection using an older unsafe version of TLS [15]. The attacker can then use known vulnerabilities in this older version to attack this connection more easily. The attack relies on some TLS implementations still supporting unsafe versions of TLS and the fact that the client-server negotiation on what version to used happens entirely in plain text. An attacker can thus intercept this negotiation and falsely claim to either the client or the server that the only version they support is such an unsafe version of TLS.

TLS implementations have also been fuzzed before. J. de Ruiter analyzed nine different TLS implementations using protocol state fuzzing in order to find implementation faults using the inferred state machines [6]. To infer these state machines a Java implementation of the L* algorithm called LearnLib was used. The method proved successful in finding security vulnerabilities in several of the TLS implementations, in particular a version of GnuTLS, Java Secure Socket Layer and OpenSSL.

Chapter 3

Background

This chapter gives an overview of virtual private networks, the different types of VPN solutions and their common implementations (Section 3.1) as well as the basic idea behind fuzzing and some tools to aid with fuzzing (Section 3.2).

3.1 VPNs

A VPN is a virtual network, built on top of existing physical networks, that can provide a secure communications mechanism for data and IP information transmitted between networks [10]. In other words a VPN allows you to set up a secure, encrypted network connection over any insecure public network (like the internet). There are three major types of VPN solutions: IPSec, PPTP and Secure Socket Layer (or Transport Layer Security) based. Generally a VPN will try to fulfill three security goals, Authentication, Integrity and Confidentiality [19]. So only those allowed to connect to the VPN can do so, transmitted data can not be read by anyone other than the receiver, and the transmitted data can not be tampered with.

As the different VPN types use different protocols and these different protocols work on different network layers, it is important to understand how these layers are stacked up and how they relate. There are two main network models that are still valid today, the OSI model and the TCP/IP model [18], which will be covered in brief detail.

The OSI model has seven layers: Physical, Data link, Network, Transport, Session, Presentation and Application. Whereas the TCP/IP model has four: Link, Internet, Transport and Application. As a packet travels down through these layers every layer adds more information to the packet. When a packet is received it travels in reverse, now every layer unpacks its own information from the packet before sending it up. Thus higher level layers are completely unaware of the lower level layers and higher level layers

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

Table 3.1: The OSI and TCP/IP models

can not provide security for what happens in lower level layers. As already said, different VPN solutions operate on different layers of the network and thus only provide security for certain layers as well. Furthermore, since the models are just that, models, it is often hard to draw a one on one comparison with reality. It is hard to place a protocol exactly on one layer. This makes stacking VPNs a complicated if not impossible feat, making knowledge of what exactly you want to protect an important factor in choosing the right VPN solution.

3.1.1 IPSec VPNs

IPSec is probably one of the most used standards for VPN communication, it operates on the Network layer. IPSec itself is a framework of open standards for private communications over public networks [10] and includes protocols for mutual authentication and the negotiation of cryptographic keys for use during a session and operates on the Network layer. Any security action it takes will thus apply to IP packets and it will not be able to differentiate between data from different application. A user of an IPSec VPN will thus see all of their data encrypted and tunneled through the VPN. An advantage to operating on this layer is the complete encryption of the IP packet, both the contents as well as the IP information can be obfuscated to the outside viewer. The downside to this is the lack of flexibility and control when the end goal is to protect a specific application. [10]

A popular example of an IPSec VPN is strongSwan[1].

3.1.2 PPTP VPNs

The Point-to-Point Tunneling Protocol was a VPN solution in part developed by Microsoft and operates on the Data Link layer. It sets up a TCP control channel between the client and a server and uses Generic Routing

[1]`https://www.strongswan.org/`

Encapsulation to encapsulate PPP packets to send over this same control channel [11]. Due to the use of PPP it can be said to operate on the Data Link layer. However, not long after its conception several serious security flaws [17] were found in the protocol and it has since been considered obsolete.

3.1.3 TLS VPNs

As the name implies these VPNs are build around the Transport Layer Security protol (formerly known as SSL- Secure Sockets Layer) which operates on the Application layer. The usage of the widely used TLS protocols mean that these VPNs are much easier in use than for example IPSec VPNs. [13] Where for example IPSec VPNs can be inflexible and hard to maintain, an TLS VPN is much easier in its setup and use, requiring nothing more than some software to setup a server without any hardware or network requirements. In its simplest form a client will need nothing other than their web browser to connect to a server. But this simplicity is a disadvantage as well, as actual user authentication is optional and can be easily left out.

There are two main types of TLS VPNs: TLS portal VPNs and TLS tunnel VPNs [9].

- An TLS Portal VPN allows a user to connect to a gateway or portal to access multiple network resources using a single TLS connection. This comes in the form of a website which, after a user has authenticated themselves, leads the user to a page through which they can access other services. This has as downside that the content that can be shown on this website is limited, as certain types of active content (Java, JavaScript, Flash, or ActiveX) cannot be displayed on this website.

- An TLS Tunnel VPN usually means relatively more effort on the user's part, in that they need to install third party software to use it. But once set up will tunnel all internet traffic through the VPN server they are connected to. This means that, with regards to an TLS Portal VPN, the communications that are protected are no longer limited to what the portal can or will show to the user, anything they do is now securely tunneled through the VPN server.

A popular example of an TLS VPN is OpenVPN, which is primarily a Tunnel VPN.

3.2 Fuzzing

Fuzzing is a software testing technique in which a program is supplied with invalid, unexpected or random data as input. The program can then be monitored to see how it handles this input, whether or not it crashes, corrupts memory or fails in any other way. Or if it behaves properly and safely handles the non-conventional input. The eventual goal of fuzzing is to find programming oversights in the program, edge or corner cases which were overlooked or not considered during development. Programs are however complex by nature, there is not a singular input method or format shared by all programs. Monitoring programs can also be done from various angles and using various tools. So there is not a single tool or fixed collection of tools that can test any and all programs. This section will therefore only list some of the more popular tools and tools found during the course of this paper's research.

3.2.1 Sulley

Sulley is an open source fuzzing framework written in python [5]. As it is a full framework it isn't just a tool to generate and send random data but also includes features to monitor the process and help automize it. Sulley's goal is to simplify the fuzzing process for the end user as much as possible, so fuzzing as a technique can become more widely used. Some of the notable features of Sulley are:

- Packet capture - Sulley can capture network packets send and received during the fuzzing process
- VMWare automation - Sully can interface with a VM to help automate the process. It will keep multiple snapshots of past stable states and can, in case of a crash caused by the fuzzing, revert to a past safe state to continue the process.
- Process monitoring - Sulley can detect faults and crashes of connected processes and when discovered, log the fault and for later analysis.
- GUI - Sulley provides a web based graphical user interface to ease up the interaction with the fuzzer and allow for easier data representation to analyze the fuzzing results.

3.2.2 American Fuzzy Lop

American Fuzzy Lop or AFL[2] is a fuzzer using a combination of brute force techniques and genetic algorithms to maximize code coverage. AFL requires a program to be compiled using its own compiler so the fuzzer can track the target's control flow. It also requires one or more user provided sample commands and input files for the program. AFL will then start the automated fuzzing process by testing the supplied file and command and check if it

[2] `http://lcamtuf.coredump.cx/afl/`

succeeds, and then trimming this input to the smallest possible form that still triggers the exact same behavior. From here on out the fuzzing process starts by modifying the input file and testing for program crashes. If a crash is detected the event is logged and the modified input file that resulted in the crash is saved for later inspection.

3.2.3 LibFuzzer

LibFuzzer[3] is a more 'direct' fuzzer that attacks a software library and its functions directly. It takes a specific fuzzing entry point, which it calls the Target Function, and supplies it with input. Then, after testing which areas of the code can be reached with this input, it generations random mutations of this input and repeats the process. Like AFL it aims to maximize code coverage from the user supplied starting point (the Target Function). An advantage of LibFuzzer over other fuzzers is the ability to target specific functions in the code and using those as starting point, rather than attacking the program from the outside and trying to path a way through the entire program to try and reach these Target Functions.

3.2.4 Debugging tools

As already stated, the process of fuzzing feeds a program with non-conventional input with as goal to find a way to make the program behave different than what is specified or expected. A program does not however always crash or exit cleanly when such a behaviour is encountered and it might in fact just crash, hang or completely lock up. In cases like these it is helpful to use debugging tools to look into the internal state of the program during runtime to try an identify what went wrong.

Valgrind is a memory debugging tool that allows one to detect any memory oddities that might occur during program execution[4]. While not a fuzzing tool by itself, it can help find the cause of a crash caused by fuzzing. While most fuzzing tools will be able to tell you when a program has crashed and what input caused the crash, figuring out why is usually outside the scope of the tool. Using a memory debugger like Valgrind allows one to look at the internal state of the program and can help determine exactly why the program crashed.

AddressSanitizer, MemorySanitizer and UndefinedBehaviorSanitizer are other examples of very specific tools. As their names suggest they each specialize in finding specific kinds of bugs and will throw more detailed (and hopefully more useful) errors if something happens to trigger them. The Clang compiler front end [5] can be used to compile a program with these

[3] `https://llvm.org/docs/LibFuzzer.html`

[4] `http://valgrind.org/`

[5] `https://clang.llvm.org/`

sanitizers enabled.

Chapter 4

OpenVPN

This chapter will delve into OpenVPN. it gives an overview of the inner workings and its protocols, the different authentication modes and the messages used in these protocols. OpenVPN does not offer a clear documentation explaining its inner workings. Therefore all the information presented in this section is gathered from various other sources like the Security Overview webpage [3] (which is partially a copy paste of some comments in the code), Wireshark traces and the doxygen generated from the code. Lastly the chapter will look into alternatives to the official OpenVPN client, or the lack thereof.

4.1 Introduction to OpenVPN

OpenVPN is a TLS based VPN solution that can tunnel any IP subnetwork or virtual ethernet adapter over a single UDP or TCP port[1]. It mostly relies on the security provided by OpenSSL though later versions also support mbed TLS.

OpenVPN uses its own protocol which, when in TLS mode, incorporates a TLS handshake to authenticate and negotiate the set of session keys. In Static Key mode this handshake is skipped and the protocol assumes the session keys have already been distributed beforehand. More on these modes and the differences are explained in Section 4.1.
After this key establishment step the data channel (for the tunneled packets) is opened. This data channel is multiplexed with the control channel (for TLS authentication and key exchange) over the same port.

[1]`https://openvpn.net/index.php/open-source/333-what-is-openvpn.html`

4.2 Authentication modes

As mentioned before, OpenVPN gives two different options for authentication. These options are further elaborated on in this section.

4.2.1 Static Key mode

In this mode it is assumed that the session keys have already been distributed prior to starting the connection. Knowledge of these keys both serves as authentication mechanism and means of encrypting/decrypting the data from the tunnel. This mode of authentication is not relevant for the purposes of this paper as this skips the TLS handshake and subsequent key exchange. This mode also does not provide forward secrecy as a new key can not be renegotiated on the fly and will need to be redistributed outside of the session. However, a possible advantage of this mode is the added control for the users of the server. Since key generation and distribution is no longer handled by OpenVPN and allowing the user to take full control of these security building blocks.

4.2.2 TLS mode

The default and, for most people, safest mode to use OpenVPN with as every client that connects will dynamically generate a new key for their session as well as renegotiate keys after certain criteria have been met. As said before this mode uses the TLS protocol to secure a channel which is then used to negotiate a key and, optionally, to authenticate the user's credentials. This session key is the equivalent of the pre-distributed key in static key mode and is used to encrypt the data tunnel. However, in TLS mode this key can be renewed at any point during the session. The combination of this automatic key distribution and renewal provides this mode with perfect forward secrecy as well as the means for new clients to connect on the fly.

TLS mode also has several different options that provide more (or less) security. Namely the different key generation modes and the tls-auth option.

Key generation modes

TLS mode has two different key generation methods which slightly alter its message structure further down the line. Which key method will be used is indicated by the reset request message that starts of an OpenVPN session.

1. The first key generation method (V1) uses the RAND_bytes function of OpenSSL to generate the keys directly. These keys are then shared over the secure TLS channel after which the encrypted tunnel can start. This method is no longer recommended and deemed less safe than the second method.

2. The second key generation method (V2) has both the server and client gather random source material first to generate entropy, this random material is then shared over the secure TLS channel after which the TLS PRF function is used to generate the keys. After this step too the encrypted tunnel can start.

Starting with OpenVPN 2.0, the V2 method became the default and recommended key generation method as it provides a higher amount of entropy than the V1 method provides. V1 has a single machine generate the entire key, if this machine's random generation becomes predictable then any key generated becomes unsafe. V2 however provides entropy from two sources, as both client and server take part in the generation.

TLS-auth

The last notable feature of TLS mode is what OpenVPN calls the "HMAC firewall" [1]. This is an optional feature enabled with the "-tls-auth" option that requires the pre-distribution of separate HMAC keys to all users. These keys can then be used to add HMAC signatures to the control channel packets before any keys have been generated, adding an extra layer of security. OpenVPN recommends using this feature "when you are running OpenVPN in a mode where it is listening for packets from any IP address" [1]. The feature description states that any packet with a non matching HMAC will be discarded and ignored, this to ensure that no processing time is wasted on a TLS session that should not or will not complete successfully.

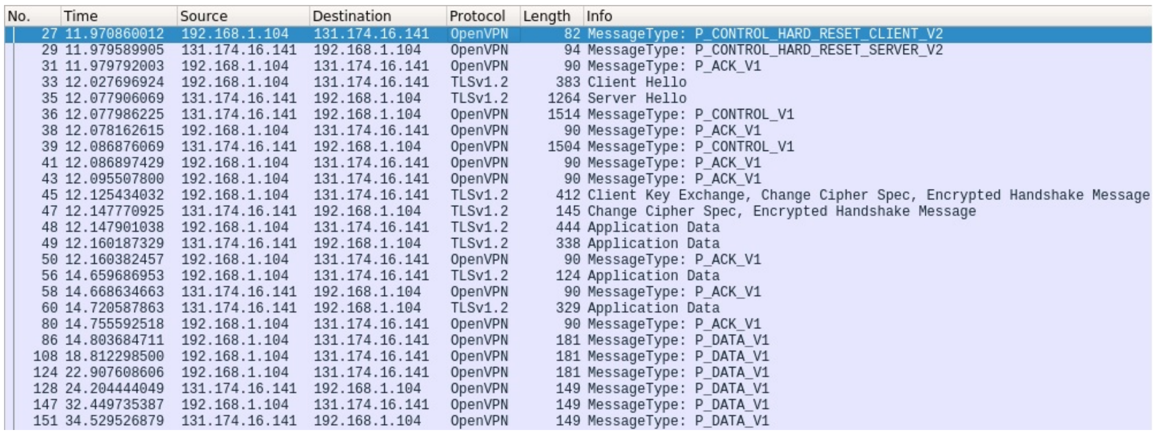
The advantage of using this option can indeed be a major one. If all packets without a valid HMAC are immediately discarded upon arrival then only the packets of authenticated, trusted users will be read. This makes it very hard, if not impossible, for attackers to send malicious data to your server. As the server won't read any data from those without permission to communicate with the server.

4.3 The OpenVPN protocol in TLS mode

This section will give an overview of the client-server communication of OpenVPN. It will provide both a high level description of the protocol itself, as well as a detailed explanation of the various message types and their contents.

4.3.1 The protocol

OpenVPN does not clearly state what its protocol looks like. However, by combining the rough description of the protocol and the messages [3] with a trace of an OpenVPN session (Figure 4.1) it is possible to piece together



```
No.   Time         Source          Destination     Protocol  Length Info
   27 11.970860012 192.168.1.104   131.174.16.141  OpenVPN       82 MessageType: P_CONTROL_HARD_RESET_CLIENT_V2
   29 11.979589905 131.174.16.141  192.168.1.104   OpenVPN       94 MessageType: P_CONTROL_HARD_RESET_SERVER_V2
   31 11.979792003 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   33 12.027696924 192.168.1.104   131.174.16.141  TLSv1.2      383 Client Hello
   35 12.077906069 131.174.16.141  192.168.1.104   TLSv1.2     1264 Server Hello
   36 12.077986225 131.174.16.141  192.168.1.104   OpenVPN     1514 MessageType: P_CONTROL_V1
   38 12.078162615 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   39 12.086876069 131.174.16.141  192.168.1.104   OpenVPN     1504 MessageType: P_CONTROL_V1
   41 12.086897429 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   43 12.095507800 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   45 12.125434032 192.168.1.104   131.174.16.141  TLSv1.2      412 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   47 12.147770925 131.174.16.141  192.168.1.104   TLSv1.2      145 Change Cipher Spec, Encrypted Handshake Message
   48 12.147901038 192.168.1.104   131.174.16.141  TLSv1.2      444 Application Data
   49 12.160187329 131.174.16.141  192.168.1.104   TLSv1.2      338 Application Data
   50 12.160382457 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   56 14.659686953 192.168.1.104   131.174.16.141  TLSv1.2      124 Application Data
   58 14.668634663 131.174.16.141  192.168.1.104   OpenVPN       90 MessageType: P_ACK_V1
   60 14.720587863 131.174.16.141  192.168.1.104   TLSv1.2      329 Application Data
   80 14.755592518 192.168.1.104   131.174.16.141  OpenVPN       90 MessageType: P_ACK_V1
   86 14.803684711 192.168.1.104   131.174.16.141  OpenVPN      181 MessageType: P_DATA_V1
  108 18.812298500 192.168.1.104   131.174.16.141  OpenVPN      181 MessageType: P_DATA_V1
  124 22.907608606 192.168.1.104   131.174.16.141  OpenVPN      181 MessageType: P_DATA_V1
  128 24.204444049 131.174.16.141  192.168.1.104   OpenVPN      149 MessageType: P_DATA_V1
  147 32.449735387 192.168.1.104   131.174.16.141  OpenVPN      149 MessageType: P_DATA_V1
  151 34.529526879 131.174.16.141  192.168.1.104   OpenVPN      149 MessageType: P_DATA_V1
```

Figure 4.1: A TLS Mode trace with Wireshark

what the protocol looks like. The protocol can be summarized as follows, while a high level representation of this protocol can be seen in figure 4.2. Note that the usage of the tls-auth feature does not change this message sequence, just the content of the messages.

1. An OpenVPN session starts with a client requesting a hard reset (1), signaling the server to reset its internal state and start a new connection. This can be seen as the Client's Hello. The server will then answer with its own version of the hard reset packet (2). This can be either a V1 or V2 message depending on the server settings (either use key generation method 1 or 2).

2. The client will then send an acknowledgment to the server for the received packet (3) and, not waiting for a response, immediately start a TLS session (4a). This session is encapsulated in OpenVPN's P_CONTROL_V1 packets. As of writing this paper no V2 version of these packets exists.

3. Messages (4a) and (4b) in the figure represent a series of messages during which a TLS handshake is performed and a temporary TLS tunnel is set up. With (4a) being the initial Hello from the client, (4b) being the rest of the handshake that follows.

4. Message (4c) represents the temporary TLS tunnel (which was set up with the handshake in (4a) and (4b)). Note that this is not the actual VPN tunnel, but rather a temporary secure channel over which the key for the VPN tunnel can be discussed. What is being sent over this tunnel depends on the exact client and server settings but the purpose of the tunnel is to communicate the session keys for the VPN tunnel. If the server wants to validate the client through a username and password then this information too is shared.

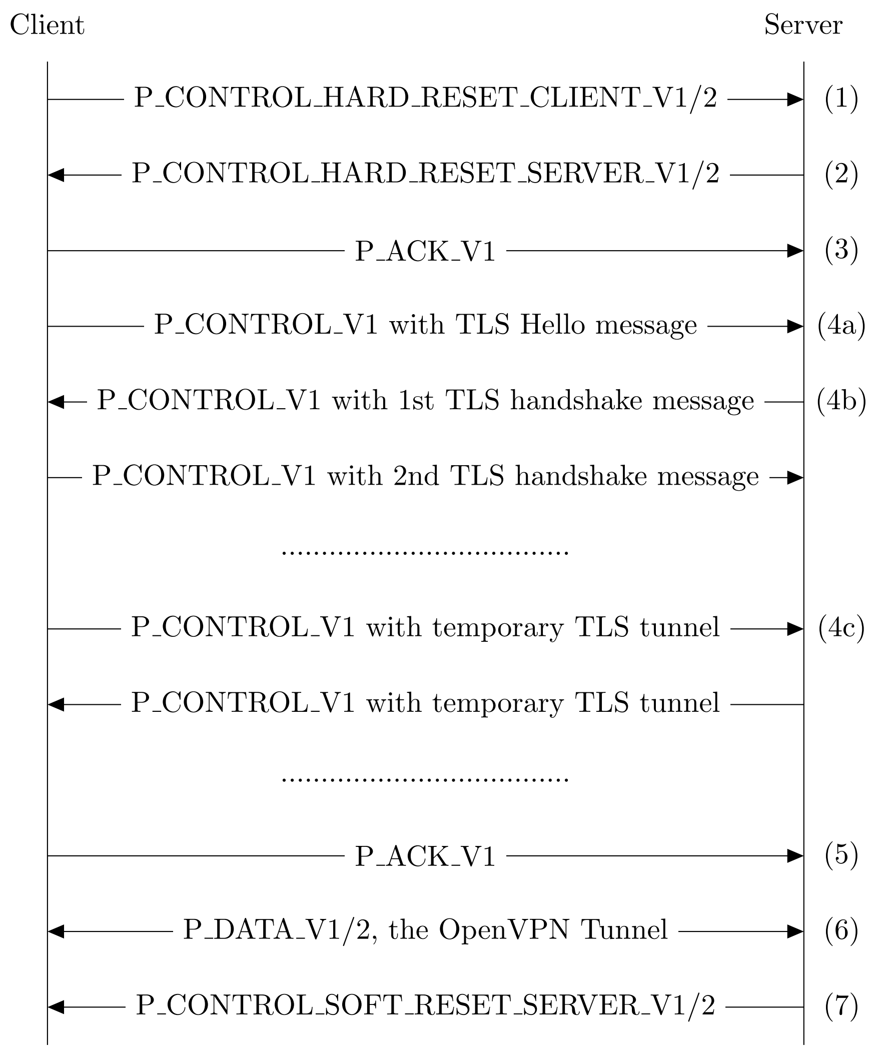

Client
Server
P_CONTROL_HARD_RESET_CLIENT_V1/2
(1)
P_CONTROL_HARD_RESET_SERVER_V1/2
(2)
P_ACK_V1
(3)
P_CONTROL_V1 with TLS Hello message
(4a)
P_CONTROL_V1 with 1st TLS handshake message
(4b)
P_CONTROL_V1 with 2nd TLS handshake message
P_CONTROL_V1 with temporary TLS tunnel
(4c)
P_CONTROL_V1 with temporary TLS tunnel
P_ACK_V1
(5)
P_DATA_V1/2, the OpenVPN Tunnel
(6)
P_CONTROL_SOFT_RESET_SERVER_V1/2
(7)

Figure 4.2: An OpenVPN session in TLS mode for both key generation methods

5. With the last needed information to start the VPN tunnel received, the client sends one last acknowledgment to the server (5).

6. Now the VPN tunnel can finally start and encrypted data is sent back and forth between the client and server.

7. At any time during the tunnel, the server can send a soft reset message. This signals the client that the current session key will expire soon and a new key needs to be negotiated, starting another string of encapsulated TLS packets to negotiate a new key.

Looking at this protocol there are two things that stand out. First the protocol does not discuss or share any OpenVPN options other than specifying which key generation method the client wishes to use. This is because the protocol assumes the specific server configuration is already known, an OpenVPN client will use a generated config file specifying all the relevant server options to connect to an OpenVPN server. To start a connection a user has to be supplied with a configuration file with all the necessary information the client needs to connect to the server. Any wrong settings in this configuration file will result in the protocol failing. Secondly the protocol messages themselves can be divided in two types.

- The P_CONTROL_* and P_ACK messages that control what happens and contain the TLS session, together these form the Control Channel.
- And the P_DATA packets that contain the tunneled VPN data. OpenVPN refers to these messages as the Data Channel.

The next section will delve deeper into these messages and how they are formatted.

4.3.2 OpenVPN message structure in TLS mode

This section is dedicated to the specifics of the different OpenVPN messages in TLS mode. OpenVPN currently has 8 different message types divided over the Control and Data Channels. These message types are listed in Table 4.1. Almost all of these messages can be seen in a single session, as shown in figure 4.2.

Data Channel packets

The data tunnel or Data Channel only contains a single message type, the P_DATA message. This message type is the most complicated due to the several layers of protection that are contained in this message. These VPN tunnel or Data channel messages contain, encapsulated in a few layers of authentication and encryption, the IP packets from and for the client. The structure is detailed in Table 4.2 and the rest of this section.

As can be seen in the table, V1 and V2 packets are mostly identical with the exception of one field. This Peer ID field serves to identify a peer, this allows a peer to change its IP address while keeping the present connection and key active. Do note that despite their confusing naming, these V1 and V2 packets here are unrelated to the key generation types V1 and V2.

Before version 2.4, OpenVPN only has different payload structures for the Data Channel: one for CBC mode and one for the no cipher mode (in case you set the cipher to none). OpenVPN versions 2.4 and later will also include three additional cipher modes (CFB, OBF and GCM) as the plan is to also allow OpenVPN clients and servers to negotiate the data channel cipher. All these cipher modes will have the same general P_DATA packet

OP code	Message type	Message purpose
0x01	P_CONTROL_HARD_RESET_CLIENT_V1	Initial message from client, request to start a new session using first key generation method.
0x02	P_CONTROL_HARD_RESET_SERVER_V1	Reply from server to request to start a new session using first key generation method.
0x07	P_CONTROL_HARD_RESET_CLIENT_V2	Initial message from client, request to start a new session using second key generation method.
0x08	P_CONTROL_HARD_RESET_SERVER_V2	Reply from server to request to start a new session using second key generation method.
0x03	P_CONTROL_SOFT_RESET_V1	Force tunnel key renegotiation.
0x04	P_CONTROL_V1	Encapsulated TLS handshake or tunnel message.
0x05	P_ACK_V1	Acknowledgment of received Control Channel message.
0x06	P_DATA_V1	Tunnel packet V1.
0x09	P_DATA_V2	Tunnel packet V2.

Table 4.1: List of OpenVPN message types in TLS mode

structure, except the payload contents themselves will be slightly different. The different payload contents are summarized in Tables 4.3, 4.4, 4.5 and 4.6.

What	Length	Notes
Packet length	2 bytes	does not include the 2 bytes of the packet length.
OP code and Key ID	1 byte	In the format XXXXXYYY, with X being the OP code and Y the key ID.
Peer ID	3 bytes	Only included in V2 data channel packets.
Payload	n bytes	Contents depend on several settings

Table 4.2: P_DATA message composition

What	Authenticated	Encrypted	Notes
HMAC	No	No	
IV	Yes	No	Consists of random bits.
Packet ID	Yes	Yes	
Timestamp	Yes	Yes	Only present in Static Key mode.
Packet payload	Yes	Yes	The tunneled packet.

Table 4.3: Data Channel payload structure for the CBC mode

What	Authenticated	Encrypted	Notes
HMAC	No	No	
IV	Yes	No	Consists of an packet ID, timestamp and optional 0 padding
Packet payload	Yes	Yes	The tunneled packet.

Table 4.4: Data Channel payload structure for the CFB and OFB modes

What	Authenticated	Encrypted	Notes
opcode/peer-id	Yes	Yes/No	peer-id is only present and field is only authenticated in P_DATA_V2 packets
Packet ID	Yes	No	
TAG	No	No	
Packet payload	Yes	Yes	The tunneled packet.

Table 4.5: Data Channel payload structure for the GCM mode

What	Authenticated	Encrypted	Notes
HMAC	No	No	
Packet ID	Yes	No	
Timestamp	Yes	No	Only present in Static Key mode.
Packet payload	Yes	No	The tunneled packet.

Table 4.6: Data Channel payload structure when no cipher is used

Control Channel packets

What	Length	Notes
Packet length	2 bytes	does not include the 2 bytes of the packet length.
OP code and Key ID	1 byte	In the format XXXXXYYY, with X being the OP code and Y the key ID.
Local session ID	8 bytes	A randomly generated local ID for this session.
HMAC	16 or 20 bytes	Only present if the -tls-auth setting is used.
Packet ID	4 or 8 bytes	Only present if the -tls-auth setting is used. Optionally contains a time_t timestamp.
Ack ID Array length	1 byte	
Ack ID Array	As long as specified	
Remote session ID	8 bytes	Only present if the Acknowledgment ID Array is present (length > 0).
Message packet ID	4 bytes	A local sequential message ID, starting at 0 and incremented by 1 with each packet send.
TLS Payload	N bytes	Only present if OP code is 0x04

Table 4.7: P_CONTROL_* message composition

While there are various control channel message types, their message structure is all more or less the same. As a result the control channel messages can be divided in two formats.

- The P_CONTROL format includes all packages starting with the label P_CONTROL. These messages have direct control over the session or contain a payload that does. They also optionally contain an acknowledgment for earlier messages received. The buildup of these packages is detailed in Table 4.7.

- The P_ACK format functions like a P_CONTROL-light, it is a message that conveys no information other than to acknowledge that certain other messages have been received. Their buildup is detailed in Table 4.8.

What	Length	Notes
Packet length	2 bytes	does not include the 2 bytes of the packet length.
OP code and Key ID	1 byte	In the format XXXXXYYY, with X being the OP code and Y the key ID.
Local session ID	8 bytes	A randomly generated local ID for this session.
HMAC	16 or 20 bytes	Only present if the -tls-auth setting is used.
Packet ID	4 or 8 bytes	Only present if the -tls-auth setting is used. Optionally contains a time_t timestamp.
Ack ID Array length	1 byte	
Ack ID Array	As long as specified	
Remote session ID	8 bytes	

Table 4.8: P_ACK message composition

A quick glance at the tables will show that the only difference between a P_CONTROL and P_ACK packet consists of a few missing fields in the ACK packet. Most of these field names speak for themselves but some require some extra explanation. This mostly focuses on the way acknowledgments are handled.

The first thing that might stand out is the two different session ID fields, Remote and Local. These two fields are different because both client and server generate their own independent session IDs for a given session. If a client then wants to acknowledge that it has received a message from the server it will take the Local session ID from the server packet and add this onto its own message as the Remote session ID. The Ack ID array is then filled with the Message packet ID's the client has received but not confirmed yet. The combination of Message packet IDs and the Remote session ID then serves as identified for the messages that are being acknowledged.

The temporary TLS tunnel

As already stated and shown in Figure 4.2, before the real tunnel starts. OpenVPN exchanges some key data and optional authentication info over a temporary TLS tunnel. This tunnel is setup through a TLS handshake embedded in the payload field of the OpenVPN P_CONTROL_V1 packets, the temporary TLS tunnel this handshake sets up is also embedded in this field. The plaintext of this message comes in two forms depending on the key generation method used. For key method 1 refer to Table 4.9, for key method 2 refer to Table 4.10. It is worth noting that the information from this section, despite being rather important to the overall security of the protocol as it details the key generation, is not mentioned on the 'Security Overview' of the OpenVPN Documentation on the official site. Instead, this information is only available if you look through the source code of OpenVPN itself. The information provided in this section has been found with a combination of looking through the source code and a set of notes compiled for OpenVPN-NL [20].

Most of these fields are self explanatory, however the Key source structure and the Options string aren't immediately clear from just the table description. So let us look a bit closer at these fields.

What	Length	Notes
Cipher key length in bytes	1 byte	
Cipher key	n bytes	
HMAC key length in bytes	1 byte	
HMAC key	n bytes	
Options string	n bytes	Null terminated, client and server options string must match.

Table 4.9: TLS tunnel plaintext for Key generation method V1, encrypted and sent as payload in step 4c of Figure 4.2

Key source structure

The `key_source` structure is made up of 3 arrays back to back and is defined as follows.

```
struct key_source {
    uint8_t pre_master[48]; \\only provided by the client
    uint8_t random1[32]; \\seed for the master secret
    uint8_t random2[32]; \\seed for key expansion
};
```

These arrays are used to exchange entropy information to be used in the generation of the cipher key for the Data Channel, but only if key generation method V2 is used (see Table 4.10). These arrays are filled slightly differently depending on whether you're acting as client or server. The client will fill all 3 arrays with random data and send this to the server. The server however will fill the pre_master array with all 0's and fill the other two with random

What	Length	Notes
Literal 0	4 bytes	
Key method	1 byte	
Key source structure	112 bytes	
Options string length	2 bytes	The possible contents for this string are listed in Table 4.11
Options string	n bytes	Null terminated, client and server options string must match.
Username string length	2 bytes	includes null. Optional.
Username	n bytes	Null terminated. Optional.
Password string length	2 bytes	includes null. Optional.
Password	n bytes	Null terminated. Optional.

Table 4.10: TLS tunnel plaintext for Key generation method V2, encrypted and sent as payload in step 4c of Figure 4.2

data like the client does. Once both client and server have received their random data sources they can use the PRF function from the TLS library to generate the keys and secret for the Data Channel.

```
master_secret[48] = PRF(pre_master_secret[48], "master secret",
           client_random[32] + server_random[32]);
key_block[] = PRF( master_secret[48], "key expansion",
           server_random[32] + client_random[32]);
```

Options string

Both key generation method V1 and V2 make use of a null terminated options string as a final check to make sure the connection will succeed. As stated before, a client trying to connect to a server has to enter all its connections settings locally as the protocol itself does not feature an exchange of options. This raises the possibility of a client attempting to connect with a faulty or outdated configuration file which can potentially lead to unpredictable behaviour. As a safe guard against this a portion of the options string is shared during the key generation, the options received options string is then compared against the local one and if the certain fields do not match the connection is terminated. An example of such an options string is the following:

```
V4,dev-type tun,link-mtu 1569,tun-mtu 1500,proto UDPv4,
cipher AES-256-CBC,auth SHA256,keysize 256,
```

`min-platform-entropy 16,tls-auth,key-method 2,tls-client`

With the exception of V4, which seems to be present by default to denote the start of an options string, all of these are simply taken from the relevant client or config file. Worth noting is that not all of these options are checked. There is a specific list of options that need to match in order for the connection to not be terminated (Table 4.11). These options are separated into Tunnel, Crypto and SSL options. Keep in mind that client/server specific options are matched against their counterpart on the other end of the connection, for example the tls-client option will be matched against tls-server and vice versa. A less obvious adherent to this rule is the ifconfig option, "ifconfig x y" has to be matched against "ifconfig y x".

Tunnel		Crypto	SSL
dev tun—tap	proto tcpserver	cipher	tlsauth
devtype tun—tap	tunipv6	auth	tlsclient
linkmtu	ifconfig x y	keysize	tlsserver
udpmtu	complzo	secret	
tunmtu	compress alg	noreplay	
proto udp	fragment		
proto tcpclient			

Table 4.11: List of options from the Options string that have to match, entries retrieved from: `https://github.com/OpenVPN/openvpn/blob/master/src/openvpn/options.c`

4.4 OpenVPN clients and servers

This section will discuss alternatives to the OpenVPN client or server.

A quick google search for OpenVPN clients will lead to several results. Once you look into the installation instructions however you will always find a dependency on the official OpenVPN package, immediately betraying the 'client' as being just a GUI for OpenVPN. To strengthen this suspicion, almost all results of this search can also be found on the OpenVPN related project page[2] under the GUI section. There are however two other results which are not listed on this page:

- OpenVPN-NL[3] is an altered version of the basic OpenVPN server and client. It was built according to guidelines set by the Dutch government's national communications security agency. In essence it is a

[2]`https://community.openvpn.net/openvpn/wiki/RelatedProjects`

[3]`https://openvpn.fox-it.com/`

stripped down and hardened version of the official OpenVPN client, with options that were deemed unsafe completely stripped away, other options mandated and now only using mbed TLS as back-end. This is the closest to a custom OpenVPN client/server there was to be found, though it is only an adaptation of the existing code.

- Pritunl[4] at first glance does seem to claim to be a new implementation. However a look at the installation script does show OpenVPN as one of its dependencies which means it is yet another GUI. A highly specialized one however that interfaces with the VPN servers and services that they sell as well as giving the user the option to use VPN solutions other than OpenVPN.

In short, other than the hardened OpenVPN-NL, there doesn't seem to be a true alternative if you want to use OpenVPN. Every other usable client and server seem to be build around the Official OpenVPN release.

[4] `https://pritunl.com/`

Chapter 5

Dummy Client and Fuzzer

This chapter will cover the fuzzing of OpenVPN and test setup used for the experiments performed in this paper. The first step towards making the fuzzer will be building a dummy client. A program that acts like a normal client, can construct OpenVPN messages and can successfully trick an OpenVPN server into setting up a secure connection. This means the dummy client can successfully go through the happy flow of the client server communication up to and including step 7 in Figure 4.2. It will try to complete this exchange using as many static messages as possible and without computing an actual VPN tunnel key, the goal is solely to follow the protocol up to the desired state. Once this is up and running this dummy client can be used as base for a fuzzer, modifying the messages it sends to fuzz the server.

This chapter is set up as follows. Section 5.1 will summarize the test setup used for the fuzzing and testing done in this paper. Section 5.2 looks at the dummy client that was made and the design decisions made along the way. And section 5.3 will go over the problems encountered while making the dummy client and fuzzer and how these were debugged.

5.1 The general test setup

For all tests two virtual machines were setup: One is running a Ubuntu 16.04 Server with OpenVPN 2.3.10 installed. The OpenVPN server was configured using the default settings from a config file provided with the installation. The other virtual machine will be running a regular Ubuntu 16.04 installation, this virtual machine will take the role of client and any fuzzing will be done from this virtual machine. The client VM will also have wireshark installed to monitor all OpenVPN traffic. The OpenVPN server also logs any connection attempts as well as errors and faults in the received messages. These logs can also be used to monitor the fuzzing attempts.

The fuzzer and dummy client are used to probe the server and are written in C/C++, just like OpenVPN. The initial idea was to attempt to take as much code as possible from the basic OpenVPN implementation but this idea was quickly scrapped. OpenVPN can function as both client and server and as a result both client and servercode exist in the same files. This combined with the fact that a single message will travel to a multitude of files and checks within these files makes it hard to pinpoint and copy single code snippets to be used for the fuzzing and dummy clients. As a result it is easier to construct messages to send to the server from scratch. The added benefit of this approach is that we can easily modify any part of the send messages, making it easier to build a fuzzer out of the dummy client. A downside of this approach, of course, is the added work that comes with writing everything from scratch.

5.2 Dummy Client

The dummy client is a simple C++ program designed to emulate standard client behavior, but simplified with no randomness or actual VPN tunnel traffic. The purpose of this client is to replicate the happy flow of the OpenVPN protocol, creating and sending regular, unaltered messages to the OpenVPN server up to the point where the VPN tunnel should open. After this it will terminate the connection and report the success. If the Wireshark logs of this exchange, as well as the OpenVPN server logs, show no oddities or differences between this and a regular client connection. Every time the dummy client successfully resolves a new step in the protocol another new step in the protocol can be fuzzed, as we now have a baseline network trace to compare the fuzzing attempts to as well as a base packet (the dummy client packet) to corrupt and manipulate for fuzzing.

However, while the dummy client can read, create and modify OpenVPN messages. It will not be build to do the same for TLS messages, as I'm not focusing on fuzzing the TLS implementation. To still handle the TLS handshake (see Figure 4.2, steps 4 to 6), the client will make use of a separate TLS client that will handle the handshake and encryption of these steps. This TLS client consists of an Mbedtls implementation capable of doing a standard handshake (and some encryption after the handshake) with some wrapper functions and a state machine around this to control its behaviour. The TLS client will communicate with the dummy client as if the dummy were a TLS server. The dummy client will only take the TLS messages, wrap them in an OpenVPN header and send these to the actual server (only acting like a man-in-the-middle). Any TLS responses from the server will undergo the reverse treatment, being unwrapped out of this header and send to the black-box as pure TLS messages. This approach will simplify the implementation of these steps of the protocol and the fuzzing later on, letting me focus purely on the OpenVPN messages. The downside of this approach is that this really makes the fuzzing of the TLS messages (apart from the TLS tunnel payloads) impossible without some heavy modifications. However, as this was already out of the experiment's scope this is not considered a problem.

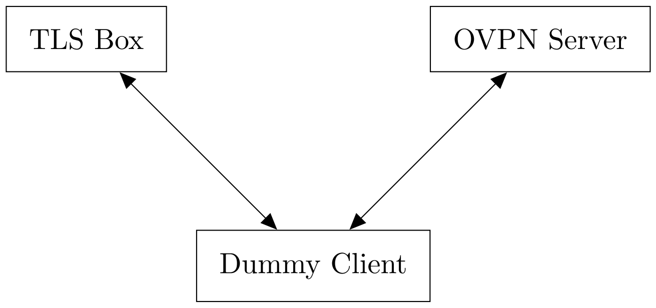


Figure 5.1: The dummy client/fuzzing setup. All TLS communication is handled by a separate TLS-box, an TLS implementation which communicates with the server through the dummy client/fuzzer

5.3 Debugging the dummy client

While constructing the dummy client there was one message that remained unclear. Step 4c in Figure 4.2, the exchange of key data and some client verification information, is not clearly documented. And unlike other messages in the Control Channel this step is fully encrypted. Figuring out what the exact format of this message is has proven to be a chore. The following methods have been tried:

- The message itself has been encrypted with TLS and we possess all certificates for both the client and the server. Wireshark has an option that lets you decrypt TLS traffic if you can provide it with all the certificates and private keys. This option however led to no results. It seems that Wireshark can not decrypt TLS traffic when the TLS authentication is two-way. Wireshark also seems to struggle with the encapsulation of the TLS protocol inside of the OpenVPN protocol.
- OpenVPN itself has two commands (–cipher and –tls-cipher) that let one alter what ciphers are used, one for the Data Channel and one for the Control Channel. While the Data Channel allows a 'none' option to be used for no encryption, the Control Channel is stricter and does not allow the user to set a null cipher.

After these two options failed there was but one left, digging through the source code itself to try and find the origin of the different segments of this message. Luckily we can get some help from the earlier mentioned

OpenVPN-NL implementation, which has a better documentation on this part of the protocol [20]. Using the more detailed information from this specification on the format of this message, together with the source code of OpenVPN, we can finally infer what the contents of these messages are (see Section 4.3.2, The temporary TLS tunnel).

5.4 the Fuzzing tool

With all the specifics of the OpenVPN protocol and message contents documented, we can start to transform the Dummy Client into a fuzzing tool. This is done by taking the mostly hard coded messages and turning these into variable objects. The full knowledge of the protocol allows us to simply define the values for all the different parts of an OpenVPN message and craft a message out of these only before sending. We can then corrupt this message by changing one or more of these values to something the server does not expect. The tool can also do the reverse, taking an OpenVPN message and extracting all info from it so it can be presented in a simpler format. This is useful to monitor server responses during the fuzzing attempts, but also to easily insert base messages into the fuzzer.

Due to time restraints however the fuzzing tool was never incorporated into an actual fuzzer, the debugging process to try and reverse engineer the entire protocol took up too much time to still create a fully fleshed out fuzzer. In the end development of the fuzzing tool choked during the implementation of the TLS Box. Nevertheless, the fuzzing tool is able to craft and dissect OpenVPN Control channel messages. The tool in its current shape has been used to run some simple, manually crafted experiments which are detailed in the next chapter. The source code of the fuzzing tool can be found at `https://github.com/Svalburg/OpenVPN-FuzzTool`.

Chapter 6

Fuzzing Results

This chapter will cover the results from the fuzzing experiments made with the Fuzzer detailed in Chapter 5. All experiments were performed on OpenVPN 2.3.10 installed on a Ubuntu 16.04 Server running in a virtual machine. All experiments were manually fuzzed and for every experiment the following information will be detailed: The specific message fuzzed (see Figure 4.2, an expectation for the result, the actual result and any potential ethical issues related to the experiment. With ethical issues we mean whether or not it is ethical to perform the experiment outside of a testing environment, for example on a real live OpenVPN server we do not own. Since we didn't manage to create an automated fuzzer and only have a fuzzing tool that can easily edit specific parts of OpenVPN messages, all experiments performed here have been crafted manually.

A. The initial message, changing the key method

Since the client initiates a connection and the initial message indicates what key method to use, and knowing that by default OpenVPN uses (and rec-

Experiment	Message	Structure	Result
A	1	Table 4.7	Server ends connection
B	1, 3	Table 4.7, 4.8	Server ends connection, but logs a weird error
C	1 to 4a	Table 4.7	Server completely ignores or ends connection
D	4a	Table 4.7	Server ends connection

Table 6.1: Experiment summary

ommends) key method 2, what happens if a client sends out a V1 message instead? This experiment changes the OPcode of Message 1 in Figure 4.2, exact contents of this message are detailed in Table 4.7.

Expectation The OpenVPN server should just throw this message away, key method 2 is safer and while key method 1 is still supported for backwards compatibility reasons, it should not be accepted unless explicitly specified in the server setup.

Result The OpenVPN server log explicitly lists this event happening, listing that an OPcode for V1 is received while key method 2 is used by the server. It then terminates the TCP connection.

Ethical Issues As this experiment does not involve any message corruption or altering and sends a completely standard V1 packet as described in the documentation, there should be no issues trying this on a random server. If the server is properly configured it will just terminate the connection. But, if for some reason the server is configured to accept key method one. Then this means the server administrators explicitly allowed this.

B. Fuzzing the hard reset header and the accompanying acknowledgment

Even though the initial packet from the client does not contain any information other than the requested key method, the header still has some fields that could be fuzzed. The fields tried were: packet length, the OPcode, Ack ID array and length and TLS payload. For length fields both extremely high and extremely low values were tried. This experiment changes the contents of Message 1 and 3 in Figure 4.2, the makeup of these messages is described in Tables 4.7 and 4.8.

This was the first experiment to actually make use of the fuzzing tool, albeit manually. A total of 7 packets were crafted and send towards the server, each modifying a single field to see how the server would behave: An abnormally low packet length, an abnormally high packet length, a different OPcode, not existing message ID's in the Ack array, an abnormally low Ack array length, an abnormally high Ack array length and a TLS payload that should not be there.
Crafting these message with the tool is extremely easy as it turns an OpenVPN packet into an editable object with getters and setters, making it both convenient for manual manipulation as well as for automatic manipulation. The code for the last experiment mentioned, adding a TLS payload to the hard reset message and writing the packet back to a buffer, looks as follows:

```
//Basic client hard reset v2 packet
unsigned char hardresetclientv2[] =
{
  0x00, 0x0e, 0x38, 0x04, 0x6c, 0x44, 0xbd, 0x10,
  0xce, 0xe3, 0x98, 0x00, 0x00, 0x00, 0x00, 0x00
};

PControl *hardreset = new PControl(hardresetclientv2);
//TLS_hello is another predefined char buffer containing
//a standard TLS hello message taken from an earlier session
hardreset->setTLSPayload(TLS_hello, 301);
hardreset->setLength((short)315);
unsigned char buffer[500];
hardreset->toPacket(buffer);
```

Expectation Since the server will already terminate if the OPcode points to the wrong key method, I'm expecting a completely wrong OPcode will result in a similar termination. Fuzzing the length, ACK ID array or payload fields might result in some interesting results but will probably result in a similar immediate termination.

Result The moment any data value falls outside of the expected parameters the server closes the connection with a clear error message in the log. A faulty length, OPcode, weirdly modified Ack array and a TLS payload field filled with corrupted values all lead to an error message pointing to the faulty element of the packet and a terminated connection. Interestingly enough, setting the length of the packet to be long, but not longer than the maximum expected length of 1546, leads the server to believe that the packet contains a TLS message payload and will then give an error that the TLS message is unreadable. Even if the packet is the first packet received from the client (which should not even have a TLS payload).

Ethical Issues Since this experiment starts modifying packets outside of normal parameters (and past CVE's have shown that at least the modification of the length and payload fields has led to server crashes) this experiment should not be used on random servers without permission. This experiment will only be performed on my own test server.

C. Attempt to start the temporary TLS tunnel negotiation early

Experiment B led to an interesting result. Even though we made the server think there was a payload attached to the hard reset message, the server only complained that the payload contained unreadable information, not that it was present. The protocol specifies that this message should not have a payload at all. What if we add a valid TLS message as payload to this message and try to start negotiating the temporary TLS tunnel earlier?

Expectation I expect this to trigger an error message. The protocol clearly states the tunnel negotiation starts at a later message so the server should end any attempt to start it earlier.

Result Two versions of the experiment were tried, one with a regular hard reset packet that just happens to have a TLS hello message as payload. And one with an actual P_CONTROL packet, skipping the hard reset packet alltogether. In the first case the server completely ignores the TLS payload, it doesn't log that it was present and simply sends a hard reset message back. In the second case the server terminates the connection immediately, logging an unable to route message error. This indicates the server did not know the client (as there was no hard reset message from the client indicating that they wish to start a connection) but did recognize the packet as a TLS message, the server then safely terminates the connection.

Ethical Issues This change to the protocol does not seem malicious, but does tread outside of the happyflow of the protocol. If the server does not terminate the connection immediately, it might lead to unpredicted behaviour. Therefore it is safest to not perform this experiment on a public server.

D. Fuzzing the first TLS hello packet

If the previous experiment did not lead to immediate termination of the connection upon receiving a slightly altered packet. What would happen once we actually reach the TLS part of the protocol and we apply the same fuzzing techniques. Will a faulty length trigger some warning flags now where they didn't before? Will a corrupted payload field be passed on by OpenVPN blindly and cause issues once the TLS implementation tries read it? Will any suspicious or changed data in the OpenVPN header (weird ack fields, a changed session ID, etc) cause an abort and how is this logged? This experiment changes the contents of Message 4a in Figure 4.2. Specifically, the TLS Payload field listed in Table 4.7.

Expectation : As a crash relating to a weird TLS payload length was part of an old CVE and they claimed to have fixed this issue. I would expect that they fixed it properly and any suspicious or wrong data in this type of packet as well will lead to the packet being ignored and the connection closed. This expectation is strengthened by Experiment B's results where attempting to read the payload while there is no payload present results in OpenVPN giving a TLS reading error, indicating that it could not properly read the TLS payload.

Result A similar result to experiment B. The same error messages logged by the server.

Ethical Issues same as the previous, this involves purposefully and maliciously altering data to try and get a non standard reaction from the server. This should only be performed in the test environment.

Chapter 7

Future Work

This chapter will look at future work that can still be done, this includes both completely different angles on this research as well as goals that weren't attained within the time frame of this thesis.

AFL[1] could potentially also be used to fuzz OpenVPN. However AFL does not natively support network services, as it expects input over stdin. This would require a rewrite of OpenVPN or some other hacks to make it work. An attempt at this has been suggested in the past[2] by the creator of the AFLize tool, a program which can automatically compile code so it can be used by AFL, but has fallen short due to a lack of developer response. In the end this approach was deemed to time consuming for this research as there were multiple other fuzzing approaches to look at.

The functionality of the fuzzing tool can still be expanded. With the ability to read and create TLS messages for the TLS handshake and tunnel portion of the protocol as well as hooking the tool into a fuzzing framework that automates the experiments and enables actual fuzzing. Furthermore, a set of extra experiments was initially planned but never performed as this state of the fuzzing tool was never reached. These experiments would look at older versions of OpenVPN and would try to replicate the outcome of several CVEs (CVE-2014-8104, CVE-2017-7478 and CVE-2017-7522 as listed in Chapter 2). If the fuzzer can find these vulnerabilities as well it would serve as a proof of concept that the fuzzer can indeed potentially produce results.

[1]`http://lcamtuf.coredump.cx/afl/`

[2]`https://sourceforge.net/p/openvpn/mailman/openvpn-devel/thread/56C5C6F7.1020903%40gmail.com/#msg34862420`

Chapter 8

Conclusions

This research initially attempted to create a fuzzer for OpenVPN and would attempt to run several experiments using this fuzzer. It became quickly apparent, however, that the OpenVPN documentation is severely lacking. At numerous parts through this research was progress halted due to either faulty or completely missing documentation of OpenVPN. OpenVPN claims to be an Open Source project, and while this is indeed true for the code itself, the exact specifics of the protocol are kept needlessly hard to find, scattered through various comments in the code, the doxygen and one or two actual (yet incomplete) explanations of the protocol on the OpenVPN website. What is especially problematic is the incompleteness of the Security Overview page [3]. This page claims to explain the security specifics of the OpenVPN protocol, but not only seems outdated (not mentioning the existence of certain packets at all) but also fails to actually explain how OpenVPN actually generates keys and exchanges entropy data to aid in this generation, which is the entire goal of the protocol. Reverse engineering this this key generation process turned out to be the biggest bottleneck of this thesis.

The OpenVPN documentation is, in short, insufficient. While there is plenty of documentation and general help available on how to use OpenVPN. Finding technical information or the exact specifics of OpenVPN's security turned out to be a chore. This not only makes it harder to actually verify the security of OpenVPN when one is not yet aware of how OpenVPN functions. It also harms the Open-source status of the software and its protocols. It is needlessly hard to replicate the software or write your own versions of the protocols in another language or on another platform. This has caused large delays for this thesis and an eventual change of goals all together. It also explains why no real alternative versions of the OpenVPN client have been found and why every supposed OpenVPN client alternative simply builds around and on top of the official implementation.

As a result of the aforementioned problems, the topic of this thesis gradually shifted from building a complete fuzzing framework to figuring out how OpenVPN operates. By gathering and compiling information from various sources, a low level documentation of the messages in the OpenVPN protocol was created (see Chapter 4). This documentation can be used to create an OpenVPN client or a fuzzer capable of creating OpenVPN messages and corrupting specific fields of those messages. A simple fuzzing tool for OpenVPN was also created, capable of acting like an OpenVPN client but corrupting specific fields of Control channel messages. This tool was then used to perform several manual experiments, none of which uncovered wrongful or dangerous behaviour from the OpenVPN server. The source code of this tool can be found at `https://github.com/Svalburg/OpenVPN-FuzzTool`.

Bibliography

[1] OpenVPN Manual. `https://community.openvpn.net/openvpn/wiki/Openvpn23ManPage`. Accessed: May 2018.

[2] OpenVPN network protocol overview documentation file. `https://github.com/OpenVPN/openvpn/blob/master/doc/doxygen/doc_protocol_overview.h`. Accessed: May 2018.

[3] OpenVPN Security Overview. `https://openvpn.net/index.php/open-source/documentation/security-overview.html`. Accessed: May 2018.

[4] The OpenVPN post-audit bug bonanza. `https://guidovranken.wordpress.com/2017/06/21/the-openvpn-post-audit-bug-bonanza/`. Accessed: July 2017.

[5] Pedram Amini and Aaron Portnoy. Sulley: Fuzzing framework. `http://www.fuzzing.org/wp-content/SulleyManual.pdf`. Accessed: November 2017.

[6] Joeri de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. PhD thesis, Radboud University Nijmegen, 2015.

[7] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol. Technical report, RFC 5246, 2008.

[8] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[9] Sheila Frankel, Paul Hoffman, Angela Orebaugh, and Richard Park. Guide to SSL VPNs. *NIST Special Publication*, 800-113, 2008.

[10] Sheila Frankel, Karen Kent, Ryan Lewkowski, Angela D. Orebaugh, Ronald W. Ritchey, and Steven R Sharma. Guide to IPsec VPNs. *NIST Special Publication*, 800-77, 2005.

[11] Kory Hamzeh, Grueep Pall, William Verthein, Jeff Taarud, W. Little, and Glen Zorn. Point-to-point tunneling protocol (PPTP). Technical report, RFC 2637, 1999.

[12] Berry Hoekstra, Damir Musulin, and Jan Just Keijser. Comparing TCP performance of tunneled and non-tunneled traffic using OpenVPN. Master's thesis, Universiteit Van Amsterdam, System & Network Engineering, 2011.

[13] Stijn Huyghe. OpenVPN 101: introduction to OpenVPN. `https://openvpn.net/papers/openvpn-101.pdf`. Accessed: 27-2-2017.

[14] Christopher Meyer and Jörg Schwenk. Lessons learned from previous ssl/tls attacks-a brief chronology of attacks and weaknesses. *IACR Cryptology EPrint Archive*, 2013:49, 2013.

[15] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[16] Tomas Novickis. Protocol state fuzzing of an OpenVPN. Master's thesis, Radboud University Nijmegen, 2016.

[17] Bruce Schneier, David Wagner, and Mudge. Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2). In *Secure Networking—CQRE [Secure]'99*, pages 192–203. Springer, 1999.

[18] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson, 5th edition, 2011.

[19] John R. Vacca. Virtual private network security. In *Complete Book of Remote Access: Connectivity and Security*, pages 251–267. Auerbach Publications, 2002.

[20] Ebo van der Laan. OpenVPN-NL protocol specification, 2017.