Bypassing deep packet inspection: Tunneling traffic over TLS VPN

Dmitriy Kuptsov
dmitriy.kuptsov@gmail.com

***Abstract*—In some countries, network operators employ deep packet inspection techniques to block certain types of traffic. For example, Virtual Private Network (VPN) traffic can be analyzed and blocked to prevent users from sending encrypted packets over such networks.**

By observing that HTTPS works all over the world (configured for an extremely large number of web-servers) and cannot be easily analyzed (the payload is usually encrypted), we argue that in the same manner VPN tunneling can be organized: By masquerading the VPN traffic with TLS or its older version - SSL, we can build a reliable and secure network. Packets, which are sent over such tunnels, can cross multiple domains, which have various (strict and not so strict) security policies. Despite that the SSH can be potentially used to build such network, we have evidence that in certain countries connections made over such tunnels are analyzed statistically: If the network utilization by such tunnels is high, bursts do exist, or connections are long-living, then underlying TCP connections are reset by network operators.

Thus, here we make an experimental effort in this direction: First, we describe different VPN solutions, which exist on the Internet; and, second, we describe our experimental effort with Python-based software and Linux, which allows users to create VPN tunnel using TLS protocol and tunnel small office/home office (SOHO) traffic through such tunnels.

I. Introduction

Virtual private networks (VPN) are crucial in the modern era. By encapsulating and sending client's traffic inside protected tunnels it is possible for users to obtain network services, which otherwise would be blocked by a network operator. VPN solutions are also useful when accessing a company's Intranet network. For example, corporate employees can access the internal network in a secure way by establishing a VPN connection and directing all traffic through the tunnel towards the corporate network. This way they can get services, which otherwise would be impossible to get from the outside world.

II. Background

There are various solutions that can be used to build VPNs. One example is Host Identity Protocols (HIP) [7]. HIP is a layer 3.5 solution (it is in fact located between transport and network layers) and was originally designed to split the dual role of IP addresses - identifier and locator. For example, a company called Tempered Networks uses HIP protocol to build secure networks (for sampling see [4]).

Another solution is Secure Shell protocol (or SSH) [5]. SSH is an application layer protocol, which provides an encrypted channel for insecure networks. SSH was originally designed to provide secure remote command-line, login, and command execution. But in fact, any network service can be secured with SSH. Moreover, SSH provides means for creating VPN tunnels between the spatially separated networks. Unfortunately, SSH connection can be analyzed and blocked (would it be as widely spread as for example TLS protocol, things could be different).

Like SSH, OpenVPN [6] runs on top of TCP protocol (in fact, OpenVPN can also operate on top of UDP transport protocol). We have evidence that in certain countries OpenVPN is successfully blocked by governments. Of course, it is harder to detect these protocols, because traffic is encapsulated inside TCP/UDP connections. Here, deep packet inspection solutions are required in order to effectively block such tunnels.

Another widely used layer 3 protocol for building the VPNs is IPSec protocol [2]. In IPSec security association can be established using pre-shared keys or using Internet Key Exchange protocols (IKE and IKEv2) [1]. Because IPsec runs directly on top of IP protocol, it can be easily detected without the usage of sophisticated packet inspection solutions.

III. Hardware, software and architecture

In Figure 1 we show the general view of the architecture.

To implement the VPN client and server we have used the Python framework and Ubuntu Linux distribution. The implementation consists of roughly 1.2K lines of code (LOC) and all functions are realized in userspace. We have exposed the implementation in our git repository [3] so that everyone can use it without any fee.

During the experiments, however, we have considered the following setup. For hardware, we have selected a micro instance from DigitalOcean. The instance had a single-core CPU, 25 GB of data storage, 1GB of random access memory, and was located in New York, USA. The VPN client was located in Tashkent, Uzbekistan, and was spinning on Raspberry PI microcontroller. This way we mimicked the SOHO router. We have also used `wget` tool to measure the throughput, and we have used `ping` utility to measure the round trip times.

IV. Configuration

We assume that both server and client are running flavour of Debian operating system (in our case the SOHO VPN box runs Raspberry Pi).

On fresh installation of Ubuntu, first install `ifconfig`:

```
sudo apt-get isntall net-tools
```

Then, install python3, pip3 and needed libraries on both client and server machines. To install the python3, run the following command on both server and client:

```
sudo apt-get install python3
```

Next install pip3 package manager using the following command:

```
sudo apt-get install python3-pip
```

Once python3 installed, install required dependencies:

```
sudo pip3 install python-pytun
```

Next make sure you have git software installed:

```
sudo apt-get install git
```

Now everything is ready for the deployment of the VPN software: make a workspace directory somewhere on your hard drive and checkout the project's repository (we need to do so on both client and server machine as usual):

```
$ git clone https://github.com/dmitriykuptsov/soho_vpn_over_tls.git
```

If needed, on server machine go to the directory `soho_vpn_over_tls/src/server` and modify the IP address of the tun interface in `config.py` file (currently it is set to 10.0.0.1, you may leave it as it is if there is no conflict with your local IP address).

On SOHO box also go to directory `soho_vpn_over_tls/src/client` and modify the IP address of the server in the config.py file (currently it is 94.237.31.77, but you have to change it to an IP address of your own server (on server machine you can check the IP address either from administrative page, like it is offered in UpCloud or DigitalOcean, or by issuing ifconfig command)).

One, probably, needs to also modify the IP address of default gateway. Currently it is set to 10.0.2.2, but it needs to be the default route of your network.

Next, you need to generate password. To do so, do the following (replace the user test password with your own password):

```
$ cd soho\_vpn\_over\_tls/src
$ python3 tools/gen.py <your secure password>
```

Then, copy the generated hash string and add it to the database.dat file in the `soho_vpn_over_tls/server/` folder (you might also want to change the default username in that database file). You also need to change the password (and probably username) in `src/client/config.py`.

Finally, generate private key and certificate for the server:

```
$ cd src/certificates
$ bash generate.sh
```

Then, copy the `certchain.pem` to SOHO VPN client machine to folder `src/certificates`.

Now everything is ready to create VPN tunnel. On server machine, go to directory `soho_vpn_over_tls/src` and run the following command:

```
sudo python3 server/server.py
```

On SOHO box (we use Raspberry PI as such), go to directory `soho_vpn_over_tls/src` and run the following command:

```
sudo python3 client/client.py
```

On your SOHO client machine (the one, for instance, you use to browse the Internet) you need to modify the default gateway so that it points to SOHO VPN box:

```
$ sudo ip route del default
$ sudo ip route add default via 192.168.0.102
```

Of course, you need to replace the `192.168.0.102` with the IP address of you SOHO VPN box.

V. Experimental evaluation

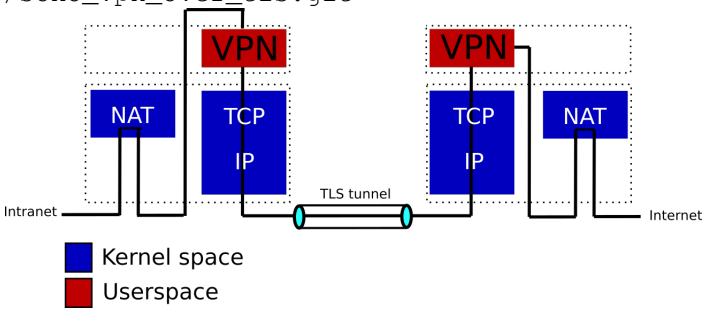

Fig. 1: Architecture of the prototype

We have made several experiments over the course of our work. The very first experiment was related to sending ping messages towards the VPN TLS server and observing the differences in round trip times (basically, we have compared the round trip times between the setting in which tunnel was present and the setting in which the tunnel did not exist). Next, we have sat down to measure the throughput between the local and remote machines. Basically, we have performed 50 measurements for a setting, in which the traffic was going inside the tunnel and the same number of measurements for the setting, in which the traffic was going normally (meaning, unencrypted and not encapsulated in TLS packets). To measure throughput, we have used `wget` tool to download Linux kernel file (1.3MB) from the Internet.

In Figure 2 we show the distribution of round-trip times (RTT). Mean RTT for VPN connection was 293.7 ms, and mean RTT for plain ICMP was 103.3 ms.

In Figure 3 we show the distribution of the obtained throughput for both TLS protected tunnel and regular TCP connections. Mean throughput value for the VPN connection was 608.7 Kb/s, and mean throughput for plain TCP connection was 1890.4 Kb/s. Given these results, we think that in this case the bottleneck is our implementation. However, we believe that our implementation of VPN tunnel is good enough for most small office settings.

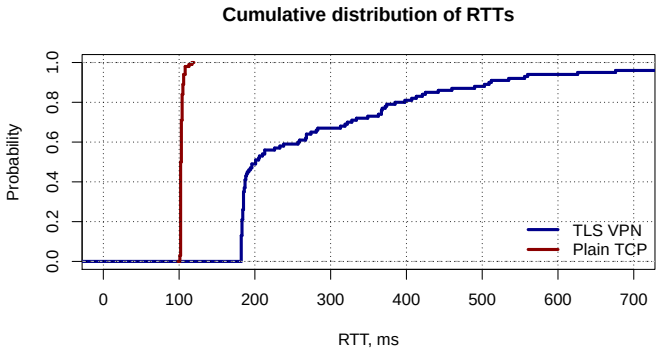

Fig. 2: Distribution of RTTs

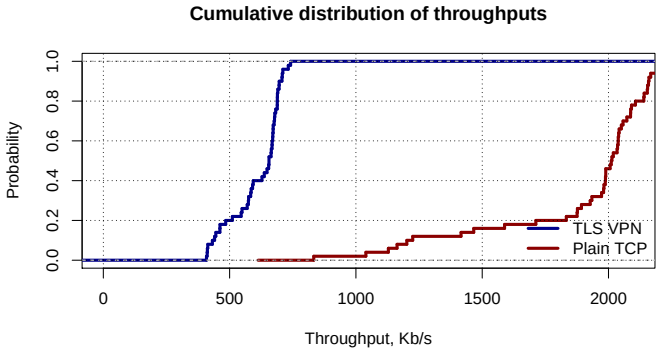

Fig. 3: Distribution of throughputs

VI. Future work

Despite that the tunnel was running for several hours without interruption and we have implemented authentication layer, we still have seen several connection attempts, although unsuccessful, from the outside world. For the future, we plan to hide the VPN server behind the web server, so that only the users who know certain secret can access the VPN server. The idea is simple: when the user knocks by sending HTTP GET request to a secret page, the web server opens a port and redirects the traffic from the client to VPN server. Once the connection is established the server closes port.

VII. Conclusions

In this short report, we have made an attempt to describe how VPN solutions can be blocked by network operators. We argued that VPN traffic can still be hidden from observers, by masking the tunnels using TLS protocol. Our experiments suggest that this approach is handy when there is a need for a secure channel, but network operators are eager to block VPN traffic.

Although it is hard to make a business with this solution (in the long run, network operators can simply block all IP addresses which belong to VPN servers, hosted by a business), we argue, however, that people can use this software and deploy it on their own cloud machines (they can keep the IP addresses of the servers in secret). In this way there is little chance that network operators can pinpoint the IP addresses of the servers: (i) the traffic will look like normal HTTPS traffic (at the end the SSL tunnel is used using default HTTPS ports); (ii) it is a rather complex task for network operators to scan through all IP addresses and find those, which are used to send VPN traffic.

We made the software available for download on a GitHub page. We hope that this project can make the Internet more liberal in certain countries.

References

[1] Internet key exchange. https://en.wikipedia.org/wiki/Internet_Key_Exchange.

[2] IPSec. https://en.wikipedia.org/wiki/IPsec.

[3] SOHO VPN client and server. https://github.com/dmitriykuptsov/soho-vpn-over-tls/.

[4] Tempered networks simplifies secure network connectivity and microsegmentation. https://www.networkworld.com/article/3405853/tempered-networks-simplifies-secure-network-connectivity-and-microsegmentation.html.

[5] D. J. Barrett, R. E. Silverman, and R. G. Byrnes. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[6] M. Feilner. *OpenVPN: Building and Integrating Virtual Private Networks: Learn How to Build Secure VPNs Using This Powerful Open Source Application*. Packt Publishing, 2006.

[7] A. Gurtov. *Host Identity Protocol (HIP): Towards the Secure Mobile Internet*. Wiley Series on Communications Networking & Distributed Systems. Wiley, 2008.