Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dominic Brütsch

Cooperating with Middleboxes in the Internet

Semester Thesis SA-2016-45
April 2016 to July 2016

Tutors: Mirja Kühlewind, Brian Trammell
Supervisor: Prof. Laurent Vanbever

Abstract

Middleboxes are ubiquitous in today's internet. While they provide many useful services, they are also responsible for more than a few problems.

Substrate Protocol for User Datagrams (SPUD) is being developed to make cooperation with Middleboxes possible over an UDP-based encapsulation.

The goal of this project was to develop a framework that tunnels existing TCP traffic over SPUD. On this basis further evaluation of this new protocol can be carried out and results provide valuable feedback for further development.

Contents

1 Introduction **5**
1.1 Motivation 5
1.2 Solution 5
1.3 Task 6
1.4 Code & Program Flow Diagrams 6

2 Background **7**
2.1 Middleboxes 7
2.2 SPUD 7
2.2.1 SPUD Header 8
2.3 TCP 9

3 Implementation **11**
3.1 Overview 11
3.2 Structure 11
3.2.1 Kernel Module 11
3.2.2 Userspace Application 12
3.3 How to run 12
3.4 Hints for debugging 13

4 Test Setup **15**
4.1 Overview 15
4.2 Debugging tunnelling 15
4.3 Evaluating the benefits of SPUD 15

5 Conclusion and Outlook **19**
5.1 Conclusion 19
5.2 Outlook 20

A List of Figures **21**

B Declaration of Originality **25**

C Originial Problem Statement **27**

D Bibliography **29**

Chapter 1

Introduction

Today the Internet is not a simple end-to-end network anymore as it was in the early days. Nowadays there are a lot of middleboxes between two endpoints.

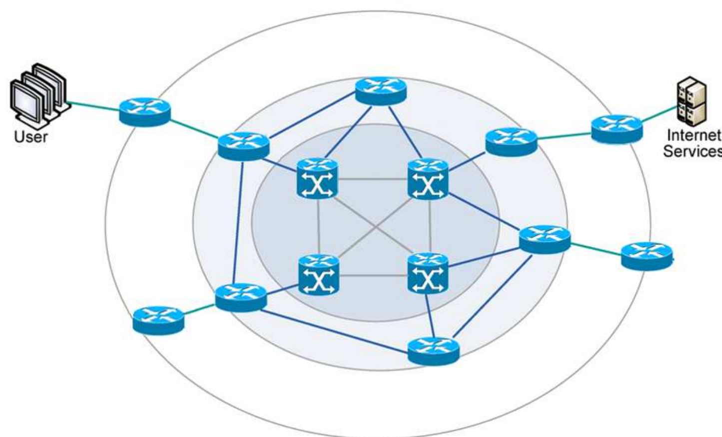

Figure 1.1: Graphic description of the Internet with middleboxes

1.1 Motivation

Middleboxes are deployed for a wide variety of reasons. Simple examples are NAT's or Firewalls but there exist much more complex constructs. A more detailed view of middleboxes is given to the reader in the background chapter. One key point for the motivation, each middlebox individual makes assumption on what can safely be manipulated.

All of this makes deploying a new transport protocol relatively difficult, thus new innovation is severely limited.

Middleboxes today rely on understanding the transport header to decide on how to handle this packet.

An additional requirement for a new transport protocol is signalling of e.g. traffic semantics. This would for example enable to signal low-latency service requirement, amongst other use cases.

1.2 Solution

The solution is called SPUD. Substrate Protocol for User Datagrams is a proposed protocol designed to avoid all these problems by doing the new protocol over an UDP based encapsulation. To assure compatibility, the packet of the new protocol is just added right below an UDP

header. Now the middlebox layer is always at the same position. With SPUD there's finally a way for middleboxes and endpoints to communicate easily. It's a bit like enhanced ICMP control messages.

1.3 Task

Following tasks needed to be completed:

- Getting familiar with SPUD, this meant reading RFC's
- Catching up about kernel module programing & netfilter
- Finding a solution to inject TCP packets
- Collecting information about C socket programming
- Designing and discussing the program flow
- Writing the actual program
- Debugging and testing

1.4 Code & Program Flow Diagrams

All sourcecode including earlier version of code and Flow Diagrams are available on Github:

https://github.com/G-GFFD/SemesterarbeitSpud

A zip file containing everything is also accompanying this document.

Chapter 2

Background

This chapter should serve as a short introduction to the most important topics of this semester project. In no way it claims to cover the theory extensively or even complete, for this reference to specific literature is advised.

2.1 Middleboxes

Middleboxes are ubiquitous in today's internet. They are deployed for a wide variety of reason. The term middlebox is defined in RFC 3234 as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host." Commonly known examples are Firewalls or NAT's.

Middleboxes manipulate, filter or inspect passing traffic. They may do so in order to improve security or performance.

Currently middleboxes analyse each packet individually to decide on how they want to handle a flow of packets. This only works if the middleboxes understands the transport protocol. There is not much cooperation in this process.

A simple example for cooperation can be signalling start/end of a flow.

Today they have only limited possibilities to communicate with the endpoints. In fact the can only communicate with the sender via ICMP messages. A wide range of different application could benefit if better cooperation is possible.

Deploying a brand new cutting edge transport protocol is a very challenging task. Too many middleboxes will simply drop packets they don't understand. This problem significantly affects any new protocol.

However, there is a new protocol, which is able to overcome those issues.

2.2 SPUD

SPUD, short for Substrate Protocol for User Datagrams, is a prototype protocol being developed to enable explicit cooperation with Middleboxes.

Technically SPUD groups multiple UDP packets together in a so called "tube". Right below the UDP header a short SPUD header is added which contains all relevant information. All data is stored below.

As data, in a first step, packets of any current transport protocol can be encapsulated. Graphically shown here with TCP, the transport protocol chosen for this project.

Figure 2.1: Encapsulation of a TCP packet in SPUD

In a later stage, data will be wrapped in CBOR[1] and all data will be encrypted for security reasons. So middleboxes can only analyse the Header, they can't tamper with the encrypted data. A non-cooperating malicious middlebox may still manipulate the SPUD header or just silently drops the packet.

In SPUD, a set of multiple related packets is called a tube. This so called tube is identified by a tube-ID which is shared amongst all those related packets. Tubes have state, they always need to be in one of the following states: opening, running, or resuming. Any communication between endpoints and middleboxes is bound to such a tube. The first packet of a group of related packet is expected to have the open command set, similarly the last is expected to carry the close command.

As a design requirement, SPUD must work in the present Internet. Due to UDP, SPUD can very easily be deployed as every current operating systems supports this. There are no modifications in the Kernel required. Userspace Application wanting to use SPUD can simply do so over a normal UDP socket, so deployment is really easy.

Only for much later in case the protocol becomes well established direct support by the kernel is envisioned.

2.2.1 SPUD Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        magic = 0xd80000d8                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              tube ID                              |
+                                                                   +
|                                                                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|cmd|a|p|  resv  |            CBOR map...                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.2: Visualization of the SPUD header - taken from the Internet-Draft

In every SPUD Header, the first 32bits are 0xd80000d8 (Hex) or 1101 1000 0000 0000 0000 0000 1101 100 (binary). This allows very quick and simple identification of SPUD packets.

[1] Concise Binary Object Representation, see RFC 7049

The 64bit tube ID identifies a group of related packets.

2 bits of command, they can be:

- 00 Data, indicates this packet only contains Data
- 01 Open, requests to open a new tube
- 10 Close, command to closes a tube
- 11 Acknowledgement, acknowledges the opening of this tube

ADEC: Application Declaration, if this bit is set the packet contains Application Declaration in the Data

PDEC: Path Declaration, if this bit is set the packet contains Path Declaration in the Data

4 Reserved Bit (must be set to zero in this version of the protocol)

If the Data is packed in CBOR, a CBOR map follows next.

Just after the header, the data of the packet is stored. Each SPUD packet contains data. Even if there is a new command or any other option that changed to be sent, there is no need to send a packet consisting just of the header. Just keep track of this state and sending can be postponed until there is data to be sent over this tube.

Tubes may time-out after a period of inactivity.

2.3 TCP

TCP, the Transmission Control Protocol is a core protocol of the Internet. It provides reliable transmission of packets on the transport layer. It's connection oriented, a connection needs to be initiated and closed. During connection setup a three-way handshake needs to be exchanged. Afterwards data can be exchange in a stream, this conceals the packet structure. Despite being developed in the 1980ies, it's still widely in use today. However the internet has since evolved and a lot of today's applications could benefit from the new protocol SPUD.

For this reason TCP is an ideal choice to be tunnelled over SPUD in order evaluate the new protocol.

TCP itself in all details is much more complex than general believed.

In all modern operating systems TCP performance is very well optimized. One resource consuming task, especially if it has to be done for countless packets is the calculation of the TCP checksum. Today, this task is often offloaded to the Network Interface Card which has dedicated hardware support to do this. As a consequence TCP packets in the Kernel do not yet contain a valid TCP checksum.

Chapter 3

Implementation

3.1 Overview

The implementation consists of a kernel module and an userspace application. It's programmed in C for Linux and tested under Ubuntu 14.04 LTS and Debian 8. Super User rights are required.

The actual SPUD version used in this implementation is called Substrate Protocol for User Datagrams (SPUD) Prototype draft-hildebrand-spud-prototype-03 according to the Internet-Draft from March 09, 2015. [1]

3.2 Structure

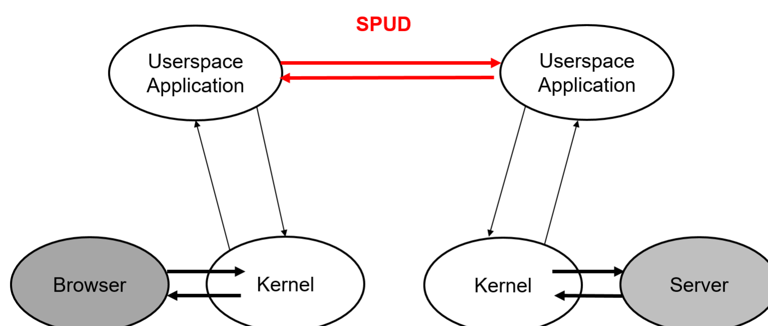

Figure 3.1: Structure of implementation, tunnelling TCP over SPUD

3.2.1 Kernel Module

The kernel modules task is to intercept existing TCP traffic and send it to the userspace application. To do so, the kernel module registers a Netfilter hook at *NF_INET_POSTROUTING* to capture all packets which are about to be passed to the Network Interface Card. Once a packet is detected, the hook function gets called. It looks at the IP header to detect if it's a TCP packet. A copy of a TCP packet is sent to usersapce, the original TCP Packet is simply discarded *(NF_DROP)*. Packets that don't contain TCP are just released *(NF_ACCEPT)*.

As the kernel module can be loaded anytime, even when the userspace application is not running, packets are only sent into userspace, after the userspace application sent a hello

[1] https://tools.ietf.org/html/draft-hildebrand-spud-prototype-03

message to the kernel module, which initiates a session. Communication between both is done via a Netlink Socket.

Error Messages are printed into the kernel log and can readily be accessed by the command *dmesg*.

3.2.2 Userspace Application

This is the main part. It has three tasks, sending SPUD, receiving SPUD and keeping state of tubes. For each task there is an own thread running.

Earlier versions simply had a while(1) loop and handled sending / receiving SPUD alternatively.

Task tcptospud (called after its name in the source code) first sets up a netlink socket and tells the kernel module that it's ready to handle TCP packets. After that, it starts a loop to do it's main task, tunnelling TCP over SPUD. After receiving a new intercepted TCP packet from the netlink socket it extracts all necessary data from the IP and TCP Header. With those information it sets up a a new sockaddr_in structure for the external receiver of the SPUD packet. After identifying the TCP tuple it checks if there already exists a matching tube or creates a new one.Most difficult task here is to correctly handle the TCP FIN Sequence, as the program is symmetric but TCP's FIN sequence not.

The task responsible for handling incoming SPUD's is called receivespud. First it sets up an UDP socket listening on a specific port where it expects to receive its SPUD packets. After receiving a packet, the SPUD header and the SPUD data need to be extracted. To handle the received SPUD, the Function HandleReceivedPacket is called. The header is analysed for the CMD, and performs all necessary actions. SPUD itself is described in more details in the preceding chapter. As all SPUD Packets contain payload, there is always TCP Data that needs to be injected locally. Injecting TCP packet is done by writing an IP packets to a Raw socket. To do this, first a new IP Header needs to be created, this includes calculating it's checksum. Then the packet is ready to be injected locally.

Finally there is a Thread Status. It keeps track of open tubes, prints status updates in the console and removes tubes after they have been inactive for too long.

Threads are implemented using pthread and semaphores protect critical sections. In our case critical sections are all access to the list of open tubes as these is shared amongst all three threads.

3.3 How to run

This section gives a brief step by step introduction how to get the current implementation up and running. On each of the two endpoints

- root rights
- git

Open a Terminal, navigate to the folder where TCPtoSPUD should be store.
Execute:

git init && git pull https:github.comG-GFFDSemesterarbeitSPUD.git

In the current version, for debugging purposes, all intercepted TCP packets get send over SPUD to a fixed IP adress. Thus, you need to adapt those defined addresses in tcptospud.c and spud.c just at the top. Also in injectcp.c the parameters for the header of the injected packet needs to be configured.

Compile userspace part with:

gcc tcptospud.c spud.c tcphandling.c injectcp.c tubelist.c -o tcptospud -lpthread

Compile Kernel module with:
sudo make

Due to the limits of this implementation, traffic cn only be tunnelled in one direction. This problem is discussed in the last chapter under 'Conclusions'. Insert the Kernel module only at one end.

sudo insmod kerneltcp.ko

Now everything is ready to run the sofware at both endpoins

./tcptospud

To remove the kernel module again:

sudo rmmod kerneltcp

On some systems, the TCP maximum segment size may be to big and UDP packets containing SPUD start to get fragmented. This results in packets without SPUD header and the purpose of the software is broken. To make sure this does not happen or if you already have observed this behaviour e.g. in Wireshark, limit the TCP MSS with:

iptables -t mangle -A POSTROUTING -p tcp –tcp-flags SYN,RST SYN -o eth0 -j TCPMSS –set-mss 1460

This sets an iptables so that all SYN and RST SYN packets contain an option which limits the MSS to 1460 bytes.

The actual value may depend on the test setup.

3.4 Hints for debugging

A lot of time consuming debugging had to be done during this project and will very likely also be necessary in further continuation of this project.

Therefore a short list of some tools used for debugging is provided next.

Add the -g option to the compile command to enable debug information.

Find memory leaks on the heap with valgrind:

valgrind –leak-check=full –track-origins=yes ./tcptospud

Or debug it with gdb.

A lot of problems are not acutall C programming bugs, instead concern corrupted packets or distorted TCP streams.

To trace any injected packet and find out where it got lost iptables was very helpful.

All TCP connection of a machine can be shown with:

ss -t -all

This is also to check on which ports the TCP server is listening for incoming TCP.

And of course the well known tool Wireshark is irreplaceable to analyse & fix corrupted packets be it TCP or SPUD. It can also trace & compare complete TCP flows.

Chapter 4

Test Setup

4.1 Overview

During development and after finishing an application, testing is always very important. To do so, we need an appropriate test setup. In a first part, we need it for debugging purpose, later for evaluating the benefits of SPUD, the main purpose of our program.

4.2 Debugging tunnelling

To quickly test the tunnelling of existing TCP traffic by the program, a simple source of TCP traffic was needed.

Using two virtual machines, one acting as client, the other as a server this was simple done by generating HTTP requests. On both endpoints our program is running.

Now due to the restriction that intercepting and injecting TCP is currently not possible on the same machine, we can only tunnel TCP over SPUD in one direction while in the other direction the TCP runs normally. On the endpoint where we want to intercept the TCP, the Kernel Module needs to be active.

If Python is enabled on the system,

python -m SimpleHTTPServer

will create a simple HTTP Server, which is perfect for our purpose.

Intercepting and analysing the TCP flow can easily be done via Wireshark

Shown in the figure with the Wireshark screenshot is the problem that took well over a month to resolve. The injected packets, injected as Ethernet packet correctly arrived at the system and showed up fully correct in Wireshark. Everything matched the packets of the same connection, which wasn't intercepted and tunnelled over SPUD.

4.3 Evaluating the benefits of SPUD

Later, to test the effectiveness of SPUD in networks with middleboxes a direct connection won't do the job anymore.

A test setup with at least one middlebox in between is needed.

Initially it was planned to write an application for middleboxes as well, however this had to be abandoned due multiple issues and finally due to time problems. This needs to be done as next

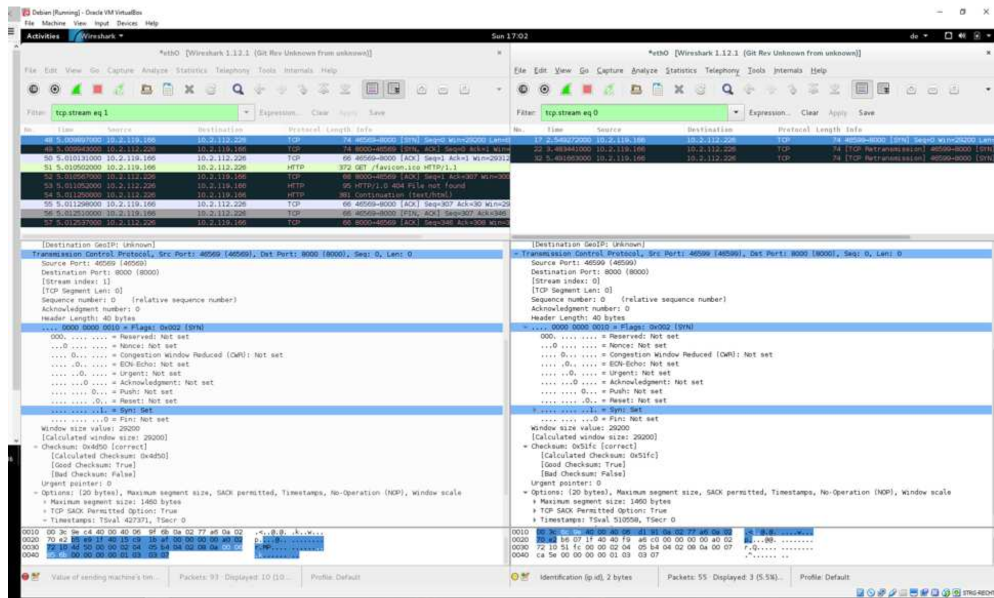

Figure 4.1: Wireshark analysis of the ignored SYN packet of the Injected TCP (right) vs a normal TCP Flow (left)

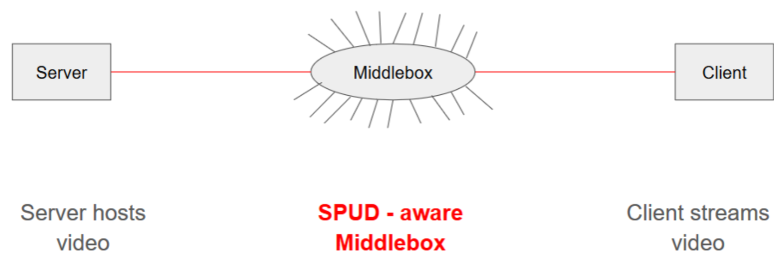

Figure 4.2: Proposed Test setup with Middlebox in between

task before evaluation of the protocol can take place.

However a lot of code and experience from the current implementation can be reused with only relatively small adoption needed.

Basically, the Kernel module here needs to intercept UDP packets and redirect them into userspace. This can simply be done by changing the protocol on which the netfilter Hook triggers. In the userspace application the packet is processed by the SPUD handling routine and re-injected accordingly.
This test setup can either be made "virtually" by having different virtual machines running with some acting as middleboxes and on the endhost the routing configured accordingly.

Another more visual way, to make it more real and less abstract, would be to make an raspberry pi acting as middlebox. One Endpoints just connects to the raspberry pi via wifi, the raspberry pi forwards all traffic to the LAN interface and vice-versa (this is called bridging).

Of course the raspberry pi acting as middlebox intercepts passing packets, detects SPUD and handles them according to defined rules.

IP forwarding needs to be activated, this is done by:

sysctl -w net.ipv4.ip_forward=1

Chapter 5

Conclusion and Outlook

5.1 Conclusion

The main task was to design and code a framework, that tunnels existing TCP traffic over SPUD. This should serve as a basis for further evaluation in a test environment to gain valuable feedback for further development of this proposed protocol.

Due to many unexpected complications things didn't move as smooth and took lots of time to resolve.

After 14 weeks it's time to look at the current state of the project:

Whats works:

- Intercepting TCP packets in the kernel
- Basic SPUD handling
- Tunnelig TCP over SPUD*
- Injecting TCP packets

However there are some restrictions, the most signigicant is it cannot tunnel a complete TCP stream of a HTTP connection, even though all steps individually with just one packet seem to work.

The main problem is that in the current implementation intercepting TCP packets and injecting TCP packets via a raw socket doesn't work together. Because of this, tunnelling actually only works in one direction until a solution is found. This appears to be a bit tricky. There exists other software doing this already, for example TCPcrypt. [1] TCPcrypt however is circumventing this problem by using the pcap API. This way the interface is not blocked an injecting packets succeeds. The big problem with this approach however is how to drop those intercepted TCP packets as pcap only supports sniffing. There's a quiet nasty hack required to solve this, so this approach was not further followed.

Additionally it must be made sure that injected packets don't get intercepted immediately again. Due to limits of the current implementation this case could not be studied yet. This may also prevents the current one-way tunnelling of TCP over SPUD to work correctly.
Currently there exists two branches of the sourcecode. Initially in addition to the TCP header and data the complete IP header was also encapsulated in the SPUD packet. This is the branch that almost works. However, the IP header should not be included instead it should be rebuilt at the endpoint. This version however currently has problems with corrupted TCP payloads and thus bad checksums.

[1] http://www.tcpcrypt.org/

5.2 Outlook

A first part may be complete but there is still a lot of work left in order to set up a complete working test environment. Only then relevant measurements can finally take place.

There is some debugging, extension, rework and fine tuning of this first part needed.

The main task of the next part should be focussed on the middlebox implementation. Of course some thinking must be spend on how to solve the problem with intercepting and injecting TCP packets simultaneously on the same endpoint and getting this first part to work completely.

With this, a first working pilot testbed should be possible. After adding a first signalling service the evaluation of SPUD can begin.

Limited evaluation may still be possible even if the problem of injecting/intercepting cannot be resolved, test cases would need to be planned accordingly.

Appart from this main points, there are lots of smaller points which need attention.

One problem is that currently opened SPUD tubes don't get acknowledged. This is due to two reasons:

1. Even though TCP and SPUD are bidirectional two separate SPUD tubes are opened for the connection, one from each endpoint as it intercepts the first TCP packet of the flow
2. A SPUD packet always contains data

Now due to the first point there is only SPUD traffic in one direction in each tube. The first packet in the opposite direction will have the ACK command flag set, however there simply is never a packet to be sent in this direction over this tube, as all packets in this direction are transmitted over the other tube. As consequence, in practice there is never an ACK to a SPUD tube.

This could cause problems during the testing phase when everything is supposed to work, if middlebox depend on seeing this ACK for a tube.

Other things that can be done in a further project:

- Expand to IPv6
- GUI for endpoints
- Filter for IPs that shouldn't be tunnelled
- Optimization of current code & debugging

Appendix A

List of Figures

List of Figures

1.1 Middleboxes & Internet, modified from Future Internet-Open Access Journal . . . 5

2.1 Encapsulation of a TCP packet in SPUD 8
2.2 Visualization of the SPUD header - taken from the Internet-Draft 8

3.1 Structure of implementation, tunnelling TCP over SPUD 11

4.1 Wireshark analysis of the ignored SYN packet of the Injected TCP (right) vs a normal TCP Flow (left) 16
4.2 Proposed Test setup with Middlebox in between 16

Appendix B

Declaration of Originality

Appendix C

Originial Problem Statement

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor/Master/Semester thesis for spring semester 2016

Cooperating with Middleboxes in the Internet

Thesis assigned to Brütsch Dominic [D-ITET] with SA-2016-45

While the Internet was designed as a simple, end-to-end network, this is not the case anymore. For every communication that is done over the Internet there are number of middleboxes on the path between the two endpoints; these are deployed for various reasons, to assist the endpoints sending traffic running through them to provide a better experience to the user. Examples of such middleboxes are network address translators to conserve scarce IPv4 addresses, carrier-grade NAT (CGN) to provide IPv4 connectivity to IPv6-only hosts, TCP performance enhancers for in mobile networks, firewalls to secure enterprise networks from unwanted traffic, or even transcoding to adapt content correctly to the the end user's device and network connection.

While this in-network functionality enriches today's Internet, it also limits innovation in new Internet services. Whether as a side-effect of their design or limits in their implementation, middleboxes may completely block or otherwise disadvantage new and unknown traffic. Especially new transport protocols such as QUIC, proposed by Google, or Multipath TCP, used in Apple's Siri application, are hard to deploy. If they do deploy, they need a lot additional engineering effort to work around existing middleboxes.

The key problem here is that middleboxes were never envisioned in the original Internet architecture, and as such, every function provided by a middlebox must in some way make implicit assumptions about the traffic passing through it and how that traffic can be safely manipulated. To address this problem in the future Internet, a new Substrate Protocol for User Datagrams (SPUD) [1,2] has been proposed in the Internet Engineering Task Force (IETF) to enable explicit cooperation with these middleboxes over a UDP-based encapsulation to improve incremental deployability.

In this project, the student will build a testbed for initial testing of the proposed protocol. An initial user-space implementation of a SPUD encapsulation protocol prototype is already available in Github[3]. In order to accelerate experimentation with this prototype, work will start with wrapping existing TCP traffic in a SPUD 'tunnel', so that explicit cooperation can be wrapped around any existing application. From this basis, the student will experiment with new cooperative in-network services in a testbed environment.

The following tasks need to be performed in this project based on the existing code:

1) Tunneling of TCP traffic over SPUD by redirecting outgoing traffic from the Linux kernel into a user-space application, which will wrap the TCP packets in the SPUD header, performing any additional protocol actions.

2) Implementation of at least one additional signaling service over SPUD, to signal application-relevant information explicitly to middleboxes to support low-latency transport, such as interactive cloud services, voice-over-IP (VoIP), or videoconferencing.

3) Setup of a pilot testbed to evaluate this implementation, including a forwarding node that provides a differentiated low latency services based on the information in the SPUD header of each packet.

Contact Person: Mirja Kühlewind, ETZ H93, +41 44 63 26932
Professor: Prof. Dr. Laurent Vanbever

Computer Engineering and
Networks Laboratory

© 2016 Laboratory TIK, ETH Zurich | Imprint | Last updated on Mon, 18 May, 2015 18:15 | t=0.0393s |

Appendix D

Bibliography

Bibliography

[1] Substrate Protocol for User Datagrams (SPUD) Prototype draft-hildebrand-spud-prototype-03,
J. Hildebrand, B. Trammell, March 09, 2015,
https://tools.ietf.org/html/draft-hildebrand-spud-prototype-03,
Last Visit: 13.07.2016

[2] Middleboxes: Taxonomy and Issues,
B. Carpenter, S. Brim, February 2002,
https://tools.ietf.org/html/rfc3234,
Last Visit: 13.07.2016

[3] Linux Kernel Networking: Implementation and Theory,
Rami Rosen, 2014,
Berkely, CA: Apress ISBN: 9781430261971,

[4] How SKB's work,
DaveM
http://vger.kernel.org/ davem/skb.html,
Last Visit: 18.07.2016

[5] The netfilter/iptables HOWTO's,
Harald Welte, Pablo Neira Ayuso
http://www.netfilter.org/documentation/index.html#documentation-howto,
Last Visit: 14.07.2016

[6] Creating a hello world kernel module in linux,
Paul Kiddie, Oktober 04, 2009
http://www.paulkiddie.com/2009/10/creating-a-hello-world-kernel-module-in-linux/,
Last Visit: 14.07.2016

[7] netlink example code in c,
arunk-s
https://gist.github.com/arunk-s/c897bb9d75a6c98733d6 Netlink,
Last Visit: 14.07.2016

[8] TCP-Packet-Injection,
unknown author
http://code.securitytube.net/TCP-Packet-Injection.c,
Last Visit: 14.07.2016

[9] The GNU C Library: Sockets,
Free Software Foundation, Inc.
http://www.gnu.org/software/libc/manual/html_node/Sockets.html ,
Last Visit: 14.07.2016

[10] CaptureSetup/Offloading,
Gerald Combs, September, 17,2009
https://wiki.wireshark.org/CaptureSetup/Offloading,
Last Visit: 14.07.2016

[11] The TCP/IP Checksum,
Lockless Inc.
http://locklessinc.com/articles/tcp_checksum/,
Last Visit: 14.07.2016

[12] Network Edge Intelligence for the Emerging Next-Generation Internet,
Jean-Charles Grégoire, Salekul Islam, 5 November 2010 *http://www.mdpi.com/1999-5903/2/4/60* Last Visit 15.07.2016

[13] C Language Examples of IPv4 and IPv6 Raw Sockets for Linux,
P. David Bucan
http://www.pdbuchan.com/rawsock/rawsock.html,
Last Visit: 14.07.2016