POLYTECHNIC OF TURIN - STOCKHOLM'S KTH

Faculty of Engineering

Master of Science in Computer Engineering

Master Thesis

MPTCP Security Evaluation

Analysing and fixing critical MPTCP vulnerabilities, contributing to the Linux kernel implementation of the protocol

Advisors:
prof. Antonio Lioy
prof. Peter Sjödin

Candidate:
Fabrizio Demaria

Company tutors
Intel Corporation Inc
Henrik Svensson
Joakim Nordell
Shujuan Chen

March 2016

Acknowledgements

Thanks to...

Summary

Abstract goes here...

Contents

Acknowledgements I

Summary II

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Benefits of Multipathing . . . 3
  - 1.1.2 Multipathing Solutions . . . 4
- 1.2 Problem Statement . . . 6
- 1.3 Methodology . . . 6

2 Multipath TCP 8
- 2.1 Transmission Control Protocol (TCP) . . . 8
- 2.2 MPTCP Design . . . 8
  - 2.2.1 Control Plane . . . 8
  - 2.2.2 Data Plane . . . 8
- 2.3 MPTCP Deployment . . . 9
  - 2.3.1 Middleboxes Compatibility . . . 9
  - 2.3.2 Implementations . . . 9
  - 2.3.3 Deployment Status . . . 9

3 MPTCP Security 10
- 3.1 Threats Analysis . . . 10
  - 3.1.1 Threats Classifications . . . 11
- 3.2 ADD_ADDR Attack . . . 12
  - 3.2.1 Concept . . . 13
  - 3.2.2 Procedure . . . 13
  - 3.2.3 Requirements . . . 15
- 3.3 Additional Threats . . . 16
  - 3.3.1 DoS Attack on MP_JOIN . . . 16
  - 3.3.2 Keys eavesdrop . . . 16
  - 3.3.3 SYN/ACK attack . . . 17

4 ADD_ADDR Attack Simulation 18
4.1 Environment Setup 18
4.2 Attack Script 21
4.3 Reproducing the Attack 23
4.4 Conclusions 24

5 Fixing ADD_ADDR 26
5.1 The ADD_ADDR2 format 26
5.2 Implementing ADD_ADDR2 26
5.2.1 Retro-compatibility 26
5.2.2 Port Advertisement 26
5.2.3 IPv6 Considerations 27
5.2.4 Crypto-API in MPTCP 27
5.3 RFC Contributions 27
5.4 Experimental Evaluation 27

6 Conclusions 28
6.1 Related Work 28
6.2 Future Work 28
6.3 Final Thoughts 28

A An appendix 29

Chapter 1

Introduction

1.1 Motivation

The last few decades have seen the most pronounced technology evolution in history, in many different areas and markets: from smartphones to robotics, from cars to medicine, etc. One of the pillars upon which all these advancements have been made possible is the Internet, or more generally the entire set of networking technologies that allow software to communicate.

The process towards interconnected devices saw a big leap forward in the early 1960s with the first research into packet switching as an alternative to the old circuit switching. But it is 1982 the year of standardization for TCP/IP protocol suite, which permitted the expansion of interconnected networks [wiki]. The Internet grew rapidly, passing from a few tens of million users in the 1990s to almost 3 billions users in 2014 [ref]. Even more astonishing is the number of networked devices and connections globally, around 14 billion in 2014 [ref]

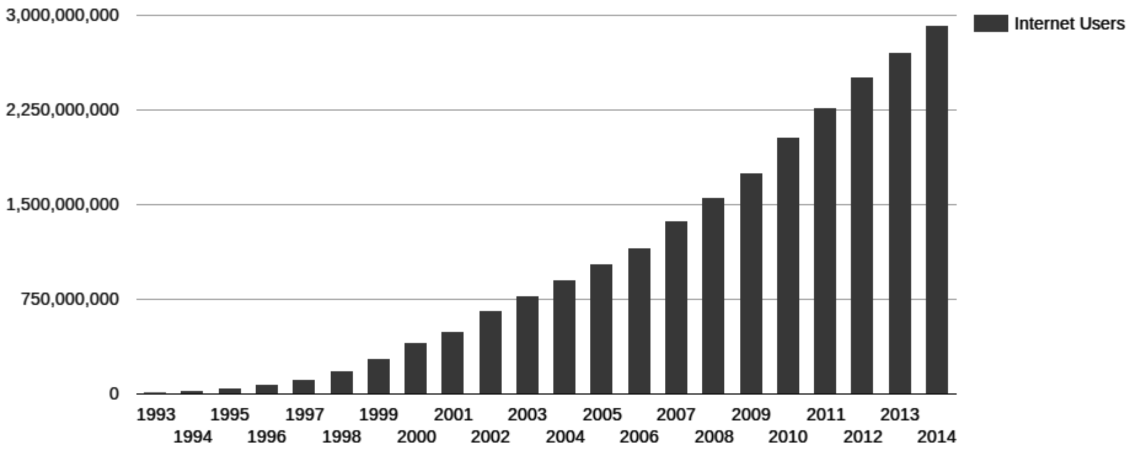

Figure 1.1: The expansion of the Internet

"Networks" is a very generic term. In the IT context it can refer to any set of nodes communicating over a certain protocol. The most widespread protocol for networking communication is the above-mentioned TCP/IP protocol, that is used in the vast majority of services like the World Wide Web, email, file transfer (for example FTP), remote system access, etc. It is also often used as a communication protocol in private networks and datacenters. The reason for its wide adoption is not because there aren't good alternatives: TCP/IP is not to most performing protocol in every network environment, but it is fairly simple and it introduces a relatively low complexity in the overall architecture, still guaranteeing all the basic connection requirements. Back in the 1980s, TCP/IP was the simpler way for applications to use most networks, thus quickly becoming a de-facto standard [ref]. During its life, the TCP/IP protocol suite have seen many updates and additional components to reach the desired levels of network congestion, traffic load balancing, handling of unpredictable behaviors, security, user-experience and so on. Such requirements became more and more strict with the uncontrollable expansion of the Internet. Albeit, after all these years the fundamental aspects of the TCP design design haven't been changed at all, mainly for retro-compatibility purposes. This inevitably causes some aspects of the protocol to look very limited in the current networking reality. A striking example is the scarcity of available IPv4 addresses: when TCP was designed in the 1970s, a 32 bits number reserved for addressing all hosts in the Internet seemed to be a very high number. Nevertheless, due to a misuse of address allocation policies and, mainly, due to the unexpected increase in the number of connected devices, the original IPv4 addresses available run out quickly, forcing the introduction of the lengthy 126bit bits address format, known as IPv6. Unfortunately, modifying the original TCP implementation to make use of the new address turned out to be a remarkably complicated procedure overall: to maintain retro-compatibility, IPv6 portion of TCP is implemented as a separate network stack component, and even if nowadays the majority of the systems are IPv6-compatible, it took 20 years to reach the current percentage of overall adoption: 10% [ref].

When the TCP protocol was first developed in the 1970s, it was certainly difficult to predict the rate of growth of the networks all around the globe, not only in terms of the number of nodes involved, but also in terms of the quantity and type of the transmitted data, the increasing need of low latency for new streaming applications, the advancement in the hardware adopted to carry the data and the computing power of the interconnected devices. Today we can count billions of interconnected devices, and we have just started the era of the IoT (Internet of Things) which aims at giving communication capabilities to virtually every object commonly used in our daily life. As a result of this process, the networks of today are becoming more intricate and devices often use multiple interfaces to stay connected. Even common appliances like smartphones provide both cellular connectivity and Wi-Fi modules; laptops have at least Wi-Fi capabilities plus an Ethernet port. The argumentation is much more complex in the back-end infrastructures' scenario, where we can find datacenters counting thousands of interconnected nodes and content-delivery servers that are capable of handling a huge number of network interfaces simultaneously. The implications of this new reality include the possibility of establishing multiple paths to transmit data between pairs of hosts, since they are now often equipped with multiple network interfaces, each with an active IP address. Back in 1970s TCP was designed to create a virtual connection between exactly two IP addresses and two port values, with almost no flexibility or dynamism in address/port addition and/or removal. In the multipath scenario typical of the infrastructures of today, to old point-to-point single-path connection provided by TCP looks quite limiting. This led to various projects aiming at exploiting the multipath concept, and MPTCP is one of them.

1.1.1 Benefits of Multipathing

Multipathing provides hosts with a resource pooling concept applied to networking access. Resource pooling allows dynamism and flexibility in requesting and handling resources and it has been a positive trend in many services and architectures, like CDNs, P2P networks, cloud computing, etc. The very concept of packet-switching is based on a resource pooling technique, where circuit utilization is not using isolated channels but the traffic in a circuit can make use of the spared capacity of other circuits [ref]. In the multipath context, "resource pooling" might assume slightly different meaning according to the specific implementation and the layer it operates on in the OSI architecture (more on this later). Nevertheless, a general set of benefits can be defined for adopting multipathing-aware technology. Such set is presented in the following list, from the point of view of MPTCP (which operates at the Transport Layer):

- Combining MPTCP multipath and multi-homing, it is possible to achieve higher throughput by exploiting multiple simultaneous connections to transfer different portions of the same piece of data. For example, a typical smartphone could use its cellular module and its Wi-Fi module simultaneously in downloading a file from a remote server, despite them having two different IP addresses.

- It is possible to introduce failure handover for the connection with no special mechanism at Network or Link Layer. If one of the interfaces goes down or the flow of data gets interrupted for any reason, data transfer can seamlessly continue through other interfaces;

- By assigning different priorities to the various interfaces, it would be possible to better handle data consumption (useful in case of a limited-capacity data-plan); for example, consider the case of a file download on a smartphone via 4G connectivity: it would be reasonable to switch the whole data transfer to the Wi-Fi interface if that becomes available in the middle of the download, starting from the point left by the cellular connection and without the need to restart the session;

- Providing multipath awareness to current network stacks could improve load balancing and exploitation of the network resources in datacenters; this is a valuable aspect, considering that the network performance in data centers is usually the bottleneck for the latency of the overall system. A similar concept applies for load balancing in ISPs network backbone.

1.1.2 Multipathing Solutions

MPTCP aims at achieving all the benefits mentioned in the previous paragraph. Before MPTCP, other proposals have been elaborated to achieve multipath benefits by introducing new technologies at the Link Layer, Network Layer and even Transport Layer (the same layer on which TCP operates). Even at the Application Layer developers can create custom protocols on top of TCP to achieve benefits similar to those that would come "naturally" by exploiting multipath natively at the lower layers. For example, most modern browsers open many TCP connection simultaneously to download the various elements of a Web-Page to improve user experience. Another example could be Skype and similar VOIP programs, which try to automatically reconnect hosts in case of problems with minimum impact on the user experience. Albeit all the solutions at the Application Layer are just clever workarounds on top of regular TCP and they fall only marginally in our argumentation regarding multipath.

The following list gives a general overview of the most important solutions other than MPTCP:

- At Link Layer there are link aggregation techniques to aggregate the capacities of different interfaces to the same switch [add a ref]. There are different modes to achieve resource pooling through link aggregation, but the basic concept is to setup multiple interfaces with the same IP address (and usually the same MAC address) so that the link aggregation is transparent to the higher-level applications and various algorithms can be used to redistribute the data packets to the various links. In order for this to work, proper configuration is needed at the host and at the next-hop switch. Despite being a common solution in ISPs' inner networks and datacenters to improve throughput and achieve higher network-access, end-users cannot directly take advantage of this technology.

- At the Network Layer there exist multiple solutions to better exploit multipathing, most notably Mobile IP and Shim6. They both provide hot-handover capabilities with no interruption of the higher-level services, with some limitations. Mobile IP requires extensive support by the underlying infrastructure and Shim6 is an IPv6 only solution. More importantly, there is a fundamental problem in introducing resource pooling at the Network Layer: TCP operates at the Transport Layer but it is closely related to the Network Layer because it statefully inspects various properties of the underlying network paths to provide performance optimizations; transparent modifications at the Network Layer would cause consistent TCP malfunctioning.

- At the Transport Layer the most notable experiment in multipath exploitation prior to MPTCP is the Stream Control Transmission Protocol (SCTP). Such protocol is, in many ways, similar to MPTCP: the first version of SCTP provided fail-over capabilities by exploiting different interfaces, and successive versions introduced multi-streaming capabilities to increase throughput via different interfaces. The major problem with SCTP is that it was thought to be an alternative enhanced version of

TCP, and the two protocol are indeed incompatible with each other. This means that a wide adoption of SCTP would require to upgrade all the middleboxes to be SCTP aware, in order not to discard unrecognized packets. Moreover, all the applications would need to be upgraded to explicitly use the new protocol to communicate. The vast global networking scenario of today, mainly based on TCP, makes these requirement virtually impossible to meet, and today SCTP remains a technology of very limited adoption.

All these previous solutions didn't get widespread adoption. Link Layers and Network Layers solutions require extensive modifications in the underlying network configurations in order to achieve the desired results; introducing a new multipath-aware protocol at the Transport Layer requires to change all the applications in order for them to communicate over the new protocol, thus allowing this solution in very limited scenarios; workarounds at the Application Layer, despite being quite effective, are far from the purpose of MPTCP.

Multipath TCP (MPTCP) is an ongoing project managed by the Internet Engineering Task Force (IETF), whose specifications have been published as Experimental standard in January 2013 [ref RFC 6824]; since then, the protocol has seen a lot of interest in the Internet community and many implementations quickly became available for the most common platforms. MPTCP primary goal is to automatically introduce the multipath benefits to current infrastructures and devices, with the minimum possible effort from users, developers, network maintainers. Engineers decided that the best way to achieve all these requirements was to still make use of TCP as fundamental block, exploiting its extensible TCP Options field to introduce all the custom options needed to handle multipath subflows: the entire protocol design works by adding MPTCP custom options into regular TCP packets, so that MPTCP-aware systems can process such options; if a system that is not MPTCP-aware receives a MPTCP connection-request, it would simply discard the MPTCP options and threat such packet as a plain TCP connection-request. All the subflows connecting the various interfaces in a MPTCP session would look like normal TCP connections to the lower infrastructure. Retro-compatibility is achieved with most of the current systems. For what regards applications, they don't need to be changed either since MPTCP would be inserted into the network stack at the operating system level, it would automatically split the data coming from the Application Layer and send it through different subflows, according to the number of available endpoints at the connected hosts.

MPTCP not only aims at providing all the benefits of multipath, but it is also designed to require the minimum possible effort in large-scale deployment. For these reasons, the new protocol got a lot of attention in the Internet community in the last few years, and many consider MPTCP as a valuable step forward for the whole global network currently relying on TCP. The final goal of MPTCP is to replace the majority of the current TCP implementations, which is a very delicate process in which all the current TCP standards in terms of robustness and security have to be maintained, if not improved. This paper is an evaluation of the security aspects of MPTCP, with an analysis of current threats and vulnerabilities affecting the protocol.

1.2 Problem Statement

MPTCP is a big effort from the IETF working group to unlock multipath networking capabilities worldwide, with many subtle implications for current infrastructures. Hence the importance of evaluating the current security status of MPTCP, both by inspecting its implications on external middleboxes and security equipment and by analyzing internal design flaws that might allow attacks to the MPTCP sessions. The main focus of this paper is on the latter case. The main vulnerability currently known for the protocol, related to the ADD_ADDR component, is tested and studied; the solution for it is implemented and evaluated. In the process, actual patches for the Linux kernel implementation of the protocol have been developed to fix the vulnerability.

A comprehensive list of all the objective for the thesis follows:

- Studying the security implications of adopting MPTCP on current infrastructures;
- Listing the known vulnerabilities affecting the current version of the protocol;
- Studying the ADD_ADDR vulnerability of the protocol;
- Evaluating the possible solutions for the ADD_ADDR vulnerability;
- Assessing the best solution for the ADD_ADDR vulnerability and developing it for the Linux kernel implementation of MPTCP;
- Developing effective and powerful simulation scenarios in order to test MPTCP (and possibly other networking protocols);
- Contributing to the upstreaming of MPTCP into the Linux kernel by developing patches and contributing to official RFC documentation.

1.3 Methodology

The thesis work has been carried out at the Intel office in Lund, Sweden. The process took 6 months in total, with a main focus on testing and developing. The entire work has been closely followed by major stakeholders in the MPTCP community, located in Sweden, Romania and the United States. Such cooperation involved patch reviewing and weekly meetings.

The work flow started with an overall study of MPTCP and how it interacts with the most common middleboxes. The next step was a more focused evaluation of the current threats for the protocol by using as reference the document RFC 7430. Within the document, only one vulnerability is considered a blocking issue in the advancement of MPTCP standardization, known as the ADD_ADDR vulnerability. The document also proposes a change in the protocol design that fixes the problem, and with such reference the actual development part of the work started.

At first, it was necessary to sync with the development status by interacting with the official MPTCP mailing list for developers [ref]; this allowed to make sure that the ADD_ADDR solution proposed in RFC 7430 was indeed the preferred one and that nobody started developing a patch for it already. Before starting to work on the fix, a first stage of the work involved a deeper analysis of the ADD_ADDR vulnerability. A connection hijacking has been executed by exploiting such vulnerability in a testing environment. This allowed to better validated the criticality of the problem and it was a useful experiment to get acquainted with MPTCP. Moreover, it was a good way to setup a proper testing environment that was indeed used during the whole patch-development process that followed.

After having reproduced the attack there was an analysis of the MPTCP modules within the Linux kernel in order to understand how the protocol implementation works inside the TCP stack. The development work that followed involved side features that came out to be necessary for a proper functioning, including some final testing.

Chapter 2

Multipath TCP

From now on the discussion becomes more technical. This chapter is all about how MPTCP works from a technical perspective. In this chapter there is no reference about the work carried out during the stage but only information taken from external documentation.

2.1 Transmission Control Protocol (TCP)

This is an explanation on how plain TCP protocol works. Even if there is no need to go into too much details here, this is a necessary starting point from where the MPTCP extension discussion can start (in the next section).

2.2 MPTCP Design

How MPTCP is added on top of TCP (with all the related design aspects) is reported here. This is the first portion of the thesis containing a more in-depth description of how MPTCP works in the networking stack. This specific section might follow closely the introductory portions of the RFC documents regarding MPTCP.

2.2.1 Control Plane

All the MPTCP options used to manage MPTCP sessions are reported and explained here, including all the details on how to set a new session and add/remove subflows.

2.2.2 Data Plane

This part concerns all the MPTCP options used to manage the data flow in a MPTCP session, including how the byte stream is subdivided into different subflow and how the original order of the packets is provided at the receiver.

2.3 MPTCP Deployment

After having explained all the technicalities about the protocol, it is now possible to talk about its deployment status and the problems encountered by pairing MPTCP with current infrastructures. This might seem a bit outside the scope of the thesis, but it is worth mentioning that the *deployment status* and *implementations* are a good indicator of how much MPTCP is important in the Internet community and they are good topics to motivate the thesis work. Also *middleboxes compatibility* is indeed a fundamental part of the MPTCP security evaluation, being monitoring and detection equipment part of these middleboxes.

2.3.1 Middleboxes Compatibility

This section will be quite technical and it is supposed to list the most important middleboxes and their impact/effect on a MPTCP connection. These boxes include NATs, proxies and firewalls. This part should clearly state why MPTCP widespread adoption is a big challenge.

2.3.2 Implementations

Despite the previously described problematics, MPTCP is a big bet in the IETF community and many implementations have been developed for the most common OSes, listed in this section (with some history notions).

2.3.3 Deployment Status

It should be interesting for the reader to go through some examples of real world's scenarios in which MPTCP is used successfully. Here it is possible to cite some important achievements related to MPTCP (for example the highest throughput ever reached with the new protocol). If available, it would be also good to show some graphs about MPTCP adoption rate or similar.

Chapter 3

MPTCP Security

3.1 Threats Analysis

A complete security evaluation of MPTCP can be subdivided into two main categories:

- A study of the vulnerabilities in the current MPTCP design that can be exploited to carry out flooding or hijacking attacks on an MPTCP session. This is an assessment on how consistently the MPTCP extension would impact the security standards of a plain TCP connection;
- A second perspective is to understand how the new protocol affects the functioning and behaviour of external security gears. This evaluation might include compatibility issues for middleboxes not yet aware of MPTCP [ref section of middleboxes] as well as more fundamental problematics related to security monitoring solutions that wouldn't work anymore with MPTCP: by splitting the logic flow of data into different paths, potentially belonging to different ISPs, it would be much harder to keep track of the content of the transmitted data over the networks. Moreover, the MPTCP ability to reroute traffic on the fly, adding and removing addresses and interfaces, would per se cause major problems with current intrusion detection and intrusion prevention equipment.

This paper focuses on the first point: MPTCP enables data transmission using multiple source-destination address pairs per endpoint and this generates *new* scenarios in which an attacker can exploit the way subflows are generated, maintained and destroyed to perform flooding or hijacking attacks. Flooding attacks are Denial-of-Service procedures that aim at overloading an MPTCP host with connection requests in order to quickly consume its resources. Hijacking attacks aim at taking total control of the MPTCP session, thus being considered the ultimate example of those threats falling in the *active attacks* category (more and this later on, in this section).

MPTCP security mechanism was designed with the primary goal of being at least as good as the one currently available for standard TCP [RFC6181]. The official MPTCP documentation and analysis reports don't cover common threats affecting both TCP and MPTCP, but only the vulnerabilities introduced by the new protocol alone. Nevertheless, it is of paramount importance that the various security mechanisms deployed as part of standard TCP, for example mitigation techniques for reset attacks, are still compatible with Multipath TCP.

Apart from the fundamental objective of keeping MPTCP at least as reliable and secure as TCP, official documents offer another set of requirements mainly related to securing subflow management in MPTCP [RFC6824bis]. These requirements are:

- Provide a mechanism to confirm that the parties in a subflow handshake are the same as in the original connection setup.
- Provide verification that the peer can receive traffic at a new address before using it as part of a connection.
- Provide replay protection, i.e., ensure that a request to add/remove a subflow is fresh.

MPTCP involves an extensive usage of hash-based handshake algorithms to achieve the required security specifications, as described in chapter 2.

Once the security requirements are clear, it follows a set of related problematics due to the way MPTCP is added to the normal TCP stack. Most notably, the entire behaviour of the protocol relies on the TCP Options field, which is of limited length of 40 bytes. This factor plays an important role in the definition of the security material to be exchanged during an MPTCP session (truncating the HMAC values and tokens is a common technique). Moreover, TCP Options field has been designed to accept any custom protocol extending TCP and for security reasons many middleboxes would discard or modify packets containing unknown options. As a last point, MPTCP approach to subflows' creation implies that a host cannot rely on other established subflows to support the addition of a new one [RFC6182-5.8]; this last requirement follows the *break-before-make* property of MPTCP, that must be able to react to a subflow failure a posteriori by establishing new subflows and automatically sending again the undelivered data. All these considerations define the fundamental boundaries and the context in which the security design of MPTCP has to be developed to meet the requirements.

3.1.1 Threats Classifications

Introducing the support of multiple addresses per endpoint in a single TCP connection does result in additional vulnerabilities compared to single-path TCP. These new vulnerabilities need proper investigation in order to determine which of them can be considered critical and might require modifications in the protocol design in order to meet the required specifications. In order to classify how critical each security threat is, it is a good starting point to define the various typologies of attack according to their requirements, rate of success and what power they can provide to the attacker.

The general requirements for an attack to be executed might be grouped into the following categories:

- *Off-path attacker*: the attacker does not need to be located in any of the paths of the MPTCP connection at any time in order to execute the attack;
- *Partial-time (time-shifted) on-path attacker*: the attacker has to be able to eavesdrop a specific set of information during the lifetime of the MPTCP connection in order to execute the attack. It doesn't need to eavesdrop the entire communication in between the hosts, and the specific direction and/or subflow for the sniffing procedure are attack specific;
- *On-path attacker*: this attacker has to be on at least one of the paths during the entire lifetime of the MPTCP session in order to execute the attack.

We can clearly state that the critical case concerns off-path attacks, which do not require any eavesdrop procedure in order to be executed. In fact, on-path attacks are not considered part of the MPTCP work, since they allows for a significant number of attacks on regular TCP already. A primary goal in the design of MPTCP is not to introduce new ways to perform off-path attacks or time-shifted attacks.

The effects of an attack over an MPTCP connection and the power that the attack can provide to the attacker can be divided into two main categories:

- *Passive attacker*: the attacker is able to capture some or all of the packets of the MPTCP session but it can't manipulate, drop or delay them, and it can't inject new packets in the current session either.
- *Active attacker*: the attacker can pretend to be someone else, introduce new messages, delete existing messages, substitute one message for another, replay old messages, interrupt a communication's channel, or alter stored information in a computer.

The rate of success of a certain attack over a MPTCP connection strongly depends on the specific requirements: two attacks falling in the same categories in terms of attacker eavesdrop capabilities and passive/active typologies might have rather different rates of success. For example, a certain kind of attack might require IP spoofing, thus being unfeasible in a network with ingress filtering [add reference]. There are no general thresholds to define when an attack can be considered a real threat according to the success rate, but this is an important factor to be studied in an attack analysis.

3.2 ADD_ADDR Attack

This paper is mainly focused on studying and testing the ADD_ADDR vulnerability of MPTCP, as well as providing an analysis of the commonly accepted fix and its implementation in Linux kernel. This section describes the attack procedure in details, while other minor residual threats are briefly reported in the next section.

3.2.1 Concept

The ADD_ADDR attack is an *off-path active attack* that exploits a major vulnerability in the MPTCP version 0 design [ref to RFC version 0]. As previously mentioned, the attacks falling into this category are usually the most critical and can easily jeopardize the protocol security requirements. With the current MPTCP model, an attacker can forge and inject an ADD_ADDR message into an MPTCP session to achieve a complete hijacking of the connection, placing itself as a man-in-the-middle. Being this an off-path attack, the attacker can *conceptually* send the forged ADD_ADDR message from anywhere in the network (as allowed by routing), with no need to be physically close to the victim machines. At the end of the attacking procedure, the attacker will be able to operate in any way on the ongoing data transmission, with no clear warning given to the original parties involved in the MPTCP session. If no protection system is used at the application layer (like data encryption), the attacker can eavesdrop all the information and even modify or generate the exchanged content. The attack vector enabled by such exploit is huge.

The culprit is indeed the format of the ADD_ADDR option, whose behaviour will be changed with the new ADD_ADDR2 for the very purpose of fixing this vulnerability [last chapter ref]. This vulnerability is entirely related to the MPTCP design, and due to its characteristics it is considered a blocking issue in the MPTCP progress towards Standard Track [7430].

3.2.2 Procedure

Let's consider a scenario in which two machines, host A and host B, are communicating over an MPTCP session involving one or more subflows. The attacker is called host C and it is operating remotely with no eavesdrop capabilities. The attacker is using address IPC and targeting a single MPTCP subflow between host A (address IPA and port PA) and host B (address IPB and port B). The scenario is reported in figure 3.1.

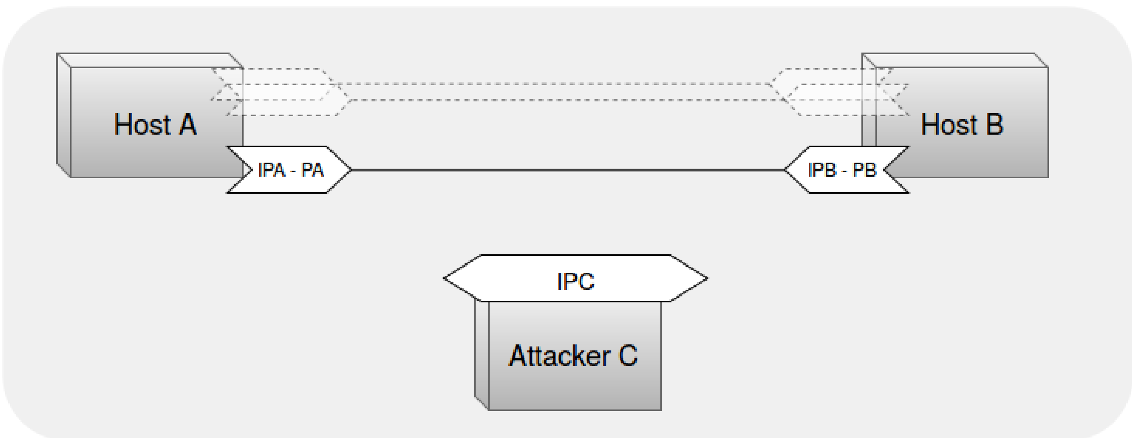

Figure 3.1: Attack scenario

Here is reported the step-by-step procedure to carry out the ADD_ADDR attack:

1. The first step performed by the attacker is to forge an ADD_ADDR message as follows: it is an ACK TCP packet with source address IPA, destination address IPB and the advertised address in the ADD_ADDR option is IPC. The ADD_ADDR option also contains the *Address ID* field, that the attacker can set to a number high enough in order not to collide with existing identifiers for the ongoing subflows between hosts A and B. The format of all the various MPTCP option can be found in chapter 2. The forged packet is then sent to host B.

2. Host B will process the forged packet as a legit request by host A of advertising a new available interface with address IPC. This most likely triggers the creation of a new subflow towards the new IP address, meaning that host B sends a SYN+MP_JOIN packet to the attacker. This packet contains all the security material needed in the first phase of the MP_JOIN three-way handshake, and the attacker does NOT need to operate over that portion of data: the attacker C simply manipulate the SYN+MP_JOIN packet by changing the source IP to IPC and the destination IP to IPA; then, it forwards such packet to host A.

3. Host A will process the incoming packet as a legit request by host B of starting a new subflow from host B's new available interface having address IPC. All the required information is present in the MP_JOIN option, like the token of host A that identifies the specific MPTCP session to which attach the new subflow to. Host A computes all the needed parameters (like a valid HMAC value), generate the SYN/ACK+MP_JOIN packet and send it to IPC. The attacker, similarly to the previous steps, manipulate the IP addresses of the packet from A by changing the source endpoint from IPA to IPC and the destination endpoint from IPC to IPB. At this point, attacker C sends the packet to host B.

4. All the parameters in the received packet looks correct to host B, which replies with an ACK+MP_JOIN packet to attacker C. The attacker changes the source address to IPC and the destination address to IPA and sends the modified packet to host A. Upon acknowledge reception, host A will verify all the parameters in the packet (which will be correct since properly calculated by host B), and create a new subflow towards IPC. At this point the attacker has managed to place itself as man-in-the-middle.

5. As a further, optional step, the attacker can send RST packets to the other subflow in order to close them thus being able to perform a full hijack of the MPTCP session between host A and host B. The attacker can now operate upon the connection in any possible way, modifying, delaying, dropping, forging packets between the two parties.

By exploit the ADD_ADDR option, the attack procedure is relatively straightforward. Albeit there are some important requirements and limitations that consistently limit the rate of success of such attack, which are discussed in the following section.

3.2.3 Requirements

A first, basic prerequisite needed by the attacker to inject the ADD_ADDR message into an ongoing MPTCP session is to know the IP addresses and port values adopted by host A and host B for the targeted subflow. It is reasonable to assume that the IP addresses are known. In a typical client-server configuration, the server's port for a certain application protocol is fixed and can be assumed to be known, too. For the client counterpart, the port value can cause problems in the presence of protection techniques like port randomization [ref]: in these cases the attacker has to start a guessing procedure whose rate of success also depends on the ephemeral port range employed [ref].

The knowledge about the above-mentioned four-tuple is a basic requirement for obvious reasons, but knowing the endpoint details is not enough to inject valid packets into an ongoing TCP session (that, in this case, can be also seen as an MPTCP subflow session): these packets have to contain SEQ and ACK sequence numbers that are compatible with the current ones within the stream. SEQ and ACK values are used in TCP to provide reliable, in-order transmission of data as well as services related to flow and congestion control [ref]. A very common protection technique is to randomize those 32-bit values at TCP connection setup, forcing the attacker (who acts off-path) to blindly guess them. TCP provides a window mechanism to deal with possible transmission's unalignments: at any given time, the accepted ACK values are those between the last ACK received and the same value plus the receiving window parameter. As a result, the number of packets to be sent in the attempt of guessing the right SEQ and ACK values and consequently the rate of success of the attack are strongly influenced by the TCP receive windows size at the targeted TCP host.

The requirements listed so far all pertain to the underlying TCP protocol. The only MPTCP specific parameter that can cause the failure of the ADD_ADDR attack procedure is the Address ID field in the option. The purpose of this value has been previously explained, and it doesn't actually offer an overall protection improvement. It is enough for the attacker to chose an ID value that is not in use by other subflow in the MPTCP session. In usual scenarios with a relatively limited number of subflow with the MPTCP session, applying a random value to this field should work just fine.

Moving away from the inner parameters evaluation and taking into consideration external protection mechanisms, it is worth mentioning that the attacker has to be able to manipulate and forge packets, including changing their source address field. This process, known as IP spoofing [ref], is a well known technique for which protection technologies have been developed, most notably the ingress filtering [2827] or source address validation [6056]. However, these methods are not vastly deployed and cannot be considered a sufficient mitigation for the ADD_ADDR vulnerability [ref on inngress filtering usage].

Lastly, the attacker has to be able to direct the malicious ADD_ADDR packet to a host that is actually capable of starting a new subflow, namely the client in a client-server model. The current Linux kernel implementation prohibits the server to instantiate a new subflow and only the client does so.

3.3 Additional Threats

In this section are presented the other residual threats under analysis by the IETF community at the time of writing. They all fall into two main kinds of attacks: flooding attacks and hijacking attacks.

3.3.1 DoS Attack on MP_JOIN

This kind of DoS attack would prevent hosts from creating new subflows. In order to be executed, the attacker has to know a valid token value of an existing MPTCP session. This 32-bit value can be eavesdropped or the attacker has to guess it.

This attack exploits the fact that a host B receiving a SYN+MP_JOIN message will create a state before answering with the SYN/ACK+MP_JOIN packet. This means that some resources will be consumed at the host to keep in memory information regarding this connection request from the other party; in this way, when the host B receives the third ACK+MP_JOIN packet, it can correctly associate it to the initial request and complete the handshake procedure. The creation of such state is required because there is no information in the ACK+MP_JOIN packet that links it to the first SYN+MP_JOIN request, so it is up to the host to remember all the ongoing requests. An attacker can exploit this by sending SYN+MP_JOIN packets to a host without providing the final acknowledge packets. This can be done until the attacked host runs out of available spots for initiating additional subflows. The initial number of such available spots depends on the implementation and configuration at the host machine.

This attack can be exploited to perform a typical TCP flooding attack. This is the perfect example of how MPTCP might introduce new vulnerabilities that might affect the underlying TCP protocol. SYN flooding attacks for TCP have been studied for many years and current implementations use mitigation techniques like SYN cookies [reference] in order to allow stateless connection initiations. But each SYN+MP_JOIN packet received at the host would trigger the creation of an associated state, while this is not the case for the attacker machine that can simply forge these packet in stateless manner. Exploiting this unbalance in resource utilisation is referred to as *amplification attack*.

A possible solution to this problem is to extend the MP_JOIN option format to include the information required to identify a specific request throughout the 3-way handshake, without requiring hosts to create associated states.

3.3.2 Keys eavesdrop

An attacker can obtain the keys exchanged at the beginning of the MPTCP session, exploiting the fact that those are sent in clear. This is in fact a partial-time on-path eavesdropper attack, whose success would enable a vast set of attacking scenarios, even if the attacker itself has moved away from the session after sniffing the aforementioned keys. The keys associated to an MPTCP session are sensitive pieces of information, used to identify a specific connection at the hosts and used as keying material for all the HMAC computations in the protocol. With such pieces of information an attacker can potentially execute a connection hijacking. This problem is encountered again when analysing the ADD_ADDR attack, Section 3.2.

Possible solutions have been proposed to protect the keys, but these are outside the scope of this paper.

3.3.3 SYN/ACK attack

This is a partial-time on-path active attack. An attacker that can intercept and alter the MP_JOIN packets is able to add any address it wants to the session. This is possible because there is no relation between the source addresses and the security material in the MP_JOIN packets. But securing the source address in MP_JOIN is not feasible if MPTCP is supposed to work through NATs: these middle-boxes operate exactly as described in this attack procedure.

Possible solutions have to reside on a different layer, perhaps securing the payload as a technique to limit the impact of such attack in a MPTCP session.

Chapter 4

ADD_ADDR Attack Simulation

4.1 Environment Setup

In order to achieve a reliable reproduction of a real world scenario, the simulation involves the setup of two User Mode Linux (UML) virtual machines running a Linux kernel with enabled support for MPTCP. These two machines act as client and server, carrying on an MPTCP connection that is the target for the ADD_ADDR attack. Using UML to proceed with the experiments allows for very fast setup and boot-up time, with good emulation of real machines and giving the possibility to work on a single hosting machine with no risk of damaging or crashing its underlying kernel.

A good resource in terms of tools, configuration files and kernel images is the official mptcp website: *http://www.multipath-tcp.org*. In particular, the website offers a python script that downloads all the necessary files to run the two virtual machines. Considering our purpose of verifying the ADD_ADDR attack feasibility, there is no need to modify or debug the Linux kernel source code, and the above mentioned components can be used out of the box. At this stage of the analysis it is actually advised to perform the attack on the official distribution as is, and develop external tools for injecting packets and monitoring the status of the connections. More specifically, the MPTCP version adopted for the tests is: *Stable release v0.89.0-rc*.

When executing the script *setup.py* retrieved from the official Website, a few files are downloaded. A *vmlinux* executable file with the MPTCP compatible Linux kernel, two file-systems for the client and the server (*fs_client* and *fs_server*) and two shell scripts to configure and run the virtual machines (*client.sh* and *server.sh*). No manual configuration is needed, and client and server should be able to connect via MPTCP right away. Here it follows the content of the *client.sh* (a similar shell script that is not reported here can be found for the server counterpart, including a single *tap2* interface setup in that case):

```
#!/bin/bash

USER=`whoami`

sudo tunctl -u $USER -t tap0
sudo tunctl -u $USER -t tap1
```

```

sudo ifconfig tap0 10.1.1.1 netmask 255.255.255.0 up
sudo ifconfig tap1 10.1.2.1 netmask 255.255.255.0 up

sudo sysctl net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/8 ! -d 10.0.0.0/8
    -j MASQUERADE

sudo chmod 666 /dev/net/tun

./vmlinux ubda=fs_client mem=256M umid=umlA eth0=tuntap,tap0
    eth1=tuntap,tap1

sudo tunctl -d tap0
sudo tunctl -d tap1

sudo iptables -t nat -D POSTROUTING -s 10.0.0.0/8 ! -d 10.0.0.0/8
    -j MASQUERADE
```

Listing 4.1: *client.sh*

These scripts call the *tunctl* command to create the tap interfaces and later assign an IP address to them by using *ifconfig*. A *tap* (namely network tap or tap interface) simulates a link layer device and it can be used to create a network bridge [wiki]. How taps are used in our simulation will become clear when observing the final network scenario. In order for the new tap interfaces to recognize each other and being able to send packets to each other it is necessary to enable the *ip forwarding* option using the corresponding *sysctl* command. It is also necessary to configure the *iptables* upon startup... The virtual machine is launched by executing the *vmlinux* file with some options to define various properties as well as attaching the newly created tap interfaces that will be used locally to sniff and inject packets (acting, in this specific case, as a physical man in the middle).

The resulting network scenario is graphically depicted in Figure 4.1.

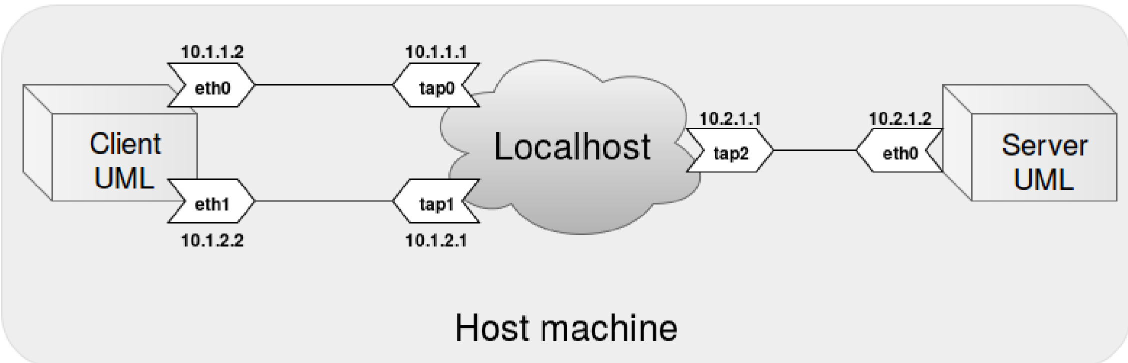

Figure 4.1: Network scenario

In order to carry out the ADD_ADDR attack it is necessary to inject forged packets into the existing MPTCP flow. In order to do this it is possible to use Scapy, a powerful interactive packet manipulation program that is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more [*http://www.secdev.org/projects/scapy*]. Moreover, there exists an unofficial version of Scapy that supports MPTCP and it can be found at the following repository: *https://github.com/nimai/mptcp-scapy*. The Python script that can be used to carry out the ADD_ADDR attack can be found here: *https://github.com/fabriziodemaria/MPTCP-Exploit*.

It is appropriate to mention here some of the limitations of the tool (that are examined more in details in Section 4.4: *Limitations and future work*): the tool has been designed to hijack a specific kind of communication involving client and server sending each others text messages using the tool *netcat*. It is very unlikely that the procedure completes with other kind of MPTCP connection setups between client and server. Nevertheless, this specific exploit serves well our purpose of assessing the danger and feasibility of the ADD_ADDR attack in general terms. Moreover, this tool simplify the attack procedure by sniffing the SEQ and ACK numbers of the ongoing connection instead of starting a procedure to try and guess the values. Also, the ports in use by the client and the server are retrieved automatically by inspecting the sniffed packets, while the IP addresses have to be provided by the user when launching the attack script. Further considerations about these simplifications can be found in Section 4.4.

The python module *test_add_address.py* in the root of the GitHub repository follows the analysis in RFC7430[] to perform the various steps necessary to hijack the MPTCP connection. All the requirements and theoretical details about this procedure have been reported in Section 3.2, and this section is limited to show and investigate the actual code implementation.

By establishing a testing connection via the *iperf* tool, two subflows are automatically generated by MPTCP, from the two interfaces of the client (ip addresses: *10.1.1.2* and *10.1.2.2*) and the single server's interface (with ip address: *10.2.1.2*).

In order to carry out the ADD_ADDR attack it is necessary to inject forged packets into the existing MPTCP flow. In order to do this it is possible to use Scapy, a powerful interactive packet manipulation program that is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more [*http://www.secdev.org/projects/scapy*]. Moreover, there exists an unofficial version of Scapy that supports MPTCP and it can be found at the following repository: *https://github.com/nimai/mptcp-scapy*. The Python script that can be used to carry out the ADD_ADDR attack can be found here: *https://github.com/fabriziodemaria/MPTCP-Exploit*.

It is appropriate to mention here some of the limitations of the tool (that are examined more in details in Section 4.4: *Limitations and future work*): the tool has been designed to hijack a specific kind of communication involving client and server sending each others text messages using the tool *netcat*. It is very unlikely that the procedure completes

with other kind of MPTCP connection setups between client and server. Nevertheless, this specific exploit serves well our purpose of assessing the danger and feasibility of the ADD_ADDR attack in general terms. Moreover, this tool simplify the attack procedure by sniffing the SEQ and ACK numbers of the ongoing connection instead of starting a procedure to try and guess the values. Also, the ports in use by the client and the server are retrieved automatically by inspecting the sniffed packets, while the IP addresses have to be provided by the user when launching the attack script. Further considerations about these simplifications can be found in Section 4.4.

The python module *test_add_address.py* in the root of the GitHub repository follows the analysis in RFC7430[] to perform the various steps necessary to hijack the MPTCP connection. All the requirements and theoretical details about this procedure have been reported in Section 3.2, and this section is limited to show and investigate the actual code implementation.

4.2 Attack Script

The very first step to be executed is the following: all the RST outgoing packets that can be generated by the hosting machine (the attacker) must be blocked during the process, when the first phases are completing and no finalized TCP connection can be actually detected by the system. This is done with the commands in Listing 4.2.

```
execCommand("sudo iptables -I OUTPUT -p tcp --tcp-flags ALL
    RST,ACK -j DROP", shell = True)
execCommand("sudo iptables -I OUTPUT -p tcp --tcp-flags ALL RST -j
    DROP", shell = True)
```

Listing 4.2: *Disable RST outgoing packets*

The Scapy built-in *sniff* function allows to retrieve packets from a specific interface, according to a custom filter function *filter_source* that inspects the source address. In this way it is possible to retrieve the IP addresses, ports, SEQ and ACK numbers of the ongoing connection between client and server. The first constructive step of the whole procedure consists in forging of the ADD_ADDR packet using the method *forge_addaddr* (Listing 4.3).

```
def forge_addaddr(myIP, srcIP, srcPort, dstIP, dstPort,
    sniffedSeq, sniffedAck):
     pkt = (IP(version=4L, src=srcIP, dst=dstIP)/
        TCP(sport=srcPort, dport=dstPort, flags="A",
        seq=sniffedSeq, ack=sniffedAck,
        options=[TCPOption_MP(mptcp=MPTCP_AddAddr(address_id=ADDRESS_ID,
        adv_addr=myIP))]))
     return pkt
```

Listing 4.3: *forge_addaddr method*

Here comes the first consideration about the script design: once the ADD_ADDR is sent to the victim client, the tool has to be already listening for the MP_JOIN sent back as a response; in order to make sure this happens, multithreading is used to start looking for the MP_JOIN packet even before ADD_ADDR is sent, with the thread named *SYNThread*. *SYNThread* just calls the method *get_MPTCP_syn* in the module *sniff_script.py*, that uses *tcpdump* with a specific filter option. In fact the Scapy *sniff* functionality proves to be unreliable in case of a high flow of packets to be processed and often skips some when the buffers reach their limits. Even if this is fine in other parts of the script where any packet capture is fine to retrieve ACK and SEQ numbers, it is mandatory not to miss the single MP_JOIN+SYN packet sent by the client upon ADD_ADDR reception. This problem concerning the sniffing function of Scapy is also reported in the official website under the section "Known bugs": *May miss packets under heavy load.* Note that this wouldn't be a problem with the slow message exchange of *netcat*, but the script can be also tested with high throughput applications like *iperf*, hence the usage of the more reliable *tcpdump*.

In order to filter out exactly the MP_JOIN packet we are looking for, the following command in Listing 4.4 is used, where *tf* is just a temporary file to store the information and *i* is the interface name passed as a parameter.

```
execCommand("sudo tcpdump -c 1 -w " + tf.name + ".cap -i " + i + "
    \"tcp[tcpflags] & tcp-syn != 0\" 2>/dev/null", shell = True)
```

Listing 4.4: *tcpdump for MP_JOIN*

A similar sniffing procedure is used for the next steps regarding SYN/ACK and ACK MP_JOIN packets, as it can be seen for the threads named *SYNACKThread* and *ACKThread*. Each time these sniffing threads are started, a sleep function is called for a time expressed in *THREAD_SYNC_TIME*, as a poor but effective mechanism that ensures that *tcpdump* is called and running in the new threads before proceeding.

The MP_JOIN packets generated and received in this way are manipulated to change the IP addresses and ports (and possibly other fields) as described in the attack procedure and then forwarded to the right host. Note that manipulating packet's fields in Scapy is different with respect to the case of ADD_ADDR where the packet is forged from scratch. All the functions *forge_ack*, *forge_synack* and *forge_syn* actually don't forge a new packet but slightly modify a copy of the received packet. While doing this it is necessary to eliminate the *checksum* value so that Scapy automatically recalculate the correct value for it, taking into consideration the updated values. Similar considerations hold for the Ethernet layer of the manipulated packets.

Once the ACK has been sent to the server, the new subflow is set up. In order to make it more visible, the next steps in the script enable again the outgoing RST packets and forge some of them to close all the subflows apart from the malicious one. By following the *print* messages in the script, this corresponds to *Phase 5*. Now, all the messages from the server to the client are sent to the attacker instead, without an explicit way for the victim to notice.

The very last portion of the script runs the method *handle_payload* that both prints the text messages (payload) received from the server and generate *data_ack* packets for the server in order to keep the connection alive.

The final tool and a step-by-step guide on how to use it can be found in the repository: *https://github.com/fabriziodemaria/MPTCP-Exploit*.

4.3 Reproducing the Attack

1. Open two terminal windows and run the client.sh and server.sh scripts to launch the UML virtual machines (user/password: root)

2. On the server machine, run the following (you can use a TCP Port of your choice here):

   ```
   netcat -l -p 33443
   ```

3. On the client machine, we first need to disable one of the two network interfaces, namely eth1. This is necessary due to some limitations currently affecting the Scapy tool and the attacking script (the connection will still be MPTCP, with a single subflow):

   ```
   ifdown eth1
   ```

4. Now you can run netcat on the client, too:

   ```
   netcat 10.2.1.2 33443
   ```

5. Try to exchange messages between client and server to verify that communication is active.

6. Now we can start the attack opening a new terminal on our local machine (it is necessary to start the Scapy script AFTER having established the netcat connection).

7. Go to the folder were you downloaded the Scapy tool and type the following:

   ```
   sudo python test\_add\_address.py 10.1.1.1 \
   10.2.1.2 10.1.1.2 tap2 tap0
   ```

   NOTE: If an import error appears, try to install the missing dependencies with:

   ```
   sudo apt-get install python-netaddr
   ```

8. Go back to the client UML terminal and start sending messages to the server. You should notice that while the messages exchange goes on, the attacking script progresses.

   IMPORTANT: it might be that the script gets stuck (it shouldn't take more than a few seconds to complete). If that is the case, close netcat and start again from step 2.

9. If you reach 100% in the attack process, just try to send a message from the server to the client and you will notice that the messages are now sent to the attacking machine instead. Further improvements would allow to also answer back to the server, thus impersonating the client.

4.4 Conclusions

The Scapy tool developed for this research targets a specific scenario to exploit the ADD_ADDR vulnerability. It is not intended to be general enough to break all the existing MPTCP implementations. Nevertheless, by succeeding in this specific case involving a *netcat* communication between two hosts, it is indeed proved the feasibility and gravity of the problem, and it should be relatively easy to extend the portability of the attacking tool to act in new scenarios.

This section mainly investigates the workarounds used to simplify the attacking process, to prove that they are not critical enough to devalue the results of the tool itself.

All the requirements for the succeeding of the attack have been already listed in Section 3.2. Here is reported a short summary:

- the four-tuple: IP and port for both source and destination;
- valid ACK/SEQ numbers for the targeted subflow;
- valid address identifier for the malicious IP address used to hijack the connection;

Regarding the last point, the Address ID chosen for the new subflow initiated by the attacker must be different from all the other IDs already used by the other subflows. It is fairly easy to choose a value quite high that has very low probability of being in use already. This value is set to 6 by default in the attack Scapy script.

It is a fair assumption that the four-tuples identifying the connection endpoints are known by the attacker[RFC7430], apart from the client side port value: in that case the difficulty in guessing the right port in use very much depends on the port randomization technique deployed at the client host [RFC6056]. Since it is anyway possible to guess the port, it is a fair simplification to simply provide it to the application in our tests: for this reason the tool has been designed to accept the IP addresses as arguments and automatically gets the ports in use to increase the rate of success in different testing scenarios, without the need for the user to provide that kind of information.

Guessing the SEQ and ACK numbers is by far more complex. Again, all the considerations about this have been reported in previous sections: it is possible to generate a big number of packets trying to guess the acceptable values for packet injection. This is out of scope in this research, so it is acceptable to simplify the attack by providing the SEQ and ACK values (by sniffing them from the ongoing connection).

It is important to emphasize that despite these workarounds, that require to act as a physical man-in-the-middle, no other information apart from the IP addresses, ports, SEQ and ACK values have been retrieved using Scapy's sniff or tcpdump, and no packet originally sent to the trusted hosts have been discarded or modified. All the sniffed values can be guessed and, despite the reduced chance of success, the exploit could be executed via a 100% off-path attack. That is why this is considered a major vulnerability for MPTCP deployment as of RFC7430 indications. In the next sections the solution to this problem and its Linux kernel implementation are discussed in the details.

Chapter 5

Fixing ADD_ADDR

This chapter is related to the second and most important part of the work performed during the Master Thesis work at Intel. The ADD_ADDR2 option is developed and added to the current MPTCP implementation for the Linux kernel in order to fix the vulnerability.

5.1 The ADD_ADDR2 format

The actual new format and the reasons why it fixes the vulnerability of ADD_ADDR are reported here. Various discussions can be added to explain why this is believed to be the best way to fix the problem. This part doesn't yet include any implementation details and/or code snippets.

5.2 Implementing ADD_ADDR2

An introductory section that shows the main architectural aspects of how MPTCP has been merged into the TCP code and the TCP modules inside the kernel. Here it starts the part with all the details about the implementation of ADD_ADDR2 in the kernel, as part of the work developed during the stage. Code snippets have to be added here. The following subsections are the side issues and side features that have been elaborated during the thesis work.

5.2.1 Retro-compatibility

Version control mechanism was not present but it is needed to negotiate which format to use in a MPTCP session: ADD_ADDR or ADD_ADDR2.

5.2.2 Port Advertisement

Port advertisement in ADD_ADDR is possible according to RFC specifications but it was not part of the implementation at the beginning of the thesis work, so it has been added.

5.2.3 IPv6 Considerations

Longer addresses brought some issues related to TCP option fields limitations.

5.2.4 Crypto-API in MPTCP

A major problem was how to deal with the new hashing requirements introduced by ADD_ADDR2. Extending the current MPTCP hashing function to deal with input messages of arbitrary size is a first point to explain. The second part has to deal with the whole analysis related to adopting the kernel CRYPTO APIs to calculate the HMAC values in MPTCP and why this is not advisable.

5.3 RFC Contributions

Another minor part of the thesis work on MPTCP is related to some small contributions to the official RFC documentation.

5.4 Experimental Evaluation

This part should include performance analysis regarding the new format introduced with ADD_ADDR2. A discussion on how the new format (and all the other modifications introduced with the patches) could impact any aspect of the protocol should be present in this section.

Chapter 6

Conclusions

6.1 Related Work

References to all the related work, including all the efforts to make MPTCP secure and stable, can be reported here.

6.2 Future Work

Here goes the list of the next steps to be taken care of in terms of MPTCP security, in order to facilitate the protocol's upstream and its widespread deployment. This section can also include a more specific discussion on the aspects to be still analysed regarding the new format ADD_ADDR2.

6.3 Final Thoughts

Some final conclusion.

Appendix A

An appendix

Appendix content goes here...