Paper ID: 419

An Alternative Approach to Authenticate Subflows of Multipath Transmission Control Protocol using an Application Level Key

Tharindu Wijethilake[#], Kasun Gunawardana, Chamath Keppitiyagama and Kasun de Zoyza

University of Colombo School of Computing, Sri Lanka

[#]tnb@ucsc.cmb.ac.lk

Abstract: Multipath Transmission Control Protocol (MPTCP) is an extension to Transmission Control Protocol (TCP) proposed by the Internet Engineering Task Force (IETF). The intention of MPTCP was to use multiple network interfaces in a single network connection simultaneously. Researches have identified that there are a considerable amount of security threats related to the connections initiated by MPTCP. In this research, we studied on the security threats generated by sharing authentication keys in the initial handshake of the MPTCP in plain text format and investigated the applicability of external keys in authenticating sub-flows with minimum modifications to the kernel and the socket APIs. To pass external keys from user space to kernel space, we used sin_zero padding in TCP socket data structure. Through the experiments we found that MPTCP sub-flows can be authenticated and certain vulnerabilities can be avoided with our approach.

Key Words: MPTCP, Computer networks, Linux kernel, Authentication keys

Introduction

TCP, the Transmission control protocol is one of the major protocols in the transport layer which was introduced in 1981 (Postel, 1981). The main objective of the TCP was to achieve the reliability of the communication channel between two hosts over a packet switching network. With the advancement of the technology, most of the modern devices such as laptops, mobile phones, and tablet PCs are having more than one network interface, such that Ethernet port, WiFi, cellular data connection like 4G/LTE and so on. However, most of the time these devices use only one network interface at any given time, and hence, researchers investigated the plausibility of employing the second network interface in order to increase the throughput and to provide redundant connectivity. To achieve this, an extension to classical TCP was introduced as Multipath TCP (MPTCP) in 2013 (Ford, Raiciu, Handley and Bonaventure, 2013).

A. Multipath TCP

Currently, Multipath TCP kernel is available for Linux operating systems, macOS, Android, and Apple iOS which can be installed separately. According to our knowledge, only Apple iOS has implemented MPTCP on their Siri voice assistant application (Bonaventure, 2014). Multipath TCP uses the normal TCP threeway handshake method to create the connections between two hosts. It does not change the currently available TCP protocol stack and the header format. All the data related to MPTCP are sent by using the TCP “option” field available in the TCP header.

To initiate the MPTCP connection between a client and the server, the client sends the normal TCP SYN message with the MP_CAPABLE options included in the TCP header. If the server is also configured with MPTCP, it will reply to the client using SYN/ACK with MP_CAPABLE. And finally, the connection is established with the ACK message from the client. In MPTCP these connections are known as subflows. To initiate an additional subflow, it has to send another SYN packet to the server with MP_JOIN option from the second network interface as shown in Figure 1.

When sending the SYN+MP_CAPABLE message at the beginning, the client sends a key to the server in plain text, as the key of the client. The server also sends a key with the SYN/ACK+MP_CAPABLE message in plain text as the key of the server. Finally, with the ACK message, the client sends both the keys to the server to confirm the connection. These shared keys are used to generate the HMAC, which will be later used to authenticate the new sub-flows that would be initialized between the two nodes (Demaria, 2016). In any case, if one of the hosts is not configured with MPTCP, it will automatically change into the normal TCP connection. So MPTCP is designed to be backward compatible and independent from the applications which are being executed on the server.

If a client needs to create a new sub-flow with the server, it will send a TCP SYN message to the server with the MP_JOIN option using the client's second interface. In this case, the client sends a token to the server to authenticate itself. This token is a part of the HMAC generated by using the keys shared in the initial key exchange. After sharing the HMACs of keys between the client and the server, MPTCP will create a new sub-flow between them as shown in Figure 1. Other than that there is an option called ADD_ADDR in MPTCP which can be used to advertise the available interfaces of a host to other hosts.

B. The Goal

There are several security vulnerabilities related to MPTCP connections. One of the major vulnerabilities is exchanging the key in plain text. A number of solutions has been proposed by IEFT for this problem. Some of the solutions were developed based on the ideas proposed in the RFC7430 (Bagnulo et al., 2015) and some are developed by combining available security protocols.

Other than that, Paasch and Bonaventure, 2013 have proposed a different approach where it uses external keys in authenticating sub-flows. However, to implement that, the existing socket API has to be modified. Changing the existing socket API must be a prudent and meticulous effort as it requires a comprehensive restructuring of the current implementation of the TCP. Therefore, in this study, our goal is to explore an alternative approach to use external keys in authenticating subflows without modifying the existing socket API.

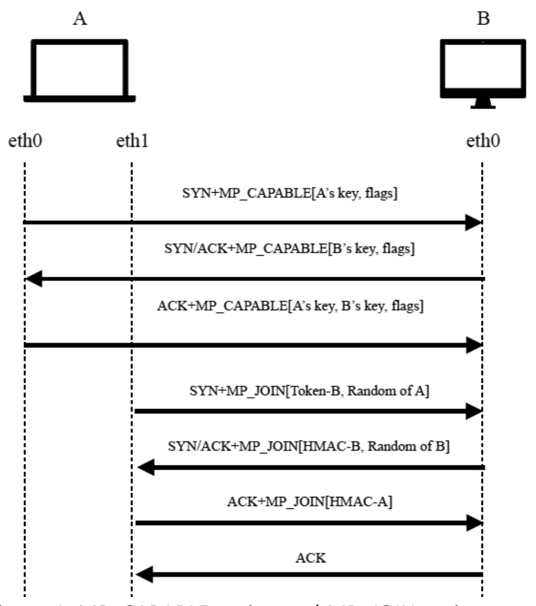

A
B
eth0
eth1
eth0
SYN+MP_CAPABLE[A's key, flags]
SYN/ACK+MP_CAPABLE[B's key, flags]
ACK+MP_CAPABLE[A's key, B's key, flags]
SYN+MP_JOIN[Token-B, Random of A]
SYN/ACK+MP_JOIN[HMAC-B, Random of B]
ACK+MP_JOIN[HMAC-A]
ACK

Figure 1: MP_CAPABLE option and MP_JOIN option

Background and Literature Review

Two of the key benefits of MPTCP mentioned in the RFC 6182 (Ford et al., 2011) are, to increase the ability to recover the connectivity in a connection failure without failing the end hosts by using multiple paths and to increase the efficiency of the connections by using multiple paths.

Creating multiple sub-flows between two hosts requires the authentication of one host to the other. As mentioned in the section I, the authentication mechanism (Ford, Raiciu, Handley and Bonaventure, 2013) employs plain text key exchange between two hosts over a public network, which opens to many security risks. If an attacker got access to these keys, the attacker can create a new sub-flow with the server and even can remove the connection between with the legitimate client and the server (Bagnulo, 2011).

A. Security Threats

The attackers in the context of MPTCP can be categorized based on their location and their actions (Bagnulo et al., 2015). The attacker based on their location can be classify into three. Those are, off the path attackers, p artial time on path attackers and on path attackers. Off path attacker is an attacker who is not in the middle of the path of MPTCP connection. Therefore, an off path attacker cannot eavesdrop the packets exchanged in the connection. The second group of attackers is the partial time on path attackers, who have access to the MPTCP connection, but not for the entire period of the connection. The third attacker is, on path attackers, who are on the MPTCP connection itself. That means they have access to one of the subflows of the connection (Bagnulo et al., 2015).

The attackers based on their actions can be classified in to two, which are eavesdropper and active attackers. Eavesdroppers collect data from the connection while the active attackers try to change the data on the connection (Bagnulo et al., 2015).

Bagnulo et al., (2015), have explicitly identified several security threats on MPTCP connections. The first security threat is ADD_ADDER attack which is a man in the middle attack where the attacker can hijack the MPTCP session. The next attack is the DoS attack on MP_JOIN which the attacker sends SYN+MP_JOIN packets to a host with a valid token, then the host will open a connection. There is a maximum number of half open connections can be maintained by a host according to the implementation. When that number is exceeded, the host becomes exhausted.

SYN flooding amplification is a denial of service attack (Eddy, 2017). The attackers send several SYN packets to a port and this make a number of half open connections which will eventually exhaust the connection.

Eavesdropper in initial key exchange is one of the main security issues in MPTCP. In this attack, the attacker collects the keys by listening to the initial key exchange and after that, the attacker can create new subflows using the captured keys (Bagnulo, 2011).

B. Current Status

For the identified security threats, several high-level solutions have been proposed in RFC 7430 (Bagnulo et al., 2015). Some of the researches based on these are as below.

1) Asymmetric key exchange: Without using plain text keys as in the original MPTCP, Kim and Choi, (2016) have proposed to use the Elliptic curve Deffie-Helman key exchange (Blake-Wilson et al., 2006) in the initial key exchange of MPTCP.

2) MPTCPsec: MPTCP secure (MPTCPsec) was proposed to satisfy two main objectives, which are detecting and recovering from packet injection attacks and to protect application level data (Jadin, Tihon, Pereira and Bonaventure, 2017).

3) ADD_ADDR2: To overcome the vulnerability in ADD_ADDR, the ADD_ADDR2 option (Demaria, 2016) was proposed.

4) Using external keys to secure MPTCP: Exchanging keys in plain text is one of the main security issues in MPTCP. One of the solutions was proposed for this problem was to use external keys such as SSL or TLS keys to authenticate the MPTCP connection. These SSL or TLS keys are already negotiated in the application layer. The proposed solution (Paasch and Bonaventure, 2013) has suggested a mechanism to transfer the

application layer keys to the MPTCP layer by using two socket options. One is MPTCP_ENABLE_APP_KEY, which is used to inform the MPTCP protocol that the application level keys are used to authenticate the connection, and MPTCP_KEY is used to provide the application level key to the MPTCP layer.

Apart from the aforementioned research work, several other attempts are also available in literature. Using hash chains (Díez, Bagnulo, Valera and Vidal, 2020), using SSL (Paasch and Bonaventure, 2013) and tcpcrypt (Bittau et al., 2018) are some of such work.

The research work presented in this paper was inspired by the idea suggested by Paasch and Bonaventure, (2013). In their proposal, they have suggested employing external keys to authenticate the subflows of MPTCP such that external keys are taken from the application layer and transferred to the kernel level. In order to achieve that, they have suggested to include two new socket options, in turn modifies the existing socket API. In this research, our goal is to explore an alternative mechanism to achieve the same objective as (Paasch and Bonaventure, 2013) without altering the existing socket API.

Methodology

As explained in section II, our goal is to obtain key information from the application level and deliver it to the kernel level without introducing new socket options. Theses key information is used to authenticate the subflows generated by the MPTCP protocol. In the first sub-flow, MPTCP uses MP_CAPABLE option to check whether both the hosts are compatible with MPTCP and share the keys which are need to authenticate the next subflows. From the second subflow onwards MPTCP uses MP_JOIN option to authenticate and join the new subflow to the same connection.

In this work, there are two assumptions to be made as given below.

- Both the hosts have to be agreed on the external keys before initiating the second subflow.
- The external keys have to be secure.

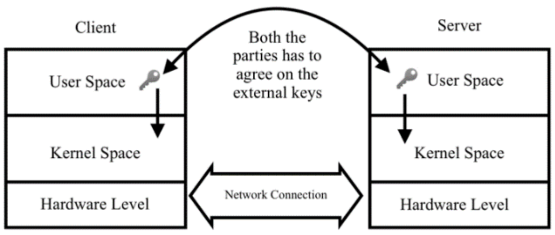


Figure 2. Kernel level, User level and External keys

Figure 2 shows the abstract picture of the solution proposed by this research. The external key means the shared secret between two hosts which was obtained from the application level. As shown in Figure 2, this key has to be transferred from the application level to the kernel level. After that, the key has to be used in the kernel level to authenticate the subflows. When the external key is available in the kernel level, it can be used to generate the authenticating material. The authenticating material has to be transferred to the server end for the authentication of the newly generated subflow.

A. Transfer application level information to kernel level

To transfer application level key to the kernel level, several avenues were explored.

1) Using the proc file system: Proc is a pseudo file system in the Linux operating systems that can be accessed from /proc (proc(5) - Linux manual page, 2020). This is an interface to the kernel data structure and most of the files in the proc directory are read-only. Some of them are writable and can be used to modify kernel variables. With this approach, a new proc directory has to be created in the /proc directory and the key value has to be written in the newly created proc directory. Then this value has to be

accessed by the kernel file. There were several overheads were identified when incorporating proc file system to transfer application level information to the kernel level. One of the issues is, the key value from the application level has to be written to a file in a /proc directly before invoking a TCP socket. That is an overhead for the normal procedure of invoking TCP socket. The other issue is, the value in the proc file has to be read by the kernel using a function and then assigned to the kernel variable. It cannot be directly assigning to the kernel variable value by the proc file system itself. Therefore when the kernel initiating a TCP connection, it has to read proc files and assign the values to the kernel variables.

2) Netlink Sockets: Netlink (netlink(7) - Linux manual page, 2020) is a Linux kernel interface which can be used to communicate between kernel space and the user space, and between different user processes also.

3) Using sin_zero of TCP socket : sin_zero is a char array in the sockaddr in data structure used in TCP sockets. This data structure contains the necessary information to create a TCP connection between two hosts. Protocol, port number, and address are some of the information contain in the sockaddr in data structure. Other than that there is another char array called sin_zero which is used as padding (struct sockaddr_in, struct in_addr, 2020). This space is not used by the sockets when creating connections. Therefore, theoretically, this space can be used to transfer data from user space to the kernel space, if it is not dropped when the information is transferred from user space to kernel space. This was further explored to identify the behavior of the sin_zero variable and tracked the functions which transfer the data from user space to kernel space. Compared to other solutions, using the sin_zero easier to send data from the application level. Char value can be easily copied to the sin_zero character array when creating the TCP socket. Therefore no need to customize the socket APIs. But the challenge was to retrieve the data from the kernel level. Theoretically, the sin_zero data should be received by the kernel space, if it was not dropped by the system calls. We have used inet functions of af_inet.c to retrieve the data from the sin_zero in the kernel kevel. Implementation was straight forward when using the inet functions of af_inet.c. Therefore the necessary functions were identified and modified to retrieve the data from sin_zero. Key value send from the application level was directly assigned to a kernel variable with this method.

B. Backward Compatibility

The backward compatibility is one of the important features in MPTCP. This means if the host machines are not compatible with MPTCP, it will automatically change into the original TCP connection. Therefore these solutions also should be backward compatible. This means if any of the machines are not configured with the proposed solution, it should use the normal MPTCP authentication mechanism.

To achieve this requirement, slight modifications to the code has to be done. It has to check whether the sin zero value is set from the user level or not. If the value is set, it has to use to the proposed solution, and if not it has to use the original MPTCP authentication mechanism.

C. Key Transferring Mechanism

The userspace key was transferred to the kernel space by using the sin_zero character array of the sockaddr in data structure of TCP socket and the data was obtained by the kernel space using inet functions of at inet.c with minimum modifications to the existing kernel implementations. Figure 3 shows the key transferring mechanism.

Evaluation and Results

For the evaluation, the modified MPTCP kernel was installed on Ubuntu 16.04 LTS and several experiments were done. All these experiments were conducted in virtual environment.

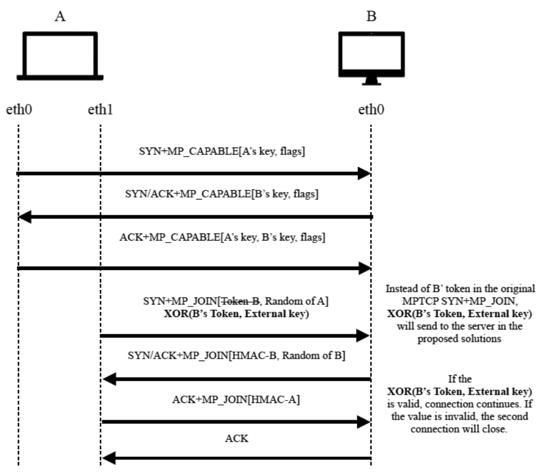

A
B
eth0
eth1
eth0
SYN+MP_CAPABLE[A's key, flags]
SYN/ACK+MP_CAPABLE[B's key, flags]
ACK+MP_CAPABLE[A's key, B's key, flags]
SYN+MP_JOIN[~~Token-B~~, Random of A]
XOR(B's Token, External key)
Instead of B' token in the original MPTCP SYN+MP_JOIN, XOR(B's Token, External key) will send to the server in the proposed solutions
SYN/ACK+MP_JOIN[HMAC-B, Random of B]
ACK+MP_JOIN[HMAC-A]
If the XOR(B's Token, External key) is valid, connection continues. If the value is invalid, the second connection will close.
ACK

Figure 3: Key Transferring Mechanism

A. Experimental Setup

TCP client and server sockets were implemented using the C programming language and executed on two virtual machines connected via the virtual network of Virtual box. The server is sending a string of data to the client and the client display that information in the terminal. Network packets were captured in the server machine and were analyzed using Wireshark.

In order to prove the concept of authenticating newly generated subflows using external keys, the authentication material was generated by XORing the token value generated by MPTCP and the external key that has obtained from the application level. This authentication material was sent to the server using the available token space in the SYN+MP_JOIN packet and the authentication material was validated in the server end.

B. Evaluation

To evaluate the proposed solution three main experiments were designed as below.

In the first experiment, it has to check whether the connection establishes when both ends use the same key. In this case, latter sub-flows should be authenticated, and data should be transmitted through all the sub-flows.

In the second experiment, it has to use different keys in both ends and check whether latter sub-flows establish or not. Since the keys are not identical, in this experiment, latter sub-flows should not be authenticated. Hence, it is expected to have only the first sub-flow of the connection.

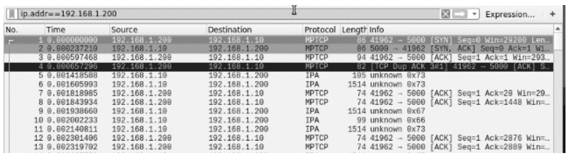
Finally, it has to check whether the solution is backward compatible when sin_zero is not assigned any value. In that case, it is expected to use the original MPTCP authentication mechanism and establish the connections.

C. Results

The experiments were conducted as mention in the above section and the results are as follows.

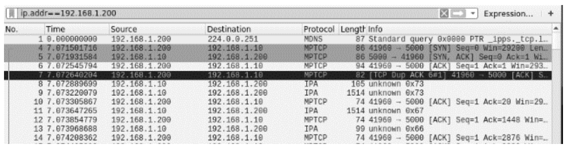
1) Using the same key on both server end and client end:

As mentioned in the experiment 1, when using the same key on both the server and the client ends, the MPTCP connection should start properly. It has to authenticate the second sub-flow using the key obtain from the application level and should start the second sub-flow. The connection was established using TCP sockets from both the server and the client machines. Figure 4 shows the packets send from interface eth0 of the client and Figure 5 shows the packets send from interface eth1 of the client. By analyzing the packets, it can come to a conclusion that both the interfaces has successfully completed the TCP three-way handshake and established the MPTCP connection on both the interfaces successfully.



```
ip.addr==192.168.1.200                                        Expression...  +
No.  Time         Source         Destination    Protocol Length Info
   1 0.000000000  192.168.1.200  192.168.1.10   MPTCP      86 41962 → 5000 [SYN] Seq=0 Win=29200 Len…
   2 0.000237210  192.168.1.10   192.168.1.200  MPTCP      86 5000 → 41962 [SYN, ACK] Seq=0 Ack=1 Wi…
   3 0.000597468  192.168.1.200  192.168.1.10   MPTCP      94 41962 → 5000 [ACK] Seq=1 Ack=1 Win=293…
   4 0.000657296  192.168.1.200  192.168.1.10   MPTCP      82 [TCP Dup ACK 3#1] 41962 → 5000 [ACK] S…
   5 0.001418588  192.168.1.10   192.168.1.200  IPA       105 unknown 0x73
   6 0.001605993  192.168.1.10   192.168.1.200  IPA      1514 unknown 0x73
   7 0.001818985  192.168.1.200  192.168.1.10   MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=20 Win=29…
   8 0.001843934  192.168.1.200  192.168.1.10   MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=1448 Win=…
   9 0.001938660  192.168.1.10   192.168.1.200  IPA      1514 unknown 0x67
  10 0.002002233  192.168.1.10   192.168.1.200  IPA        99 unknown 0x66
  11 0.002140811  192.168.1.10   192.168.1.200  IPA      1514 unknown 0x73
  12 0.002301406  192.168.1.200  192.168.1.10   MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=2876 Win=…
  13 0.002319702  192.168.1.200  192.168.1.10   MPTCP      74 41962 → 5000 [ACK] Seq=1 Ack=2889 Win=…
```

Figure 4: Packets captured from eth0 interface



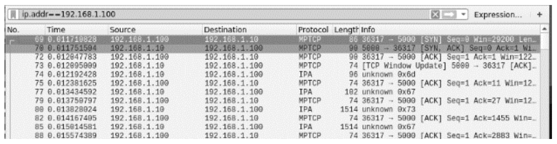
```
ip.addr==192.168.1.200                                        Expression...  +
No.  Time         Source         Destination    Protocol Length Info
   1 0.000000000  192.168.1.200  224.0.0.251    MDNS       87 Standard query 0x0000 PTR _ipps._tcp.l…
   4 7.071501716  192.168.1.200  192.168.1.10   MPTCP      86 41960 → 5000 [SYN] Seq=0 Win=29200 Len…
   5 7.071931584  192.168.1.10   192.168.1.200  MPTCP      86 5000 → 41960 [SYN, ACK] Seq=0 Ack=1 Wi…
   6 7.072545794  192.168.1.200  192.168.1.10   MPTCP      94 41960 → 5000 [ACK] Seq=1 Ack=1 Win=293…
   7 7.072640204  192.168.1.200  192.168.1.10   MPTCP      82 [TCP Dup ACK 6#1] 41960 → 5000 [ACK] S…
   8 7.072889699  192.168.1.10   192.168.1.200  IPA       105 unknown 0x73
   9 7.073220079  192.168.1.10   192.168.1.200  IPA      1514 unknown 0x73
  10 7.073305867  192.168.1.200  192.168.1.10   MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=20 Win=29…
  11 7.073647265  192.168.1.10   192.168.1.200  IPA      1514 unknown 0x67
  12 7.073854779  192.168.1.200  192.168.1.10   MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=1448 Win=…
  13 7.073968688  192.168.1.10   192.168.1.200  IPA        99 unknown 0x66
  14 7.074208362  192.168.1.200  192.168.1.10   MPTCP      74 41960 → 5000 [ACK] Seq=1 Ack=2876 Win=…
```

Figure 5: Packets captured from eth1 interface

2) Using the different keys on server end and client end:

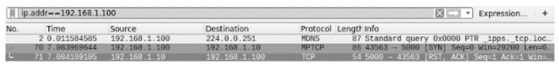
As explained in the experiment 2, two different keys were used in server end and client end when starting the sockets, and the Figure 6 shows the captured and filtered packets send from the client's eth0 interface. Figure 7 shows the packets sent from the eth1 interface of the client.

According to the captured packets, it is clear that the first connection was successfully established with the server because the TCP three ways handshake was successfully completed. But when observing Figure 7, it is clear that the TCP three ways handshake has stopped at the SYN ACK stage. Protocol fails to complete the authentication process because of having two different keys in the server end and the client end. Therefore, it has dropped the connection. Which means when a malicious party attempting to join without having the shared application key, it cannot create the latter sub-flows.



```
ip.addr==192.168.1.100                                        Expression...  +
No.  Time         Source         Destination    Protocol Length Info
  69 0.011710828  192.168.1.100  192.168.1.10   MPTCP      86 36317 → 5000 [SYN] Seq=0 Win=29200 Len…
  70 0.011751504  192.168.1.10   192.168.1.100  MPTCP      90 5000 → 36317 [SYN, ACK] Seq=0 Ack=1 Wi…
  72 0.012047783  192.168.1.100  192.168.1.10   MPTCP      90 36317 → 5000 [ACK] Seq=1 Ack=1 Win=122…
  73 0.012095009  192.168.1.10   192.168.1.100  MPTCP      74 [TCP Window Update] 5000 → 36317 [ACK]…
  74 0.012192428  192.168.1.10   192.168.1.100  IPA        96 unknown 0x6d
  75 0.012381625  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=11 Win=12…
  77 0.013434592  192.168.1.10   192.168.1.100  IPA       102 unknown 0x67
  79 0.013750797  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=27 Win=12…
  80 0.013828024  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x73
  82 0.014167405  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=1455 Win=…
  85 0.015014581  192.168.1.10   192.168.1.100  IPA      1514 unknown 0x67
  88 0.015574389  192.168.1.100  192.168.1.10   MPTCP      74 36317 → 5000 [ACK] Seq=1 Ack=2883 Win=…
```

Figure 6: Packets captured from eth0 interface



```
ip.addr==192.168.1.100                                        Expression...  +
No.  Time         Source         Destination    Protocol Length Info
   2 0.011584585  192.168.1.100  224.0.0.251    MDNS       87 Standard query 0x0000 PTR _ipps._tcp.loc…
  70 7.083969644  192.168.1.100  192.168.1.10   MPTCP      86 43563 → 5000 [SYN] Seq=0 Win=29200 Len=0…
  71 7.084109105  192.168.1.10   192.168.1.100  TCP        54 5000 → 43563 [RST, ACK] Seq=1 Ack=1 Win=…
```

Figure 7: Packets captured from eth1 interface

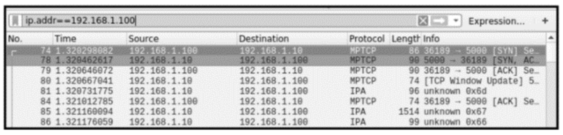
Therefore the main goal of the research was successfully achieved. This means without having the correct application level information, the second subflow cannot be initiated.

3) Backward compatibility: As explained in experiment 3, the sockets have set without assigning any value to sin_zero variable. Figure 8 shows the packets captured on both the eth0 and eth1 interfaces of the client and it has successfully established the MPTCP connection. Which shows that it has used normal MPTCP. Therefore the solution is backward compatible if both the client and server does not use application level keys to authenticate the sub-flows.

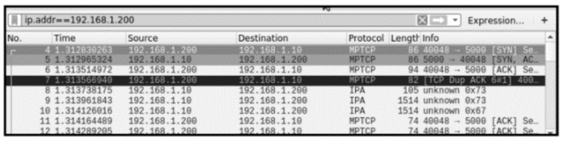
Conclusion and Future Work

Use of external keys in authentication was proposed by C. Paasch and O. Bonaventure (Bagnulo et al., 2015) where they propose to modify existing Socket API. Since it is a redesign and a major restructuring to current implementation, we explored an alternative approach to authenticate sub-flows of MPTCP connection using external keys.

In this study, we experimented the completeness of our approach in three facets. First, with a common external key, two parties were able to authenticate and establish sub-flows successfully. Second, it was shown that a party that does not possess the common external key could not establish a sub-flow due to failed authentication. Finally, when an external key is not involved, the method returns to the normal MPTCP authentication, and hence our approach is backward compatible. With these three facets, we have demonstrated the completeness of the proposed approach. Therefore, it can be used to authenticate sub-flows and eliminate the vulnerabilities in classical MPTCP to certain extent.



```
ip.addr==192.168.1.100
No.  Time         Source          Destination     Protocol Length Info
  74 1.320298082  192.168.1.100   192.168.1.10    MPTCP      86 36189 → 5000 [SYN] Se…
  78 1.320462617  192.168.1.10    192.168.1.100   MPTCP      90 5000 → 36189 [SYN, AC…
  79 1.320646072  192.168.1.100   192.168.1.10    MPTCP      90 36189 → 5000 [ACK] Se…
  80 1.320667041  192.168.1.10    192.168.1.100   MPTCP      74 [TCP Window Update] 5…
  81 1.320731775  192.168.1.10    192.168.1.100   IPA        96 unknown 0x6d
  84 1.321012785  192.168.1.100   192.168.1.10    MPTCP      74 36189 → 5000 [ACK] Se…
  85 1.321160094  192.168.1.10    192.168.1.100   IPA      1514 unknown 0x67
  86 1.321176059  192.168.1.10    192.168.1.100   IPA        99 unknown 0x66
```



```
ip.addr==192.168.1.200
No.  Time         Source          Destination     Protocol Length Info
   4 1.312830263  192.168.1.200   192.168.1.10    MPTCP      86 40048 → 5000 [SYN] Se…
   5 1.312965324  192.168.1.10    192.168.1.200   MPTCP      86 5000 → 40048 [SYN, AC…
   6 1.313514972  192.168.1.200   192.168.1.10    MPTCP      94 40048 → 5000 [ACK] Se…
   7 1.313566940  192.168.1.200   192.168.1.10    MPTCP      82 [TCP Dup ACK 6#1] 400…
   8 1.313738175  192.168.1.10    192.168.1.200   IPA       105 unknown 0x73
   9 1.313961843  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x73
  10 1.314126016  192.168.1.10    192.168.1.200   IPA      1514 unknown 0x67
  11 1.314164489  192.168.1.200   192.168.1.10    MPTCP      74 40048 → 5000 [ACK] Se…
  12 1.314289205  192.168.1.200   192.168.1.10    MPTCP      74 40048 → 5000 [ACK] Se…
```

Figure 8: Packets captured from eth0 and eth1 interfaces with no key

As for future work, it has to develop a mechanism to generate a secure authentication material and proper method to transfer the authentication material from the client machine to the server machine.

References

Postel, J., 1981. RFC 793 - Transmission Control Protocol. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc793> [Accessed 26 June 2019].

Ford, A., Raiciu, C., Handley, M. and Bonaventure, O., 2013. RFC 6824 - TCP Extensions For Multipath Operation With Multiple Addresses. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc6824> [Accessed 26 June 2019].

Bonaventure, O., 2014. Observing Siri : The Three-Way Handshake — MPTCP. [online] Blog.multipath-tcp.org. Available at: <http://blog.multipath-tcp.org/blog/html/2014/02/24/observing_siri.html> [Accessed 27 June 2019].

Demaria, F., 2016. Security Evaluation Of Multipath TCP. [online] Diva-portal.org. Available at: <http://www.diva-portal.org/smash/get/diva2:934158/FULLTEXT01.pdf> [Accessed 27 June 2019].

Ford, A., Raiciu, C., Handley, M., Barre, S. and Iyengar, J., 2011. Architectural Guidelines For Multipath TCP Development. [online] Rfc-editor.org. Available at: <https://www.rfc-editor.org/info/rfc6182> [Accessed 27 June 2019].

Bagnulo, M., 2011. RFC 6181 - Threat Analysis For TCP Extensions For Multipath Operation With Multiple Addresses. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc6181> [Accessed 30 June 2019].

Bagnulo, M., Paasch, C., Gont, F., Bonaventure, O. and Raiciu, C., 2015. Analysis Of Residual Threats And Possible Fixes For Multipath TCP (MPTCP). [online] Rfc-editor.org. Available at: <https://www.rfc-editor.org/info/rfc7430> [Accessed 30 June 2019].

Jadin, M., Tihon, G., Pereira, O. and Bonaventure, O., 2017. Securing Multipath TCP:Design & Implementation. [online] Dial.uclouvain.be. Available at: <https://dial.uclouvain.be/pr/boreal/object/boreal:184252> [Accessed 30 June 2019].

Paasch, C. and Bonaventure, O., 2013. Securing The Multipath TCP Handshake With External Keys. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/id/draft-paasch-mptcp-ssl-00.html> [Accessed 30 June 2019].

Díez, J., Bagnulo, M., Valera, F. and Vidal, I., 2020. Security For Multipath TCP: A Constructive Approach.

Kim, Y. and Choi, K., 2016. Draft-Kim-Mptcp-Semptcp-00. [online] Datatracker.ietf.org. Available at: <https://datatracker.ietf.org/doc/html/draft-kim-mptcp-semptcp> [Accessed 30 June 2019].

Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C. and Moeller, B., 2006. RFC 4492 - Elliptic Curve Cryptography (ECC) Cipher Suites For Transport Layer Security (TLS). [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc4492> [Accessed 5 July 2019].

McGrew, D., 2008. RFC 5116 - An Interface And Algorithms For Authenticated Encryption. [online] Datatracker.ietf.org. Available at: <https://datatracker.ietf.org/doc/rfc5116/> [Accessed 5 July 2019].

Eddy, W., 2017. RFC 4987 - TCP SYN Flooding Attacks And Common Mitigations. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc4987> [Accessed 5 July 2019].

Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q. and Smith, E., 2018. Draft-Ietf-Tcpinc-Tcpcrypt-10 - Cryptographic Protection Of TCP Streams (Tcpcrypt). [online] Datatracker.ietf.org. Available at:

<https://datatracker.ietf.org/doc/draft-ietf-tcpinc-tcpcrypt/10/> [Accessed 5 July 2019].

Bittau, A., Giffin, D., Handley, M., Mazieres, D. and Smith, E., 2017. Draft-Ietf-Tcpinc-Tcpeno-18 - TCP-ENO: Encryption Negotiation Option. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/draft-ietf-tcpinc-tcpeno-18> [Accessed 5 July 2019].

Dierks, T. and Rescorla, E., 2018. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. [online] Tools.ietf.org. Available at: <https://tools.ietf.org/html/rfc5246> [Accessed 5 July 2019].

Wireshark.org. 2020. Wireshark · Go Deep.. [online] Available at: <https://www.wireshark.org> [Accessed 20 July 2019].

Man7.org. 2020. Proc(5) - Linux Manual Page. [online] Available at: <http://man7.org/linux/man-pages/man5/proc.5.html> [Accessed 8 July 2019].

Man7.org. 2020. Netlink(7) - Linux Manual Page. [online] Available at: <http://man7.org/linux/man-pages/man7/netlink.7.html> [Accessed 8 July 2019].

Gta.ufrj.br. 2020. Struct Sockaddr_In, Struct In_Addr. [online] Available at: <https://www.gta.ufrj.br/ensino/eel878/sockets/sockaddr_inman.html> [Accessed 10 July 2019].

Author Biographies

Tharindu Wijethilake is an assistant lecturer at the University of Colombo School of Computing. He obtained Master's degree in Computer Science from the UCSC. His research interests include computer networks, network security and information system security.

Kasun Gunawardana is a senior lecture at the University of Colombo School of Computing. He obtained his PhD from the Monash University. His research interests are Machine Learning, Cyber Security, and Digital Forensics.

Chamath Keppitiagama obtained his PhD from the University of British Colombia and he is a senior lecturer at the University of Colombo School of Computing. His research interests are is Computer Networks, Distributed Systems and Operating Systems.

Kasun de Zoysa is a senior lecturer at the University of Colomobo School of Computing. He obtained his Phd from the Stockholm His research interests includes Information Security, Sensor Networks and Embedded Systems.