Université Catholique de Louvain la Neuve

Deep Packet Inspection and Multipath TCP

Author:
Baugnies Benjamin (6020-10-00)

Advisor:
Ramin Sadre

2014 - 2015

Contents

List of Figures

List of Tables

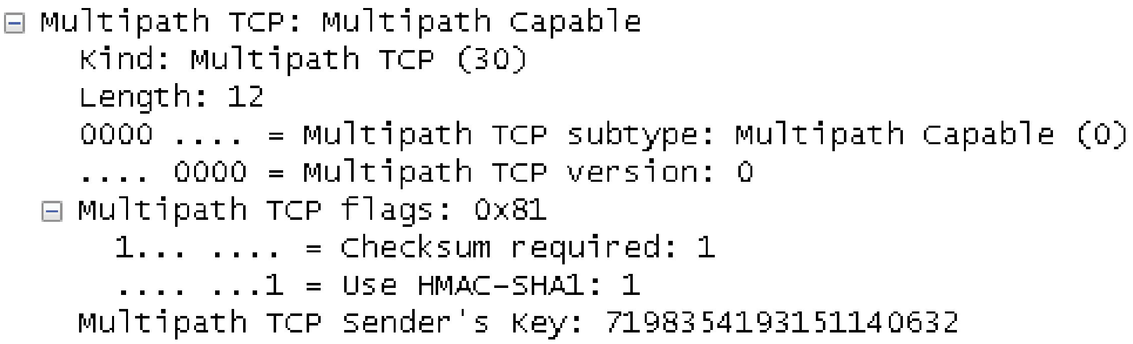
Chapter 1

Introduction

Creating a new protocol and ensuring its large scale deployment and use over the internet presents a series of wide ranging challenges. Multipath TCP (MPTCP) has taken many of these considerations into account from the start of its development. For example, it goes to great lengths to ensure retro-compatibility with regular TCP, it has mechanisms for working over Network Address Translators (NATs) and certain other middle boxes, and can use fall-back signaling to revert to TCP if needed. Thanks to this, it has met a level of success which is rare for a protocol of its type. However, there are still many challenges ahead to ensure that the protocol can reach a level of usage comparable to that of regular TCP.

One such challenge is security. While securing a protocol itself through the use of randomized sequence numbers, cryptography, and careful design is important, many attacks use legitimate protocol operations and can only be detected as attacks when looking at the bigger picture. Even when a protocol has been designed to cope with or to mitigate a problem, a network operator may still wish to be alerted when such an attack is taking place. Let's take TCP as an example. If a server receives a SYN packet, replies with a SYN+ACK, and no final ACK arrives, the connection is in a partially opened state until it times out. This is a regular operation and is certainly not a problem from the protocol's point of view. The issue arises when thousands of SYN packets arrive within a short window, a SYN flooding attack. None of the individual connections is odd, but a security system observing the traffic and capable of understanding TCP will easily identify this as an attack. Even with ways to deal with this problem, such as stateless server operation during the connection establishment, the network's operators would likely want to be informed that this attack is happening/has happened.

This is just one example to illustrate the need for security and monitoring tools that understand the protocols in use within the network. Many efforts have already been made in order to adapt commonly used tools to MPTCP. For example, Wireshark [5] was patched in order to display MPTCP options and their field values [8], as we can



```
⊟ Multipath TCP: Multipath Capable
    Kind: Multipath TCP (30)
    Length: 12
    0000 .... = Multipath TCP subtype: Multipath Capable (0)
    .... 0000 = Multipath TCP version: 0
  ⊟ Multipath TCP flags: 0x81
      1... .... = Checksum required: 1
      .... ...1 = Use HMAC-SHA1: 1
    Multipath TCP Sender's Key: 7198354193151140632
```

Figure 1.1: MPTCP option as seen in Wireshark

see in figure 1.1. However, it does not yet provide the function to “follow” an MPTCP stream as it can a regular TCP stream. The TCPdump [6] command line utility was also updated in order to display MPTCP information [3]. Both the Wireshark and TCP dump extensions are now featured in the standard distribution of these programs. The packet manipulation program Scapy [11] was also given an MPTCP update, allowing it to read, display, modify and write MPTCP options [9] .

This work is intended as a continuation of those efforts in order to bring a new security tool up to speed with MPTCP: Deep Packet Inspection (DPI). DPI is a method generally employed within intrusion detection systems (IDS). While a lot of packet analysis usually stops at headers, DPI can provide byte-level inspection of the whole packet, including the payload. While many systems use DPI, we have chosen to work with the Bro network analysis framework [13]. Bro is a powerful and flexible system which also has the great advantages of being widely used and open source. As part of our work, we have extended Bro to provide MPTCP comprehension, and built scripts to use these new functionalities for MPTCP analysis. Until now, whenever Bro encountered a packet using MPTCP, it would be treated as a regular TCP packet. Thus, though the operation of the system was not jeopardized, users were incapable of retrieving MPTCP-related information. Furthermore, the subsequent analysis of application-layer protocols became impossible if the MPTCP stream was split over multiple connections, since Bro is incapable of reassembling subflows into a single data stream.

This document is split into three main parts. We will begin by presenting the inner workings of the Multipath TCP protocol in chapter 2 and the Bro framework in chapter 3. The second part will focus on a first attempt at making the system MPTCP compliant by modifying the TCP Analyzer, for which the source code is available at `https://github.com/bbaugnies/DPI-MPTCP`. We will start by describing the modifications to the Bro Engine in chapter 4 . In chapter 5 , we will describe how the modified system can be used to extract meaningful information from analyzed traffic and packet traces through the use of scripts. With the main development explained, chapter 6 will then describe how both the Bro modifications and scripts were tested. Chapter 7 will serve as a recap for this first implementation, giving a summary of what it can be used for, as well as its shortcomings. The third and final part of this thesis will describe another way to enable Bro to handle MPTCP, specifically aiming at enabling packet reassembly. Chapter 8 will first take a deeper look at how TCP reassembly is performed by Bro. Chapter 9 contains a detailed description of the requirements needed to allow this reassembly to work with MPTCP connections. Finally, we will discuss further work which can be done on the subject in chapter 10 .

Chapter 2

An Overview of MPTCP

Multipath TCP is an extension of the TCP protocol which was published as an Experimental standard by the Internet Engineering Task Force (IETF) in RFC 6824 [1] in January 2013. Several implementations have since been developed. This work was done using version 0.89 of the Linux Kernel implementation [2]. At its core, MPTCP aims to allow a host to use multiple link-level paths for a single TCP connection, improving throughput and redundancy.

This section will cover how the MPTCP protocol works by explaining which types of packets are sent, and how the different parts of a connection are performed.

2.1 General Operation and Option Types

MPTCP initially behaves like a regular, single TCP connection. However, once the use of MPTCP has been successfully negotiated, either one of the hosts can decide to start a new TCP connection that will join the MPTCP connection. The host can then use all the individual TCP connections (called subflows) to spread the data, using the combined bandwidth of each path. Figure 2.1 shows how two hosts can use the protocol to set up several subflows.

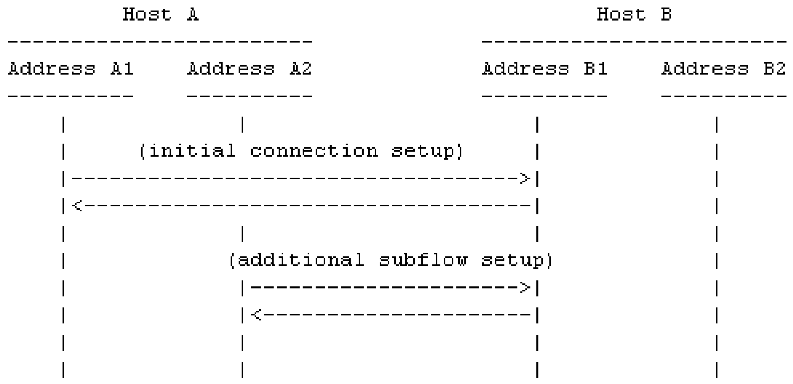
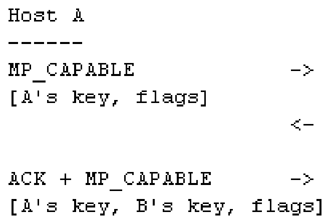


Figure 2.1: Setting up multiple subflows between two hosts

MPTCP's operation is based on the use of TCP options. The protocol uses the option kind 20, which is reserved by the Internet Assigned Numbers Authority (IANA). The option has a varying length which will be discussed in further detail later. The first four bits of an MPTCP option are always used to indicate the option subtype which is one of the eight following values:

- Subtype 0, MP_Capable: used to negotiate the use of MPTCP during the establishment of the first TCP connection.
- Subtype 1, MP_Join: used during the establishment of a secondary TCP connection to signal that it is a subflow of an existing MPTCP connection.
- Subtype 2, DSS: contains the data sequence mapping used to indicate how data from different subflows is ordered, enabling re-assembly on the receiver side.
- Subtype 3, Add_Addr: used within a subflow to signal the existence of other addresses that can be used to set up additional subflows.
- Subtype 4, Remove_Addr: used like Add_Addr, but to signal that an address is no longer available.
- Subtype 5, MP_Prio: addresses can be used as either normal subflows or backup. This subtype allows a host to change the status of an address to and from backup.
- Subtype 6, MP_Fail: this option is used to close a subflow which has been opened correctly but for which modifications to the data have been detected. If the data sequence mapping is modified, the data on that subflow cannot be used since the reassembly will fail. The connection is therefore closed using an MP_Fail option which allows the host to "forget" all the data that was sent on that subflow.
- Subtype 7, MP_Fastclose: in an MPTCP connection, a TCP rst will only close that particular subflow. This option subtype is equivalent to an rst, but for the whole MPTCP connection.

The format of each MPTCP option as defined in the RFC can be found in appendix A. A given packet can contain multiple MPTCP options.



```
Host A                                 Host B
------                                 ------
MP_CAPABLE             ->
[A's key, flags]
                       <-              MP_CAPABLE
                                       [B's key, flags]
ACK + MP_CAPABLE       ->
[A's key, B's key, flags]
```

Figure 2.2: Data exchanged for MPTCP connection setup

We will now go over how the important steps of an MPTCP connection are performed. For a more detailed explanation, please consult RFC 6824.

2.2 First Connection Establishment

When a host A wishes to establish an MPTCP connection with a host B, it will begin by initiating a TCP connection, where the first SYN packet will contain an MP_Capable option. This option contains the MPTCP version number, a number of flags which are used to negotiate the cryptographic algorithms to employ, and the 64-bit key which A will use for this connection (the whole MPTCP connection, not just the subflow).

B will then reply with a SYN+ACK. If the packet does not contain the MP_Capable option, then either the host cannot use MPTCP, or the options have been removed along the way. In any case, the connection reverts to standard TCP. If the option is present, it will contain the key that B will use. Host A responds with the final ACK packet, completing the three-way handshake. This ACK must also contain an MP_Capable option, along with both host's keys.

Each host can then compute the tokens that will uniquely identify the connection on each host. These tokens are 32-bit values computed by truncating a hash of the key to its most significant 32 bits. A token must be unique for a given host. Figure 2.2 shows the packets exchanged during this process.

2.3 Advertising New Addresses

Once a connection is established, each host may wish to indicate to the other that it has additional addresses available for new subflows. To do this, it will use an existing subflow and send a packet with the Add_Addr option. The address is mapped to an Address ID which is unique per host and per connection (the address for the original subflow has the ID 0). The host wishing to advertise a new address will therefore send its address along with the ID so that the other host can know the mapping. MPTCP supports both IPv4 and IPv6 addresses. It is also worth noting that a host can use an address that it has not previously advertised when joining a connection. Once the other host has authenticated the join, the address will be considered implicitly advertised.

If an address later becomes unavailable, the host can send a Remove_Addr option to signal that the other host should no longer attempt to connect to it. Remove_Addr exclusively uses the Address ID to reference an address.

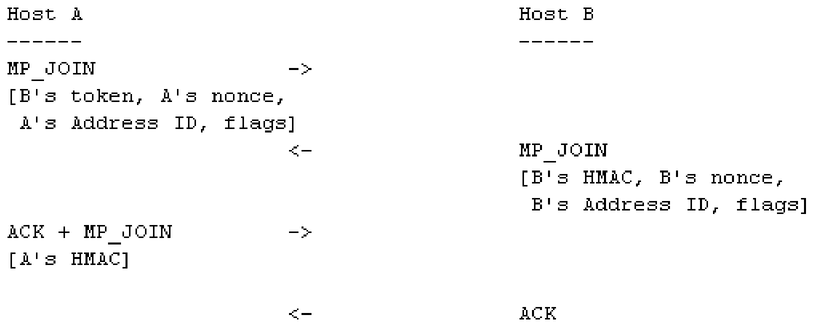
Finally, as mentioned earlier, addresses can be used as regular or backup addresses. If a host wishes to indicate that an address should be used as backup (or that a backup address can now be used regularly), it can send an MP_Prio option. This option references the address in question by its ID, and contains a single bit which indicates the address' new priority.

2.4 Joining an Existing Connection

If host A wants to start a new subflow for the MPTCP connection, it will send a SYN packet to one of host B's advertised (as seen in the previous section) addresses. Note that, as mentioned earlier, it is not strictly necessary for A to have advertised the address it wishes to use. This SYN will contain an MP_Join option. The MP_Join option contains the token to identify which MPTCP connection it concerns, as well as the Address ID of the packet's source (in case the actual header is modified). Letting just any other TCP flow join an ongoing connection would allow attackers to join it and disrupt the communication. Therefore, the join procedure uses an authentication method. To this end, the SYN also contains a 32-bit random number.

Host B will reply with a SYN+ACK containing its own random number. It will also have to use both keys that were exchanged to compute a HMAC of both random numbers. Due to option space limitations, it can only send a truncated (64 bits) version of this HMAC to host A. Unlike MP_Capable, if the SYN+ACK does not contain an MP_Join option, the TCP connection is closed.

Host A must verify that the HMAC is correct. If it is, Host A sends its own HMAC (in full this time, since no other data is needed) to host B. Even though the three-way handshake has thus been completed, host B must still verify host A's HMAC and will close the connection if it is wrong (with an rst). Figure 2.3 shows the packets exchanged during this process.



```
Host A                                Host B
------                                ------
MP_JOIN                   ->
[B's token, A's nonce,
 A's Address ID, flags]
                          <-          MP_JOIN
                                      [B's HMAC, B's nonce,
                                       B's Address ID, flags]
ACK + MP_JOIN             ->
[A's HMAC]

                          <-          ACK
```

Figure 2.3: Data exchanged for MPTCP JOIN

2.5 Sending Data

Data packets belonging to an MPTCP connection will contain a DSS option. This option contains the equivalents of TCP sequence numbers and acknowledgments, but over all the subflows of the MPTCP connection. The Data Sequence Number, Subflow Sequence Number and Data-Level Length allow the receiver to re-order the packets coming from multiple TCP connections into a single coherent data stream. The data sequencing mapping will be discussed in more detail when we cover reassembly.

DSS options also contain the DATA_FIN flag, which is used to indicate that a host has no more data to send. This flag functions like the TCP FIN flag for the entire MPTCP connection and is the regular way of closing the connection.

2.6 Fastclose and Failure

In regular TCP, a host can rapidly close a connection by sending an rst. For MPTCP, a host must close multiple TCP connections to stop a single MPTCP connection. To do this, host A will send an MP_Fastclose option on one of the subflows, along with host B's key. Host A will also send a TCP rst on each one of the other TCP subflows (leaving only one active subflow on the MPTCP connection). If host B recognizes the key, it will answer by closing down the final subflow with an rst.

If a subflow proves unsuitable for MPTCP (for example, if a middle box is modifying the payload and making the correct data sequence mapping irrecoverable), the subflow will be closed with an MP_Fail option. This option indicates the Data Sequence Number before any data was sent on that subflow, allowing the receiver to discard any corrupt information the subflow could have sent.

Chapter 3

The DPI system: Bro

Bro is an open-source, multi-layered, stream-oriented intrusion detection system capable of high-performance analysis and logging. This section will cover how Bro functions. In particular, we will go over each component's job, how it works, and how they all fit together to provide many flexible services.

3.1 System Architecture

Bro's operation is mainly split into two parts: the C++ Event Engine and the Policy Script Interpreter. Figure 3.1 shows Bro's architecture as seen in the documentation [13], though it is worth noting that packets can also be fed into the Event Engine through packet captures.

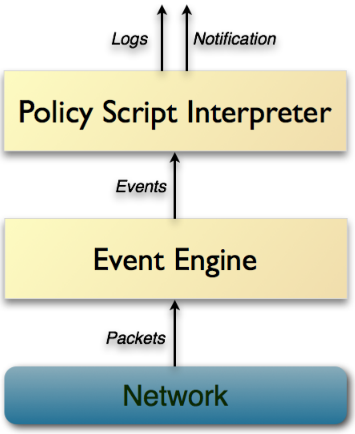
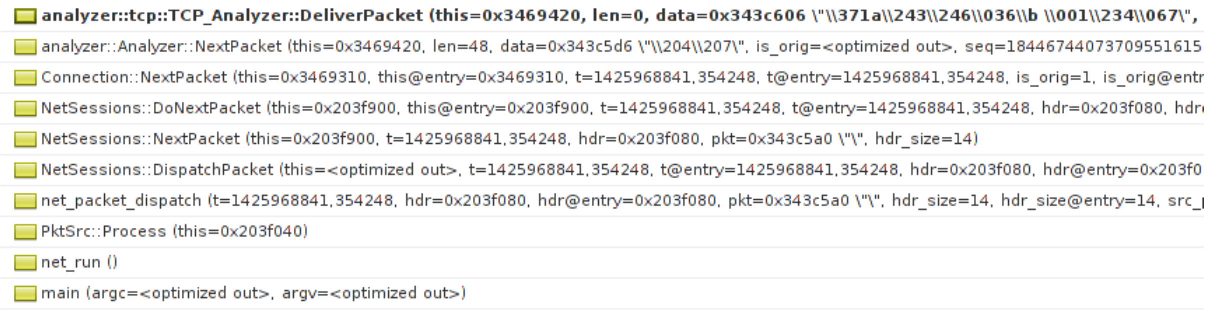


Figure 3.1: Bro's internal architecture

The Event Engine parses incoming packets and raises events based on what is seen. The engine itself is stream-based and stateful, meaning that it stores information about each stream observed in the network and that packets in a same stream are treated as such. The script interpreter is where events are processed in order to extract meaningful information about the network traffic.



```
analyzer::tcp::TCP_Analyzer::DeliverPacket (this=0x3469420, len=0, data=0x343c606 \"\\371a\\243\\246\\036\\b \\001\\234\\067\",
analyzer::Analyzer::NextPacket (this=0x3469420, len=48, data=0x343c5d6 \"\\204\\207\", is_orig=<optimized out>, seq=18446744073709551615
Connection::NextPacket (this=0x3469310, this@entry=0x3469310, t=1425968841.354248, t@entry=1425968841.354248, is_orig=1, is_orig@entr
NetSessions::DoNextPacket (this=0x203f900, this@entry=0x203f900, t=1425968841.354248, t@entry=1425968841.354248, hdr=0x203f080, hdr
NetSessions::NextPacket (this=0x203f900, t=1425968841.354248, hdr=0x203f080, pkt=0x343c5a0 \"\", hdr_size=14)
NetSessions::DispatchPacket (this=<optimized out>, t=1425968841.354248, t@entry=1425968841.354248, hdr=0x203f080, hdr@entry=0x203f0
net_packet_dispatch (t=1425968841.354248, hdr=0x203f080, hdr@entry=0x203f080, pkt=0x343c5a0 \"\", hdr_size=14, hdr_size@entry=14, src_
PktSrc::Process (this=0x203f040)
net_run ()
main (argc=<optimized out>, argv=<optimized out>)
```

Figure 3.2: Bro call stack to deliver TCP packets

The Script Interpreter is the most flexible part of the Bro system. Indeed, Bro ships with a plethora of pre-written scripts that will, for example, log connections for many standard protocols. The great flexibility, however, comes from the fact that anyone can write additional script using Bro's domain specific language (.bro extension). This language is Turing complete, has many pre-defined data structures for network analysis (such as `addr` which stores both IPv4 and IPv6 addresses or domain names), and allows the users to write their own event handlers to detect the behavior they wish to observe. The script interpreter is also highly stateful, allowing users to store any information they wish to extract from the events that are processed.

These two components are not independent, however. Indeed, the Engine is aware of the events, data structures, and even functions that are defined for scripts (in `.bif` files). The Engine's event handler is even made aware of which events are actually used in the scripts that are loaded at runtime, allowing the Engine to skip unneeded computations.

3.2 The Event Engine

As mentioned already, the Event Engine is responsible for parsing packets and generating events. Figure 3.2 shows the stack call for how an event is raised. As we can see, packets are first dispatched to their session, or stream. Once there, the Engine attempts to determine which protocols are in use. This is always done by first determining the transport layer protocol (IPv4 and IPv6 are treated the same way). Once the transport protocol is identified, the packet is sent to the corresponding analyzer through the `DeliverPacket` method. The figure shows that the packet being processed was found to be using TCP, and was therefore delivered to the `TCP_Analyzer`. The analyzer itself contains the logic to analyze the header of the given protocol and raise the corresponding event. Once the analysis is complete, the packet payload is sent to several higher-level analyzers to determine which application layer protocols are used and raise the events concerning those.

During the whole process, it is important to remember that all the packets belonging to the same stream are treated together. For some protocols, TCP included, Bro can perform re-assembly internally and deliver the reassembled stream to the analyzers instead of individual packets. This will play a role on how MPTCP reassembly must be performed. Indeed, MPTCP must reassemble content from multiple streams, meaning that the architecture of the Event Engine is not currently adapted to this. It has no way of maintaining state over multiple streams.

3.3 The Policy Script Interpreter

The Script Interpreter manages all the event handlers that are defined in the scripts. Everything that Bro outputs comes from behavior described in one or more scripts. When run out of the box Bro will produce many logs. Even these are the result of scripts that are run by default. When running Bro from the command line, the user may specify additional scripts to be used. All the events that are handled in loaded scripts are sent to the Engine in order for them to be raised when encountered.

Scripts use an event-driven language specific to Bro. Once again, the language is Turing complete, meaning that we can actually use this language for any computation we would want to do with another language. Users can store variables which will either be scoped only within the given script, globally available for all scripts running on the machine, or even within a single event handler. Bro's script language features many common data types (such as `int` and `count` which are 64-bit signed and unsigned, respectively, numbers), data structures (such as `set` for lists of unique elements, `table` to store key-value pairs...), and custom data types useful for network-related operations (`addr` for IP addresses, `port` for port numbers and their associated protocols ...). We can also define our own data structures, called records.

Scripts are organized into modules to allow one script to access variables or functions of another by using its name space. For example, calling the `Log` module's `create_stream` function will be done within another script by using `Log::create_stream`. The script itself is composed of three main parts. First, an `export` block defines all the constants that will be used. These include global constants that can be redefined, making the new version of the constant available to all scripts. Examples of this are the Notice and Log entry types which will be discussed in section 3.4. New global constants and records are also declared here. Next, users can define functions within their script. This is done as in most any other programming language. The last and most important part is the definition of event handlers.

Event handlers are what makes the language event-driven. So far, functions and constants have been defined, but the script doesn't do anything. As in any event-driven language, the script will actually do work when events happen. Even a basic "hello world" example is done by writing the traditional print command in an event handler:

```
event bro_init() {
  print("hello world\n");
}
```

An event handler is declared using the `event` keyword, followed by the type of event that will trigger it (in the hello world example, the event used is `bro_init`, which is raised when Bro is started). The arguments of the handler correspond to the data fields of that event. Each time the event is raised, the code of the handler is executed. `bro_init` doesn't send any data, but more complex events must always have the same fields as the events defined in the corresponding `.bif` file. Otherwise, the event is unknown to the system.

3.4 Output

Scripts perform the analysis of the network traffic, but once it is done, the results must be output in some form and not stored indefinitely in memory. Rather than having users write their own messages to files, Bro provides two main output methods: logs and notices. Both come with their own framework which allows users to access these high-level functionalities easily.

3.4.1 Logging

Bro's logging framework provides a high-performance way to log behavior in standard tab-separated files. Creating a new log file is done entirely in script. The first step is to make the Interpreter aware of the new log by re-defining the `Log::ID` constant in the scripts `export` block. This constant contains the IDs of each log currently being maintained, and is populated by default with many common logs such as the connections log. We redefine it using the `redef` command:

```
export {
  redef enum Log::ID += { LOG };



  ...
}
```

Once this is done, we must create a new record (in-script data structure) called `Info` which will contain the column names for our log. This is also done within the `export` block. In the record, we specify the name and data type of each column. We are not required to write every element of the record to the log; we must actually specify, for each value, whether or not it must be written to the log with the `&log` keyword. Furthermore, we are allowed to have some elements of the record be optional with the `&optional` keyword. For example, if we wanted to log an address and port number, we could create a record like so:

```
  type Info: record {
     address:   addr &log;
     port:      port &log;
  };
```

The logging framework manages the logs by creating a stream for each individual log. This stream receives all the entries that are generated for its log and is in charge of writing them to the corresponding file. The next step is therefore to create this stream. To do so, we simply call the `Log::create_stream` function. This function takes two arguments: the ID of the log to create, and the record type that will serve as the log header. The call we most often be made within the `bro_init` event handler to ensure the stream is up and ready before any entries are generated, and will most often look like this:

```
  Log::create_stream(LOG, [$columns=Info]);
```

Now that the stream is created, we simply need to generate its entries when we observe the behavior we are interested in. For example, if we wanted to log the address and port of the originator of every TCP SYN packet we saw, we could write our entry generation in the `connection_SYN_packet` event handler. This would look like:

```
event connection_SYN_packet(c: connection, pkt: SYN_packet) {
      Log::write(MyModule::LOG, [ $address=c$id$orig_h,
                        $port=c$id$orig_p]);
}
```

Here we see that we fill the fields of the `Info` record by accessing the fields of the `connection` record that was sent in the event. The resulting log would resemble:

```
#separator \x09
#set_separator ,
#empty_field  (empty)
#unset_field  -
#path MyModule
#open 2015-05-04-18-57-53
#fields address port
#types  addr  port
2a02:a03f:2214:b200:147:1f2:679d:1ecd 53013
2a02:a03f:2214:b200:147:1f2:679d:1ecd 55243
2a02:a03f:2214:b200:204:4bff:fe0a:54f9 40886
#close  2015-05-04-18-57-54
```

3.4.2 Notices

Logging is important, and parsing logs is a common way to detect that certain attacks took place. However, in many cases we may be interested in a more immediate reaction. This is the Notice framework's purpose. Notices are created in a way very similar to log entries, and they are raised much like log entries are written. Like for Logs, we first begin by redefining the list of known notice types:

```
   redef enum Notice::Type += { MyNotice, };
```

This time, we do not define our own `Info` record because the notice framework uses its own: `Notice::Info`, defined in `/scripts/base/frameworks/notice/main.bro`. A quick inspection of the record shows that it provides many fields to transmit information to the function that will handle the notice. Most of these fields, however, are optional, and we therefore need only send the information of interest for a given event.

The next step is actually raising the notice. This is done in a similar way as creating a log entry, only simpler. Indeed, we only need to call the `NOTICE` function which takes a `Notice::Info` record as argument. Returning to our earlier example of TCP SYN packets, we could raise a notice for every one instead or in addition to logging it. This would be done as follows:

```
event connection_SYN_packet(c: connection, pkt: SYN_packet) {
      NOTICE([$note=MyNote,
              $msg=fmt("SYN received from %s", c$id$orig_h)]);
}
```

Once again, most fields of the `Notice::Info` record are optional. Here, we filled `note` which is the notice type which allows the notice handler to distinguish which kind of notice it was (and the only mandatory field), and `msg` which allows us to transmit a human-readable message along with the notice.

Once the notice is raised, it will not be acted upon unless a script subscribes to notices by using a hook. The hook will be executed every time a notice is raised, and can check the note to determine which notice took place, and how to react to it. For our example, we could write a hook that sends an e-mail every time the notice is raised (a very bad idea in practice, of course):

```
hook Notice::policy(n: Notice::Info) {
    if (n$note == MyModule::MyNoticer) {
        add n$actions[Notice::ACTION_EMAIL];
    }
}
```

Chapter 4

Bro Events

Events in Bro fulfill a crucial function. They are the result of the C++ event engine's analysis of the network traffic and serve as a link to the event-driven "scriptland" DSL. Essentially, events are a condensed representation of what has been seen which is used to understand what is happening. In order to make Bro capable of working with a new protocol, enabling it to raise relevant events is the first step.

In this section, we will go over how events are generated within the event engine and describe which events have been implemented for the understanding of MPTCP.

4.1 Protocol Analyzers

Every packet that goes through Bro will go through a hierarchy of analyzers. Essentially, the system attempts to detect which protocols are present in a given stream by passing the packets through an analyzer tree which will dynamically determine the protocols in use, from transport-layer protocols right down to application-layer protocols.

Figure 4.1 shows the analyzer's class layout. In this case, we are interested in the TCP analyzer which is a Transport Layer analyzer. At this point of the analysis, protocol detection is straightforward and unambiguous, which is the reason Transport-layer analyzers serve as the root of analyzer trees. This means that we are operating at the lowest level of the stream analysis and we are receiving data directly as it appears in the packet (without modifications of previous analyzers). At runtime, the event engine is made aware of the events required by the scripts being run. The TCP analyzer will parse the incoming packets at the byte level, searching for the behavior that corresponds to the needed event. Once such behavior has been detected, the event is raised by sending a value list containing the relevant information to the event handler. The handler will enqueue the event and deliver it to the script which is how the user can act upon the information.

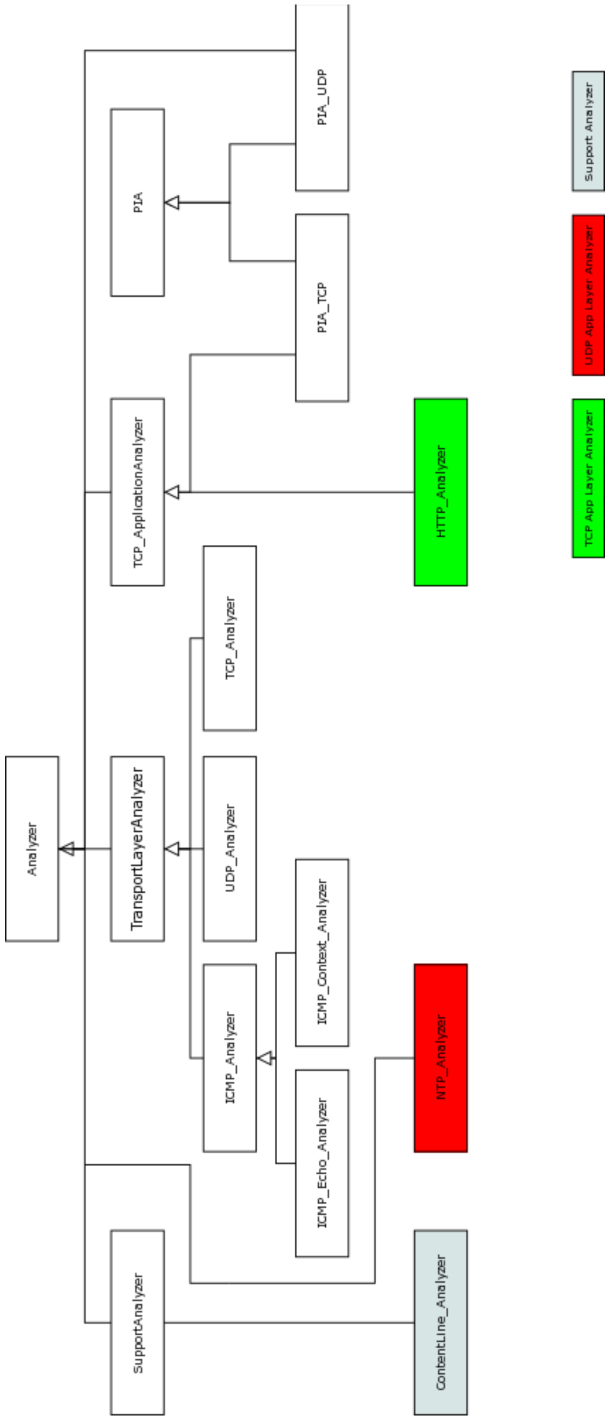


Figure 4.1: Class Hierarchy of Bro analyzers

Events can be relatively high-level (such as the establishment of a connection) or low-level (such as a TCP packet being delivered). As mentioned above, each one is represented by a list of values which provides the scripts with information to piece together what has happened. At the very least, this includes the `connection` value, which is a composite data type containing information about the stream the event happened on (for a TCP stream, this includes the source and destination addresses and ports, the state of the stream, and information about the underlying protocols). In several cases though, it is useful to provide more information for use in the script layer. When one wants to be alerted for each arriving TCP packet, for example, it is likely for a rather thorough analysis for which the `connection` value is not enough. The corresponding event, `tcp_packet`, therefore contains much more. The following code block shows the declaration and documentation of this event:

```
## Generated for every TCP packet. This is a very low-level and expensive event
## that should be avoided when at all possible. It’s usually infeasible to
## handle when processing even medium volumes of traffic in real-time. It’s
## slightly better than :bro:id:‘new_packet‘ because it affects only TCP, but
## not much. That said, if you work from a trace and want to do some
## packet-level analysis, it may come in handy.
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## flags: A string with the packet’s TCP flags. In the string, each character
##       corresponds to one set flag, as follows: ‘‘S‘‘ -> SYN; ‘‘F‘‘ -> FIN;
##       ‘‘R‘‘ -> RST; ‘‘A‘‘ -> ACK; ‘‘P‘‘ -> PUSH.
##
## seq: The packet’s relative TCP sequence number.
##
## ack: If the ACK flag is set for the packet, the packet’s relative ACK
##     number, else zero.
##
## len: The length of the TCP payload, as specified in the packet header.
##
## payload: The raw TCP payload. Note that this may be shorter than *len* if
##         the packet was not fully captured.
##
## .. bro:see:: new_packet packet_contents tcp_option tcp_contents tcp_rexmit
event tcp_packet%(c: connection, is_orig: bool, flags: string, seq: count, ack:
    count, len: count, payload: string%);
```

Each event is defined like this in a `.bif` file, which serves to define data types and functions for use in the script level. The full list of TCP events is detailed in `bro/src/analyzer/protocol/tcp/event.bif`. While it is sufficient to provide the name and data types of the event’s values, it is good practice to document each new event with a description of what it corresponds to, what each data field represents, and a list of the related events.

The MPTCP extension uses the TCP option fields (see `http://www.iana.org/assignments/`

tcp-parameters/tcp-parameters.xhtml) for its operation. However, TCP options receive very little native support in Bro. Indeed, the only relevant event is the tcp_option which provides the option kind and length. As the documentation itself states “There is currently no way to get the actual option value, if any.” Additionally, the function that raises this event is only executed on packets possessing a payload.

This is clearly problematic since MPTCP relies on the option value to establish the option subtype and the connection parameters such as the keys used to identify the connection, or the sequence mapping necessary to re-assemble the data from multiple subflows. MPTCP options are also crucial during the connections establishment (when no payload is given). In order to provide the ”scriptland” with enough information to understand MPTCP, we must first extend the analyzer to provide new, more detailed events of what is going on in the options.

4.2 Adding new Events

As we have seen, some work must be done at the analyzer level for MPTCP to be understood. The question remains whether to implement a new analyzer which would be a child of the TCP analyzer, or extend the TCP analyzer. The general architecture of the system is to use one analyzer per protocol. Additionally, the parsing of the TCP header is already done and it would be redundant and inefficient to do it twice. For these reasons, the choice was made to simply add the MPTCP parsing and event generation in the pre-existing analyzer. We will therefore work on bro/src/analyzer/protocol/tcp/TCP.cc.

Packets enter the analyzer through the DeliverPacket method. In this method we can see that the first steps are to extract the TCP header, check the flags, and update the state of the stream state. What interests us comes a little later:

```
if ( tcp_option && tcp_hdr_len > sizeof(*tp) &&
     tcp_hdr_len <= uint32(caplen) )
   ParseTCPOptions(tp, TCPOptionEvent, this, is_orig, 0);
```

This snippet of code shows us how Bro starts the analysis of the header to detect TCP options. The first line shows what we discussed in the previous section; events are only raised when the scripts call for them. This is done thanks to the boolean tcp_option being set when the script uses the event of the same name. For obvious efficiency reasons, it is not desirable to dissect the option part of the header if the end users are not interested in this behavior. If this is the case, we will call the ParseTCPOptions method.

The behavior of this function is as we would expect. It loops through each option until it reaches the end of the header, and for each valid one, it calls the TCPOptionEvent callback function which will populate the value list and raise the event.

For the simplest solution possible, it would suffice to add the value of the option to the value list and parse this in script. However, this would cause many undesired effects. Firstly, the event would be raised much more often than we would want. Secondly, this would force users to do byte-level parsing in script which the DSL is not adapted to do. Finally, scripts would be much harder to understand as they would only catch one event which would contain the whole parsing and treatment for MPTCP behavior and any other option comprehension we would want to do.

In order to improve this, we define ten new events and two new functions, analogous to the ones we have seen: `ParseMPTCP` and the callback function `MPTCPEvent`. Out of the ten events, eight correspond to the different subtypes that are used by the protocol. The last two are `mptcp`, a generic event indicating the use of the protocol, and `mp_error`, indicating an option which is in some way illegal with regards to the RFC. All the events will be described in further detail in the following section. For the two functions, the first will behave exactly like `ParseTCPOptions` except for two points: it will be called if any MPTCP event is used in script, and it will only use the callback function if the option kind is 30 (MPTCP). The `MTCPEvent` is where we parse the value of the option.

The first step is to extract the subtype which is contained in the first four bits of the third option byte. With only this information, we can already raise the `mptcp` event if needed. If further information is required by the script, we begin a case statement on the subtype. Guards are regularly re-checked to ensure as little unneeded work as possible is done. Based on the value of the subtype, we will be able to extract the relevant fields from the header, fill the value list, and raise the correct event. If, at any point of the process, an illegal value is found (such as an option length which does not correspond to the authorized values for a given subtype), an `mp_error` event is raised.

4.3 Event Description

As mentioned in the previous section, ten events were added to the analyzer in order to facilitate working with the protocol, corresponding to the eight existing subtypes, a generic event, and an error event. Each one of these new events was added to `bro/src/analyzer/protocol/tcp/event.bif` along with its documentation. While the selection of which events to implement follows an intuitive choice, it is far from the only possibility. Using multiple different events allows us to minimize the in-script parsing which improves efficiency. It also allows users to focus on the behavior they want to observe. For example, when attempting to log connection information, we might not want events being fired for every DSS option detected, or care about whether an address is considered as a backup. Using more, higher-level events than what is provided in this work could also be considered. When observing the events available for standard TCP, we can see that different events exist for each step of the connection process.

This section will now describe the added events in detail. We will go over which elements are contained in its value list and how it is populated, as well as how it might be used to understand the protocol's behavior in script. For a reminder on the format of the MPTCP options, consult appendix A. For the definition of the new events in the `.bif` format, see appendix B.

mptcp event

The first implemented event is the simple `mptcp` event. This is primarily intended as a proof of concept event, but might be used to detect mptcp activity without the will to delve much deeper into details. In addition to the connection value (discussed earlier), the `mptcp` event provides the user with the option length and subtype. As such, the event can be raised early in the parsing process since we only care about extracting the subtype (it is already necessary to get the length when iterating over the different options of the header). This is all done before even beginning the main case statement that makes up most of the `MPTCPEvent` function. Once again, the practical utility of this event is limited, but it is a good starting point for the comprehension of the function thanks to its simplicity.

There are two important things to note with this event. First, if it will potentially be raised very often and can therefore prove impractical during real-time packet analysis (it is raised once for each MPTCP option, which means at least once per MPTCP packet). The other is that, due to how early it is raised during the option parsing, it will still be raised if an error is detected later. This makes it the only event which can be raised along with an `mp_error` on the same option.

mp_error event

The `mp_error` event is one that should never arise during the use of a correct MPTCP implementation. Such errors are mainly based on the authorized lengths for the options, and either TCP or MPTCP flags. For example, a `mp_capable` option (subtype 0) should only ever be of length 12 or 20. Furthermore, it should only be of length 20 during the final step of the three-way handshake (SYN + ACK) since it is the only time where two 64-bit keys are sent. Another example is the `DSS` option (subtype 2). In this case, the length varies not based on the TCP flags, but on the MPTCP flags contained in the option itself.

Given how the fields of an event's value-list are filled, it is not possible to work with incoherent values. What key should be returned if the length of a `mp_capable` option is

10? Should we only send six bytes or assume the length is off and add two bytes from the next option or TCP payload? Given that this should not happen, we do not attempt to make the call and simply raise an error. The regular MPTCP event is not raised, and we get an `mp_error` event instead.

In order to raise this event, the parsing process is carried out as normal. For each different subtype, the length of the option is matched against the known authorized length values that the subtype can take. Further fine-tuning is done based on information from the main TCP header and the option's flags, when applicable. The values contained in the event are the same as those in the `mptcp` event, namely, the option length and subtype. Unlike the `mptcp` event however, it is raised much later in the parsing process given that the option must be analyzed in detail in order to detect the error.

It is worth noting that detecting a faulty implementation is far from the main aim of the software. For similar errors in the basic option parsing, incoherent length values are simply dealt with by returning -1. Given the relatively young age of the protocol, however, this event might find some use either due to new implementations containing errors or because of new attacks attempting to break the protocol.

mp_capable event

The first of the main events, the `mp_capable` event is raised when an MPTCP option with subtype 0 is encountered. These options are piggy-backed onto the TCP connection establishment to negotiate the use of MPTCP or lack thereof. The option value contains, in addition to the subtype, a four-bit version number of the MPTCP implementation used, eight bits of flags and one or two 64-bit keys. The option has two authorized lengths: 12 bytes for the first two steps of the three-way handshake (SYN & SYN+ACK), or 20 bytes during the final step (ACK) which the is only time the second key is present. In order to allow the user full control over what he wants to observe, each piece of information (other than the subtype since it is always 0) is added to the event's value list. The four last bits of the third byte are passed as an integer value representing the version number, the flags are bunched together in another, and finally the eight bytes representing the keys are copied into integer values. They are passed to the script level using Bro's own integer data type, called Count (which is 64 bits). Note that, since optional values are not supported, a second key will always be present in the event. It is therefore important to check the length in the script and to not take it into consideration should the length be 12.

As far as usage goes, the primary use of this event is for the detection of MPTCP connection establishment. This option tells us a lot about what is going on, both at the MPTCP level and for the TCP connection which the option belongs to. For example, a `mp_capable` event with length 12 indicates either a TCP SYN or SYN+ACK has occurred. On further inspection, the is_orig value will tell us if the packet came from the connection initiator or the responder. The first case indicates a SYN, and the second a SYN+ACK. This second packet is when the connection can generally be considered established (see the connection_established event). Finally, an option with length = 20 indicates the final ACK has been sent. We could also use this information to find instances where the use of MPTCP was denied by combining new events with those of the standard TCP analyzer. Indeed, if we were to see a mp_capable event on the SYN, but none after the connection_established event on the same connection, this would indicate that the responder did not allow the use of the protocol and has reverted to standard TCP.

Other uses include the detection of unknown MPTCP versions or cryptographic choices. Furthermore, detecting and saving the keys exchanged by both hosts as well as the choice of cryptographic algorithm allows the IDS to compute the token used to identify the connection on each host, and then use this information to validate the authentication process when additional TCP connections will attempt to join the MPTCP connection.

Although, in the end, we still receive the whole of the information contained in the option as discussed in the first solution (passing the whole option value in TCP_Option events), the advantages of this method are easy to see. We allow the script level access to the different fields of the option as variables rather than requiring byte-level parsing. We can also take the opportunity to observe how adding more events could further facilitate the comprehension of the protocol. Currently, as we have discussed in the previous paragraphs, one still has to use the same event for multiple different scenarios depending on what one wishes to observe. The mp_capable event could be broken down into further, more explicit events such as a connection attempt, the establishment of the connection or its reverting to standard TCP, the first acknowledgement... in a way similar to the events that exist in the standard TCP analyzer. Doing this would also remove the use of “placeholder” fields such as the second key which is sent on each event, but only used in one of the three cases.

mp_join event

mp_join events correspond to options with the subtype 1. Like mp_capable, they are piggy-backed onto TCP connection establishments but with one big difference: when used, the TCP connection is actually a sub-flow of the MPTCP connection it has joined. An option of the subtype contains four bits of flags and the address ID corresponding to the address the packet originated from as it is known in the MPTCP connection (independently of whether or not the actual address was modified by middle boxes along the way). The remaining fields vary depending on which step of the handshake it is on.

Each one of the three steps corresponds to a set of values and a different option length which makes determining which one is taking place easy. However, filling the value list fields is made slightly harder. Based on the length, only parts of the fields are used, and the HMAC makes things worse since the actual field's length can vary. Similarly to the `mp_capable` option, every available piece of information is sent to the script level, and it is up to the user to know which fields to use depending on the length.

In its use, `mp_join` is also similar to `mp_capable` in that it mainly serves to observe connection establishment. While it might be of some interest to simply monitor how many TCP connections are using MPTCP, we can go further and attempt to link multiple connections as being subflows of the same MPTCP connection. The token that is sent during the first SYN serves to identify which connection is being joined and should be unique on a given host. If we can observe the original key exchange for the MPTCP connection (in the MP Capable options), we can compute the tokens and match MP Joins to the corresponding connection. Of course, given that we would observe the traffic between multiple hosts, it is entirely possible to observe multiple key exchanges that would result in the same token. While unambiguous for each host, it would make matching difficult for the outside observer.

mp_dss event

DSS options, subtype 2, form the main body of the connection as they are used during the actual data exchange. This option is the first we encountered which possesses many fields that are either optional or vary in length depending not on the TCP flags but on the MPTCP flags. Filling the fields of the `mp_dss` event therefore requires parsing the five flags in order to determine which bytes correspond to which value. Like before, every individual field of the option is passed into the event.

This option is used to signal how the different subflows must be aggregated in order to re-assemble the data in the right order. As such, the `mp_dss` event will primarily serve to allow observers access to the data sequence mapping to perform the same reassembly as the hosts of the connection. However, the event can also serve to detect MPTCP connections whose establishment was not observed. Additionally, it might even be possible to determine which subflows belong to the same connection if links between the sequence mappings can be established. Finally, the DSS option is also responsible for signalling when a host has no more data to send on this connection.

An important note regarding this event is that DSS options are carried on every MPTCP packet after the connection establishment, making it a very common option when the protocol is in use. As such, the event will be raised fairly often, which may cause performance issues.

mp_add_addr event

The mp_add_addr event is raised when an MPTCP option of subtype 3 is seen. These options are part of the address signalling mechanism of MPTCP. Though rather straightforward, some field values once again depend on the information contained within the same option. The option contains the IP version of the address being added and the address ID that will be used to identify it within the MPTCP connection. These two values can be simply put into the value list as integers. The next important field is the address itself. Using the IP version lets us know how long (32 or 128 bits) the address is. In order to pass it to the script, we use Bro's built-in addr data type. This is a high-level structure which is able to handle both IPv4 and IPv6 addresses and even host names. The last piece of information is the port number. This field is one of the two optional fields whose presence is not indicated by either the TCP or MPTCP flags. The only way to detect its presence during the parsing is whether the option length is a multiple of four (in which case it is absent), or not (in which case it is present). Though simply given as an integer, Bro also has a port data structure which can indicate the protocol associated with the given port.

In order to observe connection establishment, the mp_add_addr can help reduce ambiguity with joins. As we mentioned earlier, the tokens used to identify connections are supposed to be unique for a given host, but are not expected to be so across multiple hosts. Memorizing the addresses that are advertised over a given MPTCP connection as well as its token can help reduce the chance of colliding tokens. Indeed, if two MPTCP connections A and B use the same token, and an address is advertised over connection A, a join coming from that address can probably be assumed to belong to connection A and not B.

Additionally, we can even match subflows to their MPTCP connection based solely on the addresses advertised and used during joins, which allows us to monitor connections without re-computing the cryptographic functions, or if the key exchange was not observed. However, this method cannot account for the modification of addresses by middleboxes, and we risk collisions again if two hosts advertise the same address.

mp_remove_addr event

With subtype 4, REMOVE ADDR options are the opposite of ADD ADDR. It is also one of the simplest options of the protocol; since each address that was advertised over the MPTCP connection is uniquely identified by an ID, that ID is sufficient to unambiguously remove it from the available addresses. However, it does have a small subtlety which is that a single REMOVE ADDR option can remove an arbitrary (within the limits of the TCP option space) number of addresses. The number is deduced from the option length, one byte being used per address ID. Rather than letting the script determine how many addresses are affected by a single `mp_remove_addr` event, one event is raised for each address, each one containing one ID.

The `mp_remove_addr` can be used to optimize memory use by allowing us to remove addresses from memory as soon as the MPTCP connection stops using them. Some use might also be found in trying to understand how a given host decides which addresses to make available and when.

mp_prio event

The `mp_prio` event contains the other optional field not indicated by anything other than option length. It is a very straightforward option that simply changes the status of a subflow to backup or back to regular. Its use is rather limited, but one may be interested in observing which links hosts prioritize.

mp_fastclose & mp_fail events

The last two events, corresponding to subtypes 6 and 7 respectively, are both relatively simple to generate, having few and fixed fields. The data is simply passed to the scripts as integers. In both cases, their use deals with the termination of MPTCP on the connection. `mp_fastclose`, being used for abrupt termination of an MPTCP connection, can primarily be used for resource management (removing connection state from memory when it closes). `mp_fail` has more interesting applications since it can potentially reveal the existence of middleboxes that deny the use of MPTCP between two hosts that are otherwise willing to use the protocol.

Chapter 5

Script-level Functionalities

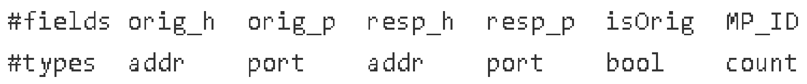
Now that the Event Engine is raising MPTCP events, we can start using them in order to track MPTCP behavior on the network. This section will describe the problems we have tackled. We will explain why it is useful and describe how we can use Bro's script interpreter to showcase certain actions.

5.1 Logging MPTCP Connections

Logging is one of Bro's main features. Logs are important in a great number of cases. They can be used for forensic analysis to determine if an attack took places, for network or protocol performance evaluations, or even to look for traffic trends or understand user behavior. With our new MPTCP events, we will try to log MPTCP traffic specifically. The `conn.log` is a log the Bro creates by default to log the different end-to-end connections that appear over the network. A quick examination of this log shows that it contains a lot of useful information, such as the transport layer protocol used, the duration of the connection, and the number of packets that were exchanged. Since this information is already available, we will concentrate on what is not, and more importantly, on what is unique to MPTCP.

Obviously, we will want to find each TCP connection that was used as an MPTCP subflow. Each one will be identified by the four-tuple that usually defines a connection: the address and port of both endpoints. Next, we will want to group TCP connections that belong to the same MPTCP connection. To do this, we will uniquely identify each MPTCP connection and add a column to the log indicating which MPTCP connection each subflow belongs to. Finally, we might want to know whether a given TCP subflow was the original subflow of its MPTCP connection or not. We will therefore add a column of booleans to show this.

In order to do the logging, we will of course use the logging framework that we dis-



```
#fields orig_h  orig_p  resp_h  resp_p  isOrig  MP_ID
#types  addr    port    addr    port    bool    count
```

Figure 5.1: Header for our MPTCP connection log

cussed in section 3.4.1. Since we have decided what information we wish to log, we can redefine the `Log::ID` constant and create our custom `Info` record like so:

```
type Info: record {
    orig_h:         addr &log;
    orig_p:         port &log;
    resp_h:         addr &log;
    resp_p:         port &log;
    isOrig:         bool &log;
    MP_ID:          count &log;
};
```

Where `orig` and `resp` stand for the connection's initiator and responder, `h` is for the address, `p` is for the port. `isOrig` is the boolean specifying if the connection is the first one of the MPTCP connection, and `MP_ID` is the MPTCP connection's unique identifier. This record will give us the log header shown in figure 5.1.

After creating the log stream, all we have left to do is create the log entries. Traditionally for TCP, Bro considers a connection established starting from the SYN+ACK response in the three-way handshake. Barring aborted connections or failed authentication, which deserves its own special treatment, we can use the same assumption for our subflows. Therefore, a new subflow will be considered set up in two cases:

1. When we see an MP Capable option of length 12 from the responder of the connection.
2. When we see an MP Join option of length 16.

As a reminder, MP Capable options are of length 12 for both the SYN and SYN+ACK since they only carry one 8-byte key, and length 20 on the final ACK since they carry both keys. The two cases where they have a length of 12 can be differentiated by using the `is_orig` boolean that is sent in the `mp_capable` event. MP Join options have different length at each step, 16 being the length of the SYN+ACK since it carries the truncated HMAC for the authentication process (8 bytes), and the responder's random number (4 bytes).

The first case is the simplest. Since the connection is using an MP Capable option, it means that the hosts are establishing a new MPTCP connection. Our `mp_capable` event handler can therefore create a new log entry immediately. The addresses and ports can be extracted by the `connection` record which is sent with the event. `isOrig` will always be true in this case since the MPTCP has just been created. For the final field, `MP_ID` we simply keep a global counter which we increment at each new MPTCP connection. The value of this counter is used as the unique identifier (the Bro count type uses 64 bits which should be enough to uniquely identify connections).

The second case is when a new TCP connection is established using the MP Join option. The logic for this case will be written into the `mp_join` event handler, after a guard ensuring the length of the option is 16. In this case, we are obviously facing an MPTCP connection that has already been established in another flow, `isOrig` will therefore always be false. The addresses and ports can once again be extracted from the connection record that is passed within the event. The main difficulty comes from finding which MPTCP connection this subflow is part of. On a single host, the MPTCP implementation identifies each active connection with a 32-bit token which is sent within the MP Join option. However, Bro will be observing traffic to and from a multitude of hosts and we will not have access to each one's token-to-connection mapping. Furthermore, given two IP addresses, we cannot even tell if they both belong to the same host, whereas the host itself is obviously aware of which addresses are his.

In order to cope with this, we will need to re-create these mappings within the script. To do this, will make extensive use of Bro's `table` structure, a traditional key-value store. We will also define our own record types. First, we define an `MP_host`. This data structure represent one endpoint of one MPTCP connection. Once again, Bro cannot know that two distinct MPTCP connections from different addresses come from the same physical host. Thankfully, we don't need to and we can consider each MPTCP connection to belong to two new hosts as long as we can find all the connection's subflows. The record contains the `MP_ID` of the connection which this host belongs to, the list and number of know addresses this host has advertised and/or used (this is also the mapping between addresses and address IDs which MPTCP uses, and we will have other uses for it later), and the identifier of the other host participating in the connection. The second record we will use is the `MP_conn` record. This data structure represents one MPTCP connection. It contains the identifiers of both `MP_hosts` involved, as well as the number of active subflows. Finally, we define an `MP_addr` record as simply an address and port pair.

The next step is creating the tables that will maintain the mappings we need. Tables are used to retrieve a value for a given key, but not the other way around. We therefore need to think about what data we will have available to use as key, and what data we will need to retrieve. When we receive an MP Join, we will have the addresses and port numbers of the TCP connection and the MPTCP token to work with. Maintaining a mapping of the tokens requires re-computing the cryptographic operations of every host, so we will avoid this for the time being. That leaves the addresses. For a host to establish a new subflow, the destination address must either be the same as that of the original subflow, or have been advertised by the other host on a known subflow of the connection. Therefore, our first table will use `MP_addresses`es as keys and map them to the `MP_host` that advertised them (or created an MPTCP connection on it). Even though the `MP_host` contains the list of its addresses, this will not be available when we receive the Join. `MP_hosts` are referenced by unique IDs. An additional table will thus serve to map IDs to the corresponding hosts.

With these structures ready, we return to the `mp_capable` event handler. When a new TCP subflow is established, we create two new `MP_host`s representing the to endpoints of the TCP connection. Each one is made aware of the other, and originally contains only one address: the one used on the current subflow. Both these hosts are added to the hosts table. Next, both the originator and responder address/port pairs are mapped to their respective host in the address table. Now, when we receive an MP Join with length 16, we can lookup the ID of the `MP_host` the destination address belongs to using the address table. With this ID, we use the host table to lookup the corresponding `MP_host` which contains the ID of the `MP_conn`. This is all the information we need in order to create the log entry. However, one issue remains: besides the addresses of the original subflow, we are not populating the address table.

In order to do this, we need to use the `mp_add_addr` event handler. Add Addr options are always sent on an existing subflow so the hosts will already be known. The first step is determining which one of the two hosts is advertising the new address. This is done simply by using the `is_orig` value of the event. If it is true, we find the host's ID by doing a lookup with the connection originator's address in the address table. If it false, we do the lookup with the responder's address. Once the correct ID has been found, we update the address table by mapping the new address to the host ID. We also update the host's address mapping at this point. An important note is that the port number in a Add Addr option is optional. As per the RFC, if it is not provided, we use the same port number as the one used to send the option.

An additional case to consider is that a host may use an address that it has not explicitly advertised to initiate a new connection. Indeed, it may send an MP Join from an unknown address to one that the other host has advertised. In order to cope with this, the `mp_join` handler will also populate the address table of a host if the incoming address of the Join message is not known.

With everything in place, we can run Bro on a packet trace while loading the script, and we find a new log consistent with the format we were expecting:

```
#separator \x09
#set_separator ,
#empty_field  (empty)
#unset_field  -
#path mp__connection
#open 2015-05-30-23-36-19
#fields orig_h  orig_p  resp_h  resp_p  isOrig MP_ID
#types  addr port  addr  port  bool  count
2a02:a03f:2220:4300:9009:ac68:352f:d0cc 48020 2001:6a8:308f:1:216:3eff:fec5:c815 80 T
    0
192.168.1.47  48190 130.104.230.45 80 F 0
2a02:a03f:2220:4300:204:4bff:fe0a:54f9 39238 2001:6a8:308f:1:216:3eff:fec5:c815 80 F 0
2a02:a03f:2220:4300:9009:ac68:352f:d0cc 48022 2001:6a8:308f:1:216:3eff:fec5:c815 80 T
    1
192.168.1.47  40568 130.104.230.45 80 F 1
2a02:a03f:2220:4300:204:4bff:fe0a:54f9 34776 2001:6a8:308f:1:216:3eff:fec5:c815 80 F 1
192.168.1.47  50760 137.110.116.31 80 T 2
192.168.1.47  50763 137.110.116.31 80 T 3
192.168.1.47  50764 137.110.116.31 80 T 4
192.168.1.47  50765 137.110.116.31 80 T 5
192.168.1.47  50775 137.110.116.31 80 T 6
192.168.1.47  50776 137.110.116.31 80 T 7
192.168.1.47  50769 137.110.116.31 80 T 8
192.168.1.47  50768 137.110.116.31 80 T 9
192.168.1.47  50770 137.110.116.31 80 T 10
192.168.1.47  50818 137.110.116.31 80 T 11
192.168.1.47  50819 137.110.116.31 80 T 12
192.168.1.47  50820 137.110.116.31 80 T 13
192.168.1.47  50820 137.110.116.31 80 T 14
192.168.1.47  50819 137.110.116.31 80 T 15
192.168.1.47  50826 137.110.116.31 80 T 16
192.168.1.47  50828 137.110.116.31 80 T 17
192.168.1.47  50827 137.110.116.31 80 T 18
192.168.1.47  50828 137.110.116.31 80 T 19
192.168.1.47  50827 137.110.116.31 80 T 20
2a02:a03f:2220:4300:9009:ac68:352f:d0cc 48178 2001:6a8:308f:1:216:3eff:fec5:c815 80 T
    21
2a02:a03f:2220:4300:204:4bff:fe0a:54f9 53632 2001:6a8:308f:1:216:3eff:fec5:c815 80 F
    21
192.168.1.47  58468 130.104.230.45 80 F 21
192.168.1.47  59310 178.254.13.90 80 T 22
192.168.1.47  59311 178.254.13.90 80 T 23
2a02:a03f:2220:4300:204:4bff:fe0a:54f9 53657 2002:b2fe:d5a::1 80 F 23
2a02:a03f:2220:4300:204:4bff:fe0a:54f9 49181 2002:b2fe:d5a::2 80 F 23
2a02:a03f:2220:4300:9009:ac68:352f:d0cc 33329 2002:b2fe:d5a::1 80 F 23
2a02:a03f:2220:4300:9009:ac68:352f:d0cc 54239 2002:b2fe:d5a::2 80 F 23
#close  2015-05-30-23-36-24
```

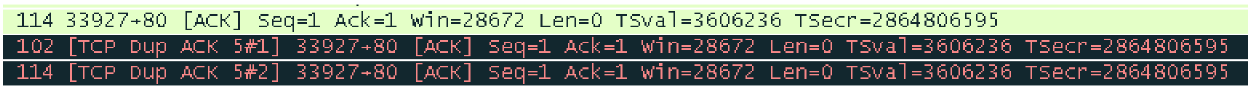
With these steps, the MPTCP connection log is functional. For performance reasons though, we still need to cleanup our data structures once they are no longer needed. In order to do this, we can use a handler for the `connection_state_removed` event. This event is automatically raised when the Event Engine removes a stream from its memory, usually when it is terminated. This event only provides us with the connection record of the stream being removed. If the connection is known within our script, we will remove all the data related to that connection. When removing the last subflow of an MPTCP connection, we also remove all the data related to the connection and the two hosts involved.

As mentioned earlier in this section, MPTCP itself uses the tokens in order to recognize which sub-flows belong to which connection. The reason for this is that, due to middle boxes such as NATs, the addresses that are advertised might not correspond to those that are observed during Joins. Since we are not computing the tokens at this time, this means that the script cannot function in the event where such modifications are taking place. While using the tokens would be an improvement, it is important to remember that they are not enough on their own. Indeed, tokens are meant to be locally unique. However, since Bro observes traffic to and from many hosts, it is not guaranteed that the use of tokens will be collision-free.

5.2 Detecting Erroneous Address Advertisement

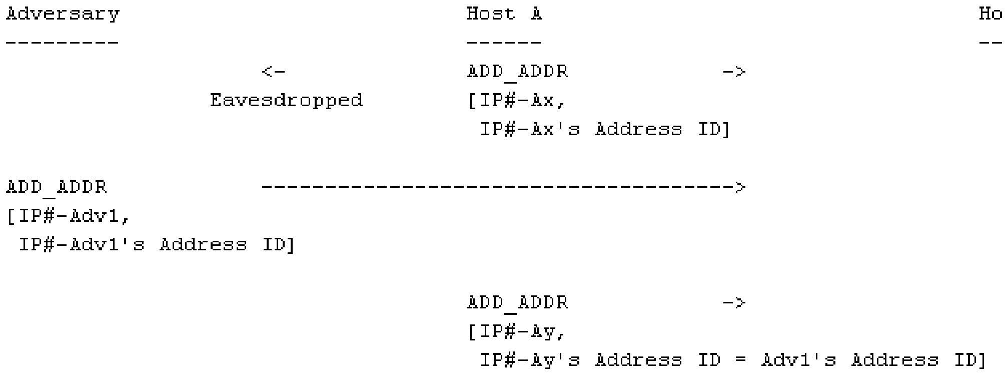
As explained in chapter 2, an MPTCP host maintains, for each one of its MPTCP connections, a mapping of its addresses and those of the other host to address IDs. When a new address is advertised, the mapping used by its owner is sent along in the Add Addr option. The address ID is used in place of the actual address for address removal, changes in path priority, and most importantly for joining connections. Indeed, due to NATs (Network Address Translators) and other middle boxes along the paths, the address in the packet header of an MP Join might not correspond to an address that was advertised. The address ID is therefore used to specify the address unambiguously. Our next task will be to detect if address advertisement goes wrong. Within a given MPTCP connection, an address ID should map to one and only one address/port pair. If we receive an Add Addr option with a already used ID, there are three possible cases:

1. The new address is the same as the existing one and the port is different. In this case, the host is changing the advertised port which is authorized.
2. The Add Addr is a duplicate, advertising the same address and port on a given ID. This is normally ignored.



```
114 33927→80 [ACK] Seq=1 Ack=1 Win=28672 Len=0 TSval=3606236 TSecr=2864806595
102 [TCP Dup ACK 5#1] 33927→80 [ACK] Seq=1 Ack=1 Win=28672 Len=0 TSval=3606236 TSecr=2864806595
114 [TCP Dup ACK 5#2] 33927→80 [ACK] Seq=1 Ack=1 Win=28672 Len=0 TSval=3606236 TSecr=2864806595
```

Figure 5.2: A Three-way handshake ACK followed by two Add Addr duplicates



```
Adversary                          Host A                                  Host B
---------                          ------                                  ------
                   <-              ADD_ADDR           ->
              Eavesdropped         [IP#-Ax,
                                    IP#-Ax's Address ID]

ADD_ADDR           ------------------------------------->
[IP#-Adv1,
 IP#-Adv1's Address ID]

                                   ADD_ADDR           ->
                                   [IP#-Ay,
                                    IP#-Ay's Address ID = Adv1's Address ID]
```

Figure 5.3: Possible Add Addr Attack

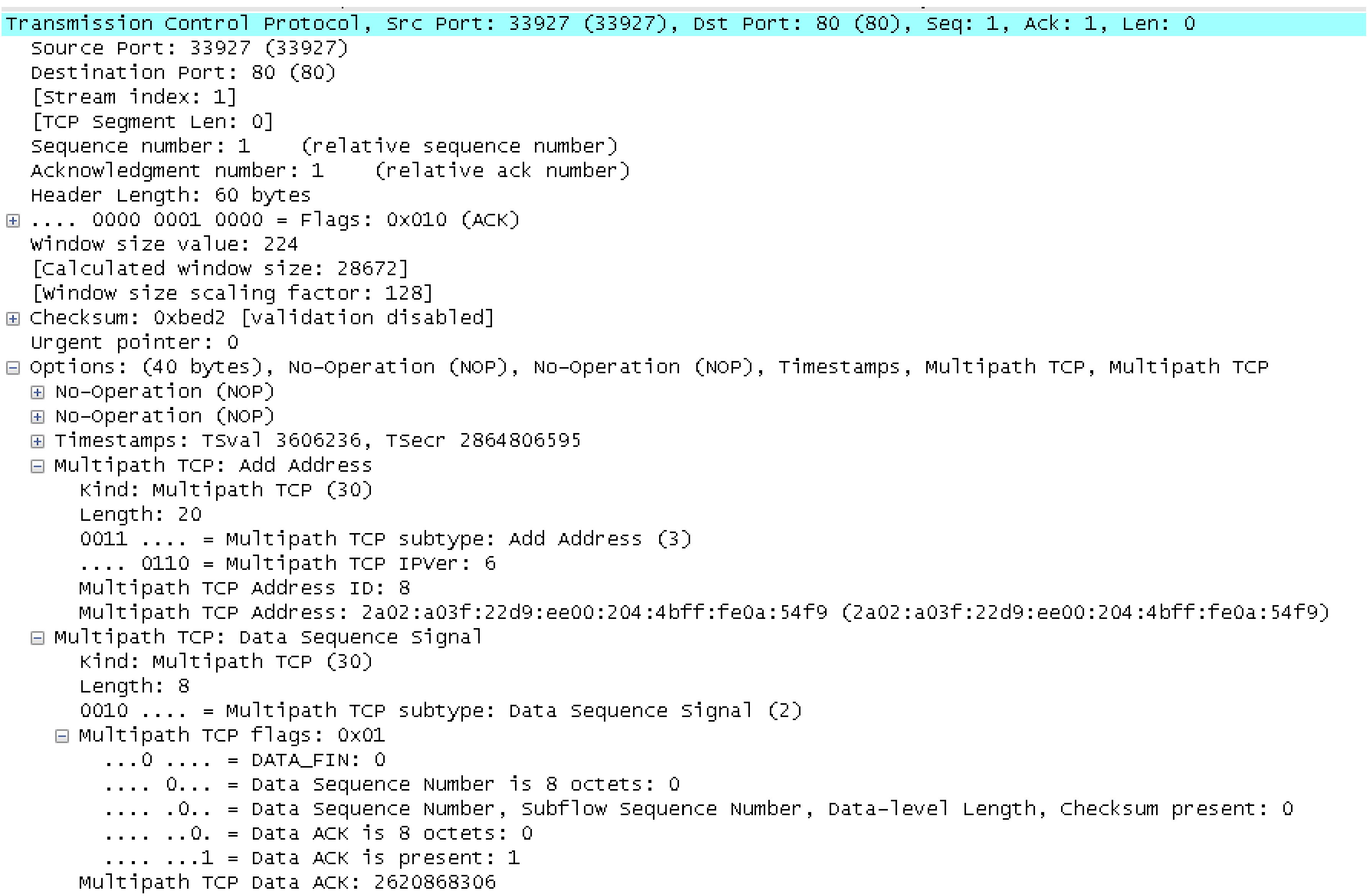
3. The new address is different, leading to two addresses mapping to the same ID. This is not allowed and is the behavior we will want to detect.

When using a correct MPTCP implementation, this third case should not happen. Detection of this behavior may therefore be and indicator of an attack. For example, an attacker may have eavesdropped on an MPTCP connection and would be attempting to advertise his own addresses as being available for the connection while pretending to be one of the two legitimate hosts. While the utility of doing this may be questionable, the possibility is real. Figure 5.2 shows three packets captured with Wireshark. The first is the final ACK of a connection establishment. The following two packets, while seeming like duplicates, are actually address advertisements. Figures 5.4 and 5.5 show the detailed TCP headers of both these packets. It is easy to see that, if an attacker were to sniff one, it would be easy to forge the second. The only variations are the option length (and only because one is an IPv4 address and the other is IPv6), the TCP checksum (easy to recalculate), and the advertised address and ID itself. Figure 5.3 shows a potential scenario where an adversary observing and Add Addr would send his own advertisement as a forged duplicate of the previous packet. Should Host A ever try to perform a legitimate Add Addr on the ID the adversary chose, the IDs would clash, allowing for detection. A possible application of this attack would consist of an attacker filling up all 254 available address IDs (8 bits, ID 0 used by the initial subflow and at least one advertised in the sniffed packet), thereby making it impossible for the connection to establish more subflows.

Should this ever happen on our network, we would therefore want to be notified about it. Unlike the logging of MPTCP connections we saw in the previous section which aims at logging normal behavior, this time we are interested in abnormal cases. For an immediate response, we will therefore make use of the Notice framework described in section 3.4.2. After adding our new notice type to the framework, we go back to the `mp_add_addr` handler, unsurprisingly. We have already used this handler to populate our address table. The first step to that process was getting the ID of host which was advertising the address. We can use this ID to lookup the host record itself in the host table. As we mentioned in the previous section, the host record maintains the mapping between its known addresses and their address IDs. This is exactly what we need, and now is the time to use it.

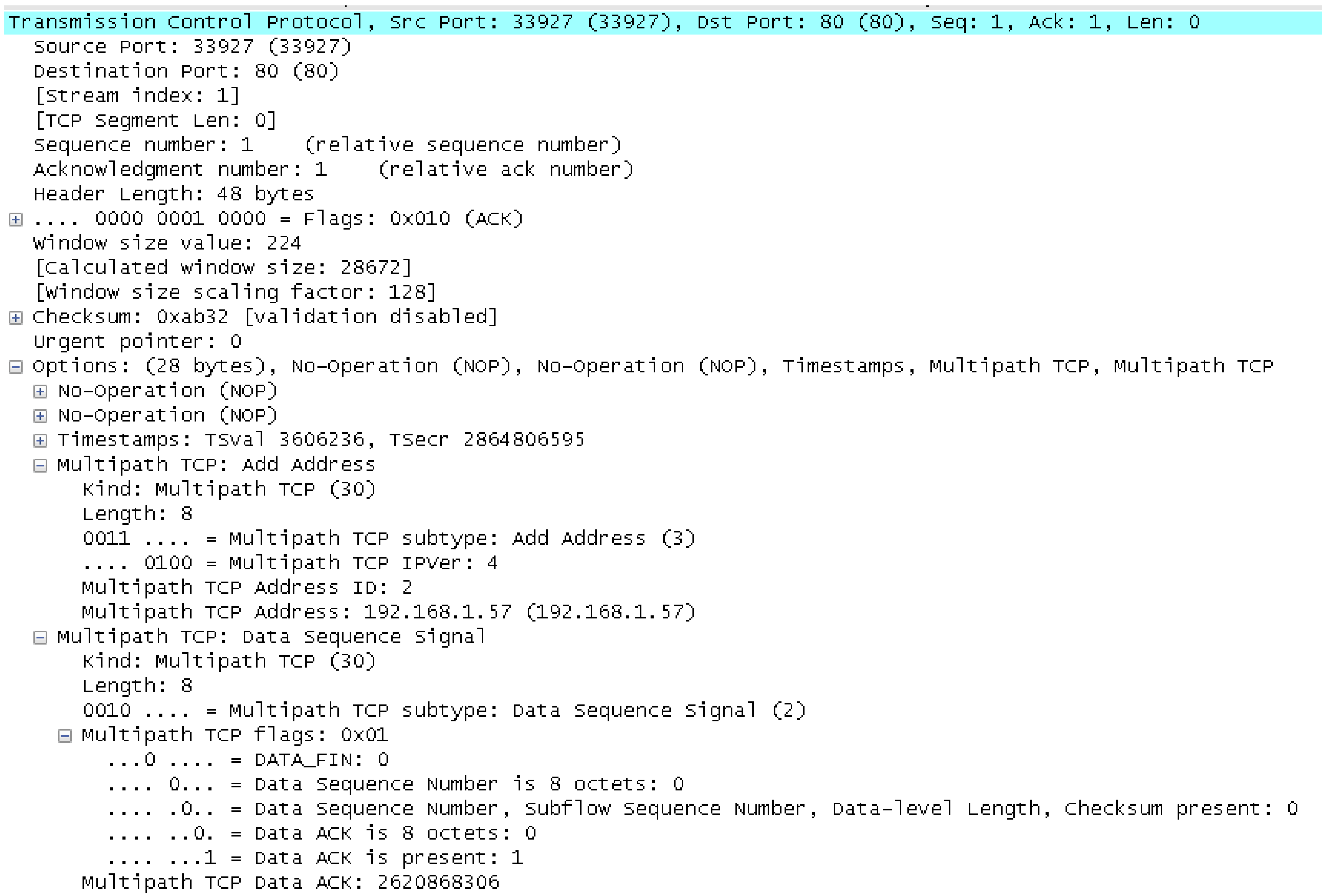
The `mp_add_addr` contains the address ID that is supposed to be used. If this ID is already present in the hosts address table, then we have to check the address itself. Once again, illegal behavior is advertising a different IP address on an existing ID, if this is the case, we raise a notice. We opted for a simple notice, containing only the note, a message, and the connection value. If the address and port number are the same as those already known by the host, a notice is also raised with a different message. Again, this is authorized, but it is redundant behavior because the receiver is supposed to ignore it.

The last step of creating a notice is the actual reaction to said notice. We therefore write a hook for the notice framework. For each notice that is raised, we check the note to match it with the kind we are interested in. When we encounter the `Duplicate_add_addr` type, we describe the action to take. In the context of this work, we have simply instructed the framework to take no action, and are satisfied simply be printing the message to the standard output. Off course, this is easily modifiable should an actual attack with this method be discovered, and any other script may opt into the notice framework to add additional responses to it.



```
Transmission Control Protocol, Src Port: 33927 (33927), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 0
  Source Port: 33927 (33927)
  Destination Port: 80 (80)
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 1    (relative sequence number)
  Acknowledgment number: 1    (relative ack number)
  Header Length: 60 bytes
⊞ .... 0000 0001 0000 = Flags: 0x010 (ACK)
  Window size value: 224
  [Calculated window size: 28672]
  [Window size scaling factor: 128]
⊞ Checksum: 0xbed2 [validation disabled]
  Urgent pointer: 0
⊟ Options: (40 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps, Multipath TCP, Multipath TCP
  ⊞ No-Operation (NOP)
  ⊞ No-Operation (NOP)
  ⊞ Timestamps: TSval 3606236, TSecr 2864806595
  ⊟ Multipath TCP: Add Address
      Kind: Multipath TCP (30)
      Length: 20
      0011 .... = Multipath TCP subtype: Add Address (3)
      .... 0110 = Multipath TCP IPVer: 6
      Multipath TCP Address ID: 8
      Multipath TCP Address: 2a02:a03f:22d9:ee00:204:4bff:fe0a:54f9 (2a02:a03f:22d9:ee00:204:4bff:fe0a:54f9)
  ⊟ Multipath TCP: Data Sequence Signal
      Kind: Multipath TCP (30)
      Length: 8
      0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
    ⊟ Multipath TCP flags: 0x01
        ...0 .... = DATA_FIN: 0
        .... 0... = Data Sequence Number is 8 octets: 0
        .... .0.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 0
        .... ..0. = Data ACK is 8 octets: 0
        .... ...1 = Data ACK is present: 1
      Multipath TCP Data ACK: 2620868306
```

Figure 5.4: The first Add Addr in detail



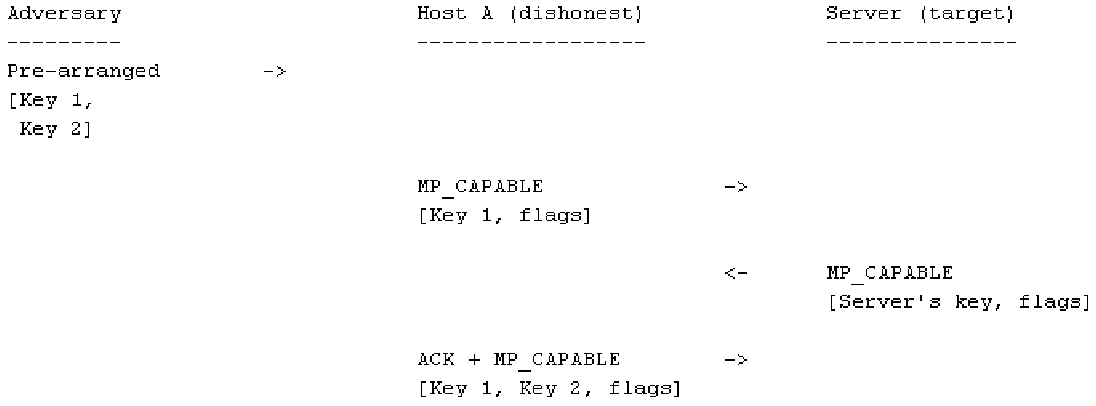
```
Transmission Control Protocol, Src Port: 33927 (33927), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 0
  Source Port: 33927 (33927)
  Destination Port: 80 (80)
  [Stream index: 1]
  [TCP Segment Len: 0]
  Sequence number: 1    (relative sequence number)
  Acknowledgment number: 1    (relative ack number)
  Header Length: 48 bytes
⊞ .... 0000 0001 0000 = Flags: 0x010 (ACK)
  Window size value: 224
  [Calculated window size: 28672]
  [Window size scaling factor: 128]
⊞ Checksum: 0xab32 [validation disabled]
  Urgent pointer: 0
⊟ Options: (28 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps, Multipath TCP, Multipath TCP
  ⊞ No-Operation (NOP)
  ⊞ No-Operation (NOP)
  ⊞ Timestamps: TSval 3606236, TSecr 2864806595
  ⊟ Multipath TCP: Add Address
      Kind: Multipath TCP (30)
      Length: 8
      0011 .... = Multipath TCP subtype: Add Address (3)
      .... 0100 = Multipath TCP IPVer: 4
      Multipath TCP Address ID: 2
      Multipath TCP Address: 192.168.1.57 (192.168.1.57)
  ⊟ Multipath TCP: Data Sequence Signal
      Kind: Multipath TCP (30)
      Length: 8
      0010 .... = Multipath TCP subtype: Data Sequence Signal (2)
    ⊟ Multipath TCP flags: 0x01
        ...0 .... = DATA_FIN: 0
        .... 0... = Data Sequence Number is 8 octets: 0
        .... .0.. = Data Sequence Number, Subflow Sequence Number, Data-level Length, Checksum present: 0
        .... ..0. = Data ACK is 8 octets: 0
        .... ...1 = Data ACK is present: 1
      Multipath TCP Data ACK: 2620868306
```

Figure 5.5: The second Add Addr in detail

5.3 Detecting Key Modifications

In many cases, it is desirable for a server, which usually responds to connection attempts, to not maintain state about connections until they are actually established. This is notably the case to mitigate SYN flooding attacks; the server does not memorize all the incoming connections which were never intended to be used. The MPTCP connection establishment was designed with this in mind. The fact that the final ACK of an MP Capable three-way handshake contains both the initiator and the receiver's keys is so that the server does not need to remember his own key until the connection is established. As stated in RFC 6824 [1]: "B's Key is echoed in the ACK in order to allow the listener (Host B) to act statelessly until the TCP connection reaches the ESTABLISHED state."

However, this could tempt an attacker into trying to modify the keys for his own gain. The keys are actually sent in clear so in most cases modifying them has little use when sniffing them is sufficient. However, we could imagine convoluted scenarios where the adversary would be unable to do so. If he were to have the cooperation of a dishonest client, the client could establish a connection with the target server and share the keys. If, for another convoluted reason, this is impossible as well, they could agree on a pair of keys to be used in advance. The client would then start the connection with the server, receive the server's key, and replace it with the pre-arranged key on the final ACK. Figure 5.6 shows this scenario.



```
Adversary                  Host A (dishonest)           Server (target)
---------                  ------------------           ---------------
Pre-arranged      ->
[Key 1,
 Key 2]

                           MP_CAPABLE           ->
                           [Key 1, flags]

                                                <-      MP_CAPABLE
                                                        [Server's key, flags]

                           ACK + MP_CAPABLE     ->
                           [Key 1, Key 2, flags]
```

Figure 5.6: Possible Key Change Attack

Unfortunately for our adversary, the authors of the RFC did take this into consideration, the rest of the paragraph reading: "If the listener acts in this way, however, it MUST generate its key in a way that would allow it to verify that it generated the key when it is echoed in the ACK.". An implementation conforming to the standard should therefore be equipped to deal with this situation, probably by ignoring it, and of course no one would dare to forget implementing a single guideline of the standard. However, a network operator may still desire to be informed that someone made a futile attempt at breaking the protocol. Let us use the notice framework once again to signal this behavior.

To handle the problem, we will create a new `Pending_conn` record composed of two booleans and two counts. The booleans are true if the originator's (resp. responder's) key was seen during the original SYN (resp. SYN+ACK). The counts each hold one of the host's key. We will also need an additional table which will map a connection (identified by the address/port four-tuple) to the related `Pending_conn`. The issue is confined to MP Capable options only. After adding the new notice type (MP_key_change) to the framework, we will therefore head back to the `mp_capable` event handler. When receiving an MP Capable SYN, we will create a new entry in the pending table, containing Host A's key. When we receive the SYN+ACK, we must check whether an entry for the connection already exists (it might not if Bro did not see the SYN packet). If the entry exists, we add Host B's key to it, otherwise we create a new entry with B's key only. Finally, when we receive the final ACK, we check to make sure the keys received match those in the table. We only check the values we have seen since, again, Bro may have missed certain packets. Should one or both keys be different from what was expected, a notice is raised. In any case, the pending entry is deleted for memory management reasons.

As in the previous section, the notice hook will not perform any action other than printing out the message.

5.4 Unknown Join Detection

The notice for unknown joins deserve a brief mention as it might not do what one might expect. In its current form, it is raised when a Join of length 16 (SYN+ACK) is seen, but the destination address does not belong to any known MPTCP connection in Bro's memory. The important thing to remember is that it is therefore raised when the responder is replying, indicating that both hosts using the subflow appear to know it, even if Bro does not.

That being said, the notice does serve a certain purpose since it indicates that some address advertisement was obviously missed. Usually, this will be due to one of three scenarios:

1. The address advertisement was done on a path that does not go through the IDS. In this case, it might indicate a poor design of the network from a security point of view.
2. The packet header was modified along the path, making the destination address

observed in the Join different from the one that was advertised by the destination. This implicitly reveals the presence of middle boxes along the path.

3. The advertisement was done before Bro was launched or was not captured if using a packet capture.

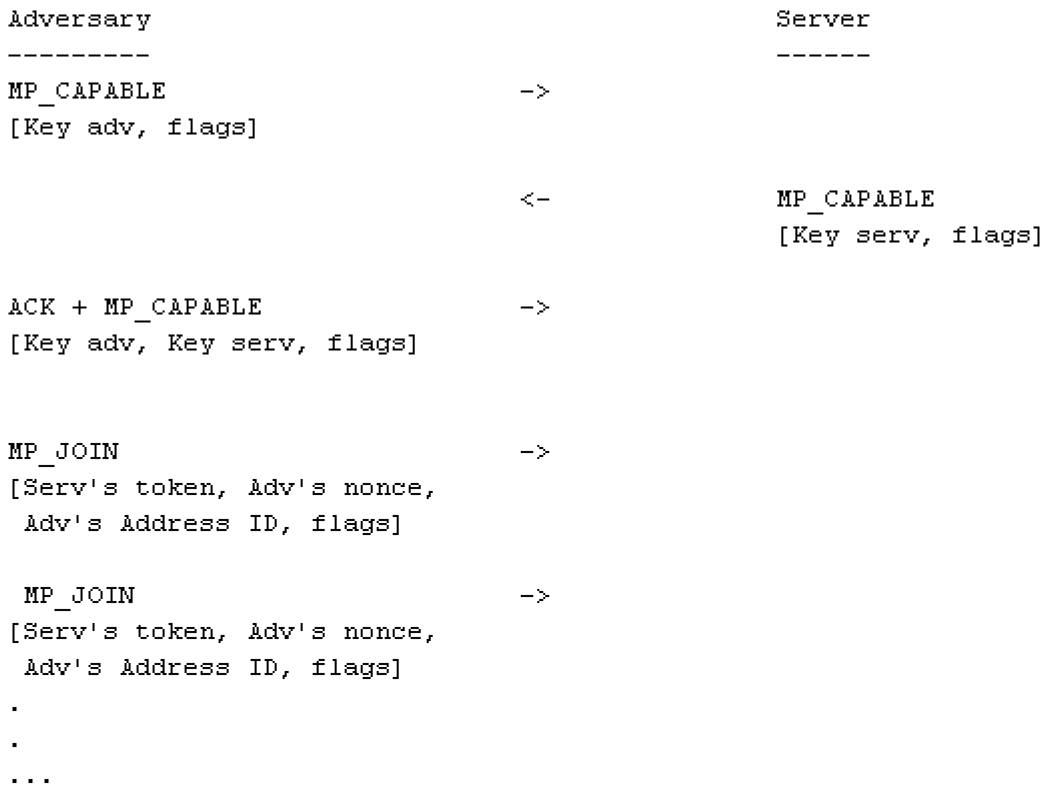
5.5 Join Flooding

The initiator of a join first sends the token identifying the connection he wishes to join, along with a random number used for authentication. The receiver must reply with his own random number, and more importantly, a HMAC (truncated to 64 bits) of the two random numbers. This means that the receiver is actually the first host which has to perform a cryptographic operation. The token serves as a form of protection against flooding the server with join demands. Indeed, if a host receive a join SYN with a token it does not know, it should reply with an rst and no calculation is performed. This is stated in the standard [1]: “Although calculating an HMAC requires cryptographic operations, it is believed that the 32-bit token in the MP_JOIN SYN gives sufficient protection against blind state exhaustion attacks”.

However, figure 5.7 shows a potentially problematic scenario. Here, we see an adversary begin a legitimate MPTCP connection, obtaining a regular key allowing him to compute the token for this session. Later, perhaps after the server has advertised a second address, the adversary begins flooding the server with Join attempts. His token is valid, forcing the server to compute the HMAC. Of course, the adversary need not reply to these HMACs since he does not intend to use the new subflows.

Should this attack be possible, could we detect it? A typical IDS would probably detect this attack as a standard SYN flooding since fundamentally, it is one. However, we are still interested in detecting for what it really is. The first reason is simply to detect MPTCP potentially playing a part in an attack. The second is that the attack is asymmetrical in the sense that a small request causes a heavy computation. An attacker might purposefully lower the sending rate to the point where an automatic SYN flooding detection would not register it, while still sufficiently disturbing the target due to the cryptographic operations.

We therefore propose a simple method to detect excess MP Join messages. For every receiver we see, we count the number of MP Join SYN packets he gets. If the number exceeds a set threshold, we raise an `MP_join_flood` notice. At set time intervals, all the counters are reset. The notice that is raised is slightly more complex than those we have used up until now. In addition to the traditional note, msg, and conn fields, we add an



```
Adversary                                        Server
---------                                        ------
MP_CAPABLE                   ->
[Key adv, flags]

                             <-                  MP_CAPABLE
                                                 [Key serv, flags]

ACK + MP_CAPABLE             ->
[Key adv, Key serv, flags]


MP_JOIN                      ->
[Serv's token, Adv's nonce,
 Adv's Address ID, flags]

 MP_JOIN                     ->
[Serv's token, Adv's nonce,
 Adv's Address ID, flags]
.
.
...
```

Figure 5.7: Potential Attack on the Join Authentication

identifier and a suppress_for value. Together, these attributes will ensure that the notice for a given address is only raised once per time interval. This is needed because, once a SYN causes the counter to exceed the threshold, every other SYN thereafter would also cause the notice to be raised.

This method of detection is similar to that used by the TCP SYN flood detection script that shipped with the early versions of Bro [10]. It was adapted to run smoothly on a packet-trace given that the real-time rate at which packets arrive is usually a key factor in such an attack. Another issue with both the original TCP and this MPTCP detection methods is the use of a static threshold.

Chapter 6

Testing

With several types of behavior now being detected, the next step is to validate that the scripts function correctly and without impacting the execution too heavily. This section will describe the different testing methods we employed to make sure the system behaved correctly.

6.1 Testing the Event Engine Modifications

The first step towards validating the system was verifying that the new events themselves worked as intended. In order to do this, we created a small auxiliary script which would have a very basic functionality: every time an MPTCP event is seen, it prints out the event type and all the values contained in the event to the standard output. Next, we created a short packet capture in which we visited an MPTCP-enabled website: `https://amiusingmptcp.com/` [7]. We checked the capture with Wireshark [5] which quickly showed us that the capture contained enough MPTCP behavior to run some tests. The capture contains two MPTCP connection establishments, each one of which advertise several other addresses on which new subflows are created, using both IPv4 and IPv6.

We then ran Bro on our new capture, specifying that our script should be loaded. the result was as expected, with the contents of the MPTCP options being printed out to the console. In order to verify that the content was correct, we performed a side-by-side comparison of the values that Bro output, and those visible via Wireshark.

6.2 Testing the Logging

In section 5.1, we describe how we wrote a script which would log all the TCP connections that used MPTCP while matching subflows to their respective MPTCP connection. This script was tested in two phases. First, we ran it over the same capture we used to test the events. The number of packets and connections was small enough that we could use Wireshark to compare the output log to what we saw hands-on. By following the capture packet by packet, we saw which connections were established with MP Capable, making them initial subflows for their MPTCP connections. The first TCP flow that was established with a MP Join happened before the second MPTCP connection was established, allowing us to easily view the token corresponding to the first connection. From then on, we could count the flows by hand and make sure each one was attributed to the correct MPTCP connection in the log.

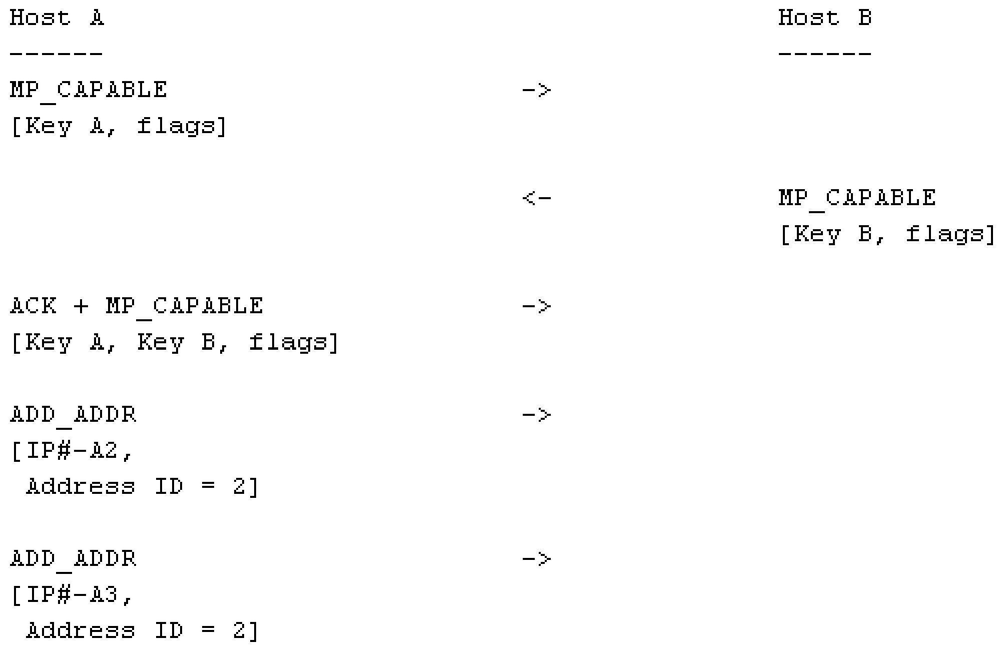
We also used this occasion to make sure the data structures we were using were being emptied out as the analysis progressed by periodically printing out the content of the tables.

Once we had ascertained that the log satisfied our requirements, we ran the script both on a large packet capture containing visits and downloads from multiple TCP-enabled websites, and on a live Ethernet interface while browsing the web. In both cases, we were not expecting to be able to validate that the contents of the log exactly matched what had happened. The test was mainly to make sure no odd behavior, errors, or incorrect log entries appeared.

6.3 Testing the Bad Behavior Detection

The last remaining script functions to test were those making use of the notice framework for the detection and notification of odd behavior. These tests also took place in two phases. First, we ran them over a series of large packet captures, and a live Ethernet capture. The main focus of this test was to make sure the script was not raising any false positives. Real MPTCP traffic is not expected to trigger any of our notices, and therefore nothing should be printed out. There is one exception though: the `Duplicate_add_addr`. As we saw in section 5.2, this notice is raised in two different cases, with a different message depending on the situation. The truly bad case is the advertisement of a new address on an existing address ID, but we also raised a notice when the same address and port pair was advertised on the same address ID. This is a case that is allowed, but where the receiver should normally ignore the duplicate advertisement. This second type of `Duplicate_add_addr` did appear during the first test, but none of the other notices were raised.

The second phase of the testing was to make sure the notices would be raised should the situation ever call for it. In order to test this, we would need packet captures showcasing behavior that only a bad implementation of MPTCP would generate. Since such captures are not readily available, we turned to the packet manipulation program Scapy



```
Host A                                           Host B
------                                           ------
MP_CAPABLE                      ->
[Key A, flags]

                                <-               MP_CAPABLE
                                                 [Key B, flags]

ACK + MP_CAPABLE                ->
[Key A, Key B, flags]

ADD_ADDR                        ->
[IP#-A2,
 Address ID = 2]

ADD_ADDR                        ->
[IP#-A3,
 Address ID = 2]
```

Figure 6.1: Generated Packet Capture for Duplicate Add Addr

[11], and more particularly the MPTCP adaption of the software [9]. Thanks to this program we were able to write short packet captures which contained the specific behavior we were attempting to detect. Due to certain limitations of the program (packets containing certain types of options such as IPv6 ADD ADDR could not be written to pcap files), we limited ourselves to just the behavior we needed, resulting in very short traces containing only a few signaling packets.

For the detection of duplicate address IDs, we started by taking the short trace we used for earlier testing. As mentioned before, this trace contains, along other things, the establishment of an MPTCP connection, the advertisement of several addresses, and the establishment of other subflows that join the connection. We then copied the three packets of the connection establishment, as well as the first address advertisement into a new packet list. Then, we copied the address advertisement into another packet. It advertised an IPv4 address, needed since the program could not print out IPv6 Add Addr options, on the address ID 2. We modified the copy of the packet so that the address would be different. We then let Scapy recompute the TCP checksum and wrote the packet list to a new pcap. The result was a 5 packet capture in which we establish an MPTCP connection, then advertise two different IPv4 addresses on the same address ID. When the script was run against this capture, a notice was raised and printed out the correct ID and addresses. Figure 6.1 illustrates the content of the test trace.

To verify whether our key change detection was working, we once again took our short trace. This trace was easier to create, we simply copied the SYN and SYN+ACK of the first connection establishment into a new packet list. Next, modified the two keys of the ACK packet and added our modified ACK to the packet list. We then wrote the three packets to a new capture. As in the previous case, the corresponding notice was raised when we ran the script on our generated trace, printing out both the old and new keys. The contents of our modified trace basically correspond to the interactions between Host A and the Server from figure 5.6.

Next, we move on the the Join flooding detection. In this case, we actually need a large trace in order to simulate a flood attack. Using Scapy and our trusty capture once more, we begin by copying one of the existing SYN packets with an MP Join option. Then, enter a loop in which we create a copy of this packet, change the sender address to a random one, recalculate the checksum, and append the copy into a new packet list. We did this 10.000 times, giving a trace with a large number of SYN packets by most standards. The threshold for detecting a Join flood was arbitrarily set at 1000 SYN's over 10 seconds. When we ran our script over the fabricated trace, the entire execution lasted less than the time interval, meaning we got 10.000 connection attempts within a single period. The notice was raised after the first 1000, as expected, and it was only raised once thanks to our suppression of the notice.

6.4 Performance Testing

The last step for the evaluation of our implementation is performance testing. For reproducible test, we limited the study to packet captures. Given that the engine is not limited by the speed at which the network delivers packets in this case, the execution time is not really representative of a real deployment. Therefore we focused on peak memory usage. Evidently, we want to compare the memory usage of Bro with and without our scripts to see if the generation and handling of event negatively impacts the system's performance.

We ran two test cases: in the first, Bro is executed on the Join flood packet capture, containing only 10.000 SYN packets. In the second case, we run Bro on a large packet capture with over 400k packets of regular network traffic. For both cases, we ran Bro 20 times while measuring the memory usage with a bash script [12]. Table 6.1 summarizes the results of this test.

The first thing we can observe is that the difference between using the script and not using it is extremely small, and the MPTCP detection is therefore unlikely to cause more performance issues than a standard usage of Bro. This is likely due to the fact that the small amount of additional memory we are using in script pales before the large amount of state maintained by the Event Engine regardless of which events must be generated.

	No Script	With Script	Diff
Flood pcap (10k packets)	142.207	142.117	-0,06%
Large pcap (400k+ packets)	90.925	91.982	+1,16%

Table 6.1: Average Peak memory [KB] usage over 20 iterations

The second observation we can make is that, both with and without the script, the flood pcap cause a much higher peak memory usage despite being much smaller than the other capture. Once again, this points to the fact that the Event Engine is the main memory user. Regardless of whether we are looking for a flood attack, the Engine will create state for the connection attempts.

Chapter 7

Section Recap

The three previous chapters describe how we have implemented and tested a first method of MPTCP traffic analysis. To do so, we have acted upon Bro's TCP analyzer to have it produce MPTCP-related events. These events correspond to the different kinds of option sub-types used by MPTCP. We then wrote scripts that used our new events in order to detect and log MPTCP behavior over the network. These scripts include connection logging and the detection of certain anomalies that might occur in the protocol's operation. However, this solution presents certain issues which we will address.

A first problem is that MPTCP hosts exchange and compute information through cryptographic functions. The session tokens, for example, are obtained by hashing the exchanged keys while the authentication of Joins is done via a keyed-hash. As we were unable to find any cryptographic libraries implemented for Bro's scripting language, performing these computations would be difficult in the current situation. Since `mp_capable` events only transmit the keys to the script layer, it is hard to find the associated token. This hints at the fact that the cryptographic operations that we need to re-perform in Bro should be done within the Engine.

More importantly, however, the current solution does not allow the Bro Engine to observe any links between different sub-flows. This causes a number of problems that render the solution incapable of performing certain important tasks, chiefly, reassembly. As mentioned in chapter 3, Bro passes the packets of a given connection through a hierarchy of protocol analyzers in order to understand all the behavior taking place in it. While the TCP analyzer can essentially make do with the packet header, certain protocol analyzers need to observe the data stream that is exchanged. In order to do this, Bro is capable of reassembling an out-of-order TCP stream in order to pass it along the the child-analyzers so that they can work on a coherent stream. Since Bro's Engine still considers sub-flows as independent streams, the child analyzers will only receive part of the data and will, in most cases, be unable to perform their tasks.

Therefore, simply adding MPTCP-related events to the engine is not sufficient. This solution only allows us to detect the behavior of MPTCP, but gives us no information on the data that it is being used to send. Though we could potentially perform this reassembly in script, it is simply too late; the reassembled stream cannot be passed back to the Engine to have the child analyzers work on it.

To cope with this issue, the following chapters will describe the requirements that must be met in order to allow Bro to perform the reassembly of MPTCP traffic within its Engine, in order to pass the reassembled stream to the subsequent analyzers.

Chapter 8

A closer look at Bro's Reassemblers

It is now clear that, with reassembly playing such an important part in the analysis of packets' payloads, it is necessary to allow Bro to reassemble data from multiple streams within the Engine. In order to assess what is necessary to achieve this, we will now examine in detail how Bro performs reassembly of regular TCP. To do this, we will explain how Bro manages individual data streams, then how it handles reassembly on streams using TCP.

8.1 Identifying Streams

As we have already mentioned, Bro takes a stream-oriented approach at analyzing traffic. This means that all the packets that belong to the same connection are processed together as a single coherent data stream. In order to do this, Bro goes through a number of steps in order to uniquely identify all the connections that it sees. Since the software aims at inspecting the payload, information that is purely routing-related is mostly ignored. To this end, removing tunneling information and MPLS labels is one of the first steps. Once this is done, Bro can then move on to the next layers of the packet. In the case that interests us, Bro will then assess that the packet uses either IPv4 or IPv6. It will then perform reassembly of fragmented IP packets, if necessary, before moving on to the transport layer.

By continuing the parsing of the packet headers, Bro will split up incoming packets according to the transport protocol in use. Once it is determined, the Engine can properly extract the Addresses from the IP header and, in the case of TCP, the port numbers. Up until this point, Bro does not maintain any long-term state about the packet, simply relying on temporary storage such as buffers for the de-fragmentation. With the four address and port values available, Bro will now start uniquely identifying connections. Connections are represented by a complex Conn data structure containing information about the stream, as well as being in charge of maintaining the analyzer tree that will be used and performing the subsequent analysis of the connection's traffic. The connection manager keeps dictionaries containing all the Conns of a given protocol. We create a Conn ID structure containing the four-tuple along with a boolean indicating the direction of the connection, and the hashed value of this data structure serves as a key in the dictionaries. It is important to note that Bro therefore limits the distinction of different connections to these five values.

For each incoming packet, the Conn ID is calculated and hashed, and the resulting key is used in a lookup in the corresponding dictionary. If a match is found, the packet is treated by the match's Conn structure. If the dictionary does not contain an entry for the key, a new Conn is created and added. It is also at this point that the connection's analyzer tree is initialized. In our case, the TCP analyzer is selected as root, given that this is the transport protocol in use. Each connection will dispose of its own instantiation of the required analyzers. To complement the analyzers, a PIA is also initialized. The PIA is the mechanism responsible for dynamically detecting which application-layer protocols are used within the connection, and will therefore build or prune the analyzer tree as the connection progresses. However, the root analyzer will not change, given that the connection will keep using TCP throughout its lifetime. The child analyzers do not concern us and we will therefore not go into detail on how the PIA works.

At this point, we have a Conn structure unique to the connection using the address, port, and direction 5-tuple. This Conn is stored within the TCP connection dictionary of Bro's session manager, allowing future packets from the connection to be matched straight to it. The structure also contains an analyzer tree rooted at a TCP analyzer. This TCP analyzer is where we performed modifications during our first attempt. In order to go further, we will now see how this analyzer reconstructs the TCP stream that it is in charge of.

8.2 Bro's TCP Endpoints and Reassemblers

Bro's TCP analyzers use several supporting classes in order to process TCP traffic both efficiently, and in a way similar to how the actual end hosts proceed. One such component is the TCP Endpoint. When a new analyzer is instantiated, it immediately creates two TCP Endpoints which each represent one of the two hosts involved in the connection. Since each host has its own parameters and receives a different data stream, this is a natural way to proceed. In practice, each TCP Endpoint's primary function is to keep track of the host's state with regards to the connection. It will, for example, keep track of whether or not the host considers the connection established or terminated, as well as the sequence and acknowledgment numbers that it has seen and is expecting.

Since the TCP Endpoints are already tracking the sequence and acknowledgment

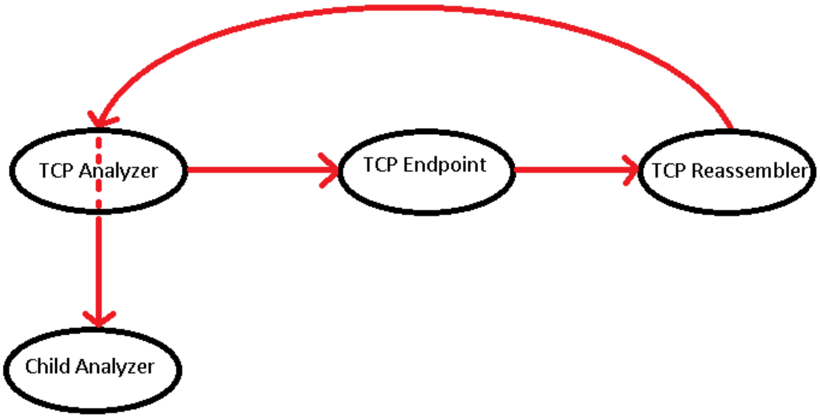


Figure 8.1: Data Path for reassembling TCP Data

numbers, this is the ideal place to perform reassembly. Additionally, each host's traffic must be reassembled separately and the distinction is already done at this point. Therefore, the TCP Reassemblers are added as elements of the TCP Endpoints. The TCP Reassembler itself will be in charge of ensuring that only contiguous data blocks are sent to the analyzer's children.

Now, whenever a packet arrives on the connection, the TCP analyzer will first determine which of its two endpoints sent it and to which one it is destined. At every point during the analysis of the TCP header, the analyzer can now notify its Endpoints of the changes that occurred. SYN or FIN flags will update the corresponding Endpoint's state. Acknowledgments and TCP payloads are explicitly transferred to the receiving Endpoint as well, allowing the corresponding values to be updated. The Endpoint itself does little with these last two elements, but in turn transfers them to its Reassembler. The Reassembler will then compare the incoming data's starting point to the data it has already delivered. If a gap of data is detected, the payload is buffered until the gap is filled. At that point, all the newly contiguous data can be delivered. The data itself is actually re-sent to the TCP Analyzer. Since the analyzer has already processed that data for any TCP-related information, it simply forwards it to its children analyzers. Figure 8.1 shows how the data travels through the TCP analyzer to the different components of reassembly.

Chapter 9

Requirements for MPTCP Reassembly

Now that we've gone over how Bro performs reassembly for regular TCP traffic, we must now adapt the mechanism to process MPTCP traffic. This section will describe which components are needed in order to achieve this objective. As we will see, these changes can be brought about in a variety of ways, and can be performed at various stages of the Engine's operation. Going forward, we will use the following assumptions:

- MPTCP traffic that we attempt to reassemble behaves correctly. If ever the protocol is misused, we will rely on scripts such as those developed in the first iteration to deal with it.
- Identifying MPTCP connections by the tokens is sufficient. As we've mentioned before, the tokens are not meant to be globally unique and may collide with traffic from multiple hosts. We will, however, consider them sufficient for the time being.

As a reminder, we saw in chapter 7 that all modifications must be made within the Engine to ensure the reassembled data can be processed by the full analyzer tree.

9.1 Maintaining an MPTCP Connection List

In order to reassemble data from multiple TCP subflows into a single MPTCP flow, it is first necessary to ensure that Bro is able to recognize which flows belong together. In order to do this, we will have to recompute the mappings from all the hosts whose traffic we see. This process is similar to what we used in the connection logging script of chapter 5. However, since the Engine grants us easy access to cryptographic libraries such as OpenSSL [4] which is already used for md5 hashing (and basing ourselves on the second assumption), we will perform the matching based on the tokens rather than on the address advertisement.

Similarly to what is done for transport layer protocols, we will maintain a dictionary of all the ongoing MPTCP Connections. In addition, we will maintain a second dictionary mapping subflows to their MPTCP connections. The following pseudo-code illustrates how one could populate such dictionaries:

```
if (MP_Capable & SYN+ACK) {
   token1 = mp_token_hash(key1)
   token2 = mp_token_hash(key2)
   mp_conns.Insert(token1, this_connection)
   mp_conns.Insert(token2, this_connection)
   mp_flows.Insert(this_connection, this_connection)
}

if (MP_Join & SYN) {
   connection = mp_conns.Lookup(token)
   mp_flows.Insert(this_connection, connection)
}
```

As we can see, we need to be able to identify which MPTCP connection a subflow is joining; this is made possible by `mp_conns`. The tokens that are found at the connection initiation are used as keys, so that joins can then perform a simple lookup to find the initial connection. Of course, if no result is found during the lookup, then the original connection was not observed and we are limited at attempting partial reassembly or defaulting to standard TCP processing. On the other hand, if the key is already present at the MP_Capable stage, then there is a token collision. If the first connection using this token has been terminated, then Bro's Engine provides us with mechanisms for clearing the dictionary entry. The other case (two different hosts trying to use the same token) is not considered given assumption two. The second dictionary, `mp_flows` is used to match the identifying 5-tuple of the TCP connection to the original MPTCP subflow. This will be necessary down the road, since subsequent packets will not carry the token.

Operations on the other stages of the TCP handshakes (both for MP_Capable and MP_Join) are not necessary. For MP_Capable, all we need are the two keys and 5-tuple which are all present in a single header at the SYN+ACK stage, and this allows us to avoid token computations if the use of MPTCP is refused at an early stage of the handshake. For MP_Join, we have already established that the token is necessary, and it is only present in the initial SYN packet. The rest of the handshake only serves for authentication. In the case of a failed authentication, the stream will not carry any data (and will therefore not affect reassembly) and will quickly be removed by Bro.

From this point on, all packets using TCP must have their 5-tuple looked up in the `mp_flows` dictionary. This has to be done whether or not the packet contains any MPTCP options. Indeed, a data packet does not need to contain a DSS option if a sufficiently large mapping has already been negotiated along its subflow. If a match is found, Bro will be aware that the packet belongs to an MPTCP connection, and not a simple TCP one.

The issue remains of where these operations should be performed. Given that all MPTCP packets will belong to a TCP stream, it is pointless to do the work before the IP addresses and port numbers have been extracted. The earliest time to start identifying MPTCP connections is therefore right before Bro identifies regular TCP connections, in `Sessions.cc`. Since the matching must be done before reassembly can be performed, the latest point is before the TCP analyzer passes the data along to its Endpoints.

The next step is now passing the data from all the subflows along with enough data to properly reconstruct the MPTCP data stream. In any event, further analysis will have two sources of data to rely on: the packet being processed itself (which cannot be modified), and the Conn data structure representing the TCP connection it belongs to.

9.2 Aggregating to a Single Connection

One of MPTCP's fundamental principles is its ability to appear like regular TCP for applications. A logical solution to transfer the data from the subflows to the reassemblers would be to apply the same principle. Upon the establishment of a new MPTCP connection, we create a new Conn structure as usual. However, subsequent subflows will not have their own Conn but will instead re-use that of the original subflow. In this way, all the subflows of a given MPTCP connection will use the same TCP analyzer (and later, the same children analyzers), as well as the same endpoints and reassemblers. To pull this off, the modification should occur before the TCP Conn of the subflows is created, leaving us with a small window (since this is done shortly after the 5-tuple is determined with `Sessions.cc`). In this way, we are attempting to make the reassemblers believe that they are reassembling a single regular TCP stream. In order for this process to work smoothly, however, there are of course a series of modifications to make.

9.2.1 Using the Proper Sequence Numbers

For the reassembler to function correctly, we obviously cannot use the TCP sequence numbers that are found within the TCP headers. Indeed, to obtain a coherent data stream over multiple subflows, one must use the MPTCP sequence numbers. As a reminder, a DSS option contains both the data sequence number (DSN) which covers the entire connection, and a subflow sequence number which is only relevant within the given subflow. Since we want the reassembler to process the whole connection, the ideal situation would be to simply substitute the TCP sequence numbers by the MPTCP DSN.

To do this, however, we must account for the difference in size between TCP sequence numbers (32 bits) and MPTCP DSNs (64 bits, sometimes truncated to 32 in the header). This change must be made throughout all the structures that use the sequence number, meaning the TCP analyzer, endpoints, and reassemblers. However, simply increasing the size of the variable will cause problems with regular TCP analysis, since incrementing beyond 2^{32} will cause differences between what Bro expects and what is seen in the TCP header. To handle this, we must ensure that the structures are able to use either a 32 or 64 bit sequence number. This could be achieved by, for example, extending the Conn value to include an MPTCP flag. If the flag is not present, then the analyzers and its sub-components should truncate the sequence number to the lowest 32 bits before comparing it to what is found in the header.

The next issue is actually getting the DSN value to use. If we keep the idea of using an MPTCP flag in the Conn value, we could imagine simply looking within the MPTCP option whenever we are working with a flagged Conn. However, it is simply not a correct solution given that not all MPTCP data packets will be carrying the DSS option. For this reason, it is clear that we will have to deal with the subflow sequence numbers as well, and keep track of how each subflow is progressing. Each TCP Endpoint will therefore have to maintain a list of the subflows used by the connection, as well as the latest subflow mappings seen for each one. As a result, whenever a packet is processed by the TCP analyzer on an MPTCP-enabled connection, the Endpoints must be updated in a specific way. Even though the Conn ID of the current Conn structure is composed of the 5-tuple of the first subflow, the actual packet's header is not modified during the analysis. Therefore, we can re-extract the tuple and use it to identify subflows within a connection. The TCP analyzer will extract the addresses, ports, direction, TCP sequence number, and (if they are present) MPTCP DSS fields and update the receiving Endpoint with these values. The Endpoint will look up the 5-tuple within the list of known subflows. If there is no match, a new entry is created. If there is one, the lookup should return the last known values for the mapping on that flow.

If the packet that is being processed contains a DSS option, then we can simply pass it to the reassembler along with the corresponding DSN, and update the Endpoint's values. If there is no DSS option, then by comparing the TCP sequence number to the last one seen in the mapping. By using the data level length, we can determine whether or not the data is covered by the mapping. If it is, we can calculate the corresponding DSN and pass it along to the reassembler. If the data is not covered by any known mapping, then the DSS containing the relevant mapping must have been lost or delayed. We must therefore buffer the data until the DSS arrives, at which point we may proceed by forwarding it to the reassembler.

There remains one problem with the sequence numbers seen by the reassembler: since the two first packets of the original flow are treated as regular TCP (before the SYN+ACK), the reassembler will originally see the TCP sequence number. Once the reassembler switches to the DSN, it is almost impossible for the two to match up, leading the reassembler to believe there is a gap. However, the problem is actually solved for us. Indeed, the reassemblers work with relative sequence numbers received from the Endpoints. Therefore, the original TCP sequence number will be 0 and we will switch to the DSN before any data is received. The first DSN will then also be resolved to 0 when converting it to the relative sequence space.

9.2.2 Maintaining Proper State on the Endpoints

Another problem arises when sending multiple connections worth of packets to a single analyzer: the Endpoint state-machine will be seeing multiple establishment and termination handshakes. It is important to deal with this in order to avoid having it crash, or even simply believe the connection is closed too early. Care should be taken to ensure that the state of an Endpoint doesn't revert back to, say, SYN_SENT after being established, simply because a new subflow is being created. However, it is even more important to avoid early termination of the analyzer. Receiving a TCP RST should only remove the corresponding 5-tuple from the list of known subflows, not mark the entire Conn as Reset.

9.2.3 On Working with Distinct Conns

The solution we have presented above sends data from multiple connections to a single Conn structure, essentially "tricking" the Engine into believing it is dealing with a single stream. We may now wonder if it would be possible to reassemble the traffic using multiple distinct Conn structures the way Bro intended it. Actually, the process to obtain a single coherent and ordered data stream would be much the same, and might even be slightly simpler. Each individual analyzer could track its own subflow sequence numbers and send the readjusted sequence numbers to a centralized reassembler.

The issue therefore isn't with reassembly itself, but rather, with what comes next. Bro's original model is that a single IP session is handled by a Conn structure which will apply a series of analyzers on the data to understand a variety of protocols. A Conn is therefore created to treat a whole stream of data from start to finish, and assuming at least one handled application protocol is used, the transport layer analysis represents only half (if not less) of the actual analysis. If we were to use multiple distinct Conn for a single MPTCP connection, we would have multiple Conns producing a single reassembled TCP stream. We could have this stream forwarded to all the Conns, but that would just imply redundant analysis. The alternative is having multiple Conns feeding a single sub-

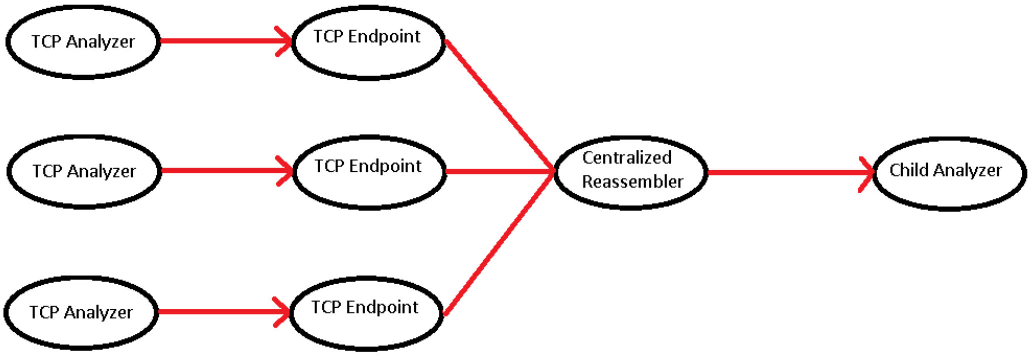


Figure 9.1: Reassembling from Distinct Conns

tree of analyzers, but that is also an inefficient use of resources; essentially, all but one Conn will stop working after only half the work is done while still consuming as much memory as any other. Figure 9.1 shows the data path in this situation.

9.3 Interaction with the Implemented Changes

As we have seen, the original changes made to the Engine do not have any contribution to reassembly. However, we may now wonder whether or not the events created in the first implementation (and the scripts that use them) will still be usable. Unfortunately, while the events would still be raised, most of them contain the Conn ID as a means to retrieve the addresses and ports in the scripts. Since all the subflows now share the same Conn ID, it would be impossible to differentiate their addresses. This could be fixed by reading the address from the header in the Engine instead of relying on the ID when necessary.

However, the addresses are mainly used for the connection logging, which can now be done in a much simpler manner. Indeed, since the subflow matching is now performed in the Engine, it would be much simpler to add new events at that point, indicating when a flow has joined an existing connection. Simply put, while the ”simpler” events can still be used, anything relating to connection establishment should be performed through new, higher-level events raised during the subflow matching.

Finally, it is worth noting that the option parsing of the reassembly cannot be turned off dynamically (depending on the scripts used) as is the case for the MPTCP events. Whether or not we are looking for any MPTCP activity, the Engine must still perform the

MPTCP-related operations in order to ensure that no other analyzer suffers. This also leads to redundant computations in the case where we are processing MPTCP events. In order to cut back on the unnecessary processing, it would be advantageous to raise MPTCP events during the first parsing of the TCP options (in `Sessions.cc`). The issue of course is that the events would be raised at a somewhat unrelated stage of the packet analysis.

Chapter 10

Further Considerations

We have now concluded the discussion of what has been implemented, as well as what would be required for MPTCP stream reassembly. This chapter will discuss what can still be done on the subject in the future. These may include improvements upon existing mechanisms, additional detection schemes, or even entirely new features.

10.1 Implementation of the Reassembly Mechanisms

With a good grasp of how internal reassembly should be implemented, it still remains to be done in practice. The source code pertaining to this thesis available on Github makes some attempts at implementing the first mechanisms. Namely, we have modified the `Sessions.cc` manager in order to perform subflow matching in the Engine, and redirect multiple subflows to the same Conn. However, the computation of the tokens was imperfect leading to a failure to match Join attempts to the correct MPTCP connection.

Examination of the MPTCP source code indicates that the computation of the tokens is done using the Linux Kernel cryptohash implementation of SHA-1 which only includes low-level hashing functions (no automatic padding, only 512 bit blocks). Since Bro already makes use of the OpenSSL library, we attempted to use it as well for the tokens, but we were unable to obtain the same results. Since SHA-1 is a standardized algorithm, we have no doubt this is simply due to an issue of getting the right message to hash. Nonetheless, the implementation was tested with a modified packet trace in which the Join tokens were replaced by those we did obtain.

10.2 Re-Implementing/Adding Higher-level Events and Redesigning Scripts

The last section of this thesis concentrated exclusively on reassembly in order to analyze the application layer protocols. The analysis of MPTCP behavior itself has therefore been neglected, and as we mentioned in section 9.3, some scripts from the first implementation were even rendered unusable. While reassembly is definitely crucial to performing true deep packet inspection (actually understanding the MPTCP payload), it still remains useful to gather data about the protocol's operation itself.

Since a number of events will no longer behave as intended given the lack of Conn distinction, it will be necessary to re-implement a number of events to bring them back to their specification. In addition, we can capitalize on the fact that a large portion of the work which was previously done in script (subflow matching and maintenance of the MPTCP connection list)is no done in the Engine. This enables us to provide events of a much higher-level. These new events would enable us to, for example, perform a more precise MPTCP connection logging with drastically simplified scripts.

Of course, once the final event set is decided upon and implemented, existing scripts may be reworked to take advantage of them. New scripts may also be implemented to detect additional odd behavior

10.3 Increased Discrimination Between MPTCP Connections

As we have stated in chapter 9, we consider the use of tokens to be sufficient to uniquely identify MPTCP connections. We know that a given host may not select keys that hash to the same token for multiple simultaneous connections. However, this guarantee does not hold when we are observing traffic to and from multiple hosts. This could potentially lead to token collisions. Given that we use tokens as keys to look up the MPTCP connection, two Joins from different hosts using the same token would end up being processed together. The tokens are 32 bits long. Obviously, over the global internet, this is not sufficient to avoid collisions. An instance of Bro is unlikely to cover the entire internet, however, so the probability of finding colliding tokens is directly dependent on how large the monitored network is. The Birthday Paradox tells us that the probability of collision remains under 1% with over 9000 hosts each picking one token. Whether this is sufficient will depend both on the security needs and the deployment scale.

In order to further reduce the probability of collision, it would be possible to combine the token-based subflow matching with an analysis of address advertisement (both explicit and implicit) similar to what was done in the connection logging script. In this case, we would only experience a collision if two pairs of hosts managed to pick the same tokens while at the same time using the same addresses and port numbers.

This solution is not without its own issues, however. By using the address advertisements as a criteria for the matching of a subflow, we expose ourselves to the risk of missing one such advertisement. This would result in correct subflows being unable to match to their parent connection. Ultimately, we may find a use for a compromise. We could, for example, store the address advertisement information, but only use it if a potential token collision is detected.

10.4 Partial View Reassembly

For regular TCP, the reassemblers will not deliver any data that would start with a gap. If the connection continues to exchange data until the maximum sending window is exceeded, Bro will raise an error. This indicates that either Bro missed a packet that still made it through to its destination, or that one of the hosts is not respecting the established parameters. In either case, this is not supposed to happen.

If we add MPTCP, however, then there are legitimate situations where this may occur. The simplest case is if Bro has an incomplete view of the network. In this event, two MPTCP hosts might have established a subflow visible to Bro, and another one who's path does not go through the IDS. Bro's reassemblers will be stuck waiting for data that it will never see, while both hosts carry on with the connection without any problems.

From a security point of view, installing an IDS with an incomplete view of the network is a sign of poor design. It should be placed at a choke-point without any traffic being able to go around it. If, for whatever reason, this is not possible or not desirable, we may still want Bro to be able to run as much analysis as possible on the datastream. For this reason, it may be interesting to include mechanisms for delivering reassembled datastreams with missing byte. This could be done when the maximum sending window has been reached but more traffic is still arriving. It is entirely possible that the subsequent analyzers' task will be compromised, but some may yet provide interesting insight.

Chapter 11

Conclusion

This concludes our work on Deep Packet Inspection with Multipath TCP. As we have seen, the development and deployment of a new protocol is not limited to conceptualizing it and implementing it. For a protocol to achieve large scale success on today's internet, compliance with commonly used networking tools is just as important as good design. In addition to the common challenges of creating a new protocol, our overview of MPTCP has shown that this protocol in particular brings about new challenges that break many common assumptions on modern networks.

In an effort to improve existing tools to work with MPTCP, we have presented the Bro IDS, a powerful and flexible tool for network analysis and intrusion detection. By explaining how the system functions, we exposed which parts needed to be worked on in order to allow Bro to understand and produce meaningful results with MPTCP. We discussed the two main components of Bro: the Engine and the Script Interpreter. The Event Engine which parses packets to raise events has been modified in order to transfer the binary MPTCP information contained in the TCP options to the powerful Policy Script Interpreter, and to do so in an easily understandable manner. With the new events being raised, we have been able to produce scripts which can exploit the information in order to showcase important behavior within the protocol. While presenting the scripts, we outlined potential attacks that may be used in the future against the protocol.

We then discussed why these changes were not sufficient. While detecting anomalies in the behavior of the protocol has its uses, it is crucial to allow Bro to run its full set of analyzers on the data stream, and not on bits and pieces. In order to make this possible, we have provided an explanation of how Bro performs TCP reassembly. Once the mechanisms were understood, we endeavored to conceive a method which would allow Bro to reassemble data from multiple subflows into a single, ordered stream. This, in turn, would allow all the existing protocol analyzers to run their course, thoroughly examining the connection's payload.

Through these efforts, we hope to pave the way for a successful MPTCP reassembly implementation within an IDS. The success of such an implementation would enable us to detect threats even if they were split over multiple TCP connections. Being able to do so is an important part of the evolution and continued development of security tools for Multipath TCP.

Bibliography

[1] M. Handley O. Bonaventure A. Ford, C. Raiciu. Tcp extensions for multipath operation with multiple addresses. https://tools.ietf.org/html/rfc6824, 2013.

[2] et al. C. Paasch, S. Barre. Multipath tcp in the linux kernel. http://www.multipath-tcp.org, 2014.

[3] Gregory Detal. Tcpdump network dissector - extended for multipath tcp. https://github.com/multipath-tcp/tcpdump, 2013.

[4] OpenSSL Software Foundation. Openssl. https://www.openssl.org/, 2015.

[5] et al. Gerald Combs. Wireshark. https://www.wireshark.org/.

[6] The Tcpdump Group. Tcpdump network dissector. http://www.tcpdump.org/.

[7] Matt Layher. Am i using mptcp? https://amiusingmptcp.com/, 2013.

[8] Andrei Maruseac. Multipath tcp patch to packet-tcp.c. https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=6705, 2012.

[9] Nicolas Maître. mptcp-scapy. https://github.com/nimai/mptcp-scapy.

[10] Vern Paxson. synflood.bro. http://www.filewatcher.com/p/bro-1.5.1.tar.gz.4327064/bro-1.5.1/policy/synflood.bro.html, 2007.

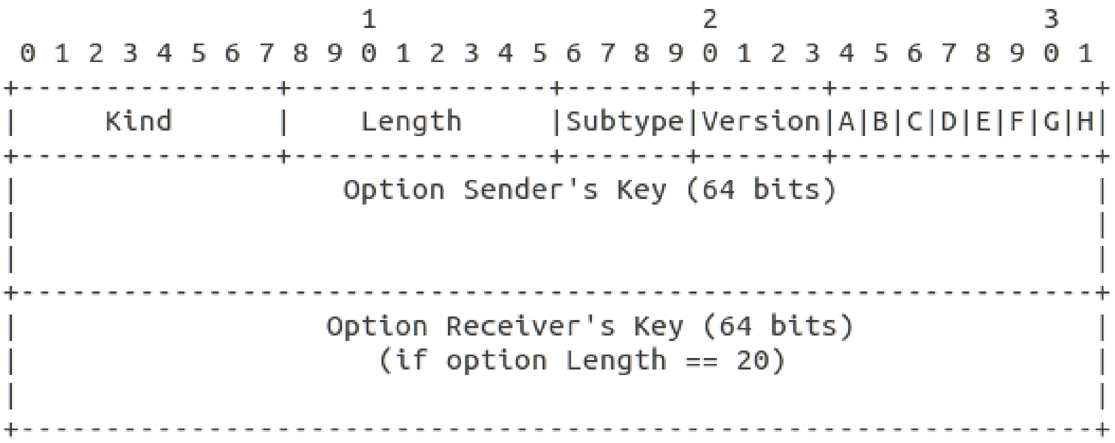
[11] secdev. Scapy. http://www.secdev.org/projects/scapy/.

[12] Jaeho Shin. memusg. https://gist.github.com/netj/526585, 2010.

[13] et al. Vern Paxson, Robin Sommer. Bro. http://www.bro.org.

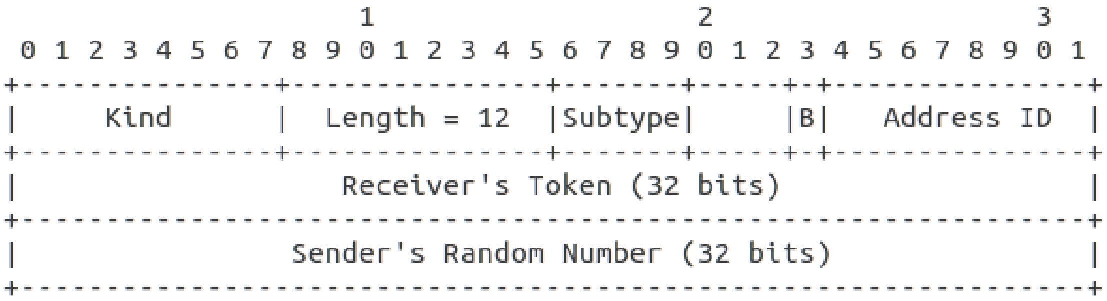
Appendix A

MPTCP Option Formats



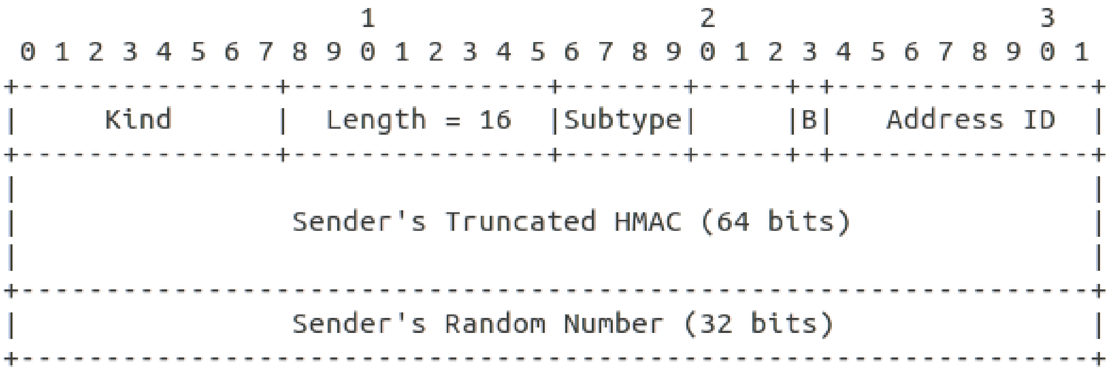
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |    Length     |Subtype|Version|A|B|C|D|E|F|G|H|
+---------------+---------------+-------+-------+---------------+
|                   Option Sender's Key (64 bits)               |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
|                  Option Receiver's Key (64 bits)              |
|                     (if option Length == 20)                  |
|                                                               |
+---------------------------------------------------------------+
```

Figure A.1: MP Capable option format



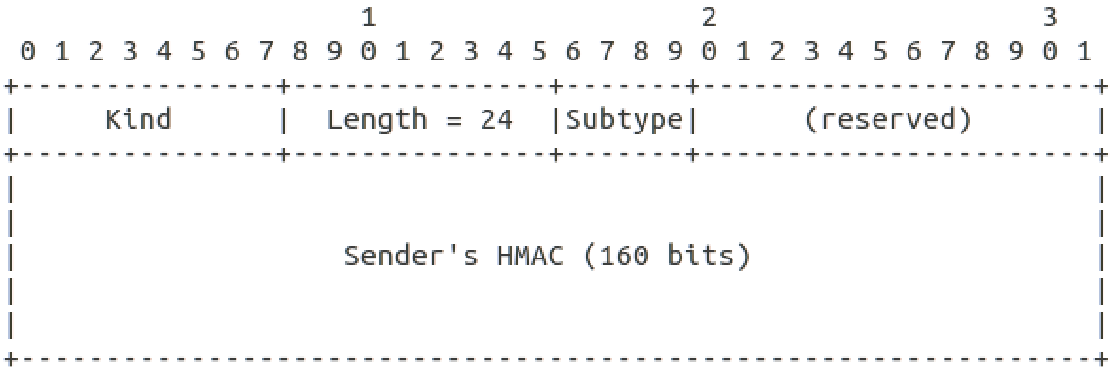
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |  Length = 12  |Subtype|     |B|   Address ID  |
+---------------+---------------+-------+-----+-+---------------+
|                   Receiver's Token (32 bits)                  |
+---------------------------------------------------------------+
|                Sender's Random Number (32 bits)               |
+---------------------------------------------------------------+
```

Figure A.2: MP Join option format for initial SYN



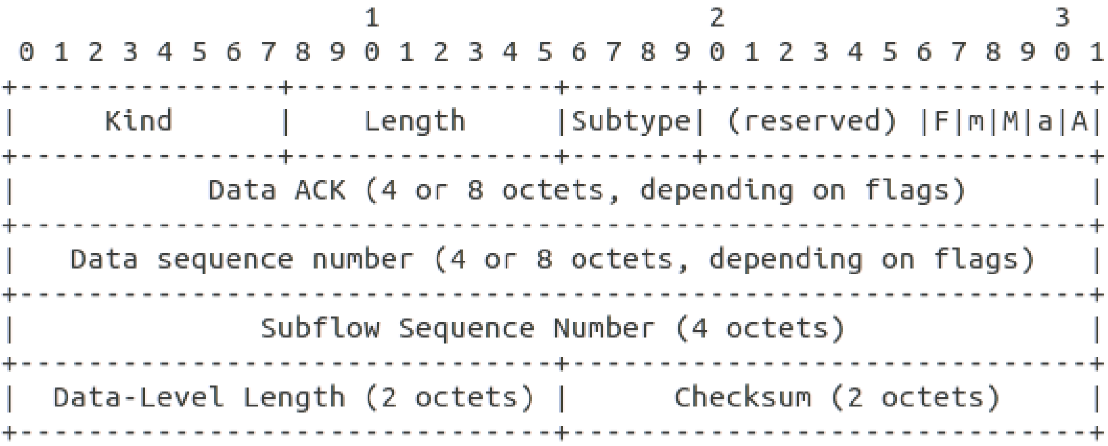
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |  Length = 16  |Subtype|     |B|   Address ID  |
+---------------+---------------+-------+-----+-+---------------+
|                                                               |
|                Sender's Truncated HMAC (64 bits)              |
|                                                               |
+---------------------------------------------------------------+
|                Sender's Random Number (32 bits)               |
+---------------------------------------------------------------+
```

Figure A.3: MP Join option format for SYN+ACK



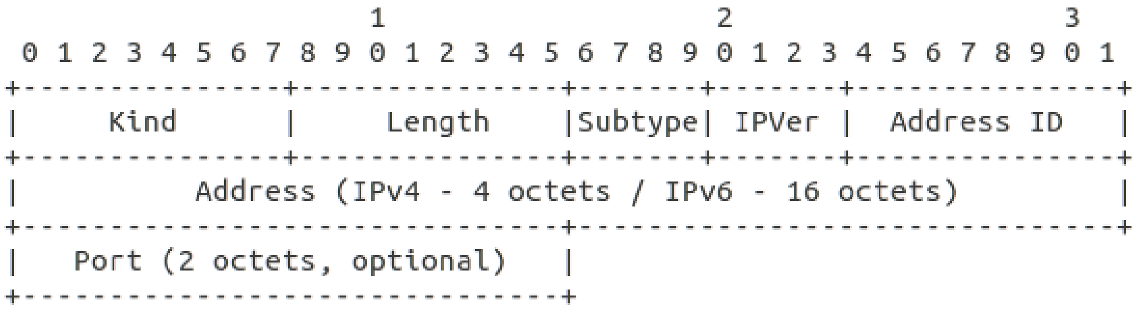
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |  Length = 24  |Subtype|      (reserved)       |
+---------------+---------------+-------+-----------------------+
|                                                               |
|                                                               |
|                   Sender's HMAC (160 bits)                    |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
```

Figure A.4: MP Join option format for final ACK



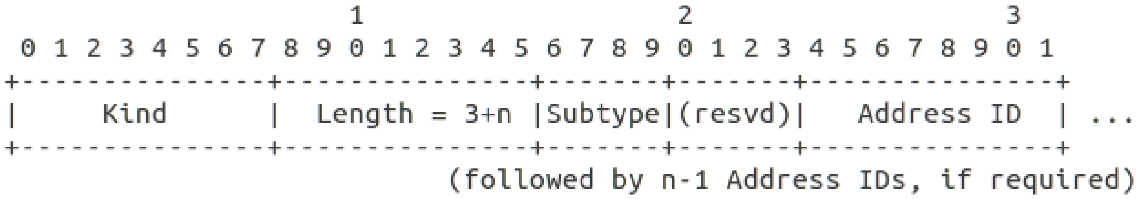
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+----------------------+
|     Kind      |    Length     |Subtype| (reserved) |F|m|M|a|A|
+---------------+---------------+-------+----------------------+
|           Data ACK (4 or 8 octets, depending on flags)       |
+--------------------------------------------------------------+
|   Data sequence number (4 or 8 octets, depending on flags)   |
+--------------------------------------------------------------+
|              Subflow Sequence Number (4 octets)              |
+-------------------------------+------------------------------+
|  Data-Level Length (2 octets) |      Checksum (2 octets)     |
+-------------------------------+------------------------------+
```

Figure A.5: MP DSS option format



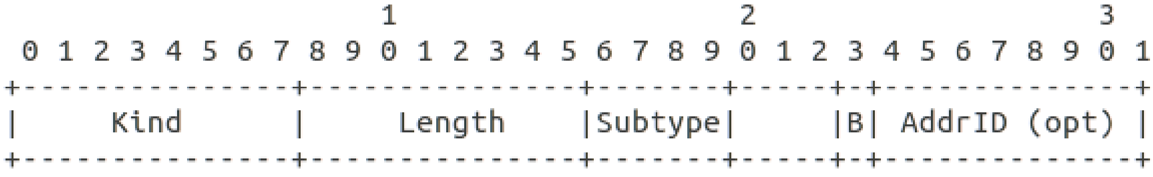
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |     Length    |Subtype| IPVer |  Address ID   |
+---------------+---------------+-------+-------+---------------+
|          Address (IPv4 - 4 octets / IPv6 - 16 octets)         |
+-------------------------------+-------------------------------+
|   Port (2 octets, optional)   |
+-------------------------------+
```

Figure A.6: MP ADD_ADDR option format



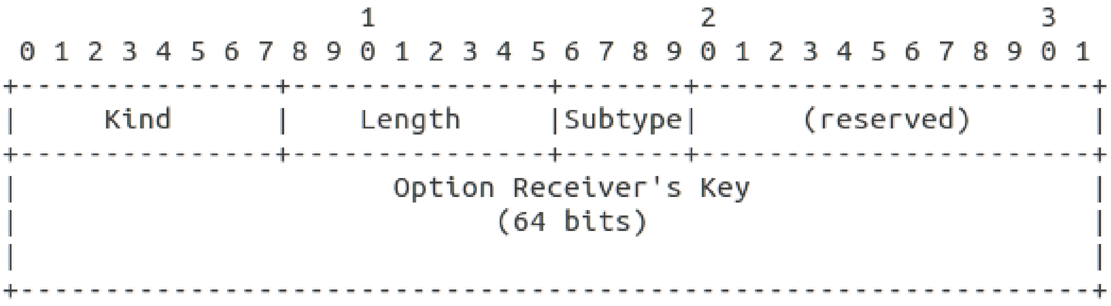
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+---------------+
|     Kind      |  Length = 3+n |Subtype|(resvd)|   Address ID  | ...
+---------------+---------------+-------+-------+---------------+
                           (followed by n-1 Address IDs, if required)
```

Figure A.7: MP REMOVE_ADDR option format



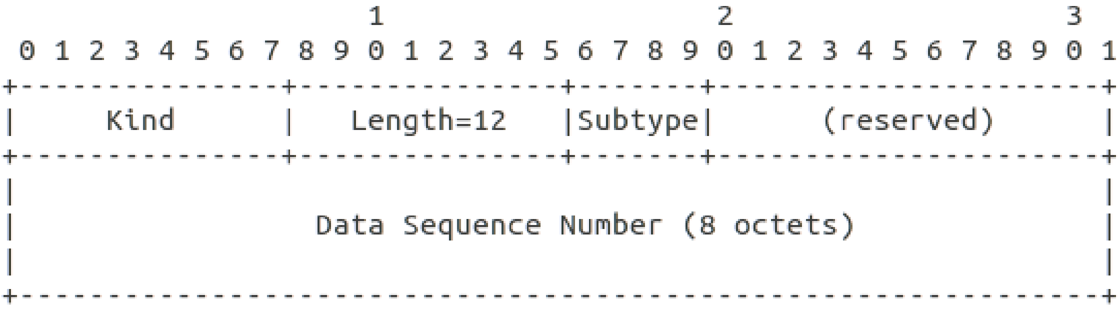
```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----+-+---------------+
|     Kind      |     Length    |Subtype|     |B| AddrID (opt)  |
+---------------+---------------+-------+-----+-+---------------+
```

Figure A.8: MP PRIO option format



```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype|      (reserved)       |
+---------------+---------------+-------+-----------------------+
|                      Option Receiver's Key                    |
|                            (64 bits)                          |
|                                                               |
+---------------------------------------------------------------+
```

Figure A.9: MP FASCT_CLOSE option format



```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |   Length=12   |Subtype|      (reserved)       |
+---------------+---------------+-------+-----------------------+
|                                                               |
|                Data Sequence Number (8 octets)                |
|                                                               |
+---------------------------------------------------------------+
```

Figure A.10: MP FAIL option format

Appendix B

MPTCP Event Definitions

```
## Generated for each MPTCP option found in a TCP header (kind = 30)
## In most cases, this event will be raised at least once for every
## packet in an MPTCP connection (read "very often if MPTCP is in use")
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## subtype: The numerical subtype of the option as defined by RFC6824
##
##
event mptcp%(c: connection, len: count, subtype: count, is_orig: bool%);
```

```
## Generated for each MPTCP option of MP_Capable subtype
## found in a TCP header (kind = 30, subtype = 0)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection's originator.
##
## len: The length of the option's value.
##
## version: The MPTPC version used (should be 0 according to RFC 6824)
##
## flags: the MPTCP flags (refer to RFC 6824)
##
## sender_key: the key chosen by the sender
##
## receiver_key: the key chosen by the receiver. Should be 0 if len != 20
##
##
event mp_capable%(c: connection, len: count, version: count, flags: count,
    sender_key: count, receiver_key: count, is_orig: bool%);
```

```
## Generated for each MPTCP option of MP_JOIN subtype
## found in a TCP header (kind = 30, subtype = 1)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection's originator.
##
## len: The length of the option's value.
##
## flags: the MPTCP flags (refer to RFC 6824) should only be 0 or 1 (backup)
##
## addr_id: ID of the address that is joining, as referenced in the MPTCP
    connection's address table
##     should be 0 on last ack of handshake
##
## rand: the random number chosen by the sender. Should be 0 in the last ACK of TCP
    handshake
##
## token: the token corresponding to the connection the sender wants to join. Should
    be 0
##      outside of initial SYN
##
## hmac: the (possibly truncated) hmac sent for authentication. Should be 0 in
    initial SYN,
##      64 bits in SYN + ACK and 160 bits in final ACK (see RFC 6824 for details)
##
##
event mp_join%(c: connection, len: count, flags: count, addr_id:count, rand: count,
    token: count, hmac: string, is_orig: bool%);
```

```
## Generated for each MPTCP option of DSS subtype
## found in a TCP header (kind = 30, subtype = 2)
## most fields are optional in length and presence. See RFC 6824 and flags.
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection's originator.
##
## len: The length of the option's value.
##
## flags: the flags used by the MPTCP option. Includes reserved bits that should be 0
##
## data_ack: connection-level acknowledgement
##
## dsn: data sequence number
##
## ssn: subflow sequence number
##
## dll: data-level length
##
## checksum:
##
##
event mp_dss%(c: connection, len: count, flags: count, data_ack: count, dsn: count,
    ssn: count, dll: count, checksum: count, is_orig: bool%);
```

```
## Generated for each MPTCP option of ADD_ADDR subtype
## found in a TCP header (kind = 30, subtype = 3)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## ipver: the version of IP used by the new address
##
## addr_id: the ID to assign to the new address in this connection
##
## address: the new address (in 32 or 128 bits)
##
## portn: the port number of the address (optional)
##
##
event mp_add_addr%(c: connection, len: count, ipver: count, addr_id: count, address:
    addr, portn: port, is_orig: bool%);
```

```
## Generated for each MPTCP option of REOMVE_ADDR subtype
## found in a TCP header (kind = 30, subtype = 4)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## resvd: reserved bits, should always be 0
##
## addr_id: ID of the address to remove. (TODO: change to accommodate multiple
    addresses)
##
##
event mp_remove_addr%(c: connection, len: count, resvd: count, addr_id: count,
    is_orig: bool%);
```

```
## Generated for each MPTCP option of MP_PRIO subtype
## found in a TCP header (kind = 30, subtype = 5)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## flags: flags used by the option. Should only be 0 or 1
##
## addr_id: The ID of the address who’s priority is changed
##
##
event mp_prio%(c: connection, len: count, flags: count, addr_id: count, is_orig:
    bool%);
```

```
## Generated for each MPTCP option of MP_FASTCLOSE subtype
## found in a TCP header (kind = 30, subtype = 6)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## resvd: reserved bits, should always be 0
##
## receiver_key: key use by the receiver
##
##
event mp_fastclose%(c: connection, len: count, resvd: count, receiver_key: count,
    is_orig: bool%);
```

```
## Generated for each MPTCP option of MP_FAIL subtype
## found in a TCP header (kind = 30, subtype = 7)
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## resvd: reserved bits, should always be 0
##
## dsn: Data Sequence Number
##
##
event mp_fail%(c: connection, len: count, resvd: count, dsn: count, is_orig: bool%);
```

```
## Generated for each MPTCP option which is invalid
## Either unknown subtype or length inconsistent with the subtype and/or flags
## Should never be raised unless someone is using a faulty implementation
## or willingly creating invalid packets
##
##
## c: The connection the packet is part of.
##
## is_orig: True if the packet was sent by the connection’s originator.
##
## len: The length of the option’s value.
##
## subtype: subtype of the option
##
##
event mp_error%(c: connection, len: count, subtype: count, is_orig: bool%);
```