1 Bro events

Events in Bro fulfil 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 event have been implemented for the understanding of MPTCP.

1.1 Protocol analyzers

Every packet that goes through Bro will go through a hierarchy 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.

Firgure 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 behaviour that corresponds to the needed event. Once such behaviour 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.

Events can be relatively high-level (such as the establishment of a connection) or low-level (such as a TCP packet being delivered). Like we just mentioned, each one is represented by a list of value 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 & destination addresses & 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 is not enough. The correspofing event, `tcp_packet`,

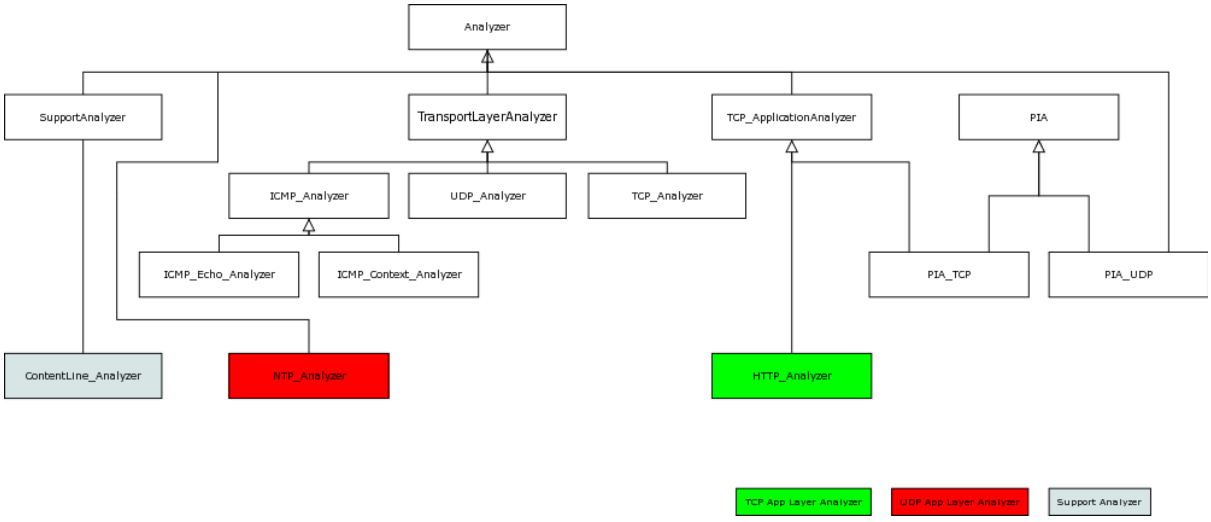

Figure 1: Class Hierarchy of Bro analyzers

therefore contains much more:

- The packets TCP flags
- Whether or not the packet was sent by the connection's initiator
- The sequence & acknowledgement numbers
- The packet length
- The packet payload

Each event is defined 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` 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.

1.2 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; event 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 behaviour. If this is the case, we will call the `ParseTCPOptions` method.

The behaviour 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. First off, 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 behaviour 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

two the ones we have seen: `ParseMPTCP` and the callback function `MPTCPEvent`. Out of the ten event, 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, 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 raise 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), and `mp_error` event is raised.

1.3 Event description

As mentioned in the previous section ten events were added to the analyzer in order to facilitate working with the protocol, corresponding the 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 behaviour 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 event 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 behaviour in script.

1.3.1 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 wince we only care about extracting the subtype (it 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.

1.3.2 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. For example, a `mp_capable` option (subtype 0) should only ever be of length 12 or 20. Furthermore, it should only of length 20 during the final step of the three-way handshake (SYN + ACK) since it 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 happen, we do not attempt to make the call and simply raise an error.

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 know authorized length values that the subtype can take. Further fine-tuning is done base 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 analysed 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.

1.3.3 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 only time the second key is present. In order to allow the user full control over what he want 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 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 he wishes to observe. The `mp_capable` event could be broken down into further, more explicit event 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.

1.3.4 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 middleboxes 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 part 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 of the length.

In its use, `mp_join` is also similar to `mp_capable` in that it can mainly server 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.

1.3.5 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 allows observers access to the data sequence mapping to perform the same re-assembly as the hosts of the connection. However, the event can also serve to detect MPTCP connections who's 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.

1.3.6 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 the 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 who's presence is not indicated by either the TCP or MPTCP flags. The only way to detect its presence during the parsing is whether the 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.

1.3.7 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 is does has 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 stop using them. Some use might also be found in trying to understand how a given host decides which addresses to make available and when.

1.3.8 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 subflows to backup or back to regular. Its use is rather limited, but one may be interested in observing which links hosts prioritize.

1.3.9 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.