Combining kTLS and BPF for Introspection and Policy Enforcement

Daniel Borkmann
Cilium.io
daniel@cilium.io

John Fastabend
Cilium.io
john@cilium.io

ABSTRACT

Kernel TLS is a mechanism introduced in Linux kernel 4.13 to allow the datapath of a TLS session to be encrypted in the kernel. One advantage with this mechanism compared to traditional user space TLS is that it allows sendfile operations to avoid using otherwise expensive bounce buffers to do encryption in user space. Additionally, as of kernel 4.17 the Linux kernel has supported implementing socket based BPF policies by attaching SK_MSG programs to sockets. These can be used to monitor TCP sessions and enforce policies by allowing or dropping messages using an administrator supplied BPF program. However, until recently these features have not been allowed to coexist. Users had to choose between performance improvements offered by kTLS or applying BPF policies using SK_MSG programs. Perhaps worse, BPF policies operating with traditional TLS in place, like those supported by OpenSSL, had minimal visibility into TCP based messages due to receiving already encrypted traffic. In this paper we describe the new kTLS/BPF stack implementation and its user API.

1 INTRODUCTION

Linux, starting with kernel 4.17, provides a mechanism to attach a BPF program to a socket. Once attached the BPF program, known as a SK_MSG program, is then executed on every sendmsg or sendpage call issued on the socket. The programs can inspect, drop, redirect, or modify messages as they are sent by the socket. To accomplish this, the socket operations sendmsg and recvmsg among others are modified to use BPF specific calls. This is similar to the mechanism known as Upper Layer Protocols (ULPs) used by kTLS. One primary difference between BPF SK_MSG programs and ULPs is the attach method. ULPs use a socket option from user space side and currently can only be attached, not detached. On the other hand, BPF SK_MSG programs use a special purpose BPF set of maps known as a sockmap and are modified from within the kernel.

Linux Plumbers Conference'18, Nov 2018, Vancouver, BC, Canada
2018. :

kTLS, or kernel TLS, introduced in kernel 4.13 allows the kernel to do the datapath encryption portion of a TLS session. The performance benefits are described in detail by Dave Watson[1], the sendfile case shows particularly good results because buffers that previously had to be moved through user space can now be handled completely by the kernel. However, because both BPF SK_MSG programs and kTLS work by using their own set of callbacks assigned to the socket operations, the callback structure used by sockets to implement protocol specific handling, they previously could not coexist. This was further complicated by kTLS and BPF SK_MSG programs using different sets of data structures and helpers each with their own subtleties.

To resolve this, we first provided a common set of data structures, the struct sk_msg, and functions that can be used by both kTLS and BPF SK_MSG as well as other ULPs that need a common mechanism to access data contained in scatterlists. Once kTLS and BPF SK_MSG were converted to a common set of data structures, support for BPF SK_MSG programs was added to kTLS. This allows kTLS and SK_MSG programs to coexist so that BPF policies implemented in SK_MSG programs can continue to be applied once kTLS is in use.

The rest of this paper describes the networking stack and BPF programs that can be implemented with SK_MSG while deploying kTLS. In Section 2, the basic stack is outlined and a description of SK_MSG programs is given. Section 3 gives a brief introduction to kTLS. In Section 4 the developer implementation details are provided. Finally, in Section 5 we will talk about remaining issues with the existing implementation and ongoing work.

2 KTLS AND SK_MSG STACK

BPF SK_MSG allows inserting BPF policies into the socket layer. TCP is currently the only supported socket type. BPF SK_MSG programs and kTLS integrate with the TCP stack as shown in Figure 1. Next we describe the UAPI details for SK_MSG and kTLS.

[1]kTLS: Linux Kernel Transport Layer Security, Dave Watson. Facebook. October 2016. https://netdevconf.org/1.2/papers/ktls.pdf

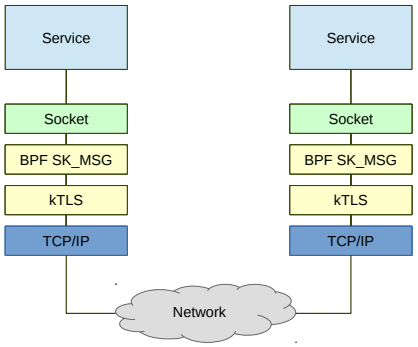

Figure 1: SK_MSG programs and kTLS integrated

2.1 SK_MSG Lifetime

BPF_PROG_TYPE_SK_MSG is a BPF program type, shortened here to SK_MSG, that is executed on every sendmsg and sendpage call of a socket. The lifetime of a SK_MSG program is as follows:

- The BPF program is loaded using the BPF syscall command BPF_PROG_LOAD. After this, the SK_MSG program is identified using its file descriptor.
- A BPF map of type BPF_MAP_TYPE_SOCKMAP (sockmap) or BPF_MAP_TYPE_SOCKHASH (sockhash) is created. These are key/value maps where the key is an exact match or hash similar to the regular BPF array or hash maps implemented in BPF except the value is a socket in each case.
- The SK_MSG program is attached to the above map using the bpf syscall command. At this point any sockets added to the map will have the attached SK_MSG program applied.
- Sockets are added to the sockmap or sockhash maps using either the BPF syscall command BPF_MAP_UPDATE_ELEM or from a BPF program using the helper bpf_sock_map_update() or bpf_sock_hash_update(). This allows BPF programs to monitor TCP state events, such as entering socket state ESTABLISHED, and adds the socket to an appropriate sockmap or sockhash map directly from a BPF program without userspace intervention.
- Once sockets are added to a sockmap or sockhash with a SK_MSG program attached, the attached SK_MSG program will be invoked on sendmsg and sendile.
- SK_MSG programs can be detached from a map using the BPF syscall command BPF_PROG_DETACH. Additionally, programs will be automatically removed if the map is destroyed. Once a program is detached any existing SK_MSG will be removed and no longer run on that socket.

In addition to the above there are some rules that apply to when and how SK_MSG programs are attached/detached to sockets and the sockhash/sockmap maps. First, although a socket may exist in multiple maps no more than one of those maps may contain a SK_MSG program[2]. This avoids the case where its unclear what program is being run on a socket. Second, when a SK_MSG program is detached from a map it will not be removed from sockets running in the map. Sockets must be explicitly removed from the map to have the SK_MSG program removed (e.g. no longer invoked on sendmsg/sendfile) regardless of the attach/detach state of the BPF program. This is easily done by BPF syscall BPF_MAP_DELETE_ELEM or alternatively if the map is removed, all sockets will be removed from the map first and hence SK_MSG programs removed. Finally, attaching a SK_MSG program to a map with existing sockets will not retroactively apply the SK_MSG program to existing sockets in the map. Rather, the SK_MSG program only applies to sockets added after the program has been attached. For this reason for most use cases we suggest using the above lifetime as it avoids having to be concerned with these details. That said, when managing many sockets and multiple policies, the above rules may be helpful to understand the proper workflow.

2.2 SK_MSG Program

A SK_MSG program has the following function signature,

```
int bpf_sk_msg_prog(struct sk_msg_md *msg)
```

With the sk_msg_md shown here,

```
struct sk_msg_md {
 void *data;
 void *data_end;

 __u32 family;
```

[2]A socket may exist in multiple maps with different types of programs. For example, a map with a SK_MSG program described here and a BPF program run on the ingress path of a socket. This is not discussed in the scope of this paper.

```
 __u32 remote_ip4;
 __u32 local_ip4;
 __u32 remote_ip6[4];
 __u32 local_ip6[4];
 __u32 remote_port;
 __u32 local_port;
};
```

The sk_msg_md provides pointers to the message data being passed into the program as well as a set of metadata describing the session the data is being sent over. Similar to other BPF program entry points, for example XDP, direct data access must be guarded with data bounds checks. A typical check would test for the availability of 'N' bytes in a message as follows,

```
if (data + N > data_end)
 goto error;
```

However, for performance reasons unlike in XDP where all data is contiguous and in a single buffer a SK_MSG message is stored in a scatter-gather ring data structure. Because BPF expects data access to be linear the [data, data_end) memory may only reference the first element in a scatterlist. In general the infrastructure will try to maximally populate the first scatterlist element typically packing as much data in this element as possible. However, depending on memory pressure and application patterns, this can not be guaranteed and data may need to be read after the first scatterlist element. To support this, SK_MSG supports the following helper,

```
bpf_sk_msg_pull_data(struct sk_msg *msg,
      u32 start, u32 end,
      u64 flags)
```

This helper will update [data, data_end) pointers to reference [start, end). When accessing this data the helper will try to avoid expensive memory allocations and copy operations, but if the user asks for a large memory range (e.g. greater than a page size) or a range that crosses a scatterlist boundary an alloc/copy may be required. By crossing a scatterlist boundary we simply mean that start and end are part of two different scatterlist elements. Also because we optimize for performance in the zero-copy case, in the common case for sendfile, no data will be available by default on sendfile calls. Instead, the data pointers will be initialized as follows [data=0, data_end=0) and users will need to use sk_msg_pull_data() to read any necessary data. This reflects that the infrastructure does not have enough information about the BPF program to know which bytes will be read by the program. Thus, we force the programmer to be explicit. This avoids unnecessary overhead at the cost of exposing some complexity to the user.

There are two other scenarios that can occur when dealing with applications that are handled by the helpers listed below. First, applications may supply less data than is needed to reach a policy verdict. In this case the SK_MSG program may wait until the application provides the required remaining bytes. The second case occurs when we reach a verdict early in a message and want to avoid running the SK_MSG program on a specific number of future bytes. This pattern is exemplified by a header with a large payload.

```
bpf_msg_cork_bytes(struct sk_msg_buff *msg, u32
    bytes)
bpf_msg_apply_bytes(struct sk_msg_buff *msg, u32
    bytes)
```

The bpf_msg_cork_bytes() helper will delay passing any verdict to the SK_MSG infrastructure and instead wait until the specified number of 'bytes' have been received before calling the SK_MSG program again to obtain a corrected verdict.

The bpf_msg_apply_bytes() helper will skip calling the SK_MSG program until the number of 'bytes' specified has been sent. Using the above two helpers allow SK_MSG programs to handle applications that (a) send too little data or (b) large amounts of data that is not relevant to the SK_MSG program.

The above describes basic data handling in a SK_MSG program. Once the policy implemented by the SK_MSG program or monitoring is complete the program can return a verdict. The SK_MSG infrastructure supports two verdicts, SK_PASS and SK_DROP. As the names imply SK_PASS will send the message on to the lower layer stack which may be TCP or kTLS. With the addition of the bpf_msg_redirect() helper two additional types of SK_PASS can be supported and are documented below. SK_DROP on the other hand will return the error EACCESS to the application and the message will not be sent. In most cases the return codes are easily understood, but when mixed with bpf_msg_cork_bytes() and bpf_msg_apply_bytes() there can be some subtle interactions.

First, after a bpf_msg_cork_bytes() call has been issued, data exists in the scatter-gather ring from multiple sendmsg/sendfile calls. In the event that SK_DROP is returned when running the SK_MSG program on cork data all data will be removed.[3] This includes the data in the buffer that was submitted from previous sendmsg calls, potentially confusing an application that is not

[3]The alternative would be to not free the data, but without any API to remove it, we could get the BPF program stuck. In the future, we may provide flags for the BPF program to specify if it should free all or some of the data.

prepared to deal with a EACCESS error because the application may believe older data was successfully sent. SK_MSG programmers should keep this in mind while building programs that implement policies.

Next, it may not be immediately obvious, but because the bpf_msg_apply_bytes() may indicate fewer bytes than the application sends in a single sendmsg/sendfile call, multiple verdicts may be given per call. In this case if a SK_DROP verdict is reported from any part of the message, processing of the message is halted and the sendmsg/sendfile syscall returns. However, if the SK_MSG program has accepted some subset of the message sent by the sendmsg/sendfile call, we report the number of bytes passed to the lower layer not the EACCESS error. This allows the application to have a consistent view of how many bytes have been consumed by the lower layer stacks.

A note on precedence, if bpf_msg_cork_bytes() requests N bytes and bpf_msg_apply_bytes() specifies M bytes in the same BPF program, first N bytes will be corked and then the M bytes specified from bpf_msg_apply_bytes() will be considered. However, the BPF program will always be called after cork bytes is reached regardless if the apply bytes is larger. The rational is that bpf_msg_cork_bytes() has a higher precedence than bpf_msg_apply_bytes(). SK_MSG programs will need to consider this when mixing the two helpers. In practice we have found that this seldomly happens and best programming practices should dictate avoiding having outstanding bpf_msg_cork_bytes() and bpf_msg_apply_bytes() on the same message.

In addition to indicating the message is to be allowed, SK_PASS can also be used to redirect a message to another socket when paired with one of the redirect helpers.

2.3 SK_MSG Redirect

```
int bpf_msg_redirect_map(struct sk_msg_buff *msg,
    struct bpf_map *map, u31 key, u64 flags)
int bpf_msg_redirect_hash(struct sk_msg_buff *msg,
    struct bpf_map *map, void *key, u63 flags)
```

The bpf_msg_redirect_map() helper is used with a map of type BPF_MAP_TYPE_SOCK_MAP and bpf_msg_redirect_hash() is used with a map type of BPF_MAP_TYPE_SOCK_HASH. Multiple helpers exist to aid the BPF verifier in type checking. In each helper the 'key' is used to lookup a socket to redirect to within the corresponding map. Two flags, BPF_F_INGRESS and BPF_F_EGRESS, influence if the redirect should send the message on the EGRESS path or the INGRESS path. On the egress path the message is submitted to the lower layer of the specified socket, which at this time may be TCP or kTLS. On the ingress path, the message is put into a message receive queue that the application will read from using standard recvmsg semantics.

Finally, the last SK_MSG program helper which is specific to SK_MSG programs is bpf_msg_push_data(). This helper is used to push bytes into a message. Typical use cases include adding headers to a message and inserting metadata to be read lower in the stack perhaps by another BPF program. It can be used in combination with any of the above helpers and/or verdicts.

This completes a brief description of the SK_MSG UAPI. For further details, the kernel source UAPI documentation has additional description of each helper and verdict. Sample SK_MSG programs exist in the Linux kernel tree along with test programs which may provide some insight into expected behavior. Finally, for a container network security example, much of which motivated this work, review Cilium [4] which uses SK_MSG program types.

3 KTLS

kTLS, or kernel TLS, allows the Linux kernel to perform the datapath operations of TLS, but the more complex TLS handshake is kept in userspace. This allows applications to avoid bounce buffers in user space when implementing sendfile type workloads. Removing context switches and copies gives performance improvements especially when considering tail latencies [5].

For most applications kTLS will be transparently enabled by the SSL library being used. OpenSSL, for example, can enable kTLS using a socket option as shown below. Once set data encryption and decryption takes place in the kernel's datapath.

```
struct tls12_crypto_info_aes_gcm_128 tls_tx = {
  .info = {
    .version = TLS_1_2_VERSION,
    .cipher_type = TLS_CIPHER_AES_GCM_128,
  },
  .key = [...], [...]
}, tls_rx = {
  [...]
};
setsockopt(fd, SOL_TCP, TCP_ULP, "tls", sizeof("tls
    "));
setsockopt(fd, SOL_TLS, TLS_TX, &tls_tx, sizeof(
    tls_tx));
```

[4] Cilium, https://cilium.io

[5] kTLS: Linux Kernel Transport Layer Security, Dave Watson. Facebook. October 2016. https://netdevconf.org/1.2/papers/ktls.pdf

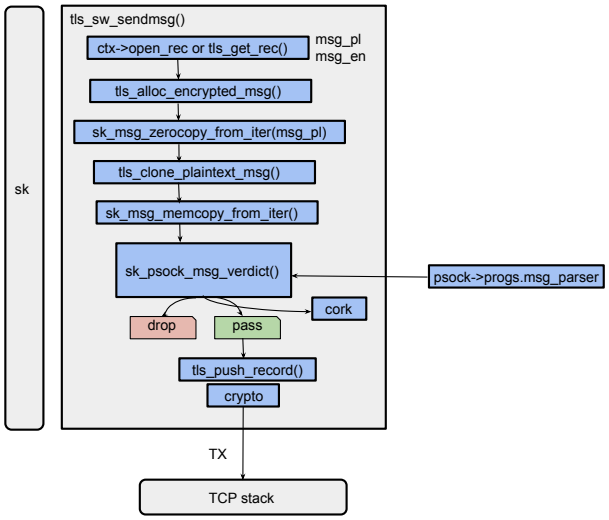

Figure 2: SK_MSG with kTLS egress path

```
setsockopt(fd, SOL_TLS, TLS_RX, &tls_rx, sizeof(
    tls_rx));
```

kTLS at the time of this writing support TLS 1.2 with AES-GCM and 128 bit of keysize.

Further details are available in kernel Documentation in Documentation/networking/tls.txt.

4 IMPLEMENTATION

Sockmap and kTLS when used together or separately, starting with Linux kernel 4.20, use the struct sk_msg as their base data structure to track data and metadata. The sk_msg structure is listed below. It is primarily a ring buffer of scatterlist elements with associated metadata containing the current context. Each scatterlist element points to data provided by the user. This data may be zero-copied or not depending on context, this is tracked in the sk_msg copy field. For example, data from a sendfile call will not be copied until the BPF program requests access to it. If it has not been copied, BPF programs may not read or write the data. This is required to avoid applications from modifying shared data during or after a BPF program has accessed the memory. The helper, documented in Section 2, sk_msg_pull() can be used to copy the data for reading if the BPF program needs to read the data. This implementation detail was chosen to optimize zero-copy where possible because in general it is unknown which bytes the BPF program may read at program load time. This is also analogous to tc BPF programs using direct packet access when they need to pull in non-linear data from the skb upon demand through the BPF helper bpf_skb_pull_data().

```
struct sk_msg_sg {
 u32    start;
 u32    curr;
 u32    end;
 u32    size;
 u32    copybreak;
 bool    copy[MAX_MSG_FRAGS];
 /* The extra element is used for chaining the
  * front and sections when the list becomes
  * partitioned (e.g. end < start). The crypto
  * APIs require the chaining.
  */
 struct scatterlist  data[MAX_MSG_FRAGS + 1];
};

struct sk_msg {
 struct sk_msg_sg  sg;
 void    *data;
 void    *data_end;
 u32    apply_bytes;
 u32    cork_bytes;
 u32    flags;
 struct sk_buff   *skb;
 struct sock   *sk_redir;
 struct sock   *sk;
 struct list_head  list;
};
```

We chose to implement sk_msg as a fixed size ring buffer to minimize runtime allocations as well as to simplify the implementation. It is possible that a future version may support chained buffers, which would allow the ring to grow as needed instead, if the buffer limits are reached where the user will receive a EAGAIN error.

When kTLS and SK_MSG programs are used together Figure 2 shows the packet egress flow. First, a TLS context is created for the message, next we build the sk_msg_sg using a zero-copy iterator if possible. The data will later be copied during encryption before returning to the application. If the zero-copy iterator fails, then a full copy will be performed. Once the sk_msg_sg is completed, it is passed to the BPF verdict engine sk_psock_msg_verdict(). This will ensure all required cork data is available and if there are any outstanding applied data counters. However, if cork data counters indicate more data is needed, the sk_msg_sg will then be buffered and the SK_MSG program will not be executed until specified bytes are available. Similarly, if it is determined by the apply bytes counter that the SK_MSG program does not need to run, sk_psock_msg_verdict()

will also be skipped. Otherwise, assuming the above two conditions are met, the SK_MSG program will be run. Once a verdict is reached the data will either be sent to the TLS encryption block, assuming SK_PASS or dropped if SK_DROP is received.

5 CONCLUSION

The above provides a detailed description of the kTLS and SK_MSG BPF program interaction. The above implementation is sufficient to meet many practical use cases, such as those in use by Cilium. Further, this allows network policy and TLS to coexist, something that until this work has not been possible without complicated middlebox and key distributions.

However, the reader may have noticed the above only addresses the egress path. The ingress path although supported by kTLS and with a different but similar BPF hook similar policies to those described above can be applied. Unfortunately, these two programs can not yet be used together. The main barrier to this support is kTLS uses a BPF hook to build complete messages. This hook happens to be the same hook used by the BPF infrastructure to implement its policies. At the moment we have two challenges, first we have no way to run multiple programs at this hook. Second if BPF and kTLS have conflicting policies those would need to be resolved. We plan to address this in a future kernel release.

Another limitation is that the current kTLS implementation only supports TLS1.2 with 128 bit key sizes. However, kTLS is designed to allow additional key sizes and can be extended to TLS1.3. This will be addressed in future kernels releases as well.

Otherwise, there are a handful of performance optimizations that can be implemented in both the TLS and SK_MSG datapaths. These should help with performance but are primarily minor. To date we have mostly focused on functionality and have spent little time micro-optimizing the stack. We expect this line of work will prove useful for performance metrics, but will not have any user visible API changes.

Finally, SK_MSG programs currently support a limited set of BPF helpers. We have identified a handful of additional helpers that may be useful when building BPF policies. For example retrieving the cgroup id may be useful for building Cgroup aware policies. We expect as more policies are built more helpers will likely be needed.

In summary, we have enabled SK_MSG programs and kTLS so that they can now co-exist as of the Linux kernel 4.20. This allows message level policies, like those typically used by microservices, to be exposed transparently to the applications even when TLS is in use. As far as we are aware, this is the first such implementation of its kind. We expect this to become used more widely as the 4.20 Linux kernel becomes generally available.

6 ACKNOWLEDGMENTS

The authors would like to thank Alexei Starovoitov, David S. Miller, and Eric Dumazet for review and feedback of this work both on the patch submissions, but also on the early designs. Thomas Graf for identifying the problem space way back at the kernel's Netconf 2017, countless hours of discussions and reviewing/integrating our work with Cilium. And finally, we thank everyone we missed who provided feedback on the mailing list and in person.