Building socket-aware BPF programs

Joe Stringer
Cilium.io
joe@cilium.io

ABSTRACT

Over the past several years, BPF has steadily become more powerful in multiple ways: Through building more intelligence into the verifier which allows more complex programs to be loaded, and through extension of the API such as by adding new map types and new native BPF function calls. While BPF has its roots in applying filters at the socket layer, the ability to introspect the sockets relating to traffic being filtered has been limited.

To build such awareness into a BPF helper, the verifier needs the ability to track the safety of the calls, including appropriate reference counting upon the underlying socket. This paper describes extensions to the verifier to perform tracking of references in a BPF program. This allows BPF developers to extend the UAPI with functions that allocate and release resources within the execution lifetime of a BPF program, and the verifier will validate that the resources are released exactly once prior to program completion.

Using this new reference tracking ability in the verifier, we add socket lookup and release function calls to the BPF API, allowing BPF programs to safely find a socket and build logic upon the presence or attributes of a socket. This can be used to load-balance traffic based on the presence of a listening application, or to implement stateful firewalling primitives to understand whether traffic for this connection has been seen before. With this new functionality, BPF programs can integrate more closely with the networking stack's understanding of the traffic transiting the kernel.

KEYWORDS

BPF, firewalls, Linux, networks, packet processing, sockets

INTRODUCTION

The Berkeley Packet Filter [16], introduced in 1992, provided a model for the implementation of flexible packet filtering by allowing runtime modification of kernel logic from userspace applications. BPF quickly became the de facto technology for monitoring the traffic flow within POSIX systems. Originally, this was restricted to applying filtering upon sockets attached to devices and not to implement filtering capabilities directly for traffic flowing through a system. Subsequent work investigated using BPF as a basis for packet filtering on devices [15], however this design does not appear to have been integrated into major operating systems until the introduction of extended BPF into Linux [8].

In recent years, multiple efforts have been made to build new filtering implementations based upon BPF [2, 3, 14, 19, 21, 22]. Many of these implementations aim to provide the functionality of a stateful firewall [9] - that is, to allow network policies to be defined to allow traffic from one network endpoint to talk to another network endpoint, and to implicitly allow responses to pass back through the firewall. Many implementations over the past decade have developed this functionality in a manner that is agnostic to the locality of the endpoints. This paper revisits the possibilities for stateful firewalling when the endpoint socket is co-located with the kernel providing firewalling functionality, effectively combining the ideas of [9, 12, 15].

This paper is laid out as follows. Firstly, we discuss the constraints that the BPF verifier places upon BPF programs and how those constraints translate into requirements on the implementation of socket lookup helpers in the Linux BPF API. The implementation of reference tracking functionality is presented. Secondly, we describe extensions to the BPF API to provide socket introspection capabilities, taking into account flexibility and performance. Thirdly, we present some use cases for the new functionality, including more detail on the stateful firewall case and an additional usage in handling management traffic in forwarding devices. Finally, we conclude and assess the position of this work.

EXTENDING THE VERIFIER

The BPF verifier must ensure that all BPF programs that are loaded into the kernel are safe to execute. For accessing socket pointers, this requires validation that memory accesses are within the bounds of the memory allocated for the structure, and will not make any modifications to the socket fields that would be incompatible with the core socket handling logic. This is provided through extension of existing functionality which performs bounds verification and offset conversion for programs operating on CGroups hooks. Additionally, the underlying memory must remain associated with the socket for the duration of accesses from the BPF program, which may be guaranteed by taking a reference on sockets for the

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

duration of their use. The following sections explain how this is implemented through the introduction of a new pointer type in the verifier, and through the extension of the verifier to recognise assembly instructions that represent reference acquisition and release.

Pointers to Sockets

Earlier work [1] introduced a `bpf_sock` structure to the BPF API and implemented bounds and access type checking, along with offset rewriting for converting the access of socket attributes from BPF instructions into equivalent accesses of the underlying kernel types. This work was introduced with new BPF hook points which provide the socket structure as the context of the BPF program.

To allow BPF programs to retrieve and access a socket pointer, the verifier must be made aware of when a register contains a pointer of this type, and also how to validate pointer access. The existing verifier logic handles context pointers in a generic manner via `bpf_verifier_ops`. However, the packet event hook points targeted in this paper already contain a context pointer of a type that is different from `bpf_sock`, so some additional logic was required in the verifier to understand this pointer type. The patch series associated with this paper introduced a new pointer type specific to sockets, and linked its validation directly to the aforementioned socket verification functions without using the verifier operations abstraction. Write operations into the socket are rejected in the current implementation.

With the verifier now aware of socket pointer types, this could be built upon to implicitly associate socket pointers with references to kernel resources.

Reference tracking

Sockets in Linux are reference-counted to track usage and handle memory release. In general to ensure the safety of socket for the duration of access in BPF programs, this reference count needs to be incremented when sockets are referenced from BPF programs. Two options were considered for how to guarantee this: Implicit reference tracking and explicit reference tracking.

Implicit reference tracking. This calls for the BPF infrastructure to handle reference tracking, and to hide this detail from the BPF API. Whenever a socket lookup function is called, a reference is taken and the socket is added to a reference list. At the end of the BPF program execution, the core code could walk this reference list and release each reference. However, collecting references and releasing them at the end of the BPF program invocation has the unfortunate overhead that even the execution of BPF programs that do not make use of the new socket lookup helpers would need to pay the cost of checking the list of references to free. When dealing with sufficiently demanding use cases such as those which use eXpress Data Path (XDP) [11], even the few instructions required to implement such a check may have a measurable impact on performance.

Explicit reference tracking. This calls for reference acquisition and release semantics to be built into the API. If references must be taken, then requiring BPF program writers to explicitly handle these references ensures that the program writer must understand the potential impact this may have on the operation of the program (including atomic operation cost), and it also ensures that the cost is only borne by programs that make use of this feature.

Implementation. Based upon the tradeoffs described above, explicit reference tracking was chosen. From an implementation perspective, this requires tracking pointer values through each conditional path in the program, and rejecting the load of programs that fail to "balance" resource acquisition (lookup) with release. This implementation works as follows.

Calls to helper functions which acquire references to resources are annotated in the verifier to associate the acquisition and release functions. When the instruction that acquires a resource is processed, a resource identifier is allocated for this resource. This identifier is kept in a list in the verifier state, and it is also associated with the register which receives the resulting resource pointer. When processing a corresponding release helper call, if a parameter to the function contains a register that is associated with a resource identifier, then the identifier is removed from the verifier state. If any paths in the program reach the final (`BPF_EXIT`) instruction and the verifier state contains any resource identifier association, then the program is considered unsafe as it has leaked the resource.

Some instruction calls are restricted while holding a resource reference to avoid leaking references, for example the `bpf_tail_call` helper function, which would otherwise leak the pointer reference to a subsequent BPF program.

Runtime protection

The verifier is tasked with ensuring that all BPF program execution flow is safe to the extents where it interacts with the BPF API. In general, BPF programs, once verified and triggered by event hooks, will execute from the first instruction through the final instruction for a particular path through the program. Therefore, after execution of a BPF program containing resource acquire and release functions which are verified by the logic above, references to those resources should not leak.

One exception to this is when classic BPF `LD_ABS` or `LD_IND` instructions are used in the BPF program. These instructions were previously used to provide direct packet access, with the semantics that if the access offset exceeds the length of the packet, the BPF program would be terminated prior to the final state. Without mitigation, this could lead to leaking of references at runtime, even if the instructions appear to balance acquisition and release at verification time. Newer BPF APIs provide better alternatives to these instructions for packet access, so we disallow the use of these instructions while holding a socket reference.

EXTENDING THE BPF API

To provide an API that implements explicit reference tracking, at least two functions are needed to handle referencing of sockets: A socket lookup function and a socket release function. This section describes the API considerations for each of these.

API definition

The following considerations were made for implementing the lookup helpers. The consideration of each of these items reflects the definition of the lookup helpers in Listing 1 and the structure definitions in Listing 2.

- The Linux stack may contain multiple network namespaces, where the BPF program may operate attached to a device in one network namespace while the desired socket may exist in another network namespace.
- BPF programmers may wish to discover sockets that are not directly related to the BPF program context (packet, socket, etc).
- Current use cases are focused around UDP and TCP sockets, however there may be a desire to extend this in future.

Namespacing. Linux network devices are associated with a particular network namespace which provides logical network stack separation [4, 5]. Sockets are also inherently associated with an application that operates in a particular network namespace [6]. The simplest path for handling namespacing from socket lookup handlers operating from a network device would be to allow BPF programs operating on a device within a network namespace to only find sockets in the same network namespace. By passing the BPF program context into the socket lookup helper, the implementation can derive the source network namespace from this context. Beyond this, with the growing popularity of containers it may be desirable for a network orchestration tool to make use of socket properties inside container network namespaces to influence decisions made on packets outside the container—for instance, on packets being received into a node prior to passing the packet into the network namespace. By allowing the netns ID to be specified, this use case can be supported as well.

Tuple definition. A common use case for the socket lookup functions would call them from a packet context, where the `__sk_buff` has the full 5-tuple information available and easily accessible. One could imagine a simpler helper function that allows the BPF program writer to provide the `__sk_buff` to the helper function, then the helper function would look up the socket based upon the packet metadata. This would however limit the potential uses for such a helper. Two cases that would be more limited with this model are usage from XDP (which does not inherently parse packets or collect packet metadata into the context), or if the BPF program implements a form of network address translation. From XDP, if the raw `xdp_md` context is provided to such a helper, then the helper would need to parse the packet to understand the 5-tuple. This would duplicate the standard stack paths which are executed after XDP where the `sk_buff` is built, but this information would be subsequently thrown away upon return of the helper call. Adding to this, if the program performs network address translation between the local application and the remote destination, then the local stack may not contain a socket associated with the tuple that is directly in the packet. While these cases could be worked around with a simpler helper, it was deemed more powerful and generic to allow the BPF program writer to provide the tuple for lookup.

Extensibility. New kernel APIs that have any scope for extension should contain a flags argument [13]. For the socket lookup helpers, a few ideas had been considered as a potential future alternatives to the existing behaviour. The lookup functions follow the standard socket lookup paths in Linux which have predetermined methods for selecting a socket when the application uses `SO_REUSEPORT`. Some subsequent discussion on the mailinglist proposed allowing BPF program writers to influence the socket selection mechanism [7].

Result. Another aspect of extensibility is the ability to lookup sockets which are not TCP or UDP. The initial RFC of this patch series proposed a single lookup function which would choose the Layer 4 protocol based upon a field in the tuple [20], however this would make it more difficult for BPF program writers to detect the support for different protocol types at compile time. When a socket type is unsupported, the implementation would return `NULL`, implying that there is no such socket. There is not a significant number of supported Layer 4 socket types in the Linux stack today, and the number is not expected to increase drastically, so it was considered simpler and easier to split out each Layer 4 socket lookup function into an independent helper call, so that the result could simply return a socket or `NULL` and not need to encode a representation of other error conditions.

Optimizations

Multiple optimizations were proposed during development of this feature to reduce the runtime overhead of using the socket lookup helpers. Two such optimizations are described below: Skipping reference counting when unnecessary to save on atomic instructions, and allowing the use of direct packet pointers for the tuple.

Avoid reference counting when unnecessary. For some socket types, such as UDP sockets, or TCP listen sockets, memory access safety can be achieved with minimal implementation: The destruction of such sockets is already governed by standard RCU rules, meaning that while the RCU lock is held, they can be safely accessed without holding a reference; Once the RCU grace period is reached, the memory may be freed and references to the socket are no longer safe to use. For these socket types, since BPF programs run under the RCU lock, the properties of these sockets can be accessed directly without taking a reference on the socket. As such, the implementation of the socket lookup and release functions can avoid the atomic reference count increment operation.

The API retains explicit reference tracking to ensure consistency of the API and to allow multiple underlying implementations to handle reference tracking in a way that is safe for the implementation. For instance, for TCP sockets that are not governed by RCU, the networking stack uses reference counting to manage socket memory instead, so the socket lookup and release functions would take and release references on the socket, respectively.

Allow lookup using direct packet pointers. The tuple structure is defined in such a way that if the IPv4 or IPv6 packet is immediately followed by the TCP or UDP header without IP options in between, then a pointer to the packet data at the offset of the Layer 3 addresses may be passed to the lookup function, allowing the implementation to directly pull the addresses and ports from the packet buffer rather than requiring the BPF program to first extract these onto the stack and pass a pointer to the stack copy of the tuple.

Future work

The implementation introduced in the Linux v4.20 release cycle includes support for looking up TCP and UDP sockets for IPv4 and IPv6 traffic, using the `SCHED_CLS`, `SCHED_ACT` and `SK_SKB` hook points. A proposed patchset extends this to allow the same helpers to be used from the `XDP` hook points [10].

USE CASES

Stateful Firewalling

Stateful firewalling is distinguished from stateless firewalling in that it provides the ability to associate two directions of a connection together and apply a filtering policy based upon the direction of the connection. This commonly involves using a connection tracker to reconstruct the connection state of each endpoint of the connection without specific knowledge from those endpoints [17, 18].

In cases where Linux is configured as a firewall between applications that exist in distinct Linux kernel copies running in the network, this makes sense; The information that is needed to perform stateful firewalling is not present on the Linux instance that needs this information, so the information must either be shared from the peers (assuming that the firewall orchestrator has control over the peer endpoints), or this information must be synthesised based upon the packets that are seen on this intermediate Linux instance. However, in many cases this model is used when filtering traffic where one endpoint is co-located with the Linux stack that performs the firewalling, which already contains information about the local end of the connection. In this use case, the connection tracking table effectively duplicates knowledge that Linux already creates and maintains.

Using the socket lookup functions described in this paper, in conjunction with the socket properties including local Layer 3 and Layer 4 information available in the `bpf_sock` (Listing 2), BPF programs can identify the directionality of connections for packets ingressing or egressing the Linux stack on a network device. Between this directionality information and a network policy provided by a userspace program, BPF programs are able to identify locally-sourced or locally-destined traffic and allow or deny traffic ingressing or egressing the Linux stack.

Filtering management traffic in forwarding devices

A common design of devices providing routing and forwarding functionality is to place high-bandwidth forwarding elements into a system with a low-bandwidth CPU. In many cases, the goal of the CPU is to only to handle control traffic for configuring the forwarding path, which is a fraction of the traffic handled by the actual forwarding path. Existing implementations may direct only traffic for a specific IP into the management CPU, or could plausibly use XDP to configure DoS filters so that only legitimate traffic is forwarded up the stack. Such implementations may not provide the degree of restriction on locally destined traffic that is desired, or may require additional co-ordination with the socket layer that requires significant orchestration.

Using the upcoming XDP support for socket lookup, early in the driver level of the Linux stack the traffic could be compared with the socket table and only passed up the stack if the traffic corresponds to a local application. As this functionality uses the socket table directly, it is always kept in sync with the current applications without additional orchestration of BPF maps. Traffic which is not intended to be handled locally can then be either handled by additional BPF logic programmed in the XDP hook.

CONCLUSION

This paper describes the contribution of new BPF verifier functionality in the Linux kernel to track references to kernel resources and uses these to provide access to socket introspection capabilities from packet handling hooks. The introduction of reference tracking logic into the verifier provides useful base infrastructure for supporting acquire and release semantics for kernel resources which may prove useful for other helpers in future, and the socket lookup API makes the packet hooks more powerful to support use cases such as stateful firewalling or load-balancing based on the sockets that are open on a system.

ACKNOWLEDGMENTS

The author would like to acknowledge Alexei Starovoitov, Daniel Borkmann, Martin KaFai Lau, Nitin Hande and Thomas Graf for guidance, review and testing of the patches.

REFERENCES

[1] David Ahern. net: Add bpf support for sockets. Linux kernel v4.10, commit 6102365876003, March 2018.

[2] Cilium Authors. API-aware networking and security. https://cilium.io.

[3] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. *Netdev Conference 2.1*, 2017.

[4] Eric W. Biederman. [NET]: Basic network namespace infrastructure. Linux kernel v2.6.24, commit 5f256becd868, September 2007.

[5] Eric W. Biederman. [NET]: Add a network namespace tag to struct net_device. Linux kernel v2.6.24, commit 4a1c537113cd, September 2007.

[6] Eric W. Biederman. [NET]: Add a network namespace parameter to struct sock. Linux kernel v2.6.24, commit 07feaebfcc10, September 2007.

[7] Daniel Borkmann. Re: [PATCH bpf-next] bpf: Extend the sk_lookup() helper to XDP hookpoint. https://www.spinics.net/lists/netdev/msg529778.html, October 2018.

[8] Jonathan Corbet. Extending extended BPF. *https://lwn.net/Articles/603983/*, 2014.

[9] M. G. Gouda and A. X. Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137, June 2005.

[10] Nitin Hande. [PATCH bpf-next] bpf: Extend the sk_lookup() helper to XDP hookpoint. https://www.spinics.net/lists/netdev/msg529727.html, October 2018.

[11] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *CoNEXT'18: International Conference on emerging Networking EXperiments and Technologies*. ACM Digital Library, December 2018.

[12] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 190–199, 2000.

[13] Michael Kerrisk. Flags as a system call API design pattern. https://lwn.net/Articles/585415/, February 2014.

[14] Alexander Kurtz. Application-level firewalling using systemd socket action and eBPF filters. https://github.com/AlexanderKurtz/alfwrapper.

[15] Kurt J. Lidl, Deborah G. Lidl, and Paul R. Borman. Flexible packet filtering: Providing a rich toolbox. In *Proceedings of the BSD Conference 2002 on BSD Conference*, BSDC'02, pages 11–11, 2002.

[16] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, volume 46, 1993.

[17] Pablo Neira Ayuso. Netfilter's Connection Tracking System. *LOGIN;, The USENIX magazine*, 31(3):34–39, June 2006.

[18] Justin Pettit and Thomas Graf. Stateful connection tracking and stateful NAT. In *Open vSwitch 2014 Fall Conference*, November 2014.

[19] Justin Pettit, Ben Pfaff, Joe Stringer, Cheng-Chun Tu, Brenden Blanco, and Alex Tessmer. Bringing platform harmony to VMware NSX. *SIGOPS Oper. Syst. Rev.*, 52(1):123–128, August 2018. ISSN 0163-5980.

[20] Joe Stringer. [RFC bpf-next 07/11] bpf: Add helper to retrieve socket in BPF. https://www.spinics.net/lists/netdev/msg501166.html, May 2018.

[21] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. Building an Extensible Open vSwitch Datapath. *ACM SIGOPS Operating Systems Review - Special Topics*, 51(1):72–77, 2017.

[22] Huapeng Zhou, Doug Porter, Ryan Tierney, and Nikita Shirokov. Droplet: DDoS countermeasures powered by BPF + XDP. *Netdev Conference 1.1*, 2017.

Listing 1: BPF API helper functions for socket lookup

```
struct bpf_sock *
bpf_sk_lookup_tcp(void *ctx, struct bpf_sock_tuple *tuple,
                  u32 tuple_size, u32 netns, u64 flags);
struct bpf_sock *
bpf_sk_lookup_udp(void *ctx, struct bpf_sock_tuple *tuple,
                  u32 tuple_size, u32 netns, u64 flags);

void bpf_sk_release(struct bpf_sock *sk);
```

Listing 2: BPF API structures for socket lookup (Linux v4.20)

```
struct bpf_sock_tuple {
        union {
                struct {
                        __be32 saddr;
                        __be32 daddr;
                        __be16 sport;
                        __be16 dport;
                } ipv4;
                struct {
                        __be32 saddr[4];
                        __be32 daddr[4];
                        __be16 sport;
                        __be16 dport;
                } ipv6;
        };
};

struct bpf_sock {
        __u32 bound_dev_if;
        __u32 family;
        __u32 type;
        __u32 protocol;
        __u32 mark;
        __u32 priority;
        __u32 src_ip4;          /* Allows 1,2,4-byte read.
                                 * Stored in network byte order.
                                 */
        __u32 src_ip6[4];       /* Allows 1,2,4-byte read.
                                 * Stored in network byte order.
                                 */
        __u32 src_port;         /* Allows 4-byte read.
                                 * Stored in host byte order
                                 */
};
```