eBPF

Security Threat Model

Jack Kelly, James Callaghan & Andrew Martin

control-plane.io

Driving the Technical Direction and Vision of eBPF

The eBPF Foundation focuses on advancing the state of the art for eBPF by directing upstream development, promoting the use of the technology and its benefits, and improving the security and robustness of eBPF as a whole.

ControlPlane is a cloud native security consultancy, unlocking next-generation technologies for regulated industries including Tier 1 banks and national governments. Headquartered in London, UK with presences in North America and Australasia, their 60 staff across 12 countries assure some of the world's most secure organisations.

Members

Platinum

CROWDSTRIKE
HUAWEI
intel
ISOVALENT
Meta
NETFLIX
Google

Silver

DATADOG

Introduction

Document Purpose

This document was commissioned by the eBPF Foundation to provide security information and guidance to large enterprises using or looking to adopt eBPF-based tools. The goal of the paper is to promote the security benefits of eBPF over traditional tooling, whilst raising awareness of potential risks that could arise in common use cases.

A Threat Modelling approach is taken to outline eBPF's defences through an attacker's lens, highlighting inherent controls built into eBPF which ensure code runs safely and securely at the kernel level. As with any technology, it is not possible to prevent all threats using built-in controls. Where this is the case, end-user recommendations and awareness statements are provided. This approach will allow anyone looking to adopt eBPF solutions to make effective, risk-based architecture and deployment decisions, and to enhance the security of their systems by extending this threat model for their organisation.

Scope

eBPF was originally created for the Linux Kernel, so this whitepaper assumes that all compute instances are running Linux-based operating systems. eBPF implementations outside of the Linux kernel are beyond the scope of this threat model.

The pace of innovation and change in the eBPF ecosystem is rapid, therefore new controls against some of the threats presented here may be developed in the future. New eBPF developments can be followed via ebpf.io and ebpf.foundation, and Linux kernel changes which have affected this document are recorded in the Kernel changes of note appendix.

Document Structure

The Threat Modelling approach applied here is structured around Shostack's four questions:

1. **What are we building?** This involves understanding what eBPF is, and how eBPF programs work.
2. **What can go wrong?** Following the definition of a simple, high-level scenario in the Threat Model Scope, we develop attack trees to explore how an attacker could utilise eBPF for nefarious purposes.

3. **What can we do about the things that can go wrong?** Once a list of threats has been established, inherent eBPF controls and end-user recommendations are mapped against them.
4. **Are we doing a good job?** Finally, the threat model's outcomes are reviewed to provide practical guidance for eBPF adopters.

In accordance with this blueprint, the rest of the paper is structured as follows:

1. The remainder of the introduction introduces the basic concepts of eBPF, examines some common use cases, and starts to introduce properties which may be of interest to an attacker. In the context of the threat modelling framework, this section serves to answer the question of 'what are we building?'
2. The Threat Model section answers the question of 'what can go wrong?' and presents attack trees for the following scenarios:
    a. Unauthorised access to sensitive information
    b. Denial of service
    c. Evasion of platform security tools (helpful for an attacker looking to realise one of the above goals)
    d. Note: Information tampering does not have an attack tree of its own, but threats to integrity are either covered within the above three attack trees or called out as separate threats in Detailed Threats and Controls
3. The Mitigating Controls and Recommendations section answers the following questions:
    a. 'What can we do about the things that can go wrong?'
    b. 'Are we doing a good job?'

    It consolidates the results of the threat model and mitigating control derivations into a set of best practice guidelines.

This report was created by ControlPlane through sponsorship of the eBPF Foundation. It does not express the opinions of the eBPF Steering Committee (BSC).

Background

eBPF is a technology that allows pre-analysed and validated programs to be run in the Linux kernel or other privileged execution contexts. It is used to safely and efficiently extend the capabilities of the kernel without requiring a change to kernel source code, or loading kernel modules. The name “eBPF” was originally an initialism, but as its usage and capabilities have expanded it is now a standalone term.

eBPF enables tools to leverage low-level kernel access within security guardrails. Its safety comes from the eBPF verifier (explained in eBPF Technical Overview), Just-In-Time (JIT) compiler, and some automatic mitigations, and it also enables more granular permission grants via capabilities. Due to these guardrails, eBPF is proposed as the first option to consider over kernel modules or patches.

Creating eBPF code that conforms to the verifier can make some tasks more difficult, but its Turing completeness enables its many use cases which are more tightly secured than equivalent direct kernel manipulation. Many large companies utilise eBPF as the safest method to write kernel-level tooling and produce highly performant solutions: common use cases include performance monitoring, observability, tracing, networking, and security detection and enforcement.

eBPF Technical Overview

This subsection presents a brief technical overview of eBPF, using material from ebpf.io and our other references.

eBPF programs are generally written in pseudo-C code or Rust and compiled into eBPF bytecode[1] which can then be run within the Linux Kernel. eBPF programs are loaded into the kernel by a userspace program using the bpf() syscall, commonly using a library such as libbpf (C) or ebpf-go (Go). When a program is loaded, the bpf() syscall returns a file descriptor to the program being loaded.. The program subsequently remains in memory until the file descriptor is closed. If a process can obtain a file descriptor to an eBPF object, future operations on that object are allowed.

eBPF programs are event-driven, and are run when the kernel or an application passes a certain hook point. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others. If a predefined hook does not exist for a particular requirement, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.

---

[1] Writing eBPF bytecode by hand is of a similar difficulty to hand-writing Assembly.

Before the program can be attached to the appropriate hook, the program must pass through the eBPF verifier, which confirms that the program is safe to run, for example by checking that:

- The process loading the eBPF program holds the required capabilities (privileges)
    - Unless unprivileged eBPF is enabled, only privileged processes can load eBPF programs
- The program does not crash or otherwise harm the system
- The program always runs to completion (i.e. it does not sit in an infinite loop)

Once a program is verified, it is Just-in-Time (JIT) compiled to translate the eBPF bytecode into machine-specific instructions. eBPF bytecode can be interpreted or JIT compiled, but JIT compilation is preferential for superior performance, and to avoid certain Spectre-related vulnerabilities[2]. The vast majority of modern architectures support JIT, but there may be less common platforms which do not support it.

eBPF programs are restricted to a fixed set of memory regions with a fixed-size stack and a context that is dependent on the “program type”. They also use statically sized key/value dictionaries called maps to store and retrieve data.

eBPF programs can make function calls into helper functions, a well-known and stable API offered by the kernel. The next section highlights certain helper functions that may have security side-effects and thus be of interest to attackers.

---

[2] Mitigations automatically applied vary depending on architecture and capabilities used.

eBPF Helper Functions of Interest to Threat Actors

If a threat actor can load and run eBPF code, the following helper functions[3] have particular security relevance:

1. bpf_probe_write_user
2. bpf_probe_read_user
3. bpf_override_return
4. bpf_send_signal
5. bpf_map_get_fd_by_id

Inspecting the relevant kernel headers and relevant lines in the source code reveals the minimum Linux capabilities (KC-cap-bpf) required by the userspace process which loads an eBPF program using each helper function:

Helper Function	Purpose	Required Minimum Capabilities
bpf_probe_write_user	Write to any process's user space memory	CAP_SYS_ADMIN (& kernel lockdown[4])
bpf_probe_read_user	Read any process's user space memory	CAP_BPF & CAP_PERFMON
bpf_override_return	Alter return code of a kernel function	CAP_SYS_ADMIN
bpf_send_signal	Send a signal to kill any process	CAP_SYS_ADMIN

Additionally, the libbpf function bpf_map_get_fd_by_id, which can obtain eBPF programs' eBPF maps fd (and as a libbpf function, runs in userspace instead of eBPF context), requires CAP_SYS_ADMIN.

CAP_BPF will let a process load its own eBPF programs and maps. To load some specific program types, it must be paired with another capability. For example, CAP_NET_ADMIN for loading network programs, and CAP_PERFMON for tracing programs and some networking use cases. CAP_SYS_ADMIN allows any helper function to be called.

---

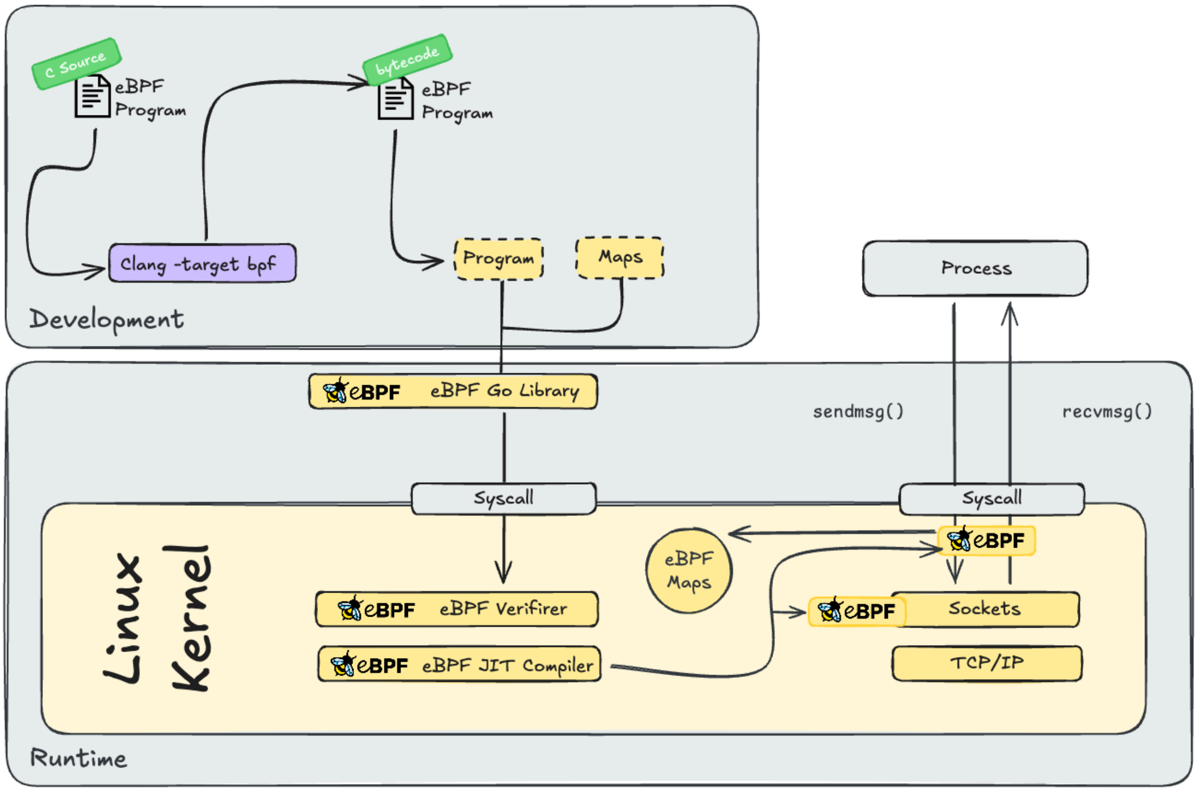
[3] For the full list of eBPF helper functions consider reading the relevant manual page bpf-helpers(7), and consult this paper.

[4] This helper is also blocked by the Kernel Lockdown feature in "integrity" mode or above. For additional details see the recommendations in Detailed Threats and Controls.

Data Flow Diagrams (DFDs)

High-Level eBPF DFD

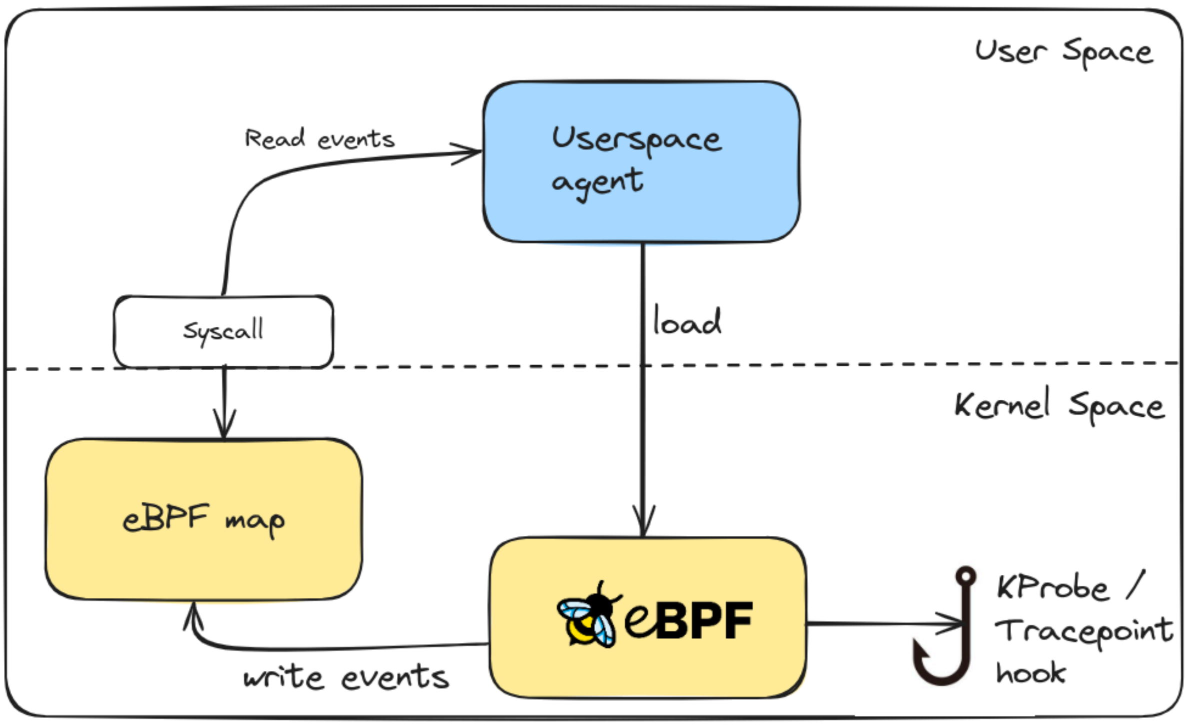
This diagram from ebpf.io shows the creation, verification, loading, and running of an eBPF program, along with communication with userspace processes via maps:



DFDs for different eBPF use cases

Performance monitoring, observability and tracing

Given the low-level insights that can be gained by running custom code in the kernel, eBPF provides an excellent foundation for observability. For examples of how eBPF can be used for performance observability and tracing, see the blog [Next-Generation Observability with eBPF](#).

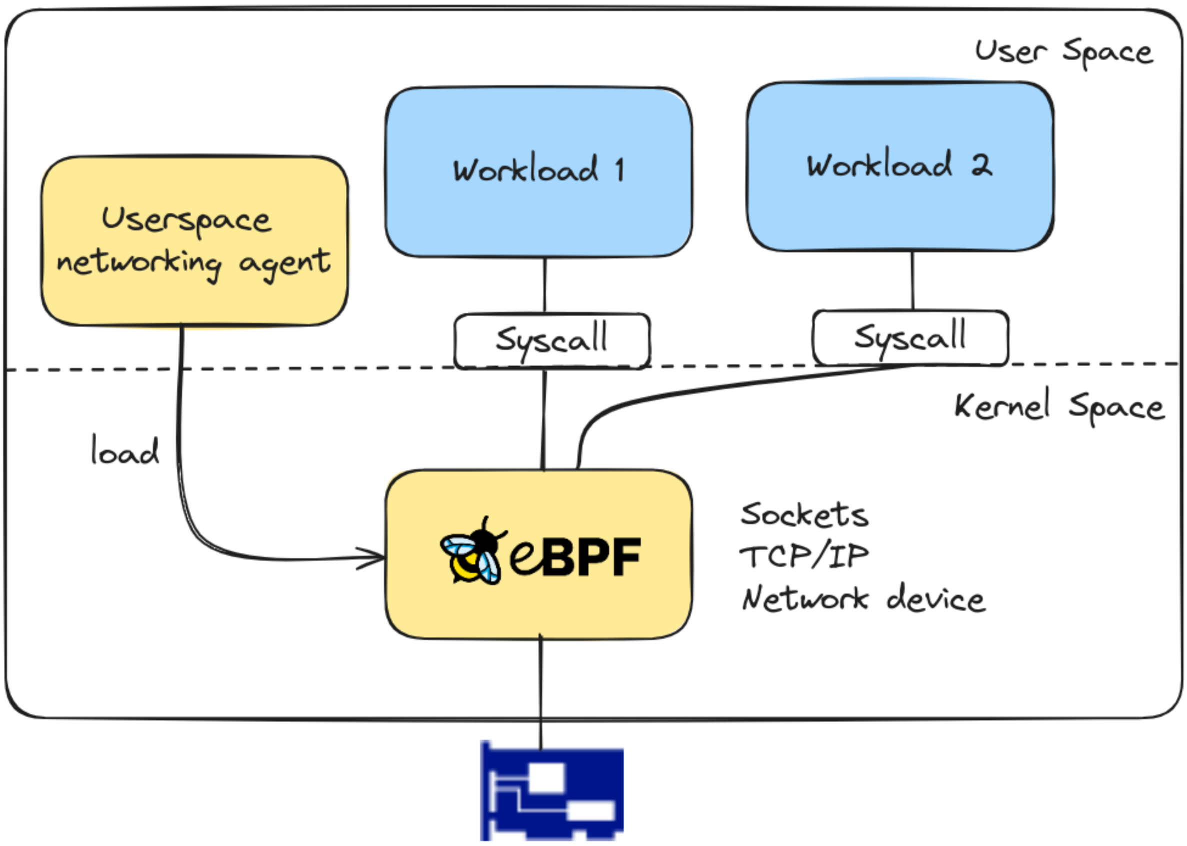


Networking

eBPF enables high-performance packet filtering and processing at various points within the networking stack. eBPF programs attached to hooks in the kernel can inspect, modify, or drop network packets without the need for user-space intervention. This avoidance of mode switching reduces packet latency and improves throughput.

XDP (eXpress Data Path) provides a framework for eBPF programs to be run as soon as the network driver receives a packet on ingress. The kernel's Traffic Control (TC & TCX) layer can be used in the networking data path at both ingress and egress and has access to the Socket Buffer structure (sk_buff). Additionally TC hooks into generic layers and does not require driver support, which makes it more flexible.

The TC hook is executed after the packet has been processed by the XDP hook if it was attached to the interface.



Security

eBPF security tools can combine the deep observability features of performance monitoring and tracing tooling, while additionally making contextualised handling decisions for anomalous events that may correspond to a threat being realised. Policies can be defined to detect classes of events, and in some cases prevent their ongoing execution. Prevention can be coarse (killing a suspicious process) or fine-grained (denying an activity). For finer-grained preventions, fmod_ret can be used to reject syscalls by altering their return values, and some network program types also enable accept/reject semantics.

Time-of-Check Time-of-Use (TOCTOU) issues can arise in security tooling. The goal is to analyse the actions that the system will perform accurately. However, if the security tool reads values from user-space memory and then those values are changed before the kernel acts on them, what is “used” by the kernel could differ from what you “checked” in user-space. TOCTOU races can be prevented by ensuring that the security tooling observes values after they have been transferred to kernel memory. The two main ways to do this are LSM (Linux Security Module) eBPF programs and directly hooking kernel internal functions via kprobe/kretprobe/fentry/fexit

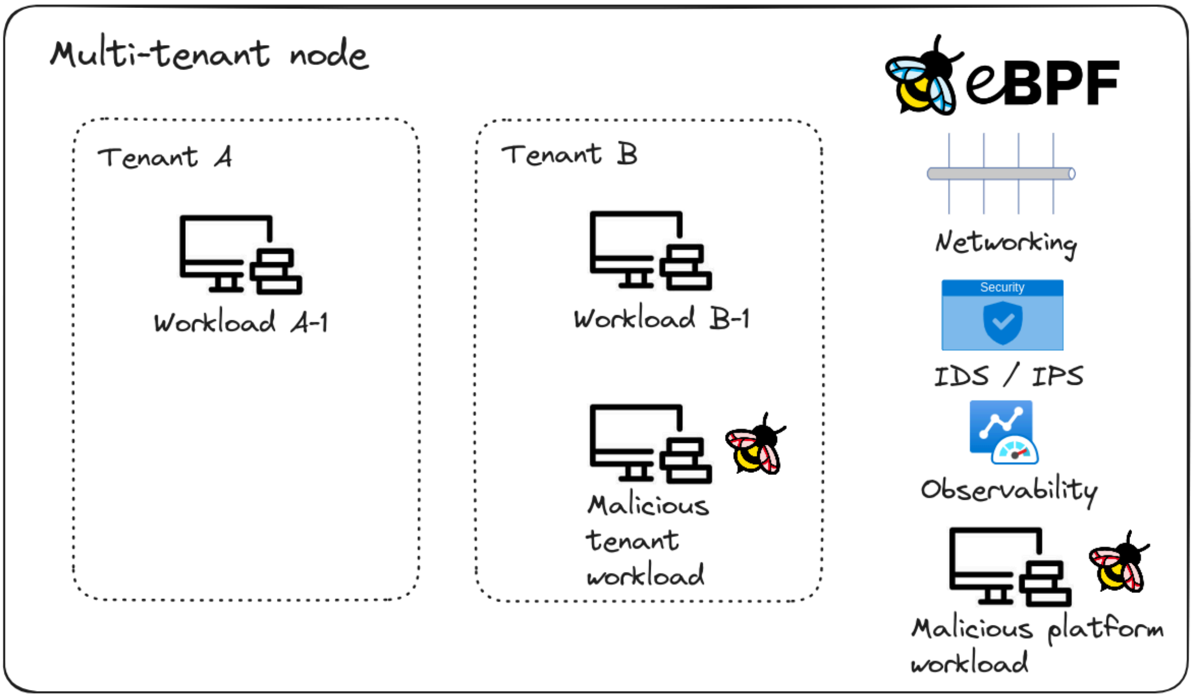
Threat Model

Threat Model Scope

To derive the most general set of end-user recommendations, we will make the following assumptions:

1. User space workloads run in a multi-tenant platform, with different tenants' workloads potentially running on the same Linux host
2. Workloads running on the shared platform are orchestrated by an unspecified external mechanism
3. eBPF programs are run by a central 'platform team', for any combination of the following three reasons:
    a. Observability and tracing for all workloads running on the shared platform
    b. Highly performant networking to route traffic to and from tenants' workloads, or to block any traffic that doesn't have a valid business justification
    c. Security tooling to detect and optionally block specified events such as Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS)

We do not make any assumptions about the security context of tenant workloads. Where workloads run with elevated privileges, malicious tenants could attempt to run eBPF programs to bypass security controls.



Attack trees

The following attack trees have been created as examples without any inherent controls in place. As such, without additional exploitable vulnerabilities, some of the theoretical attack paths will not be possible due to controls such as the verifier.

It is important to note that the eBPF runtime itself runs in the kernel, so vulnerabilities in controls such as the verifier may open these attack paths. Many of these threats are equivalent to those outside of the eBPF ecosystem.

Once the key attack scenarios have been explored through the attack trees, the threats are consolidated and summarised in [Detailed Threats and Controls](). The attack trees cover platform confidentiality and integrity, availability, and evading security tooling.

The initial compromise could come from a supply chain vulnerability, a Remote Code Execution (RCE) attack, or an internal threat, such as an employee.

The format of the attack trees is a top-to-bottom flow. It works from the "attacker goal" downwards, through the requirements and steps to achieve it. Leaf nodes are grey, logical AND nodes are blue, and logical OR nodes are green.

Confidentiality and Integrity

This threat model considers privileged, platform-level workloads running eBPF programs. If an external attacker can compromise a user-space agent which loads eBPF code, their goal may be to read sensitive data processed by one of the platform tenants. An external attacker may then leverage such a weakness in a user-space agent[5] which loads eBPF code to read sensitive data processed by one of the platform tenants.

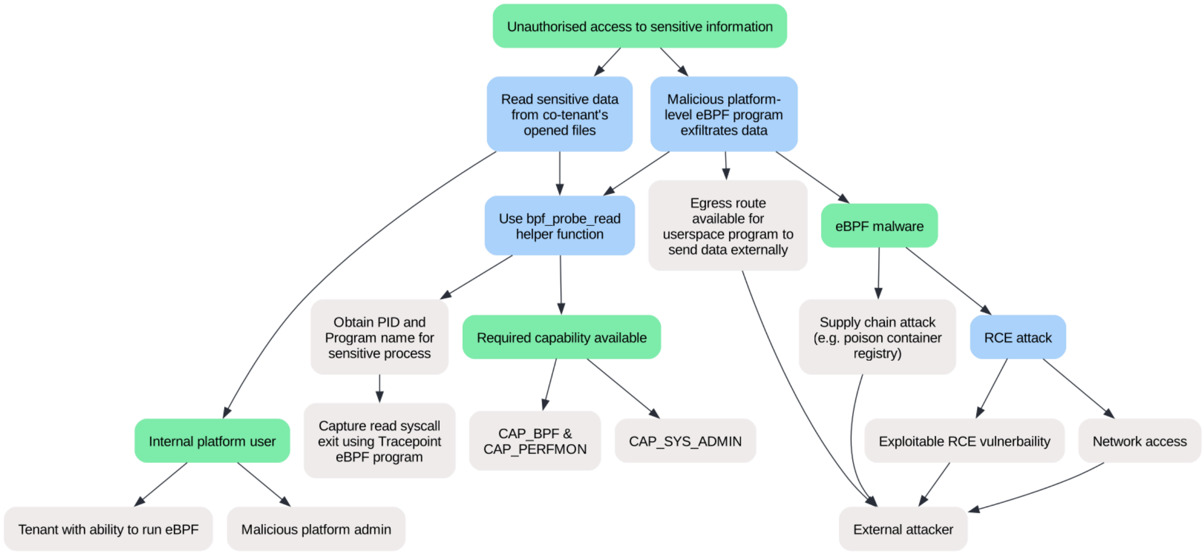
Provided the user-space agent's process has the correct privileges, it can attach a Tracepoint eBPF program that observes read syscalls (or utilise kprobe/fentry) and then use the bpf_probe_read helper function (i.e. at least with CAP_BPF and CAP_PERFMON capabilities added) to read sensitive data.

---

[5] Note that this also applies to any other privileged process which can load eBPF. According to the threat model scope we are considering user-space agents with these privileges, which we call 'platform-level eBPF tools'.

Given access to that process, an external threat actor would then require an egress route available to the user-space agent, to exfiltrate the collected data.

If no egress route is available, an external threat actor cannot carry out this attack. Instead, it may be undertaken by internal threat actors (e.g. privileged platform administrators) or privileged tenants permitted to run some workloads at the same level as the platform team.



A similar tree could be drawn if the attacker's goal were to compromise the integrity of data processed by a tenant workload. In this case, our eBPF program would need to use the bpf_probe_write_user helper function. From eBPF Helper Functions of Interest to Threat Actors, we can see that this would require the CAP_SYS_ADMIN capability[6] (whereas bpf_probe_read can be used with CAP_BPF plus CAP_PERFMON). The use of this helper has been included as an independent threat in Detailed Threats and Controls.

Availability

As with any code executed in the kernel, bugs or maliciously written eBPF code may crash nodes that run tenant workloads. In Mitigating Controls, we investigate how the eBPF verifier can mitigate these threats. At this stage of the threat model we are simply enumerating as many types of inherent threat as possible.

Similar to the confidentiality tree, the offensive eBPF helpers which could be used to carry out a denial of service (DoS) attack depend on the available capabilities. For example, a security tool acting as an IPS would need to kill processes matching indicators of compromise in a policy. To do this for legitimate purposes,

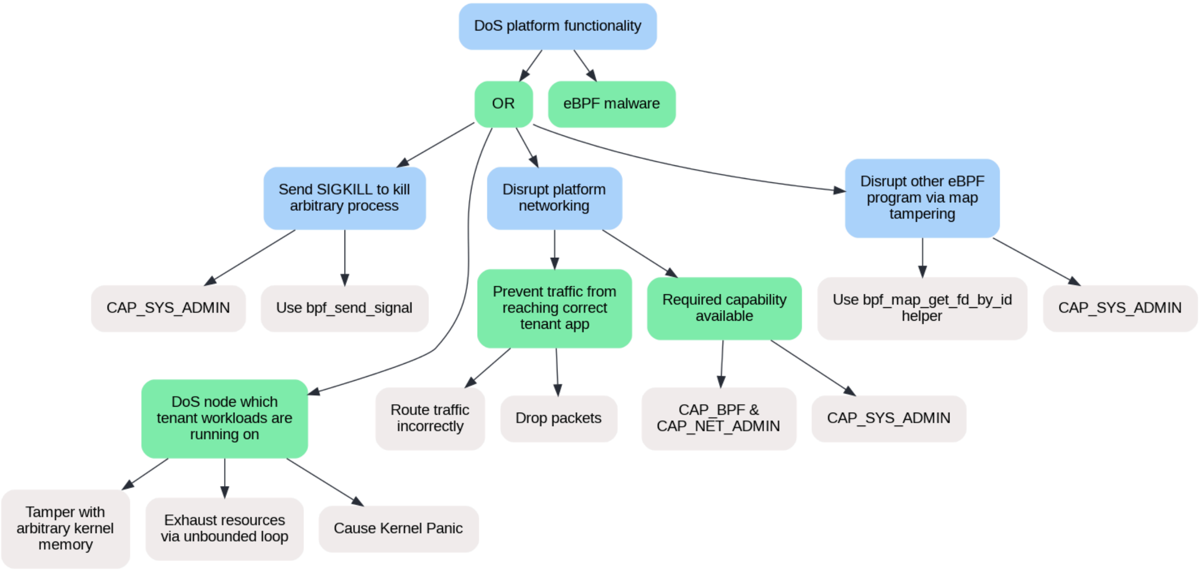
---

[6] And additionally, a kernel prior to the addition of lockdown mode, or inactive lockdown. If lockdown is enabled in "integrity" mode or above this helper cannot be used and this threat is mitigated (see KC-lockdown).

it would need to run with CAP_SYS_ADMIN (or as root), as CAP_PERFMON does not permit the use of bpf_send_signal (as seen in the kernel source code). If such a tool were to be compromised, arbitrary processes could be killed by sending a SIGKILL, causing a denial of service to platform tenants.

If an eBPF networking tool were to be compromised (or any process with at least the CAP_BPF and CAP_NET_ADMIN capabilities added), a denial of service attack could be carried out by preventing traffic from reaching its intended destination.

Finally, a malicious eBPF program could disrupt a legitimate platform eBPF tool via map tampering. The malicious program could access a map created by a legitimate platform eBPF tool if it could use the bpf_map_get_fd_by_id libbpf function in combination with CAP_SYS_ADMIN. By altering the data in these maps, a denial of service attack could be carried out, for example, by tampering with a tail call map facilitating a jump to another eBPF function.



[7]

Evade Security Tooling

eBPF's great power enables possible evasion of traditional security tooling, or attempts to execute attacks in a manner not identified by an eBPF security tool.

In the first instance, if an attacker has already gained root access to a node, eBPF could be used to build stealthy rootkit functionality. For example, the bpf_override_return helper function can alter data returned from the kernel, so malicious files and processes could potentially be hidden from user-space apps.

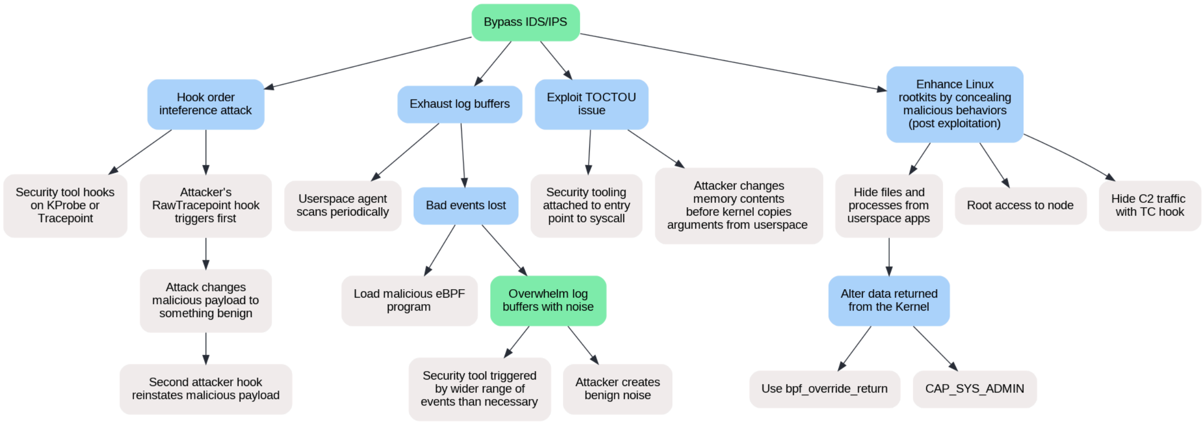
---

[7] For the leaves of the eBPF malware branch see the Confidentiality and Integrity attack tree above.

eBPF could also be used to hide command and control (C2) traffic from clients on the host by redirecting traffic to an attacker-controlled server using TC hooks. This attack can be detected with traditional network detection tooling outside of the host.

Three methods for evading eBPF security tooling are explored in the below attack tree:

- Exploiting TOCTOU issues with tools which attach to entry points of syscalls. This may allow an attacker to modify memory contents before the kernel copies arguments from userspace. In this case, what is ‘checked’ by the security tool may not be what is ‘used’.
- Executing a hook order interference attack, whereby the attacker installs one hook converting the input from malicious to benign before the security tool checks it and another to revert it to the malicious payload after checking.
- Attempting to blind security tooling by creating a large amount of benign traffic or logging. Large outputs can obscure, or even cause the loss of, data intended for monitoring due to overflows.



Mitigating Controls and Recommendations

A full suite of mitigating controls for the threats identified in the attack trees is included in Detailed Threats and Controls, and are summarised in this section.

After passing capability checks for loading eBPF programs, the next line of defence when loading an eBPF program is the verifier. The verifier performs the following checks to ensure that eBPF code is safe to run:

1. Program Analysis: The verifier performs a detailed analysis of the eBPF bytecode to ensure that the program adheres to safety and correctness constraints.
2. Control Flow Validation:
    a. Loop Detection: checks for the presence of loops in the eBPF program's execution flow. eBPF programs must have a predictable runtime, and loops can cause unpredictable execution times. The verifier ensures the program has a bounded loop, or is loop-free.
    b. Instruction Limits: ensure the program does not exceed the maximum instruction limit (see appendix).
3. Memory Safety:
    a. Bounds Checking: verifies that all memory accesses are within valid bounds to prevent buffer overflows. This includes checking and validating pointers.
    b. Stack Safety: ensures that the stack usage is within limits and properly managed.
4. Type Safety:
    a. Register State Tracking: tracks the type and value ranges of data in registers throughout program execution to ensure type-safe operations.
    b. Function Calls: checks that calls to helper functions follow the correct calling conventions, and that the arguments passed are of the correct type and within valid ranges.
5. No Undefined Behaviour:
    a. Instruction Semantics: ensures instructions don't perform illegal operations such as division by zero or accessing uninitialised memory.
    b. State Transitions: validates that each state transition (changes in register values, memory accesses, etc.) leads to a defined state.
6. Resource Constraints:
    a. Execution Time: ensures that the program will terminate in a finite amount of time, to avoid infinite loops or excessively long execution times.

   b. Resource Access: verifies that the program accesses kernel resources in a controlled manner, preventing resource leaks or unauthorised access.

7. Helper Function Safety: ensures that the program only uses allowed helper functions and that these functions are used correctly.

The verifier also checks that the process now attaching the eBPF program holds the required capabilities for both this eBPF program type[8] and the points it is attaching to. Root privileges or the CAP_SYS_ADMIN capability are not always necessary to run an eBPF program, and as such, it is important to understand which capabilities are required and to apply the principle of least privilege.

For example, an eBPF networking tool may be able to be run with CAP_BPF and CAP_NET_ADMIN, and a tracing tool may work with CAP_BPF and CAP_PERFMON. However, a security tool (acting in prevention mode) may need to use helper functions such as bpf_send_signal, and hence require root or CAP_SYS_ADMIN.

Given the privileged nature of eBPF tools for networking, observability and security, separation of duties and access control should be considered the purview of system administrators. In a multi-tenant scenario, it is recommended that a central platform team is responsible for the configuration and maintenance of these tools. For this reason, disabling unprivileged eBPF is advised and commonly the default in Linux distributions.

As with any software, regardless of the privileges it needs to run, software supply chain security is paramount. If an attacker could compromise the source code, build process or release artefacts of any application that runs with elevated privileges (including platform-level eBPF tools), any threats explored in the Threat Model could be realised. A set of supply chain security best practices can be found in the CNCF Software Supply Chain Best Practices paper.

If closed-source eBPF software is used, some due diligence and audit activities may not be possible (e.g. using OSSF Scorecard to see whether an open source project complies with best practices). In this case, using a complementary open source tool may be an option to detect or block suspicious activity.

Regardless of whether an eBPF tool is open or closed source, there will always be the question of “who watches the watcher?”. Although some eBPF tools act as security controls themselves, it is recommended that organisations maintain technical threat models which consider the case that these tools themselves are compromised. Devising controls for these threats will depend on the organisation’s threat environment and risk appetite, but may involve using complementary eBPF tools to detect specific classes of attack.

---

[8] A list of eBPF program types is in the kernel documentation. Program types have been added over time, and as such may not exist in older kernels.

It is recommended that this whitepaper is used alongside the materials it references to inform a bespoke threat model for your organisation's systems, accounting for the specifics of your system's eBPF workload orchestration and security. Once you've created a list of threats, determine the monitoring use cases to detect these threats, and based on those use cases choose the best tool to detect those events.

As an example, if you are concerned about the threat of an eBPF networking program using TC hooks to hide C2 traffic, it can help to use an external source of network monitoring data, as the true destination of packets will not be hidden for external monitoring tools.

Detailed Threats and Controls

This table expands upon threats from the attack trees. The "Inherent eBPF controls" column outlines in-built protections, and "Recommendations" covers controls to implement. Threats are unordered, and controls or recommendations may be common to multiple threats.

In general, it is recommended to create a threat model, derive monitoring use cases for bad things that could happen, monitor for dangerous events and alert on them. An eBPF tool could be considered to perform detections.

ID	Threat	Inherent eBPF controls	Recommendations
1	Malicious eBPF program uses bpf_probe_write_user helper function to write to memory of other process	When an eBPF program using this helper is attached, a warning including PID and process name is printed to kernel logs.	Apply the principle of least privilege. Rather than give tools the CAP_SYS_ADMIN capability, where possible use CAP_BPF, and additional capabilities such as CAP_PERFMON or CAP_NET_ADMIN as required (KC-cap-bpf). Consider the kernel lockdown feature (KC-lockdown) at "integrity" mode which (among other things) blocks this helper. In most Linux distributions, lockdown in "integrity" mode is enabled by default.
2	Malicious eBPF program tampers with arbitrary kernel memory	The verifier checks that the program is not trying to access memory outside the stack, or that it is not trying to access memory that is not mapped. Can BPF Overwrite Arbitrary Kernel Memory? - kernel.org/doc	
3	Denial of service via malformed eBPF program exhausting	The verifier checks for unbounded loops. Every instruction it verifies is counted and the	Consider applying a cgroup to limit the resources available to the eBPF loading process, as eBPF programs inherit the cgroup of their parent

	resources via unbounded loop	program is rejected if the number reaches a limit. As of kernel 5.2 it's one million instructions (KC-limit-increase) (which can be extended with tail calls) or 4096 for unprivileged eBPF. As the verifier improves, limits may be further relaxed. You can check for changes here or review the current source code.	(KC-ebpf-cgroup). Resource limits - Isovalent eBPF Docs - dylanreimerink.nl
4	Denial of Service via malformed eBPF program causing kernel panic	Verifier checks include: - Pointer bounds checking - Verifying that the stack's reads are preceded by stack writes - Preventing the use of unbounded loops - Register value tracking - Branch pruning eBPF programs use a predefined set of helper functions.	Maintain good hygiene in updating kernel versions to the latest provided by the vendor to fix any applicable kernel verifier bugs. If you encounter a crash not resolved upstream, engage with the kernel community to get it resolved and backported to stable releases.
5	Malicious eBPF program disrupts platform networking		Maintain awareness that every platform-level eBPF program launched with CAP_BPF and CAP_NET_ADMIN, or CAP_SYS_ADMIN can load eBPF programs to disrupt platform networking.
6	Malicious eBPF program tampers with map used by other eBPF program, causing	If a process can obtain a file descriptor to an eBPF object then it is assumed that future operations on that object, through the	Apply the principle of least privilege. Rather than give tools the CAP_SYS_ADMIN capability, where possible use CAP_BPF, and additional capabilities such as CAP_PERFMON or

	a denial of service	descriptor, are allowed. However, the bpf_map_get_fd_by_id libbpf function requires root or CAP_SYS_ADMIN to be used.	CAP_NET_ADMIN as required.
7	Denial of Service attack by malicious eBPF program terminating processes using bpf_send_signal helper function		With the high levels of privilege needed to run them, eBPF tools should be administered by a dedicated platform team in the case of a multi-tenant environment. Some eBPF tools (e.g. Tetragon) allow policies to be written which allow preventative action (e.g. kill suspect processes) to be taken when potentially dangerous events take place. It is recommended to run tools in audit mode first, to understand the impact of running a policy in prevention mode.
8	Malicious platform-level eBPF tool exfiltrates data		Implement egress controls limiting programs which load eBPF code to only communicate with approved external services.
9	eBPF malware via supply chain attack (e.g. poisoned container registry)		Due to the high level of privilege needed to run the types of eBPF tools considered in this report, establishing explicit trust relationships with the project(s) used is essential. Tools exist to assess open source projects for indicators of good maintenance and following security best practices, for example OSSF Scorecard. If deploying eBPF tools via containers, you can enforce image signing. Projects and Vendors following SLSA guidance and delivering provenance and a Software Bill of Materials (SBOM) provide additional confidence as to the artefact's origin.
10	RCE attack via an		Implement a vulnerability management

	exploitable vulnerability in platform userspace workload that loads eBPF program		process whereby CVE scans are performed on public artefacts before they are ingested. Information from vulnerability scans will include CVEs and their associated CVSS, but also can be supplemented with information about the exploitability (e.g. KEV or EPSS information) and software scorecards (e.g. OSSF Scorecard). Minimise the API surface in platform userspace programs that load eBPF programs, so that in the event of an RCE attack eBPF access is not readily available. Furthermore, privileged eBPF loaders should use mechanisms to authenticate the origin of BPF programs (e.g. with signing) to avoid being tricked into loading malicious programs. These loaders should also be hardened against memory corruption issues which can allow an attacker to gain control (read / write / execute) over privileged code.
11	Userspace program receives data via eBPF map and sends information externally to be accessed by an external attacker		Implement egress controls limiting programs which load eBPF code to only communicate with approved external services.
12	Malicious platform admin or tenant with the ability to run eBPF reads sensitive data from co-tenants' opened files		Administrative access should follow standard break-glass best practices of multi-party authorisation, and admin access should be limited to vetted programs (such as signed BPF programs and tooling scripts).

13	Enhance Linux rootkits by concealing malicious behaviours (post exploitation), e.g. modifying syscalls arguments or return code	bpf_override_return is available if the kernel was compiled with the CONFIG_BPF_KPROBE_OVERRIDE configuration option, and operates on functions tagged with ALLOW_ERROR_INJECTION in the kernel code, which is enabled by default in many distros. fmod_ret programs can also perform these actions for syscalls and functions prefaced with "security_". This helper helper is only available for the architectures having the CONFIG_FUNCTION_ERROR_INJECTION option. As of this writing, x86, ARM64, s390, powerpc, and csky architectures support this feature. (Helper Function 'bpf_override_return' - Isovalent eBPF Docs - dylanreimerink.nl).	Consider complimentary eBPF tooling to detect potentially malicious eBPF activity.
14	Exploit TOCTOU issue to evade security tooling, which hooks on the entry point to syscalls		Maintain awareness of potential TOCTOU issues when using a security tool which hooks on entry points to syscalls. Leverage bpf-lsm, or kprobe/kretprobe/fentry/fexit, to operate on kernel instead of user-space memory (KC-bpf-lsm).
15	Overwhelm log buffers with noise to blind userspace agents		Using separate ring buffers for various classes of events. e.g. separate ring buffers for on-host detection events and general audit logs. Even within audit events, one could consider using separate logging buffers for subclasses (e.g. process, network, module loads

			events).
16	Execute hook order interference attack to evade security tool which hooks on KProbe or Tracepoint		Be aware that with monitoring or security tooling the order actions are processed could affect what is seen. In order to accurately process actions the system will perform, tools must check at the last opportunity to change the actions or ideally after the action can no longer be changed. With LSM BPF you can check system actions (and act upon them) once they are unchangeable. (KC-bpf-lsm).
17	Unprivileged user exploits kernel vulnerability by using unprivileged eBPF program type (such as BPF_PROG_TYPE_SOCKET_FILTER, BPF_PROG_TYPE_CGROUP_SKB)	The mainline Linux kernel and most distributions disable unprivileged eBPF out-of-the-box (KC-disallow-unpriv-bpf).	It is recommended that unprivileged eBPF is disabled. In most distributions, this is the case by default. Since kernel 5.13 (KC-disallow-unpriv-bpf) many Linux distributions disable unprivileged eBPF (i.e. `/proc/sys/kernel/unprivileged_bpf_disabled` is set to `2`). This is a sensible default for most machines that allows re-enabling unprivileged eBPF. To fully block unprivileged eBPF (until reboot) set the value to `1`.
18	Lack of timely OS patching leads to an exploitable kernel vulnerability	Programs which are tightly coupled with a kernel version can delay update adoption[9]. eBPF programs can utilise Compile-Once Run-Everywhere (CO-RE). This in combination with BPF's stable application binary interface (ABI) means you can update without changes. The one exception is BPF helpers	Utilise eBPF programs with CO-RE and leveraging BTF so they're a reduced blocker to kernel updates. eBPF programs compiled for your exact kernel might not be a blocker if you or your distribution provider can recompile them. Dedicate time to maintaining up-to-date eBPF programs for target platforms, through ongoing testing and integration into newer kernels as they are released. Leverage CO-RE and BTF to more easily adapt to a range of target platforms.

---

[9] Kernel CVE announcements commonly include the quote: "The Linux kernel CVE team recommends that you update to the latest stable kernel version for this, and many other bugfixes."

		which walk kernel data structures or call kfuncs. BTF can help alleviate issues on kernel updates but it is still not a stable ABI and no guarantees are given and testing is necessary. [Does BPF have a Stable ABI? - kernel.org/doc]() [BPF Kernel Functions (kfuncs) - Lifecycle Expectations - kernel.org/doc]() [BPF CO-RE reference guide - nakryiko.com]() [eBPF Tutorial by Example: Learning CO-RE eBPF - eunomia.dev]()	Failures may still occur in production, despite integration testing especially when there are a lot of kernels to test. Thus loading and functionality of BPF programs should be monitored in production. Maintainers of eBPF tools leveraging kfuncs should monitor kernel releases for BTF changes.
19	TC is used to hide C2 traffic		External monitoring tools or hardware are sufficient to detect such attacks, as once the packet has been processed by the kernel, it is possible to see its actual destination. ([KC-bpf-link]()) [eBPF Offensive Capabilities - sysdig.com]()

Conclusions

eBPF is a deeply powerful foundational technology, with many benefits for the future of infrastructure software. By safely enabling custom, kernel-level software without requiring kernel recompilation or reboot, it provides options for increased security over traditional approaches due to its rigorous validation of user-supplied code.

As kernel changes or module loading are not required, eBPF is a more stable, observable, and predictable option in environments where module-based approaches may have been used previously. eBPF is supported by a growing number of tools and frameworks (e.g. bpftrace, Cilium, Falco, Hubble, Tetragon) that make it easier to implement complex use cases and effectively tackle sophisticated networking, observability, and security challenges.

Given its generic scope, this paper is intended to inform bespoke threat models tailored to an organisation as it plans eBPF adoption. When replacing existing tooling with eBPF-based tools, existing policies, controls, and profiles (such as seccomp or LSMs) should be updated accordingly as interfaces to the kernel are different (e.g. the BPF syscall). With this approach, the many benefits of eBPF can be realised, while risks are captured and mitigated by defence-in-depth controls.

The elevated privileges that eBPF requires do not introduce novel vulnerabilities beyond what superuser access could achieve; rather, eBPF provides a platform for building additional security controls that make systems more robust. While eBPF could simplify certain attack paths for an attacker already possessing substantial privileges, it does not make these attack vectors feasible on its own.

eBPF’s abilities enable more precise operations, making it easier to limit the risks associated with privileged processes and improving an organisation's security posture. By following security best practices, such as the principles of least privilege and separation of duties, organisations can fully leverage eBPF to enhance security and observability.

Appendix

Kernel Changes of Note

This list attempts to track kernel changes that affect the Threat Model and the controls and recommendations in the document. It is impractical to list every change to eBPF, but these are the most significant alterations we have identified for the Threat Model.

As of writing kernel 4.19 is the oldest release still under mainline maintenance. View the current maintenance policy here. Changes prior to this release are unlikely to be noted.

Identifier	Version	Change
KC-disallow-unpriv-bpf	5.13 - commit	Introduced the ability to disallow unprivileged eBPF
KC-disallow-unpriv-bpf (Related)	5.16 - commit	Disallowed unprivileged eBPF by default in the mainline kernel
KC-cap-bpf	5.8	Introduced CAP_BPF and CAP_PERFMON
KC-bpf-lsm	5.7	Introduced BPF LSM
KC-bpf-link	5.19	Introduced BPF Link (TCX order)
KC-lockdown	5.14 - commit	Lockdown integrity mode now blocks probe_write_user
KC-limit-increase	5.2 - commit	Increased the complexity limit for privileged eBPF from 131,072 to 1,000,000
KC-ebpf-cgroup	5.11	Switched from rlimit to cgroups for eBPF memory accounting

References

- eBPF.io website
- eBPF Foundation website
- Shostack's 4 Question Frame for Threat Modeling - github.com
- eBPF for Anything! - isovalent.com
- Cross Container Attacks: The Bewildered eBPF on Clouds - He, Yi, et al. 2023 - usenix.org
- Learning eBPF by Liz Rice - oreilly.com
- Introduction to CAP_BPF - mdaverde.com
- Linux 6.10 - include/uapi/linux/capability.h - git.kernel.org
- Linux 6.10 - kernel/bpf/helpers.c - git.kernel.org
- bpf-helpers(7) - Linux manual page - man7.org
- Helper functions - Isovalent eBPF Docs - dylanreimerink.nl
- bpf(2) - Linux manual page - man7.org
- Next-Generation Observability with eBPF - isovalent.com
- Linux 6.10 - struct sk_buff - docs.kernel.org
- BPF and XDP Reference Guide - Traffic Control - docs.cilium.io
- eBPF Offensive Capabilities - sysdig.com
- https://github.com/ossf/scorecard
- Linux 6.10 - eBPF Program Types - docs.kernel.org
- Program types (Linux) - Isovalent eBPF Docs - dylanreimerink.nl
- Linux 6.10 - BPF Design Q&A - kernel.org/doc
- BPF Kernel Functions (kfuncs) - kernel.org/doc
- Active Kernel Releases - kernel.org
- https://github.com/pathtofile/bad-bpf
- On Bypassing eBPF Security Monitoring - doyensec.com
- Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface - hovav.net
- eBPF Tutorial by Example: Learning CO-RE eBPF - eunomia.dev
- An Analysis of Speculative Type Confusion Vulnerabilities in the Wild - Kirzner, Ofek, et al. 2021 - usenix.org
- IETF RFC 9669: BPF Instruction Set Architecture (ISA) - rfc-editor.org