EPF: Evil Packet Filter

Di Jin
Brown University

Vaggelis Atlidakis
Brown University

Vasileios P. Kemerlis
Brown University

Abstract

The OS kernel is at the forefront of a system's security. Therefore, its own security is crucial for the correctness and integrity of user applications. With a plethora of bugs continuously discovered in OS kernel code, defenses and mitigations are essential for practical kernel security. One important defense strategy is to isolate user-controlled memory from kernel-accessible memory, in order to mitigate attacks like ret2usr and ret2dir. We present EPF (Evil Packet Filter): a new method for bypassing various (both deployed and proposed) kernel isolation techniques by abusing the BPF infrastructure of the Linux kernel: i.e., by leveraging BPF code, provided by unprivileged users/programs, as attack payloads. We demonstrate two different EPF instances, namely BPF-Reuse and BPF-ROP, which utilize malicious BPF payloads to mount privilege escalation attacks in both 32- and 64-bit x86 platforms. We also present the design, implementation, and evaluation of a set of defenses to enforce the isolation between BPF instructions and benign kernel data, and the integrity of BPF program execution, effectively providing protection against EPF-based attacks. Our implemented defenses show minimal overhead ($< 3\%$) in BPF-heavy tasks.

1 Introduction

The security of a computer system can only be as good as that of the underlying OS kernel. The kernel provides a relatively simplistic abstraction for programs to build on top of, and mediates their access to system resources. Hence, the confidentiality and integrity of user programs rely solely on the security of the OS kernel itself. Yet, the kernel is hard to defend, due to its unique execution model, and the sheer size and complexity of its code. With the development of automated kernel-code testing tools, such as syzkaller [13], thousands of bugs have been found across different OSes [12]. It is even pointed out that bugs are discovered faster than they are fixed [113]. The abundance of errors and vulnerabilities in OSes amplifies the importance of protection mechanisms that reduce the exploitation potential of kernel vulnerabilities.

Standard defenses, such as W^X [6] and ASLR [6], are also adopted by the Linux kernel, aiming to stop and mitigate code-injection- [64] and code-reuse-based [40] attacks. More specifically, these defenses aim to limit the attacker's ability to successfully mount an attack. However, attacks such as ret2usr [68], which completely encode their payloads in userspace, indicate that kernel exploitation is made significantly easier by the relatively *weak separation* between the kernel and userspace (as opposed to a user application, such as web server, where the interface between it and the clients is much more well-defined and limited). Similarly, protection mechanisms like SMEP [112] try to stop kernel attacks by limiting the attacker's ability to encode attack payloads in a kernel-accessible manner (more attacks and defenses in this direction are discussed in Section 2.1).

In this paper, we explore the possibility of abusing the *BPF infrastructure* in the Linux kernel—more specifically, by leveraging BPF programs as attack payloads. We have identified three properties that make BPF programs a promising candidate for such a task. First, BPF programs can be created by *unprivileged users*, and contain memory contents chosen by the user (such that they can be used to encode malicious contents). Second, BPF programs can be created in *large amounts*, such that during exploitation valid references to them can be constructed (e.g., by just guessing) with good probability. Third, BPF programs are created by users, but "*consumed*" by the OS kernel. Access to such payloads cannot be prevented by existing, strong kernel-user isolation mechanisms (e.g., XPFO [67]), because they cannot be differentiated from regular data that the kernel operates upon.

Yet, unlike previous techniques where the payload content is encoded inside regions acting effectively as "byte buffers", a BPF programs's in-memory representation has a non-trivial structure. To address this problem, we develop special code-reuse strategies, dubbed as *EPF*, to utilize payloads with the constraints introduced by the corresponding BPF structure. BPF programs can aid exploitation because they are not strongly-isolated from normal kernel data.

And to defend against such EPF-style attacks, we develop a set of defenses that enforce strong BPF-to-kernel isolation. Recognizing the fact that BPF is essentially a "virtual architecture", we follow a roadmap similar to that of isolating native code from pure data: (1) BPF code should not be read or written as regular, kernel data; (2) regular kernel data should not be executed as BPF code; and (3) BPF code should not be reused as semantically different BPF code.

To summarize, we make the following contributions:

- We introduce a novel, high-volume, undefended method to inject payloads in kernel space for aiding the exploitation of memory errors in kernel code, and we create systematic techniques to utilize them in different architectures. Our methods enable attacks that bypass state-of-the-art defenses that focus on enhancing kernel-user isolation.
- We develop defenses against exploitation that (ab)uses the BPF infrastructure by enforcing strong isolation between BPF code and regular kernel data, and the integrity of BPF execution under attack.
- We evaluate both our attacks and defenses. We create exploits using our techniques on four different real-world vulnerabilities. We integrate our defenses into Linux kernel and demonstrate low overhead ($< 3\%$) on BPF-heavy tasks.

2 Background

2.1 Kernel Exploitation and Defense

Exploitation is the process of tampering with a victim system/software codebase by abusing vulnerabilities in it [95]. In the case of operating systems (OSes), because OS kernels are typically written in memory- and type-unsafe languages, like C, C++, and ASM, the most common approach to their exploitation entails abusing memory errors in kernel code [73,74,88,96]. In general, two are the dominant exploitation strategies (re: memory errors): *code-injection* [64] and *code-reuse* [40]. Code-injection leverages memory corruption vulnerabilities to place malicious code (i.e., *shellcode*) in the victim's address space before corrupting control data (e.g., *return address*, *function pointers*, *dispatch tables*) to steer execution to it. In contrast, code-reuse stitches together *existing* (i.e., benign) code snippets, in an out-of-context manner, to perform the respective (malicious) computation.

In addition to the above, there exist *kernel-specific* exploitation techniques that address unique challenges of the OS kernel setting. Throughout the evolution of kernel attacks and defenses, regarding memory errors, the ability to create attacker-controlled, exploit-time-accessible payloads has been at the forefront of the subject matter. For example, in *ret2usr* [68] attacks, the adversary first corrupts a code pointer, and then diverts the control flow to userspace code, tricking the kernel into executing malicious (shell)code with elevated privileges. Alternatively, a different flavor of such an attack employs code-reuse techniques (e.g., ROP [100]) with payloads placed in userspace [74, 77]. An essential property of ret2usr-like attacks is the placement of their payload (e.g., shellcode or code-reuse payload) in userspace.

To provide protection against exploits that follow the ret2usr approach, various defenses focus on stopping the respective payloads from being *accessible*. CPU features such as SMEP [112], SMAP [49], PXN [2], PAN [7], as well as software solutions such as kGuard [68], PaX's KERNEXEC [89] and UDEREF [90, 91], were introduced to prevent userland code/data from being executed/accessed freely by the OS kernel. Seeking new ways to provide payloads for exploiting kernel vulnerabilities, without accessing userspace, in the past we proposed to (ab)use the implicit memory sharing between userspace and the kernel: i.e., the `physmap` region [67].

For performance reasons, modern OS kernels keep a continuous mapping of the physical memory (or part of it) in kernel space, which naturally contains user-controlled content. In such attacks, dubbed *ret2dir*, the adversary tries to allocate enough physical pages containing the respective payload, and through the implicit sharing of `physmap`, the payload will be utilized/accessed in a later stage via code injection or reuse. To defend against ret2dir-based attacks, we introduced the concept of *XPFO* (eXclusive Page Frame Ownership) [67]. XPFO prevents the kernel from accessing any memory page that houses userland content, using a kernel (`physmap`-resident) address. Again, the defense impedes the attackers' ability to access their payload.

In this work, we show that unintended access and implicit sharing are not the only reliable sources of *payload injection*. As it turns out, the ability to "push" BPF programs [83] in kernel space provides the attacker with enough control over the contents of kernel memory, to the extent that BPF can be used as an *arbitrary* payload-encoding mechanism. As BPF programs are designed to *live in*, and *used by*, the OS kernel, such payloads bypass all defenses that rely on the strong isolation of kernel- from user-space [66].

2.2 BSD Packet Filter

Design and Usage The BSD Packet Filter (BPF) [83] was originally designed for *filtering* packets during network monitoring: by providing the kernel with "instructions" regarding how to filter packets, before delivering them to a monitoring process in userspace, BPF eliminates unnecessary data copying and context switching. The design and implementation of the Linux Socket Filter (LSF; i.e., our main subject of study) [60] was inspired heavily by BPF. However, it has recently evolved into a generic utility, acting as a *universal* in-kernel virtual machine [46]: its execution is strictly *sandboxed*, and no unintended *side-effects* escape confinement.

A wide range of kernel components, and applications, make use of the expressiveness and security provided by BPF. Docker [10], Firefox [9], and Chromium [1], as well as automated system call (syscall) filtering schemes, like `sysfilter` [52], Confine [55], and Chestnut [42], use `seccomp-BPF` [80] to specify syscall filtering policies. In addition, BPF programs can be attached to Kprobes (kernel probes) [105], giving rise to powerful kernel tracing tools [38], while BPF-based networking frameworks are developed to enable agile packet processing [15, 78]. Lastly, BPF programs have also been proposed to be used in FUSE (Filesystem in USErspace) [34] to reduce context switching overheads.

Features Two different flavors of BPF exist in Linux: classic BPF (cBPF); and extended BPF (eBPF). Internally, cBPF is converted to eBPF, and therefore only a single execution engine (for eBPF) exists nowadays [70]. cBPF programs are used for socket filtering (`setsockopt`), and syscall filtering (`prctl`, `seccomp`), while eBPF programs are managed with the `bpf` syscall, allowing kernel subsystems to implement different methods for attaching/invoking eBPF code.

cBPF is similar to the original BPF, with two general-purpose registers and 16 (addressable) scratch memory slots, all of which are 32-bit wide. In contrast, eBPF has 10 general-purpose, 64-bit registers, and is equipped with a set of helper functions to access either eBPF *maps* (i.e., a family of key/-value store data structures) or other, *internal functionality* (like getting the current process-ID or generating random numbers) [3]. eBPF maps can be made accessible from multiple eBPF programs or user processes [4].

Both cBPF and eBPF specify a RISC-like instruction set, allowing: (a) loading and storing operations (re: immediate values or scratch memory slots); (b) moving values between registers; (c) performing arithmetic and logical operations; and (d) branching. To guarantee termination, there is no *indirect* branching, and the branch instructions can only *jump forward*. In addition, eBPF provides specific instructions for invoking helper functions and other eBPF programs. JIT (Just-In-Time) compiling for eBPF [45] instructions, down to machine code, allows for performance gains [99], compared to using the eBPF interpreter. Lastly, popular tools have also added support for eBPF. LLVM supports BPF as a backend since v3.7 [81], so that developers can choose to write BPF programs using a syntax similar to C. Similarly, BCC [61] and `libbpf` [8] are frameworks that allow developers to easily create and interact with loaded BPF programs.

Security The isolation between the BPF runtime and OS kernel is crucial for the security of the latter. For performance reasons, BPF favors static, *ahead-of-time* checking (over dynamic, runtime checking), when enforcing such isolation. The static checker for cBPF ensures the following properties: (1) jumps (i.e., branches) do not go backwards; and (2) scratch space accesses target initialized locations only. Any program that cannot be statically verified by the checker is rejected. In case of packet filtering, the validity of access to a packet cannot be determined statically, as packet sizes may differ at runtime. Hence, such accesses will be translated to calling helper functions, and bounds are enforced at runtime.

The checker for eBPF is more complex [84]. Same as the cBPF checker, it needs to make sure that control flow does not go backwards. It also validates, and replaces, eBPF map and helper function references, and analyzes register value types, such that the corresponding operations lead to predictable memory accesses to safe locations. The verifier is known to be prone to errors, such as incorrect value range analysis [16, 17, 21] and insufficient protection against speculative execution-related vulnerabilities [19, 22]. Hence, the unprivileged `bpf` syscall is disabled by default [48] (`defconfig` in x86 Linux), while distributions are adopting a "on/off" choice [51, 106].

BPF has also been used as aid in the exploitation of *transient execution* vulnerabilities [72], as well as in settings that involve limited memory corruption capabilities (wrt spatial ranges and/or value choices) [63, 107]. Concerns regarding the former have led to removing the interpreter entirely, and always using JIT-compiled BPF, in certain settings, in order to reduce the risk of Spectre-like attacks (i.e., via `CONFIG_BPF_JIT_ALWAYS_ON` [103]). However, BPF JIT comes with its own set of security concerns.

First, because BPF JIT provides fine-grain control of native code (i.e., instructions) in kernel space, it is favored by specific transient-execution attacks [32, 71] (against the OS kernel). Second, it has been shown that the JIT engine can be used to facilitate *code injection* [82, 98], and hardening techniques such as *constant blinding* [36] and *code-offset randomization* [53] are added to counter these attacks. Lastly, the JIT compiler itself also suffers from errors. Nelson et al. [86] proposed the use of formal methods to ensure the correctness of JIT compilation, in which they report 82 bugs, demonstrating the difficulty of developing a correct (BPF) JIT compiler. The above issues have made the choice between the BPF interpreter and JIT compiler a difficult one, because both sides come with different security trade-offs. Notably, `defconfig` in x86 Linux, as well as popular distributions, like Debian, choose to keep the BPF interpreter compiled-in, which is what we assume in this work.

3 Threat Model

3.1 Adversarial Capabilities

We consider an attacker who is a local, *unprivileged* Linux user, aiming at escalating their privileges. More specifically:

Unprivileged Access The attacker is able to perform anything an unprivileged user can, like executing arbitrary code in userspace, invoking syscalls, accessing the filesystem, and interacting with OS interfaces (`procfs` [14], `sysfs` [11]).

BPF Functionality The attacker has the ability to *push* cBPF programs in kernel space. First, BPF should be enabled in the kernel. This is true for any Linux kernel compiled with network support, because the BPF subsystem is turned on by `CONFIG_BPF`, which is selected by `CONFIG_NET` [104]. Second, the attacker should have access to the BPF infrastructure, which is always the case because the `setsockopt`, `seccomp`, and `prctl` syscalls are *not* privileged. Third, the interpreter needs to exist in kernel code, which means that `CONFIG_BPF_JIT_ALWAYS_ON` should be off (§2.2). We do not require the `bpf` syscall being available to the attacker, which can be used to create eBPF programs. Actually, the *unprivileged* `bpf` syscall is disabled in all major distributions [51, 106].

Memory Errors We also assume that the attacker has access to a (at least one) memory corruption vulnerability in kernel code, and that by abusing this vulnerability they are able to overwrite code and data pointers, either in a *temporal* (e.g., use-after-free [23, 24]) or *spatial* (e.g., out-of-bound access [27, 28]) manner. We do not assume any error, or bug, in the BPF interpreter, verifier, or JIT compiler. Although the correctness of these components is challenging [25, 26, 86], this is something orthogonal to our attack(s)/EPF.

3.2 Hardening Assumptions

On the kernel side, we assume that W^X [6] is enforced, such that code injection is not possible. In addition, we consider that the kernel is hardened against ret2usr attacks, using SMEP [112]/PXN [2] and SMAP [49]/PAN [7]. Furthermore, we assume that no implicit memory sharing can take place between userland processes and the OS kernel, and hence ret2dir attacks are not attainable. Note that this is a strong assumption, as currently Linux does not employ a comprehensive defense against ret2dir, like XPFO [67]. We also assume that the page tables are protected against tampering with page table integrity mechanisms, such as PT-Rand [50] or xMP [94]. Lastly, kernel ASLR [54] is orthogonal to our attack(s); if deployed, EPF leverages known techniques to bypass it (§7).

4 Evil Packet Filter

We use the term EPF (Evil Packet Filter) to refer to a set of attacks that (ab)use the BPF infrastructure for injecting malicious payloads in kernel space. EPF allows bypassing existing isolation mechanisms [7, 49], which prevent user-controlled content from aiding kernel exploitation (§3.2). Due to the nature of the in-memory representation of BPF programs, making use of them as an attack vector is quite challenging. First, we describe certain design/implementation details of BPF, how BPF programs are created, what malicious content can be "hidden" in them, and how an attacker can locate them. Then, we describe two EPF-based attacks: *BPF-Reuse* (EPF v1—variant 1) and *BPF-ROP* (EPF v2—variant 2).

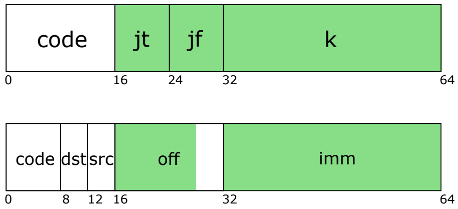

Figure 1: Fields and their sizes (in bits) in cBPF (top) and eBPF (bottom) instructions. Green regions are the parts of the instruction that can be controlled by an attacker.

4.1 Linux BPF Internals

BPF-code Management cBPF programs are primarily created via `setsockopt`, `prctl`, and `seccomp`, which can be used to "push" (cBPF) filtering code in kernel space; eBPF code can be copied in kernel space using the `bpf` syscall.

Both cBPF and eBPF programs are represented using the same data structure: that is, `struct bpf_prog`. We refer to this data structure as the "BPF program"; BPF programs pushed by `setsockopt`, `prctl`, and `seccomp` are considered 'cBPF', while those copied in kernel space by `bpf` are 'eBPF'. `struct bpf_prog` is always *page-aligned* when allocated. After a cBPF program is loaded into kernel space, the cBPF instruction array is duplicated, and stored separately—we call this instruction array as the *original cBPF code*. This code is referenced from `struct bpf_prog` by a member pointer `orig_prog`, allowing the corresponding process to retrieve the original cBPF code later (if needed). Then, the cBPF code is statically verified for safety, and translated *in-place*, becoming a *verified BPF program* (§2.2). Notably, the verification of cBPF programs is different from the one of eBPF programs, and after the cBPF instructions are translated to eBPF instructions, they are not verified again.

Lastly, a verified BPF program goes (optionally) through the process of being JIT-compiled (when `/proc/sys/net/core/bpf_jit_enable = 1`), while the memory permissions of pages that host the BPF program become *read-only*.

Data Structures The BPF program data structure (in the case of cBPF) consists of a header, which includes a pointer to the *interpreter* function, and a pointer to the original cBPF code, and an array of instructions. After verification, the array of instructions contains only eBPF code, due to the in-place cBPF⇝eBPF translation. (We refer to this array as the eBPF code. Notice that, internally, both cBPF and eBPF programs are represented with eBPF code.) BPF programs are allocated from the `vmalloc` region, whereas the original cBPF code is duplicated using `kmalloc` and lives in `physmap` [67].

In cBPF instructions (Figure 1, top), the `code` field is the opcode, defining the respective operation; `jt` and `jf` are two fields used for specifying where to jump if a predicate is true or false; and `k` is used for encoding immediates. Similarly, in eBPF instructions (Figure 1, bottom), both `src` and `dst` encode a number between 0–10, corresponding to one of the

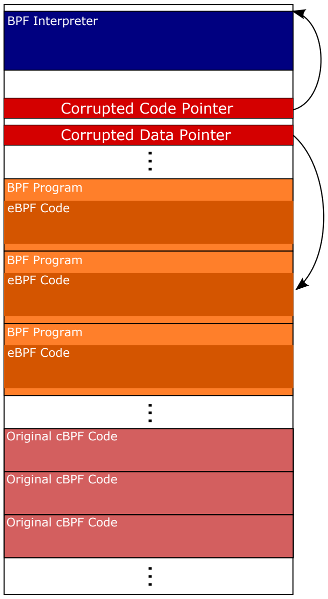

Figure 2: Memory layout during an EPF v1 (BPF-Reuse) attack. The corrupted code pointer points to the BPF interpreter function, and, at the time of its invocation, its argument points to one of the malicious BPF programs.

10 general-purpose registers or the frame pointer; the `off` field is used by memory load/store instructions, and jump instructions, to specify the offset of such operations; and `imm` stores the value of instructions that require an immediate.

Payload-encoding Challenges Ideally, the attacker would like the memory region they control to be of *arbitrary* size and content. However, this is not the case for BPF. Both cBPF and eBPF instructions correspond to 8 bytes, and not all bytes can take arbitrary values, because some of them have restrictions due to verification or translation. Only `imm` and `k` can be used in an *unconstrained* manner, and therefore we will only use these fields for EPF purposes—this corresponds to every other 4 bytes in the {c, e}BPF instruction array. Other fields can be *partially-controlled* or are only controllable iff the BPF program is constructed in a specific way. Hence, more fine-grain control is also possible (see Figure 1), but this is something that we do not explore for mounting an EPF attack.

BPF-code Spraying Although an attacker can control parts of a BPF program's content (both in the case of translated eBPF code and the original cBPF code), they still need to create *reliable references* to such BPF instructions. This can be achieved by *spraying*: i.e., saturating kernel space with BPF programs, and, as a result, a randomly-chosen location will likely contain (malicious) BPF code [67]. Both cBPF and eBPF code are page-aligned, and hence no additional care is needed to locate memory offsets within a page. In the Linux kernel, all allocated objects can be found in the `physmap` region, which maps the whole RAM.

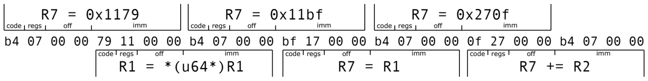
R7 = 0x1179
R7 = 0x11bf
R7 = 0x270f
code
regs
off
imm
b4 07 00 00 79 11 00 00 b4 07 00 00 bf 17 00 00 b4 07 00 00 0f 27 00 00 b4 07 00 00
R1 = *(u64*)R1
R7 = R1
R7 += R2

Figure 3: A snippet of the same eBPF code interpreted from different offsets (above vs. below).

During our experimental evaluation (§6.2), we discovered that `setsockopt` is the most effective spraying apparatus: it can be used to saturate ≈80% of all RAM, if the attack can utilize both the translated eBPF code and the original cBPF code, or ≈40% when only one of the two can be used.

BPF Interpreter The interpreter function receives two arguments: a *context* pointer and an (eBPF) instruction pointer. Context is the input to the BPF program—for example, in packet filtering, context points to the respective network packet. The value of the context pointer is loaded onto eBPF register `R1`. (The interpreter will not use this value unless it is used by the BPF program.) The second argument points to the eBPF code that is assumed to be verified. Hence, the interpreter does not validate any (eBPF) operation during execution. Because of its expressiveness, and relatively few side-effects, the interpreter is useful, as a code-reuse target as we will demonstrate in our EPF v1 (BPF-Reuse) attack.

4.2 EPF v1 (BPF-Reuse)

As we explained earlier (§4.1), an attacker can reliably control a large portion of kernel space, by (ab)using the BPF infrastructure, but they can only inject arbitrary values on every *other* 4-byte word—and thus cannot encode traditional code-reuse (e.g., ROP) payloads, which would require controlling 8 consecutive bytes in 64-bit platforms, like x86-64. However, the Linux kernel contains a powerful subsystem that can be leveraged to perform malicious computations with relatively sparse memory control: i.e., the *BPF interpreter*.

Figure 2 depicts the memory layout during an EPF v1 (BPF-Reuse) attack. First, the attacker *encodes* their payload in *valid* BPF programs, and sprays them in kernel space. Then, using a memory corruption vulnerability, they overwrite a code pointer and redirect the control flow to the BPF interpreter. By carefully choosing the respective code-pointer value, the attacker can control the *context* in which the (overwritten) code pointer will be invoked, thereby allowing them to specify an *arbitrary eBPF instruction* to start the interpretation from (i.e., by selecting/controlling the second argument of the invoked function), when the control flow is redirected to the BPF interpreter. Finally, since the attacker has sprayed BPF programs in kernel memory, they can easily find their payload in the direct mapping (i.e., `physmap`) region, with high probability (see Section 6.2).

Eventually, the problem becomes: *Can the attacker embed malicious eBPF code inside a benign BPF program, which escalates their privilege upon execution?*

```
R0 = R0 ^ R0   /* R0 =    {0} */
R1 = R1 / R1   /* R1 =    {1} */
R0 += R1       /* R0 =    {1} */
R0 += R0       /* R0 =   {10} */
R0 += R0       /* R0 =  {100} */
R0 += R1       /* R0 =  {101} */
R0 += R0       /* R0 = {1010} */
R0 += R1       /* R0 = {1011} */
```

Listing 1: Register-only BPF instructions that load the value 11 (`1011` in binary) onto register `R0`. (Values in comments are shown in binary.)

The answer is *positive*. The attacker can offset the eBPF instruction pointer by, say, 4 bytes during the attack, so that the `imm` fields will be in the original locations of `code`, `dst`, `src`, and `off`. A snippet of our eBPF payload is shown in Figure 3. When executed normally, it is an array of instructions that load immediates. However, when executed from a +4b offset, the semantics of that instruction stream are completely different. In what follows, we explain how one can perform different computations under the aforementioned model.

Arithmetics and Register Manipulation All instructions that do not use immediates, such as those moving values, or doing arithmetic, between registers, can be embedded into the `imm` field, as unused immediates are ignored by the interpreter. All *unaligned* instructions shown in Figure 3 do not use any immediates. If the attacker wishes to use constant values, they first need to load the constant into a register, and then perform the respective operation(s) with registers. Loading *arbitrary constants* into a register, without using the `imm` field, can be done solely with arithmetic operations.

Despite not being able to encode arbitrary numbers into `imm`, the attacker can load a non-zero value into a register. Then, say, `div` a register by itself, which leaves the value `0x1` in the register, or `xor` a register with itself, resulting in a `0x0` value. Similar to Listing 1, starting from `R0 = 0x0` and `R1 = 0x1`, any constant can be obtained by repeatedly adding `R0` to itself, and (optionally) adding `R1` to `R0`.

Control Flow Register-based comparisons and jumps can be encoded as usual because they do not use immediates—so conditionals can be done easily. *Looping* is also possible in malicious eBPF code. `off` field is a *signed* 16-bit value: it is signed because the `ld` and `st` instructions can use negative offsets for memory accesses. The eBPF verifier statically checks for non-negative offsets in jumps, but at runtime the interpreter does not check for this property (§2.2).

Memory Access `ld` and `st` instructions can directly access the whole kernel memory. Normally this does not create a security issue because of the (register) range analysis performed by the static checker, which ensures that no such instruction will be performed on a register that (potentially) points outside of the desired bounds. However, similar to branching offsets, this property not enforced at runtime by the interpreter.

By combining the three aforementioned techniques, the attacker can embed malicious eBPF code inside a benign BPF program. With a piece of unverified eBPF code operating in kernel memory space, there are different ways to escalate privilege. In our exploits, we chose to locate `init_task`, a global symbol that is placed in the same linked list with all `task_structs`. Then, we iterate over all processes until we find the attacking process. Lastly, we overwrite the credentials of the attacking process with those of `init_task`, giving the attacking process the highest privilege.

Combining eBPF and cBPF When spraying BPF programs, the RAM can be filled close to full, but not with only the translated eBPF code. The original cBPF code, the user process, the sockets created to attach the BPF programs to, and the JITed eBPF (if enabled), also reside in physical memory and compete for space. It is possible to encode payloads in both eBPF code and in the original cBPF code, and greatly increase the effectiveness of spraying, which translates to a higher probability of successful exploitation.

The problem is that although the allocations are always page-aligned, eBPF code and cBPF code do not start from the same offset *within* their respective pages. For example, in Linux v5.10, the eBPF code starts at a `0x38`b offset, within its page, while cBPF code starts at the beginning of the page. This can be mitigated by using a technique similar to a `NOP`-sled [30]: the attacker encodes malicious eBPF instructions inside benign instructions by offsetting the starting point by 4 bytes. Hence, in the example above, the attacker can start the malicious eBPF code with 7 (=`0x38/8`) instructions that are `xor R9, R9`; these instructions have no effect on the malicious functionality. However, if the attacker now aims at byte `0x3c` within a random page, they will be hitting a memory location that contains either eBPF or cBPF code.

4.3 EPF v2 (BPF-ROP)

Although the memory layout of eBPF code forbids the attacker from embedding 64-bit pointers, it is still possible to do so on 32-bit platforms. This facilitates ROP [100] (or, in general, code-reuse [35, 40, 57]) attacks. We will demonstrate this on x86 by introducing EPF v2 (BPF-ROP). To initiate, say, a ROP attack, the attacker usually needs to overwrite a code pointer with a stack-pivoting gadget [93], moving the stack pointer to the payload, which is an array of code pointers pointing to other gadgets. An immediate strategy would be to use the eBPF code as the payload. Doing so would encounter two challenges: (a) not every 4 bytes can encode return addresses, and (b) such a stack would be read-only.

Since the attacker has control only over every 4 other bytes, some gadgets would make `%esp` point to the gaps in between. If the execution hits a `ret` instruction in this case, it will crash the kernel and terminate the attack. More specifically, if a gadget does not move the stack pointer by +4X bytes, where X is an *odd* number, then the attack will fail.

Access Type	Regular Data	BPF Programs	
		Aligned	Unaligned
Normal Access	Allowed	BPF-ISR	BPF-ISR
BPF Execution	BPF-NX	Allowed	BPF-CFI

Table 1: How regular memory accesses and BPF-code fetches are hardened by our defenses.

To overcome this, EPF v2 filters gadgets based on how they move the stack pointer. Namely, the attacker can use gadgets that have the form: `...; pop %reg; ret`, where the `pop` instruction offsets the return address to a correct location.

EPF v2 divides a ROP (or code-reuse) attack into two stages. The first stage uses stack lifting gadgets only, and bootstraps a new ROP payload for the second stage, at a writable memory area (preferably in the original stack). This strategy requires simpler semantics for the more restricted first stage ROP, which makes gadget-finding much easier. For example, the following is a set of gadgets that can be used to bootstrap a new ROP payload: (1) `pop %edx; pop %ecx; pop %ebx; ret`, (2) `dec %eax; pop %ebp; ret`, and (3) `mov %ecx, (%eax); pop %ebx; ret`. Assuming `%eax` points to a writable region, gadget (1) can load a constant in the BPF filter into register `%ecx`, and gadget (2) can move that value to the future stack; then gadget (3) moves the future stack pointer further down for the next value. In the second stage, the attacker can use arbitrary gadgets. There are a lot of options once the attacker reaches this point. A common practice is to invoke `commit_creds(prepare_kernel_cred(0))` [88]. The ROP representation of the above, in x86 Linux, is just three addresses on the stack: a gadget to clear `%eax`, the address of `prepare_kernel_cred`, and the address of `commit_creds`. (This would not be able to execute during the first stage, as it would require a writable stack.)

5 Hardening BPF against EPF-style Attacks

5.1 Goals and Objectives

Our attacks (EPF v1 and v2) have demonstrated design weaknesses in the BPF infrastructure on Linux. Specifically, they reveal the weak separation between BPF programs and regular kernel memory: arbitrary kernel objects can be used in lieu of eBPF instructions in BPF-Reuse (EPF v1); and BPF code is used as native code pointers in BPF-ROP (EPF v2). More importantly, the problem is exacerbated by the weak runtime checks performed by the eBPF interpreter.

Taking inspiration from hardening native code, we propose to enforce the following properties (shown in Table 1):

① **Prevent regular kernel data from being used as eBPF instructions.** From the perspective of the BPF execution engine, this is analogous to data being non-executable [6]. Without this guarantee, we cannot realistically enforce any property on BPF programs, since they can be counterfeited using regular data. This property stops BPF-Reuse that utilizes the original cBPF code.

② **Ensure that BPF execution starts from benign addresses.** This is similar to control-flow integrity (CFI) on native code [29]. Since all BPF jumps have hard-coded offsets, no indirect branching is possible. The only way to divert the intended BPF control flow is to start from an unintended/unaligned address. This property stops BPF-Reuse that utilizes eBPF code.

③ **Prevent eBPF instructions from being used as regular (control) data.** By isolating BPF programs from regular kernel data, regardless of the amount of BPF programs created, the kernel cannot be misled to access malicious payloads that are embedded inside BPF programs. This property stops BPF-ROP attacks.

Besides the above security goals, we also want the respective code changes to incur negligible runtime overhead.

```
u64 bpf_interpreter(struct bpf_prog *prog)
{
    ...
    enter_bpf_mode();
    check_bpf_cfi(prog);
    initialize_context();
    mask = prog->mask;
    ...
    insn = prog->insns;
select_insn:
    tmp_insn = *insn;
    check_bpf_nx(insn);
    check_bpf_mode();
    tmp_insn = unmask(tmp_insn, mask);
    execute_bpf_insn(tmp_insn);
    if (finished) {
        goto done;
    }
    else {
        insn++;
        goto select_insn;
    }
done:
    leave_bpf_mode();
    return result;
}
```

Listing 2: Pseudocode of the BPF interpreter, instrumented with our defenses against EPF.

5.2 Design

Shown in Listing 2 is a pseudocode implementation of the BPF interpreter; colored lines correspond to our defenses.

BPF-NX To achieve objective ①, we reserve a region in the kernel's address space that is used *exclusively* for allocating BPF programs (not cBPF code, as it is not interpreted).

The interpreter can tell the difference between eBPF code and normal data by just checking an address range (ln. 12[1]). The check is performed for every eBPF instruction load, which is analogous to how a CPU enforces that the instructions are fetched from an executable page. Such frequent checking is required because in a code-reuse attack, the attacker might branch to the middle of the interpreter and bypass any one-time check outside of the main execution loop. In such scenarios, we still need the interpreter to reject instructions from invalid memory ranges.

BPF-CFI Objective ② is essentially defending against code-reuse attacks in BPF programs. BPF-CFI is challenging since we cannot simply check the alignment of the eBPF instructions. Although our attack uses unaligned eBPF instructions for simplicity, it can still be dangerous to even start executing aligned eBPF instructions from the middle of the eBPF code, because the static verifier's security guarantees only hold for executions starting from the beginning of the eBPF code. So the interpreter needs to make sure that the instruction array begins from the correct position in the eBPF code. We add such a check at the beginning of the interpreter (ln. 5).

However, this is not enough. The integrity of the control-flow [29] of the whole interpreter (function) is also necessary. Otherwise, the security check can just be skipped by leveraging a code-reuse attack that starts the execution from the "middle" of the interpreter. Kernel-level CFI [47, 85] incurs some non-negligible overhead, because it protects all code, and cannot be scaled down to protect selected parts of the kernel. Instead, we introduce a *sentinel* variable that is not corruptible by normal execution. The sentinel is used to ensure the control-flow integrity of the interpreter. The sentinel is set at the start (ln. 4), indicating that the interpreter is properly executed; at the end (of the interpretation) the sentinel is cleared (ln. 24). The interpreter checks the sentinel on each eBPF instruction fetch (ln. 13) to ensure control-flow integrity. If the control flow enters the interpreter without going through the correct entry point (i.e., the beginning of the function), it will be caught before any eBPF instruction is executed.

BPF-ISR To achieve objective ③, an obvious solution is to make use of the separation we have done in BPF-NX: whenever the rest of the kernel wants to access normal data, check whether the data lives in a BPF region.

However, it would be inefficient to instrument every memory access the kernel makes. Instead, we adopt the idea of ISR (Instruction Set Randomization), a defense originally designed to counter code injection [33, 65, 101]. ISR is suitable in this case as it is very easy to implement in a software-based interpreter. Every time the attacker tries to allocate a BPF program, the in-memory representation is chosen randomly from one of 2^{32} possibilities. Under normal BPF execution, the mask is extracted at the beginning (ln. 7), then used to unmask every instruction during interpretation (ln. 14).

The attacker can no longer benefit from tricking kernel code into accessing BPF programs as normal memory because the content is randomized and unpredictable.

5.3 Implementation

We implemented our defenses in x86-64 Linux.

BPF-NX In 64-bit Linux, there exist gaps in the kernel's address space that are not used. We reserve a 512GB *unused* region exclusively for BPF programs (`struct bpf_prog`). Originally, the BPF programs are allocated using `vmalloc`, which is a wrapper function around `__vmalloc_node_range`, whose parameters indicate the target range in which the memory is allocated from. `vmalloc` uses a *fixed* range, specified by `VMALLOC_START` and `VMALLOC_END`. To allocate in our reserved eBPF region, we add our own wrapper `bpf_vmalloc` that calls `__vmalloc_node_range` with the proper range. A small change to the page fault handler is also needed, because of the lazy propagation of changes in kernel page tables. The BPF program region needs to be handled similarly to `vmalloc`, where its page table entries are populated on-fault.

BPF-CFI We implement the sentinel variable using the `AC` flag in `RFLAGS`. This flag is the switch for SMAP: turning off the flag allows the CPU in supervisor mode to access user data. We assume that the interpreter itself is benign and does not need the protection against unintended user-space memory accesses. The sentinel variable is set by "turning off" SMAP, and cleared by turning it on. This way, the sentinel variable can easily be checked by an access to a user memory page. During kernel initialization, a dedicated memory page is marked as a user-mode page, and reserved for the access check. At the beginning of the interpreter, we execute the instruction `clac` to disable SMAP. Whenever the interpreter fetches an eBPF instruction, it also reads from the user-mode page, verifying that SMAP is indeed disabled. And at the end, the interpreter executes `stac` to re-enable SMAP.

For the check re: the starting point of the eBPF instruction array, we added 8 `0xff` bytes in `struct bpf_prog` as a *magic* number, which eBPF instructions cannot forge. By checking that this exists at the correct position inside the BPF program header, we assert that it is the right starting point.

BPF-ISR To implement ISR in the eBPF interpreter, we add a new field `mask` in `struct bpf_prog` to store the mask value for each BPF program. After a BPF program is initialized, all the pages are changed to read-only. Right before the permission change, we choose a random 4-byte value as mask, and `xor` the `imm` field in every eBPF instruction with our mask. This ensures that the memory content can not be arbitrarily chosen by the attacker. Since original cBPF code is also stored inside kernel memory, naturally it needs to be masked too. We do not mask the rest of the eBPF instructions because the attacker does not have much control over them. Lastly, the interpreter unmasks each instruction in the stack during the execution of the eBPF instructions.

[1] All line numbers in this section refer to Listing 2.

CVE	Vulnerability Type	Context	Method
CVE-2021-43267	Heap overflow	Process	EPF v1
CVE-2017-7308	Heap overflow	Process	EPF v1
CVE-2016-8655	Use-after-free	Interrupt	EPF v1
CVE-2017-7308	Heap overflow	Process	EPF v2
CVE-2017-6074	Use-after-free	Process	EPF v2
CVE-2016-8655	Use-after-free	Interrupt	EPF v2
CVE-2013-2094	Arbitrary write	Process	EPF v2

Table 2: List of vulnerabilities exploited with EPF.

6 Evaluation

To evaluate EPF (§4), and the set of defenses we developed against it (§5), we used a host armed with a 16-core 3.7GHz Intel Xeon W-2145 CPU and 64GB RAM, running Ubuntu 18.04 LTS (64-bit).

In our evaluation we focused on the following questions:

- **RQ1:** Are EPF attacks realistic?
- **RQ2:** How effective is EPF-based payload injection? How does it compare to other methods?
- **RQ3:** How much overhead does our set of defenses against EPF introduce?

6.1 Effectiveness of EPF (RQ1)

To demonstrate the feasibility of EPF-style attacks, we applied BPF-Reuse and BPF-ROP on existing Linux vulnerabilities with publicly-available exploits (see Table 2).

For BPF-Reuse (EPF v1) we chose 3 vulnerabilities. CVE-2021-43267 and CVE-2017-7308 are heap overflow bugs; our exploit dereferences an overwritten code-pointer in process context. CVE-2016-8655 is a use-after-free bug; a controlled function-pointer is dereferenced in interrupt context. Our payload loops through the linked list of *all* `task_structs` and changes the credentials on the one with the proper `pid`, using the interpreter (§4.2)—as a result, our strategy works in both contexts. This shows the expressiveness and versatility of EPF-based exploitation.

For BPF-ROP (EPF v2) we chose 4 vulnerabilities that also cover interrupt and process contexts with different types of vulnerabilities; 3 vulnerabilities happen in process context, and the respective payloads are setup to invoke `commit_creds(prepare_kernel_cred(0))` (§4.3), whereas for the interrupt context scenario, the ROP payload marks the memory page(s) that host the BPF program itself as executable, and transfers control to (x86) shellcode that in encoded as valid BPF instructions. With native shellcode embedding, we also employ the same strategy of looping through active processes and performing privilege escalation.

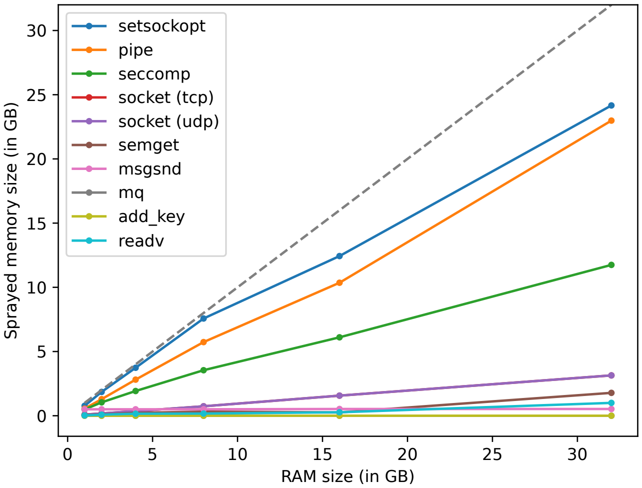

Figure 4: Effectiveness of spraying using different syscalls.

6.2 Spraying Effectiveness (RQ2)

We measured the effectiveness of EPF-based spraying and compared it to other methods. We found reasonable candidates for our comparison. Namely, we chose syscalls as spraying aids using the following three criteria:

• *Syscalls with variable-sized arguments*: In this case, we aim to find syscalls that copy variable-sized data into the kernel. This is done by filtering out syscalls that have a `const` pointer as argument, accompanied by a `size_t` or integer argument specifying the array size. This set includes `add_key` and `readv` (representing all vectorized I/O syscalls).

• *Syscalls returning writable file descriptors*: In this case, we aim to find syscalls that create file descriptors, which can later be used together with syscalls such as `write` to inject data into the kernel. As a result this set includes `pipe`, `bpf`, `socket`, and `landlock_create_ruleset`. The latter is not available in latest Debian (v11), and `bpf` is disabled for unprivileged users by default, so we exclude them.

• *Multiplexed syscalls*: In this case, we manually investigated syscalls with multiplexed arguments (e.g., `ioctl`, `fcntl`) and tried to find semantically similar syscalls for spraying. This set includes `setsockopt` and `seccomp`.

All our experiments took place on a Debian v11 (bullseye) VM, running on the benchmarking host, with its RAM size set at 1, 2, 4, 8, 16, and 32GB, respectively. Specifically, we first start a *monitor* process, which then forks a *sprayer* process and communicates with it via a set of Unix pipes. When the latter stops consuming additional memory, the monitor will spawn a new sprayer. For each method, the probability that the attacker can locate one of the sprayed objects is denoted as *success rate*. Lastly, the respective spraying method can bypass strong kernel-user isolation mechanisms (like XPFO [67]), unless we mention otherwise. As shown in Figure 4, creating cBPF (using `setsockopt` or `seccomp`) is among the most efficient methods.

setsockopt and seccomp We are using setsockopt with SO_ATTACH_FILTER to attach cBPF programs to sockets, and SECCOMP_MODE_FILTER to attach cBPF programs to processes. These two methods do not have quotas/limits set from the underlying kernel. In fact, if we keep allocating cBPF programs, the system will invoke the OOM-killer and try to reclaim memory at some point. If both the translated eBPF programs and the original cBPF copies are utilized (as described in Section 4.2), setsockopt can take up to 70% of all physical memory, while seccomp can take up to 34%; otherwise, the ratios reduce to 35% and 17%, respectively. In conclusion, spraying cBPF programs has a 70% or 35% success rate depending on the mode used, and the data structure allows controlling every 4 other bytes.

msgsnd The msgget syscall creates message queues for SysV-based IPC, and msgsnd adds messages to such queues. The maximum memory occupied by the messages is 500MB, regardless of the RAM size. The probability of locating sprayed content is 3%–24% (depending on the total RAM size) with continuous control over the sprayed region.

semget The semget syscall creates (SysV) semaphores. By generating 32K sets, each with 32K semaphores, the total limit will be reached. Each semaphore takes up 64 bytes of kernel memory, so the semaphores can consume ≈60GB of memory. But, a significant drawback of this approach is that of the 64-byte data structure only the semaphore variable can be controlled by the attacker, which is only 4 bytes. Therefore semget is not realistically useful.

pipe We create 20K Unix pipes for each sprayer process, and write exactly one page of content into each. This method can fill up to 60% of the total physical memory. Since the memory used to store pipes is page-aligned, this translates to 60% success rate, with complete attacker control. However, the sprayed content can be easily isolated using strong kernel-user separation mechanisms (e.g., XPFO).

mq By default, an unprivileged user can have 800KB of data stored in-kernel using POSIX message queues. The attacker will have less than 1% chance of locating the sprayed objects. Additionally, the sprayed content can also be isolated by strong kernel-user separation mechanisms, like XPFO.

socket We use the following strategy to spray with TCP sockets: for each process, create one TCP socket, send messages until it blocks, and then spawn as many processes as possible until socket creation fails. UDP sockets are different; they do not block with failed sending. Hence, we send until the occupied memory in /proc/net/sockstat does not change. In both cases, we are able to spray about 10% of the physical memory. This method has 10% success rate and allows for complete control over the sprayed region.

readv and add_key readv gives the attacker less than 2% success rate, controlling 6 out of every 8 bytes, while the total amount of memory that can be allocated by add_key is 20K bytes. So the success rate of the latter is less than 1%.

6.3 Hardening Overhead (RQ3)

We mainly evaluate our defenses on syscall filtering [80], socket filtering, and XDP [78] skb mode.

Syscall Filtering sysfilter [52] is an automated syscall filtering tool. It analyzes an application, creates the set of syscalls that the application needs, and then enforces it using seccomp-bpf. We evaluate the overhead our defenses incur on Nginx and Redis, when hardened by sysfilter, with no BPF-JIT, and the SSB mitigation disabled (by setting SECCOMP_FILTER_FLAG_SPEC_ALLOW [80]). The network requests in each test are sent over the loopback (lo) device. Both packages are installed from Ubuntu's repository.

To benchmark Nginx, we used wrk [56] with 2 running threads, each performing 128 connections, for 1 minute continuously. Nginx is configured to have 2 working processes. To maximize the time spent on BPF execution, we picked the smallest size of requested file from sysfilter, which is 1KB. Additionally, we manually inspected the CPU utilization, ensuring it was close to 100%. To benchmark Redis, we used memtier [97], spawning 2 worker threads, each with 128 clients, running for 1 minute continously. The ratio between GET and SET operations was set to 10:1, and each data object was 32 bytes. Again, we also made sure that the CPU utilization was close to 100%. Our results are shown in Figure 5. Our defenses introduce an additional 1.8% throughput decrease on Nginx, and 1.5% on Redis.

Socket Filtering We focus on the usage of socket filtering, similar to a traffic monitoring scenario. A simple traffic generator will send UDP packets in a tight loop, with a body size of 64 bytes to the DUT (device under test); and on the DUT there is a server that receives them. Both machines have 1GbE NICs, and the sending speed is tuned to be slightly higher than the processing capability of the DUT. We use small packets because large packet sizes will mask the BPF processing time.

To simulate a traffic monitor, we attach a raw socket to the ethernet interface and call setsockopt to attach a cBPF filter on the raw socket. We use a set of 6 different filter rules, similar to previous network monitoring studies [111], with some changes: (1) the rules are slightly modified to make sure our packets go through the maximum possible execution paths; and (2) we also modify the return instruction to always reject packets and not spend any time reading them, resulting in all overhead showing up on the receiving end.

The cBPF filters are described in the following (PCAP):

1. `""` (empty expression that allows everything)
2. `"ip"`
3. `"ip src net 10.116.70.0/24 and dst net 10.0.0.0/8"`
4. `"ip src or dst net 192.168.2.0/24"`
5. `"ip and udp port (10 or 11 or 12 or 13 or 14)"`
6. `"ip and (not udp port (80 or 25 or 143)) and not ip host ..."` (32 IPv4 addresses)

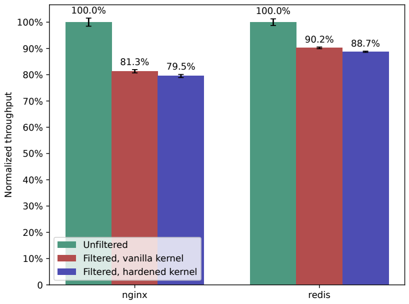

Figure 5: Normalized throughput between vanilla Linux and EPF-hardened Linux when running `seccomp`-protected Nginx and Redis.

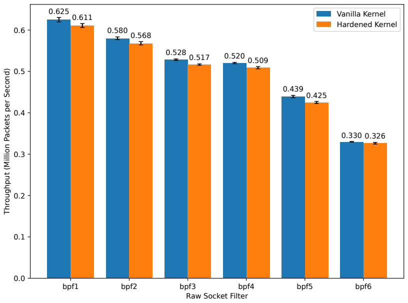

Figure 6: Traffic monitoring performance using different socket filters.

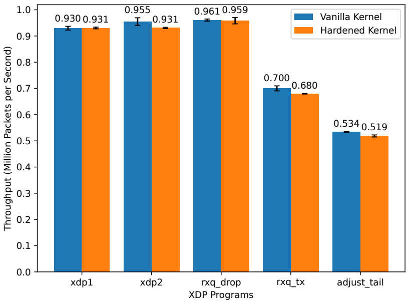

Figure 7: Throughput of XDP programs when run on vanilla kernel vs. an EPF-hardened kernel.

The results are shown in Figure 6; there is a 0.5%–3% reduction in the respective throughput.

XDP Since all cBPF programs are executed as eBPF programs, our defense affects interpreted eBPF programs too. XDP uses eBPF programs to passthrough, drop, re-transmit, or re-route incoming packets. In our experiment, we run XDP in `skb` mode, which executes eBPF programs when packet handling enters the device-agnostic part of the kernel, as the other modes require specific hardware. The experiment setup is the same as for socket filtering. To demonstrate the performance impact on eBPF programs, we run XDP programs (shown below) from the Linux source tree, similar to previous works [39, 59]. The XDP programs will intercept and run on every incoming packet generated by the UDP client.

- `xdp1`: Parse the IP header, count incoming packets and update a counter in a BPF map, then drop the packets.
- `xdp2`: Same as `xdp1`, but re-transmit the packets.
- `xdp_adjust_tail`: Change incoming packets into ICMP packets and send them back, keeping a total count in a BPF map.
- `rxq_info(drop)`: Count incoming packets for each receive queue and drop them.
- `rxq_info(tx)`: Count incoming packets for each receive queue and re-transmit them.

We tuned the traffic generator to send at a higher rate than the maximum throughput. Additionally, we pinned the NIC interrupts to one CPU core and manually verified that CPU utilization was close to 100%. The results are shown in Figure 7: we observed 0%–3% throughput degradation.

7 Discussion

BPF-CFI Considerations To implement BPF-CFI we utilize the `AC` flag, and, as we mentioned in Section 5.3, this flag also controls SMAP. We chose this approach because SMAP was designed to accommodate low-overhead switching.

However, this also means that during the execution of the BPF interpreter SMAP cannot prevent the interpreter from incorrectly accessing userspace data. At first glance this might be a security loss, but in fact the impact is very minimal. Firstly, this is a confined attack surface that is relatively easy to maintain, instead of a complicated invariance to be respected across the whole kernel codebase. Secondly, SMAP is designed to stop the kernel from incorrectly accessing user-controlled data during an attack, but the semantics of BPF programs already grant the interpreter access to user-controlled content such as BPF maps [5], context metadata, and more. In conclusion, SMAP does not provide notable security gains in the BPF interpreter context, while our defense provides better overall security by re-purposing this extension.

KASLR Bypass Although KASLR [54] is sometimes bypassed by arbitrary memory disclosure vulnerabilities, much weaker primitives exist that circumvent KASLR, including bounded memory disclosure vulnerabilities [18, 20, 44] and side-channel attacks [41, 58, 62, 76]. By spraying, our attacks can locate attacker-controlled objects *within the heap*. If the attacker deploys one of the methods to de-randomize KASLR, then they can find *where the heap is located at*. Combining these two capabilities, our attacks can work exactly the same as they would without KASLR.

8 Related Work

BPF JIT Spraying BPF JIT spraying [82, 98] is an exploitation technique that takes advantage of the BPF's JIT engine generating predictable code. By carefully crafting and spraying the JITed code, the attacker can control a piece of code in kernel context, which effectively nullifies defenses against ret2usr such as SMEP and PXN, and re-enables attacks that redirect control flow to user-controlled code—JITed BPF code. BPF-Reuse and BPF-ROP utilize the BPF program data structure itself and aim to control memory content.

Thereby turning BPF programs into a mechanism for injecting payloads for code-reuse attacks. Importantly, the targeted features and the goals of BPF JIT- vs. EPF-based spraying are completely different. Moreover, BPF JIT spraying is already defended against in recent Linux kernel [79], whereas BPF-Reuse and BPF-ROP bypass all existing isolation mechanisms, including XPFO [67], which is not yet deployed in mainline Linux.

eBPF-based Speculative Type Confusion Kirzner et al. [71] pointed out that the eBPF verifier performs extensive analyses, and safety checks, to ensure the execution of eBPF programs is sandboxed, but they did not take into account speculative execution paths. As a result, some eBPF programs deemed safe by the verifier can be vulnerable to transient execution attacks and leak confidential kernel data. The root cause is addressed by adding analyses that account for possible speculation [37]. Our attacks are possible due to a different underlying reason: BPF programs are a type of user-controlled memory object that cannot be easily isolated from the rest of kernel data, and they can be created in bulk.

Other Popular Kernel Exploitation Techniques In the post-ret2usr era, where defenses such as SMEP, SMAP, PXN, and PAN are present, kernel exploitation has evolved to allow for creating addressable payloads at exploitation time. Apart from ret2dir [67], which is discussed in Section 2.1, there are also two other popular strategies to bypass ret2usr defenses. The first strategy is careful heap manipulation that combines heap fengshui [102, 108] with elastic objects (systemized and termed by Chen et al. [43]), which results in disclosing the address of the user-controlled memory object that will become the attack payload. The strategy is used by some real-world exploits [87, 92], and takes advantage of the predictability of the heap layout, whereas EPF abuses the design of BPF functionality. The second strategy is calling functions inside the kernel, which can disable protection mechanisms. It is used by real-world exploits [75], as well as automated exploit generation frameworks [110]. The drawback is that it relies on how defense features are implemented.

After the payload is placed in kernel space, there are several ways to actually realize privilege escalation. One popular method is to overwrite the `modprobe_path` global variable [69], substituting an attacker-controlled binary to be executed with `root` privilege (instead of `/sbin/modprobe`). This is used by exploit authors and the CTF community [109, 114]. Another method is `ret2bpf` (termed by Jin et al. [63]), which is popular in ARM kernel exploits [31] because it does code-reuse using the BPF interpreter, significantly simplifying the search for code gadgets. It tricks the kernel to use attacker-controlled memory as BPF instructions, essentially doing "BPF-code injection", which notably can be defended against by BPF-NX. `ret2bpf` has similarities to BPF-Reuse, but, most importantly, it differs in the method of supplying the attack payload: `ret2bpf` requires the attacker to have the ability to create a payload in kernel space, whereas BPF-Reuse solves exactly this problem (i.e., payload injection).

9 Conclusion

In this paper, we have shown that BPF, a kernel subsystem that allows unprivileged users to push data structures, freely, onto the kernel address space, is inherently susceptible to attack-payload injection. We developed two attacks, BPF-Reuse (EPF v1) and BPF-ROP (EPF v2), and demonstrated how to inject certain payloads. Further, we showed the practicality, and effectiveness, of our attacks by combining them with real-world vulnerabilities to exploit the Linux kernel. We also developed comprehensive defenses that enforce the stronger isolation between BPF code and normal kernel data, and the integrity of BPF program execution, thwarting the abuse of the BPF infrastructure. Our defenses were evaluated on tasks that result in heavy BPF usage, and were shown to have negligible overhead.

Availability

Our prototype implementation of BPF-{NX, CFI, ISR} and the exploits we ported to EPF are available at: `https://gitlab.com/brown-ssl/epf`

Acknowledgments

We thank our shepherd, Michael Le, and the anonymous reviewers for their valuable feedback. We also thank Alexander Gaidis for providing comments on earlier drafts of our paper. This work was supported in part by the CIFellows 2020 program, through award CIF2020-BU-04, and the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or CRA.

References

[1] A safer playground for your Linux and Chrome OS renderers. https://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html.

[2] ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a/BABCEADG.

[3] bpf-helpers(7) – Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[4] bpf(2) – Linux manual page. https://man7.org/linux/man-pages/man2/bpf.2.html.

[5] eBPF maps. https://www.kernel.org/doc/html/latest/bpf/maps.html.

[6] Kernel Self-Protection. https://www.kernel.org/doc/html/latest/security/self-protection.html#executable-code-and-read-only-data-must-not-be-writable.

[7] Learn the architecture – AArch64 memory model. https://developer.arm.com/documentation/102376/0100/Permissions-attributes.

[8] libbpf. https://github.com/libbpf/libbpf.

[9] Mozilla wiki – Security/Sandbox/Seccomp. https://wiki.mozilla.org/Security/Sandbox/Seccomp#Use_in_Gecko.

[10] Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/.

[11] sysfs – The filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html.

[12] syzbot. https://syzkaller.appspot.com/openbsd.

[13] syzkaller – kernel fuzzer. https://github.com/google/syzkaller.

[14] The /proc Filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[15] XDP – eXpress Data Path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html.

[16] CVE-2017-16996. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16996, November 2017.

[17] CVE-2017-17853. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17853, December 2017.

[18] CVE-2017-17864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17864, December 2017.

[19] CVE-2019-7308. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7308, February 2019.

[20] CVE-2020-25662. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25662, September 2020.

[21] CVE-2021-45402. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45402, December 2020.

[22] CVE-2021-31829. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31829, April 2021.

[23] CVE-2021-32606. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32606, May 2021.

[24] CVE-2021-33034. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33034, May 2021.

[25] CVE-2021-3444. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3444, March 2021.

[26] CVE-2021-3490. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490, April 2021.

[27] CVE-2021-3612. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3612, June 2021.

[28] CVE-2021-42008. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42008, October 2021.

[29] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[30] Periklis Akritidis, Evangelos P Markatos, Michalis Polychronakis, and Kostas Anagnostakis. Stride: Polymorphic Sled Detection through Instruction Sequence Analysis. In *IFIP International Information Security Conference (SEC)*, pages 375–391, 2005.

[31] Brandon Azad. An iOS hacker tries Android. https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html.

[32] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security Symposium (SEC)*, pages 971–988, 2022.

[33] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, 2003.

[34] Ashish Bijlani and Umakishore Ramachandran. Extension Framework for File Systems in User Space. In *USENIX Annual Technical Conference (ATC)*, pages 121–134, 2019.

[35] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 30–40, 2011.

[36] Daniel Borkmann. bpf: add generic constant blinding for use in jits. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b.

[37] Daniel Borkmann. BPF and Spectre: Mitigating transient execution attacks. https://github.com/gojue/ebpf-slide/blob/master/eBPF_advanced/eBPF-Summit-2021-BPF-and-Spectre-Daniel-Borkmann-Final.pdf.

[38] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html.

[39] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 973–990, 2020.

[40] Bugtraq. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.

[41] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break it, Fix it, Repeat. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 481–493, 2020.

[42] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. In *ACM Cloud Computing Security Workshop (CCSW)*, pages 139–151, 2021.

[43] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1165–1184, 2020.

[44] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[45] Jonathan Corbet. A JIT for packet filters. https://lwn.net/Articles/437981/.

[46] Jonathan Corbet. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/.

[47] Jonathan Corbet. Control-flow integrity in 5.13. https://lwn.net/Articles/856514/.

[48] Jonathan Corbet. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/.

[49] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/.

[50] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] Debian Documentation Project. Linux disables unprivileged calls to bpf() by default. https://www.debian.org/releases/stable/amd64/release-notes/ch-information.en.html#linux-unprivileged-bpf.

[52] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[53] Eric Dumazet. x86: bpf_jit_comp: secure bpf jit against spraying attacks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=314beb9.

[54] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[55] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 443–458, 2020.

[56] Will Glozer. wrk – a HTTP benchmarking tool. https://github.com/wg/wrk.

[57] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, 2014.

[58] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)*, pages 368–379, 2016.

[59] 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 *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 54–66, 2018.

[60] Gianluca Insolvibile. The Linux Socket Filter: Sniffing Bytes over the Network. *Linux Journal*, 86:53, 2001.

[61] IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[62] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM Conference on Computer and Communications Security (CCS)*, pages 380–392, 2016.

[63] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf.

[64] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, 2003.

[65] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, 2003.

[66] Vasileios P Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

[67] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*, pages 957–972, 2014.

[68] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *USENIX Security Symposium (SEC)*, pages 459–474, 2012.

[69] Linux Kernel. Documentation for /proc/sys/kernel/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#modprobe.

[70] Linux Kernel. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[71] Ofek Kirzner and Adam Morrison. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security Symposium (SEC)*, pages 2399–2416, 2021.

[72] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[73] Karsten König. Exploit for CVE-2019-5596. https://www.exploit-db.com/exploits/47829.

[74] Andrey Konovalov. Exploit for CVE-2017-1000112. https://www.exploit-db.com/exploits/47169.

[75] Andrey Konovalov. Exploit for CVE-2017-7308. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[76] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–321, 2020.

[77] LEXFO. Exploit for CVE-2017-11176. https://www.exploit-db.com/exploits/45553.

[78] Linux Kernel. AF_XDP. https://www.kernel.org/doc/html/latest/networking/af_xdp.html.

[79] Linux Kernel. Documentation for /proc/sys/net/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/net.html#bpf-jit-harden.

[80] Linux Kernel. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[81] LLVM Project. LLVM 3.7 Release Notes. https://releases.llvm.org/3.7.0/docs/ReleaseNotes.html#non-comprehensive-list-of-changes-in-this-release.

[82] Keegan McAllister. Attacking hardened Linux systems with kernel JIT spraying. https://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[83] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*, 1993.

[84] David Miller. BPF Verifier Overview. https://www.spinics.net/lists/xdp-newbies/msg00185.html.

[85] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel. *Black Hat Asia*, 2017.

[86] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and Verification in the Field: Applying Formal Methods to BPF Just-In-Time Compilers in the Linux Kernel. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 41–61, 2020.

[87] Andy Nguyen. CVE-2021-22555: Turning \x00\x00 into 10000$. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html.

[88] Andy Nguyen. Exploit for CVE-2021-22555. https://www.exploit-db.com/exploits/50135.

[89] PaX Team. Better kernels with GCC plugins. https://lwn.net/Articles/461811/.

[90] PaX Team. UDEREF/amd64. http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html.

[91] PaX Team. UDEREF/i386. http://grsecurity.net/~spender/uderef.txt.

[92] peter@haxx.in. Exploit for CVE-2021-43267. https://haxx.in/posts/pwning-tipc/.

[93] Aravind Prakash and Heng Yin. Defeating ROP through Denial of Stack Pivot. In *Annual Computer Security Applications Conference (ACSAC)*, pages 111–120, 2015.

[94] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 563–577, 2020.

[95] Tim Rains, Matt Miller, and David Weston. Exploitation Trends: From Potential Risk to Actual Risk. In *RSA Conference*, 2015.

[96] rebel. Exploit for CVE-2016-8655. https://www.exploit-db.com/exploits/47170.

[97] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[98] Elena Reshetova, Filippo Bonazzi, and N Asokan. Randomization Can't Stop BPF JIT Spray. In *International Conference on Network and System Security (NSS)*, pages 233–247, 2017.

[99] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance Implications of Packet Filtering with Linux eBPF. In *International Teletraffic Congress (ITC)*, pages 209–217, 2018.

[100] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[101] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. Reviving Instruction Set Randomization. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28. IEEE, 2017.

[102] Alexander Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[103] Alexei Starovoitov. bpf: introduce BPF_JIT_ALWAYS_ON config. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=290af86629.

[104] Alexei Starovoitov. bpf: split eBPF out of NET. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f89b7755f517cdbb755d7543eef986ee9d54e654.

[105] Alexei Starovoitov. tracing, perf: Implement BPF programs attached to kprobes. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2541517c32be2531e0da59dfd7efc1ce844644f5.

[106] SUSE Support. Security Hardening: Use of eBPF by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545.

[107] Qualys Research Team. Sequoia: A deep root in Linux's filesystem layer (CVE-2021-33909). https://www.qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt.

[108] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards Automated Heap Feng Shui. In *USENIX Security Symposium (SEC)*, pages 1647–1664, 2021.

[109] willsroot. CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers. https://www.willsroot.io/2022/01/cve-2022-0185.html.

[110] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-after-free Vulnerabilities. In *USENIX Security Symposium (SEC)*, pages 781–797, 2018.

[111] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A Fast Dynamic Packet Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 279–292, 2008.

[112] Fenghua Yu. Enable SMEP CPU Feature. https://lore.kernel.org/lkml/1305581685-5144-1-git-send-email-fenghua.yu@intel.com/.

[113] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel. In *USENIX Security Symposium (SEC)*, pages 3201–3217, 2022.

[114] Xiaochen Zou and Zhiyun Qian. CVE-2022-27666: Exploit esp6 modules in linux kernel.