Randomization can't stop BPF JIT spray

Elena Reshetova
Intel OTC, Finland
elena.reshetova@intel.com

Filippo Bonazzi
Aalto University, Finland
filippo.bonazzi@aalto.fi

N.Asokan
Aalto University and
University of Helsinki, Finland
asokan@acm.org

1 Introduction

Nowadays, more and more Linux attackers focus their attention on the kernel rather than on userspace applications, especially in mobile and embedded devices. The primary reason for this change is the extensive work done over the years to limit the damage when a userspace application is exploited. For example, the latest releases of Android have well-designed SEAndroid policies that do not allow a compromised application to get any significant controls over the OS itself. On the contrary, finding a vulnerability in the kernel almost always leads to compromise of the whole device.

Many kernel (and userspace) vulnerabilities are the result of programming mistakes, such as uninitialized variables, missing boundary checks, use-after-free situations *etc*. While developing tools that help finding these mistakes is important, it is impossible to fully avoid them. Moreover, even when a vulnerability is discovered and fixed in the upstream kernel, it takes approximately 5 years for the fix to be propagated to all the end user devices [6].

The Kernel Self Protection Project (KSPP)[1] tries to eliminate whole classes of vulnerabilities that might lead to creation of successful exploits, by implementing various hardening mechanisms inside the kernel itself. An important part of the project is to create Proof Of Concept (POC) attacks that demonstrate the need for certain additional protection mechanisms, since this helps to get wider acceptance from kernel subsystem maintainers.

The Linux kernel Berkeley Packet Filter (BPF) Just-In-Time (JIT) compiler was an important focus of the project, since it is widely used in the kernel and has seen successful attacks in the past. In 2012, a JIT spray attack against Linux BPF/JIT was presented (Section 3). Consequently, some countermeasures were implemented in the Linux kernel version 3.10 (Section 3.2). In this whitepaper, we show how these countermeasures can be circumvented on a modern Linux 4.4 kernel (Section 4). We also describe the recent measures that have been added to the Linux kernel to eliminate these types of attacks altogether (Section 5).

2 Background

The need for fast network packet inspection and monitoring was obvious in early versions of UNIX with networking support. In order to gain speed and avoid unnecessary copying of packet contents between kernel and userspace, the notion of a kernel *packet filter* agent was introduced [13, 12]. Different UNIX-based OSes implemented their own versions of these agents. The solution later adopted by Linux was the BSD Packet Filter introduced in 1993 [11], which is referred to as Berkeley Packet Filter (BPF). This agent allows a userspace program to attach a filter program

[1] kernsec.org/wiki/index.php/Kernel_Self_Protection_Project

onto a socket and limit certain data flows coming through the socket in a fast and effective way.

Linux BPF originally provided a set of instructions that could be used to program a filter: this is nowadays referred to as *classic BPF* (cBPF). Later a new, more flexible, and richer set was introduced, which is referred to as *extended BPF* (eBPF) [5, 14]. In order to simplify the terminology throughout this paper, we refer to the latter set of instructions simply as *BPF instructions*. Linux BPF can be viewed as a minimalistic virtual machine construct [5] that has a few registers, a stack and an implicit program counter. Different operations are allowed inside a valid BPF program, such as fetching data from the packet, arithmetic operations using constants and input data, and comparison of results against constants or packet data. The Linux BPF subsystem has a special component, called `verifier`, that is used to check the correctness of a BPF program; all BPF programs must approved by this component before they can be executed. `verifier` is a static code analyzer that walks and analyzes all branches of a BPF program; it tries to detect unreachable instructions, out of bound jumps, loops *etc*. `verifier` also enforces the maximum length of a BPF program to 4096 BPF instructions [14].

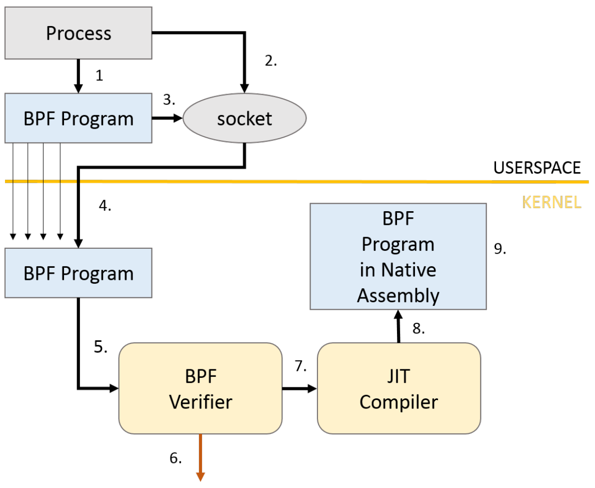

Figure 1: Typical flow of a BPF program

While originally designed for network packet filtering, nowadays Linux BPF is used in many other areas, including system call filtering in seccomp [2], tracing [15] and Kernel Connection Multiplexer (KCM) [8].

In order to improve packet filtering performance even further, Linux utilizes a Just-In-Time (JIT) compiler [7, 14] to translate BPF instructions into native machine assembly. JIT support is provided for all major architectures, including x86 and ARM. This JIT compiler is not enabled by default on standard Linux distributions, such as Ubuntu or Fedora, but it is typically enabled on network equipment such as routers.

Figure 1 shows a simplified view of how a BPF program is loaded and processed in the Linux kernel. First, a userspace process creates a BPF program, a socket, and attaches the program to the socket (steps 1-3). Next, the program is transferred to the kernel, where it is fed to the `verifier` to be checked (steps 4-5). If the checks fail (step 6), the program is discarded and the userspace process is notified of the error; otherwise, if JIT is enabled, the program gets processed by the JIT compiler (step 7). The result is the BPF program in native assembly, ready for execution when the associated socket receives data (steps 8-9). The program is placed in the kernel module mapping memory space, using the `vmalloc()` kernel memory allocation primitive.

3 JIT spray attack

JIT spraying is an attack where the behavior of a Just-In-Time compiler is (ab)used to load an attacker-provided payload into an executable memory area of the operating system [3]. This is usually achieved by passing the payload instructions encoded as constants to the JIT compiler and then using a suitable OS bug to redirect execution into the payload code. Normally the exact location of the payload is not known or controlled by the attacker, and therefore many copies of the payload are “sprayed” into OS memory to maximize the chance of success. JIT spray attacks are dangerous because JIT compilers, due to their nature, are normally exempt from various data execution prevention techniques, such as NX bit support (known as XD bit in x86 and XN bit in ARM). Another feature that makes JIT spray attacks especially successful on the x86 architecture is its support for *unaligned instruction execution*, which is the ability to jump into the middle of a multi-byte machine instruction and start execution from there. The x86 architecture supports this feature since its instructions can be anything between 1 and 15 bytes in length, and the processor should be able to execute them all correctly in any order [1]. The first attack that introduced the notion of JIT spraying and used this technique to exploit the Adobe Flash player on Windows was done by Dion Blazakis in 2010 [4].

3.1 Original JIT spray attack on Linux

The first original JIT spray attack against the Linux kernel using the BPF JIT compiler [10] was presented by Keegan McAllister in 2012. The POC exploit code[2] used a number of key steps to obtain a root shell on a Linux device.

3.1.1 Creating the BPF payload

The POC creates a valid BPF program containing the payload instructions: `commit_creds(prepare_kernel_cred(0))`. This is a very common way for exploits to obtain `root` privileges on Linux: the combination of these function calls sets the credentials of the current process to `root`. The addresses of the `commit_creds` and `prepare_kernel_cred` kernel symbols are resolved at runtime using the `/proc/kallsyms` kernel interface. The payload instructions are embedded into the filter program using the BPF load immediate instruction (`BPF_LD+BPF_IMM`), which loads a 4 byte constant into a standard register (`eax` in x86). When compiled, this instruction is transformed into the x86 `mov $x, %eax` instruction, which corresponds to the byte sequence

```
b8 XX XX XX XX
```

where `b8` is the instruction opcode and the following 4 bytes are the instruction argument `$x`. While the attacker is able to set these 4 bytes freely, in practice only the first 3 can be arbitrarily chosen; the last byte needs to be defined so that, when combined with the following `b8` instruction opcode during unaligned execution, it produces a harmless instruction. For this purpose, the last byte is chosen to be `a8`: the `a8 b8` byte sequence represents the harmless `test $0xb8, %al` x86 instruction. When the BPF load immediate instruction is repeated multiple times, this results in the byte sequence

```
b8 XX XX XX a8 b8 XX XX XX a8 b8 ...
```

Figure 2 shows how the payload is transformed from BPF pseudocode to the x86 machine code using the JIT compiler, and how the machine code looks like when starting the unaligned execution from the second byte of the payload.

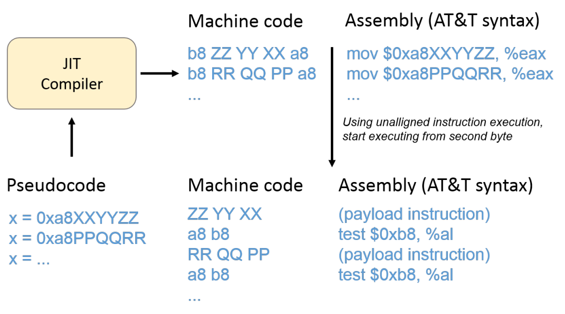

Figure 2: BPF payload JIT compilation and unaligned execution

3.1.2 Loading the payload in memory

In order to load many copies of the BPF filter payload in kernel memory, the attacker's process needs to create many local sockets, since each socket can have only one BPF filter attached to it. While Linux limits the number of open file descriptors that a process can posses at any given time, McAllister used a special trick to circumvent this limit. The trick is to open one local Unix socket and send the resulting file descriptor over another local Unix socket, and

[2] github.com/kmcallister/alameda

then close the original socket. Linux does not free the memory for the closed socket, since this might be still read by a process that receives it on the other end. Therefore, the socket is not counted towards the process socket limit, but it is kept in kernel memory regardless. By using this clever trick McAllister managed to create 8000 sockets, and correspondingly load 8000 BPF filters containing the payload in kernel memory.

3.1.3 Redirection of execution

Last, the proof of concept code contains a tiny kernel module (`jump.ko`) that jumps to the address specified by a userspace process using the interface provided by the `/proc` virtual filesystem. This extremely insecure module simulates the job of actually finding a real bug in the kernel to redirect the execution flow. `jump.ko` needs to be loaded using root privileges before the attack, which is obviously impossible for an attacker looking to obtain root privileges. This kernel module is simply used to provide an entry point to demonstrate that the JIT spray attack works.

3.1.4 The attack

After the attacker's program populates the kernel memory with 8000 filters containing the payload, it starts a loop where it attempts to jump to a random page within the kernel module mapping memory space and execute the payload at a predefined offset. The key to the attack's success is the fact that, in kernels older than 3.10, the BPF JIT compiler allocated each filter at the beginning of a memory page: since the length of the BPF filter is fixed, the attacker always knows the correct offset to jump to on a page in order to hit the payload. Each guessing attempt is done by a child process: this way, in the likely case of landing on the wrong page and executing some invalid instruction, only the child process is terminated by the Linux kernel, and the parent process can continue the attack.

It is important to note that when an attacker jumps to a page that doesn't contain a filter, the machine behavior is unpredictable. In most cases, if the landing instruction is invalid or harmless, the child process is simply killed and the attack can continue; however, if the instruction tampers with some key machine register, the whole OS can hang and the machine needs a hard reboot to recover.

3.2 Community response

After the attack was publicly released, the Linux kernel community scrambled to produce different countermeasures.

3.2.1 Upstream Linux kernel fix

The upstream Linux kernel merged a set of patches that randomized the loading address of a BPF program inside a page: instead of starting at the beginning of a page, the filter would be located at a random offset inside the page. In addition, the space between the page start and the filter - called *hole* - is filled with architecture specific instructions that aim to hang the machine if executed by an attacker. For x86, the hole is filled with repeated `INT3` (`0xcc`) instructions, which cause SIGTRAP interrupts in the Linux kernel [1]. This approach made the success probability of the attack much lower, because now the attacker needs to not only guess the correct page, but also the correct offset inside the 4KB page where the filter starts.

Furthermore, when the attacker jumps to a page that contains a copy of the filter, guessing the wrong offset is likely to be heavily punished: executing the `INT3` instruction will have more severe consequences than just executing an illegal instruction: in practice it most commonly results in kernel panic and full OS freeze. The cause of it is not properly understood at the moment and we continue investigation on the kernel behavior in this case.

3.2.2 Grsecurity fix

Another kernel security project, known as Grsecurity[3], released a different hardening mechanism to defend against the attack: using a technique called "*constant blinding*", they prevented the attacker from loading the payload instructions as constants, therefore blocking the code injection vector at the source. The idea behind constant blinding is to avoid storing the constants in memory as they are, and instead storing them XORed with some generated random number. When the constant needs to be accessed by a legitimate operation, it can be XORed again with the random number to obtain the correct value.

The provided implementation of this feature only supported the x86 architecture. It was never merged into the upstream kernel due to several reasons: the desire to have an architecture independent approach, the performance implications of the feature, various political reasons, but most of all the belief that the randomization measures implemented in the upstream kernel would be enough to stop BPF JIT spray attacks. No real attack against the hardened upstream kernel was publicly demonstrated to date.

4 Our attack

As part of the Kernel Self Protection Project together with one of the kernel BPF/JIT maintainers, Daniel Borkmann, we started to look into further securing BPF/JIT and considered the constant blinding approach proposed by Grsecurity. Our objective was to prove that the existing measures implemented in the upstream kernel are not enough to stop JIT spray attacks. If one could show a real attack against the upstream BPF/JIT, this would significantly raise the chances of additional protections being merged in the upstream kernel. This was the goal behind developing our version of the attack.

[3] `grsecurity.net`

The main part of the work has been done at the end of 2015/beginning of 2016, on Ubuntu 15.10 with the latest available stable kernel at that time (4.4.0-rc5) compiled with default Ubuntu configuration, running in a KVM-driven virtual machine. The whole setup was done for the x86_64 architecture.

We developed two different attack approaches, which we discuss below. One common issue that we had to deal with was the inability to obtain the location of kernel symbols (specifically `commit_creds` and `prepare_kernel_cred`, needed for the attack) using the `/proc/kallsyms` kernel interface. This is because the 4.4 kernel already implements kernel pointer protection, which hides the values of kernel pointers to userspace applications. This can be disabled by explicitly setting the `kptr_restrict` option to 0; however, this operation requires root privileges. One way to overcome this difficulty is to hardcode the addresses of these symbols for a specific kernel version, after obtaining them on a machine with `kptr_restrict` disabled. This is currently possible on Ubuntu and similar distros with a 4.4 kernel, because they do not utilize KASLR yet, and therefore kernel symbols are located at a deterministic address for all copies of a specific compiled kernel (*e.g.* `4.4.0-42-generic #62-Ubuntu SMP`). Then, at runtime, our attack can just resolve the correct symbol addresses by looking up the machine kernel version in a table.

4.1 Approach 1

While the size of a BPF filter program is limited to 4096 BPF instructions [14], this is more than enough to to obtain a compiled BPF filter larger than the 4KB of a kernel memory page. When the filter size grows beyond one page but under 2 pages, we can be sure that in 50% of the cases when we jump to the beginning of a page containing the filter program, we land on the program instructions. The probability could be even higher if we extended the filter to be longer than 2 pages, by increasing the number of BPF instructions to the maximum value of

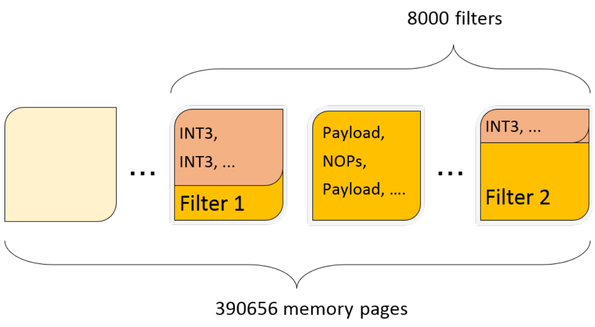

Figure 3: Attack approach 1

4096: however, the practice has shown that a filter larger than 2560 BPF instructions gets rejected by the `setsockopt()` function with ``‘‘Out of memory error’’``.

In Approach 1, we changed the original attack to generate much bigger BPF programs, containing 2471 BPF instructions (program length is 12439 bytes), which take between 3 and 4 4KB pages: we did this by repeating the payload multiple times with additional NOP instructions used as padding to align the beginning and end of the payload to 16 bytes (relative to the BPF header). Figure 3 illustrates this attack approach.

When we jump to a random page, we try to execute the first 10 offsets before moving to the next random page. The number of offsets (10) corresponds to the max number of NOP instructions added as padding between each payload. The number of sockets, and therefore loaded BPF filters, is the same (8000) as in the original attack. If we happen to jump to a page which does contain a copy of the filter, but not at the beginning of the page, we hit the hole padded with `INT3` instructions, which leads to a VM hang and causes our attack to fail.

4.2 Approach 2

Our next approach is based on how the allocation of a BPF filter program happens and how the random offset of a BPF filter is computed. This is done by the `bpf_jit_binary_alloc()` function, which is shown in Listing 1.

The function first calculates the total memory size to be allocated for a program (line 223), where `proglen` is the actual BPF program length in bytes, `sizeof(*hdr)` is 4 bytes and `PAGE_SIZE` is 4096 bytes. Next, all this space is pre-filled with illegal architecture-dependant instructions (`INT3` for x86) (line 229). The actual starting offset of the BPF filter is calculated last (line 234). What can be deduced from the above steps is that if we can make `proglen` to be `PAGE_SIZE - 128 - sizeof(*hdr)`, we will end up with only one page allocated for the BPF filter, with a max hole size of 128 located right at the beginning of a page. While the actual size of the hole is random, the maximum size (128) is static: jumping at offset 132 (128 `+ sizeof(*hdr)`) will guarantee landing on the payload. This way we can fully avoid the inserted `INT3` instructions and their negative impact. Figure 4 illustrates this attack approach.

In our experiments, we were able to bring the filter size to 3964 bytes and successfly jump over the first 132 bytes, called `hole_offset`. Trying both `hole_offset` and `hole_offset + 1` protects us from the unlucky case where our selected jump destination contains the `b8` byte deriving from the BPF load immediate instruction: jumping to `b8` would mean executing not the payload, but the actual `MOV %eax, XXXXXXXX` instruction. Jumping to two adjacent offsets guarantees that at least one of them will not contain `b8`.

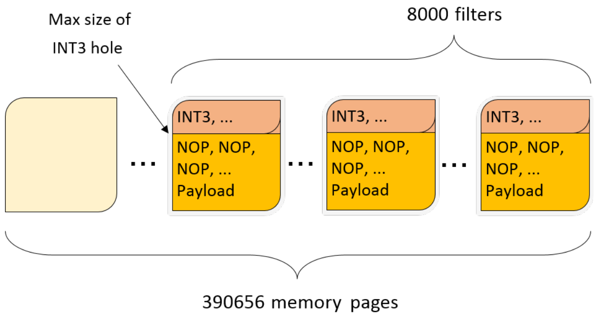

Figure 4: Attack approach 2

Listing 1: `bpf_jit_binary_alloc()` function from `kernel/bpf/core.c`

```c
struct bpf_binary_header *
bpf_jit_binary_alloc(unsigned int proglen, u8 **image_ptr,
                     unsigned int alignment,
                     bpf_jit_fill_hole_t bpf_fill_ill_insns)
{
        struct bpf_binary_header *hdr;
        unsigned int size, hole, start;

        /* Most of BPF filters are really small, but if some of them
         * fill a page, allow at least 128 extra bytes to insert a
         * random section of illegal instructions.
         */
        size = round_up(proglen + sizeof(*hdr) + 128, PAGE_SIZE);
        hdr = module_alloc(size);
        if (hdr == NULL)
                return NULL;

        /* Fill space with illegal/arch-dep instructions. */
        bpf_fill_ill_insns(hdr, size);

        hdr->pages = size / PAGE_SIZE;
        hole = min_t(unsigned int, size - (proglen + sizeof(*hdr)),
                     PAGE_SIZE - sizeof(*hdr));
        start = (get_random_int() % hole) & ~(alignment - 1);

        /* Leave a random number of instructions before BPF code. */
        *image_ptr = &hdr->image[start];

        return hdr;
}
```

5 Mitigation measures

In the time since our attack was developed, a number of mitigation measures have been merged to the upstream kernel. The main measure that has been developed by Daniel Borkmann in parallel to the attack as part of the Kernel Self Protection Project is the upstream support for blinding the constants in eBPF[4]. In contrast to the Grsecurity implementation, which was specific to the x86 architecture, Daniel's design provides almost fully architecture independent implementation: this is obtained by blinding constants already at the eBPF instruction level, and feeding the blinded constants to the JIT compiler. This not only allows to have a unified and solid design for BPF/JIT hardening among all architectures, but also further improves security by having one well-reviewed hardening implementation. This protection has been merged to the upstream kernel in May 2016 and was released as part of version 4.7.

More hardening has been done to prevent exploiting Unix domain sockets. Willy Tarreau merged a patch[5] blocking the trick of circumventing the resource limit on the amount of opened file descriptors. This protection has been released in the kernel as part of version 4.5.

Kernel Address Space Layout Randomization (KASLR) for x86_64, an important feature that aims to prevent exploits from relying on static locations of kernel symbols, has been released in the kernel as part of version 4.8. If

[4]`git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b`

[5]`git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=712f4aa`

enabled, this feature randomizes the physical and virtual memory location of where the kernel image is decompressed, and makes it significantly harder for attackers to discover the location of kernel symbols needed for attacks. For example, it is not possible anymore to rely on binary-specific locations of `commit_creds()` or `prepare_kernel_cred()` symbols based on kernel version. An attacker would have to instead use various information leaks to obtain these values [9].

While KASLR is important, it still does not provide full protection from all exploits. For example, the addresses where `vmalloc()` allocates kernel memory are still not randomized, which can provide additional information to improve an attack.

6 Conclusions

In this whitepaper we presented two different approaches to make a successful attack against Linux BPF/JIT. The main point that we demonstrated is that in order to fully fix a vulnerability, one needs to address the actual cause and not its symptoms. Another important lesson is that relying on something to be probabilistically hard is not a reliable security measure, since an attacker may find another attack path which changes the success ratio. More information about the attack and its improvements can be obtained from our project page[6].

7 Acknowledgments

The authors would like to especially thank Daniel Borkmann for his helpful discussions about BPF and JIT and his readiness and enthusiasms to make the Linux kernel BPF/JIT more secure.

[6] ssg.aalto.fi/projects/kernel-hardening

References

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual. www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, 2016.

[2] SECure COMPuting with filters. www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2016.

[3] Piotr Bania. Jit spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

[4] Dion Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf, 2010.

[5] Daniel Borkmann. On getting tc classifier fully programmable with cls_bpf. www.netdevconf.org/1.1/proceedings/papers/On-getting-tc-classifier-fully-programmable-with-cls-bpf.pdf, 2016.

[6] Case Cook. Status of the Kernel Self Protection Project. outflux.net/slides/2016/lss/kspp.pdf, 2016.

[7] Jonathan Corbet. A JIT for packet filters. lwn.net/Articles/437981/, 2012.

[8] Jonathan Corbet. The kernel connection multiplexer. lwn.net/Articles/657999/, 2015.

[9] Yeongjin Jang, Sangho Lee, and Taesoo Ki. Breaking Kernel Address Space Layout Randomization with Intel TSX. www.blackhat.com/docs/us-16/materials/us-16-Jang-Breaking-Kernel-Address-Space-Layout-Randomization-KASLR-With-Intel-TSX-wp.pdf, 2016.

[10] Keegan McAllister. Attacking hardened Linux systems with kernel JIT spraying. mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html, 2012.

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

[12] J Mogul. *Efficient use of workstations for passive monitoring of local area networks*, volume 20. ACM, 1990.

[13] Jeffrey Mogul, Richard Rashid, and Michael Accetta. *The packer filter: an efficient mechanism for user-level network code*, volume 21. ACM, 1987.

[14] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). `www.kernel.org/doc/Documentation/networking/filter.txt`, 2016.

[15] Alexei Starovoitov. Tracing: attach eBPF programs to kprobes. `lwn.net/Articles/636976/`, 2015.