Randomization Can't Stop BPF JIT Spray

Elena Reshetova[1], Filippo Bonazzi[2], and N.Asokan[2,3]

[1] Intel OTC, Espoo, Finland
[2] Aalto University, Helsinki, Finland
[3] University of Helsinki, Helsinki, Finland

Abstract. The Linux kernel Berkeley Packet Filter (BPF) and its Just-In-Time (JIT) compiler are actively used in various pieces of networking equipment where filtering speed is especially important. In 2012, the Linux BPF/JIT compiler was shown to be vulnerable to a JIT spray attack; fixes were quickly merged into the Linux kernel in order to stop the attack. In this paper we show two modifications of the original attack which still succeed on a modern 4.4 Linux kernel, and demonstrate that JIT spray is still a major problem for the Linux BPF/JIT compiler. This work helped to make the case for further and proper countermeasures to the attack, which have then been merged into the 4.7 Linux kernel.

Keywords: Network Security, Berkeley Packet Filter, JIT Spray

1 Introduction

Attackers seeking to compromise Linux systems increasingly 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 SEAndroid policies that do not allow a compromised application to get any significant control 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 it is important to develop tools to help finding these mistakes, 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 end user devices [9].

The Kernel Self Protection Project (KSPP)[4] tries to eliminate whole classes of vulnerabilities that might lead to 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.

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

The Linux kernel Berkeley Packet Filter (BPF) Just-In-Time (JIT) compiler has been an important focus of the project, since it is widely used in the kernel and has seen successful attacks in the past. In 2012, the first JIT spray attack against the Linux BPF/JIT compiler was presented. Consequently, some countermeasures were implemented in the Linux kernel from version 3.10 that rendered this attack unsuccessful. The main measure was randomization of the memory offset where BPF programs are allocated, therefore making it difficult for an attacker to locate BPF programs in memory. However, as we show in this paper, this protection can be easily bypassed by further adjusting the attack payload and taking advantage of specific features of the randomization algorithm.

In this paper we make the following contributions:

– **Analysis and characterization** of **original POC attack** against BPF/JIT as well as proposed countermeasures (Section 3).
– **Design, implementation and analysis of our new attack** that shows how these countermeasures can be circumvented on a modern Linux 4.4 kernel (Section 4).
– **Overview of recent measures** that have been added to the Linux kernel from version 4.7 to eliminate these types of attacks altogether (Section 6).

2 Berkeley Packet Filter and JIT compiler

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 the kernel and user spaces, the notion of a kernel *packet filter* agent was introduced [20,19]. 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 [18], which is referred to as Berkeley Packet Filter (BPF). This agent allows a userspace program to attach a single filter program 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) [7,21]. In order to simplify the terminology throughout this paper, we refer to the latter instruction set simply as *BPF instructions*. Linux BPF can be viewed as a minimalistic virtual machine construct [7] with a few registers, a stack and an implicit program counter. Different operations are allowed inside a 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 [21].

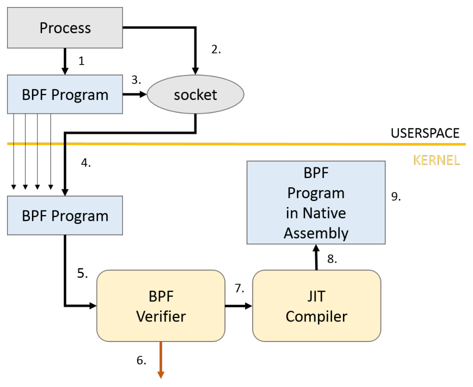


Fig. 1. Typical flow of a BPF program

While originally designed for network packet filtering, nowadays Linux BPF is used in many other areas, including system call filtering in seccomp [3], tracing [23] and Kernel Connection Multiplexer (KCM) [11].

In order to improve packet filtering performance even further, Linux utilizes a Just-In-Time (JIT) compiler [10,21] 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 `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's *module mapping space*, using the `vmalloc()` kernel memory allocation primitive.

3 JIT spray attack

JIT spraying is an attack where the behavior of a JIT compiler is (ab)used to load an attacker-provided payload into an executable memory area of the system [5]. This is usually achieved by passing the payload instructions encoded as constants to the JIT compiler and then using a suitable system vulnerability 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 [2]. 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 [6].

3.1 Original JIT spray attack on Linux

The original JIT spray attack against the Linux kernel using the BPF JIT compiler was presented by McAllister in 2012 [17]. The proof-of-concept (POC) exploit code[5] used a number of steps to obtain a root shell on a Linux device.

Creating the BPF payload: The POC creates a valid BPF program crafted to contain a small payload, comprised of the Linux kernel function calls `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 superuser (username `root` and uid `0` in Linux). The payload is located at the beginning of the BPF filter program (after a fixed offset containing the BPF header and other mandatory parts of a BPF program), and must be executed starting from its first byte in order for the attack to succeed. The addresses of the `commit_creds` and `prepare_kernel_cred` kernel symbols are resolved at runtime using the `/proc/kallsyms` kernel interface exposed through the `/proc` filesystem. The payload instructions are embedded into the BPF program using a standard BPF instruction: the *load immediate* instruction (`BPF_LD+BPF_IMM`), which loads a 4 byte constant into a standard register (`eax` by default for the x86 architecture). When compiled to native assembly on x86, this instruction is transformed into a `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` as an immediate value. 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 BPF program is transformed from BPF code to x86 machine code using the JIT compiler, and how the machine code looks like when an attacker succeeds in triggering unaligned

[5] `github.com/kmcallister/alameda`

execution from the second byte of the BPF program. The last 3 bytes of the constant (that are used for the payload) are shown in red, and it is easy to see how they get propagated from the BPF program code to the x86 machine code.

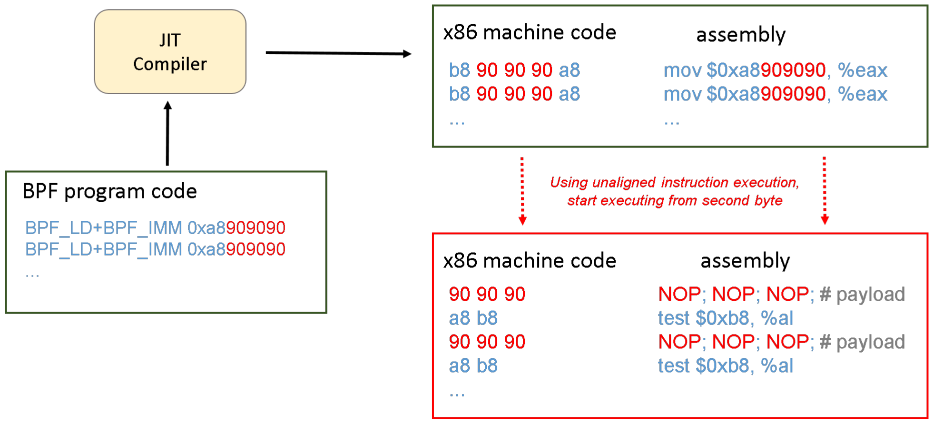


Fig. 2. BPF payload JIT compilation and unaligned execution. This sample payload contains six `NOP` instructions.

Loading the payload in memory: In order to load many copies of the BPF program payload in kernel memory, the attacker's process needs to create many local sockets, since each socket can have only one BPF program 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: open one local Unix socket and send the resulting file descriptor over another local Unix socket, and 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 programs containing the payload in kernel memory.

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 models the presence of an actual kernel bug that an attacker can use to redirect the execution flow during a real attack. `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.

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's *module mapping 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 BPF program at the beginning of a memory page: since the length of the BPF program is fixed, the attacker always knows the correct offset to jump to on a page in order to hit the payload. Figure 3 illustrates this. One thing to note is that all memory allocations of BPF programs are done using the `vmalloc()` kernel function, which leaves a one page gap of uninitialized memory between subsequent allocations; this is done in order to protect against overruns [13].

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.

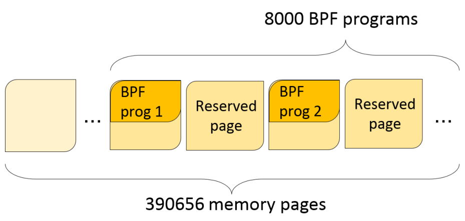


Fig. 3. Original attack

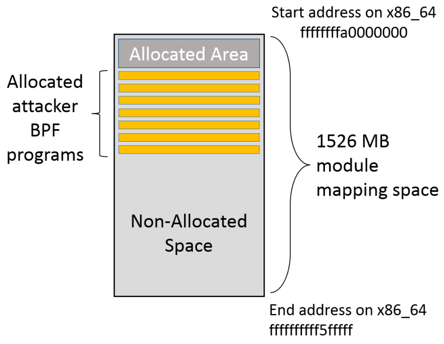

Start address on x86_64 ffffffffa0000000
Allocated Area
Allocated attacker BPF programs
1526 MB module mapping space
Non-Allocated Space
End address on x86_64 ffffffffff5fffff

Fig. 4. BPF program allocations on `x86_64`

The kernel's *module mapping space* for `x86_64` is 1526 MB (or 390656 4KB pages)[6]. The attacker has populated 8000 4KB pages with the payload code. The resulting success probability for a single guess is

$$P_{pre} = \frac{\#\ of\ pages\ containing\ the\ payload}{\#\ of\ kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

which is enough to make the multi-guess attack successful in most of the cases. It is important to note that when an attacker jumps to a page that doesn't contain the attacker's BPF program, the machine behavior is unpredictable. There are three possible cases:

- *Invalid instruction.* If the landing instruction is invalid, the child process is simply killed and the attack can continue.
- *Valid instruction with bad consequences.* If the landing instruction tampers with some key machine register, the whole OS can hang and the machine

[6] www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

needs a hard reboot to recover. It is hard to estimate the theoretical probability of executing such an instruction: it depends on CPU state, content of registers and the specific instruction itself.
– *Valid instruction with no visible consequences.* It is also possible that executing a valid instruction doesn't harm the system, and the process continues to the following instruction, where all three cases are again possible.

3.2 Community response

After the attack was publicly released, 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 BPF program would be located at a random offset inside the page. In addition, the space between the page start and the BPF program - 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 [2]. 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 BPF program starts. This is important because in order for the attack to succeed, the attacker needs to start execution exactly at the first byte of the attack payload. The resulting success probability for a single guess dropped to

$$P_{post} = P_{pre} \cdot \frac{\#\ of\ correct\ locations\ in\ a\ page}{\#\ of\ locations\ in\ a\ page} = \frac{8000}{390656} \cdot \frac{1}{4096} \approx 0.0004\%$$

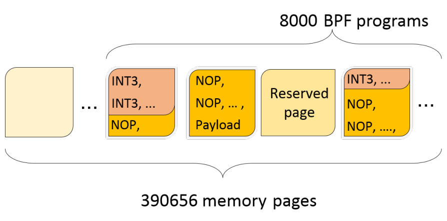
Furthermore, when the attacker jumps to a page that contains a copy of the BPF program, guessing the wrong offset is likely to be heavily punished (executing the `INT3` instruction will, in practice, result in a kernel panic and OS freeze), considerably slowing down the attacker.

4 Our attack

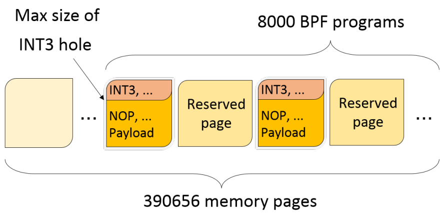
As part of the Kernel Self Protection Project together with one of the kernel BPF/JIT maintainers, we started to look into further securing BPF/JIT: our objective was to show that the existing measures implemented in the upstream kernel were not enough to stop JIT spray attacks.

The main part of the work was 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, discussed below. One issue common to both approaches 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



(a) Approach 1



(b) Approach 2

Fig. 5. Our attack

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 possible on Ubuntu and similar distributions with a 4.4 kernel, since they do not utilize KASLR yet, and kernel symbols are located at a deterministic address for all copies of a specific compiled kernel (*e.g.* `4.4.0-rc5`). 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 program is limited to 4096 BPF instructions [21], this is more than enough to to obtain a compiled BPF program larger than the 4KB of a kernel memory page. When the BPF program size grows beyond one page but is under 2 pages, jumping to a page containing the attacker's BPF program has a 50% chance of success. The probability could be even higher if we extended the BPF program to be longer than 2 pages, by increasing the number of BPF instructions to the maximum value of 4096. In Approach 1, we changed the original attack to generate longer BPF programs, containing 1318 BPF instructions, which take 2 4KB pages: we did this by filling the payload with NOP instructions and placing the actual attack code at the end. Figure 5 (a) illustrates this attack approach.

When we jump to a random page, we try to execute the first 2 offsets (0 and 1) before moving to the next random page. This 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`. The number of sockets, and therefore loaded BPF programs, is the same (8000) as in the original attack. If we happen to jump to a page which does contain a copy of the BPF program, 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.

The success probability for a single guess of our Approach 1 attack can be calculated as follows. Having 8000 filters, each occupying two pages, results in 16000 pages in total. Out of these, in the worst case only half (those which contain BPF program code at the beginning) are usable for the attack: in the worst case, whenever we hit a page starting with `INT3` instructions, we hang the VM, failing the attack. Therefore, the number of usable pages drops to 8000: in other words, with Approach 1, the attacker can restore his success probability for a single guess to the original value (Section 3.1), despite the new fix.

$$P_{app1} = \frac{0.5 \cdot \#\ of\ pages\ containing\ the\ payload}{\#\ of\ kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

However, the probability of a bad guess resulting in VM hang (jumping to one of the 8000 pages containing `INT3` instructions) is approximately the same 2%; this unfortunately renders the attack not useful in practice.

4.2 Approach 1 improved

To raise the success rate of Approach 1, we used the following observations.

The memory for BPF programs is allocated using the `module_alloc()` function, that in turn uses the `vmalloc()` function to allocate memory regions from the kernel's *module mapping space*. When we allocate 8000 filters, these allocations happen to be mostly adjacent in memory, and start after already allocated areas. Figure 4 illustrates this. For all x86_64 kernels, the kernel's *module mapping space* starts at the address `0xffffffffa0000000`. Given a certain kernel version and a default setup (loaded modules, filters *etc.*), the location where at least some of the attacker's filters are going to be loaded is rather predictable. For example, for our `4.4.0-rc5` kernel compiled with the default Ubuntu configuration and with no additional modules or filters loaded, there are always BPF programs allocated from the `0xffffffffa1000000` address. For the `4.4.0-36-generic#55-ubuntu` kernel provided with Ubuntu 16.04, with no additional modules or filters loaded, the allocations start around `0xffffffffc0000000`.

Knowing this, we can further narrow down the range of pages where we want to search for our payload. Instead of exploring all 390656 pages within the 1536 MB *module mapping space*, we only search through the address range from `0xffffffffa1000000` to `0xffffffffa1fff000`), which corresponds to 4095 pages - a significant reduction. Ideally, an attacker would like to have all 4095 pages allocated with filters, because this would increase the success rate. However, as already explained in Section 3.1, all memory allocations of BPF programs are done by the `vmalloc()` kernel function, which leaves a one page gap of uninitialized memory between subsequent allocations. Therefore, this smaller region only contains 1366 two-page filters, for a total of 2730 pages, and in the worst case only 1366 useful pages for an attacker. Following these observations, the attack achieves the following success probability for a single guess:

$$P_{app1_{improved}} = \frac{0.5 \cdot \#\ of\ pages\ containing\ the\ payload}{\#\ of\ pages\ in\ the\ search\ region} = \frac{1366}{4095} = 33.4\%$$

At the same time, this smaller region still contains 1366 pages likely to start with `INT3` instructions. Therefore, the probability to jump to one of these pages is the same as the success probability of a single guess. The remaining 1363 pages from the smaller region are in an uninitialized state, and jumping there equals to jumping to a randomly initialized memory location.

4.3 Approach 2

Our second approach is based on how the allocation of a BPF program happens and how the random offset of a BPF program is computed. This is done by the `bpf_jit_binary_alloc()` function, implemented in the `kernel/bpf/core.c` file.

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-dependent instructions (`INT3` for x86) (line 229). The actual starting offset of the BPF program 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 program, 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 5 (b) illustrates this attack approach.

In our experiments, we were able to bring the BPF program size to 3964 bytes and successfully jump over the first 132 bytes, called `hole_offset`. As in Approach 1, trying both `hole_offset` and `hole_offset + 1` protects us from the unlucky case where our selected jump destination contains the `b8` byte. The attack success probability for a single guess can be calculated as follows and equals to the original attack success rate:

$$P_{app2} = \frac{pages\ containing\ the\ payload}{kernel\ module\ mapping\ pages} = \frac{8000}{390656} \approx 2\%$$

Since - as explained above - the attacker can avoid jumping into pages containing `INT3` instructions, the global success rate of the attack is increased significantly, as shown in Section 5. The single guess success probability could be increased even further by optimizing Approach 2 with the same techniques applied to Approach 1 improved: restricting the search region to assumed BPF program locations in specific kernel versions, the single guess success probability could be increased to 50%. However, as in Approach 1 improved, this would come at the cost of a loss of generality; since the success rate for Approach 2 is already sufficiently high, we do not need to perform any such optimization which needs to be updated to each kernel version.

5 Experimental evaluation

While the theoretical single guess success probability for each attack approach is interesting on its own, what matters in practice is the global attack success rate, *i.e.* the probability for an attacker to obtain root privileges before hanging the VM. This success rate is hard to theoretically calculate, due to the difficulty of characterizing the behavior of several factors which influence it; these factors are - for example - the CPU state and kernel memory layout at the time of the attack, the specific random number generator used (which may generate random numbers according to a non-ideal distribution), *etc.* Therefore, we experimentally evaluated our different attack approaches using the following setup.

The attack was run in a Linux virtual machine powered by `qemu`, running the `4.4.0-36-generic#55-ubuntu` kernel provided with Ubuntu 16.04. The virtual machine had 2048MB of RAM and access to host 2 CPUs (Intel Core i7-3520M). We collected many attack runs per each approach: each run terminates either with an attack success (the attacker's process obtains root privileges) or by an attack failure (the VM hangs). If an attacker's process succeeds in obtaining root privileges, it is terminated, all filters are unloaded and a new run is performed from scratch. If the VM hangs, the virtual machine is forcefully restarted and a new run is performed. The measurements are shown in Table 1.

Attack	***# guesses***	***P single guess***	***# runs***	***P global***	***Mean(ℓ)***	***SD(ℓ)***	***Median(ℓ)***
4.1	12280	2.0 %	410	58.3 %	29.6	28.2	21
4.2	858	33.2 %	438	65.1 %	2.1	1.5	2
4.3	80190	2.0 %	1636	99.6 %	49	47.3	35

Table 1. Experimental results. ℓ is the length of a successful run.

The single guess success rate is calculated as the number of successful guesses divided by the total number of guesses. The global attack success success rate is calculated as the number of successful runs divided by the total number of runs. The mean and standard deviation of run lengths are calculated over the lengths of successful runs.

6 Mitigation measures

Our attack demonstrated conclusively that randomization alone is insufficient to stop BPF JIT spray attacks. The main reason is that randomization does not address the underlying problem of the attacker being able to deterministically control what instruction would be executed when the attacker can jump to a location in the payload of the BPF program and trigger unaligned execution.

Right after the original JIT spray attack in 2012, the Grsecurity[7] kernel security project released a blinding-based hardening mechanism to defend against

[7] `grsecurity.net`

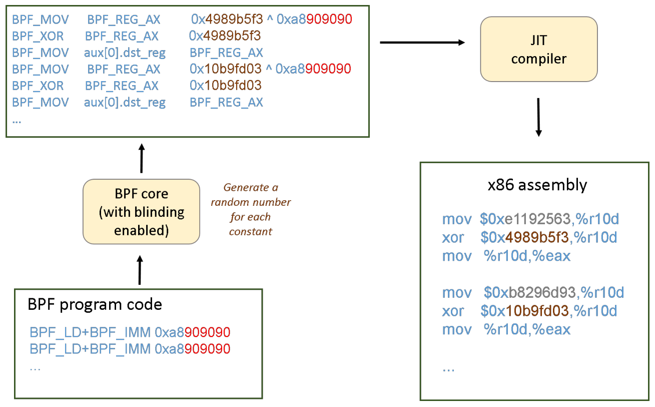
the attack. Their implementation only supported the `x86` architecture. At that time it was not merged into the upstream Linux kernel due to the desire to have an architecture independent approach, the performance implications of the feature, and most of all the belief that the randomization measures implemented in the upstream kernel would be enough to stop BPF JIT spray attacks.

Since our attack was demonstrated, a number of mitigation measures have been merged to the upstream kernel. The main measure is blinding the constants in eBPF[8]. Blinding consists of XORing each constant with a random number at compile time, and XORing it with the same random number immediately before using it at runtime. This way, constants never appear in memory "in clear", and an attacker cannot deterministically control the content of the memory area in which they are stored. Blinding functionally turns the attacker's payload code into a set of random values, and therefore blocks the code injection which allowed the JIT spray attack: the probability of finding instructions that can lead to the behavior desired by the attacker is equal to the probability of finding them in a randomly initialized memory area. The feature changes the BPF program code in the following way, shown in Figure 6. First, a new random number is generated for each constant; this number is XORed with the constant, and the result is used as the parameter of a `BPF_MOV` instruction added to the BPF program. This means that, at runtime, the XORed constant will be written to a register. Then, an instruction is added to the BPF program to XOR the same random number with this register: this is used to recover the original value of the constant at runtime. Once recovered, the original value of the constant is then used by the actual instruction which contained the constant in the original code. This is achieved by modifying the original instruction to operate on the freshly recovered value from the register, and not on an immediate constant. As a result, the x86 assembly code produced by the JIT compiler does not contain any constants in clear anymore.

In contrast to the `x86`-specific Grsecurity implementation, the new implementation is architecture independent: this is obtained by blinding constants already at the eBPF instruction level, and feeding the blinded constants to the JIT compiler. This implementation does require minimal support from each architecture that would like to enable this security feature, but the required support is straightforward. This not only allows a unified and solid design for BPF/JIT hardening for all architectures, but also further improves security by having a single, well-reviewed hardening implementation. The performance overhead of blinding was measured to be around 8% for the test suite cases, and expected to be even smaller in real world scenarios[9]. 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. A patch[10] has been merged to prevent circumventing the resource limit on the amount of open file descriptors, which was released in kernel version 4.5.

[8] git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b
[9] patchwork.ozlabs.org/patch/622075/
[10] git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=712f4aa


JIT compiler
BPF core (with blinding enabled)
Generate a random number for each constant
BPF program code
x86 assembly

```
BPF_MOV   BPF_REG_AX       0x4989b5f3 ^ 0xa8909090
BPF_XOR   BPF_REG_AX       0x4989b5f3
BPF_MOV   aux[0].dst_reg   BPF_REG_AX
BPF_MOV   BPF_REG_AX       0x10b9fd03 ^ 0xa8909090
BPF_XOR   BPF_REG_AX       0x10b9fd03
BPF_MOV   aux[0].dst_reg   BPF_REG_AX
...
```

```
BPF_LD+BPF_IMM 0xa8909090
BPF_LD+BPF_IMM 0xa8909090
...
```

```
mov  $0xe1192563,%r10d
xor  $0x4989b5f3,%r10d
mov  %r10d,%eax

mov  $0xb8296d93,%r10d
xor  $0x10b9fd03,%r10d
mov  %r10d,%eax

...
```

Fig. 6. BPF constant blinding. The attacker payload is shown in red; the random numbers used for blinding are in brown; the blinded constants are in grey.

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 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 [15]. 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.

7 Related work

With the development of various hardening mechanisms, such as stack protectors [12], DEP [1] and ASLR [24], it became harder for attackers to perform code injection and code reuse attacks. However, if a system features a JIT compiler, this provides an attractive attack vector: the JIT compiler produced code can be largely controlled or predicted by an attacker, and it is executable by default. Since the first successful JIT spray attack on Adobe Flash Player on Windows [6], many similar attacks have been done on different JIT compilers. At the same time, a number of JIT hardening mechanisms have been proposed. Chen *et al.* [8] proposed *JITDefender*, a system for userspace JIT engines that marks all JIT-produced code as non-executable and only marks it executable during the actual execution by the JIT engine. This design is not applicable to the kernel BPF JIT engine, since there may be many different small BPF programs loaded at the same time in memory, and their execution would need to be constantly enabled/disabled depending on each passing packet and allocated sockets. Homescu *et al.* [14] developed the userspace *librando* library, which, in addition to performing constant blinding, post-processes JIT-emitted code in order to randomize and diversify its location. Athanasakis *et al.* [4] demonstrated that JIT spray is still possible if constants of 2 bytes or less are left unblinded. They performed their attack successfully on the JIT engines of Mozilla Firefox and Microsoft Internet Explorer. Their conclusion was that blinding all constants regardless of their size stops the attack, but this measure was found to be too performance costly for the mentioned userspace JIT engines. Jangda *et al.* [16] propose the *libsmack* library as an alternative to constant blinding. Their idea is to replace each constant with a randomized address that in turn stores the value of the constant. The library demonstrates slightly better performance compared to simple blinding of constants. In some cases, in order to optimize the performance, the code produced by a JIT engine is made both executable and temporarily writable in a cache. Song *et al.* [22] showed that this can be successfully exploited by an attacker through code cache injection techniques.

Fortunately, many of the above problems and challenges are not applicable to the Linux Kernel BPF JIT compiler. The JIT-produced code is only executable - and never writable - after it has been placed in memory. Since the size of each BPF constant is fixed to 4 bytes regardless of its actual value, it is only possible to apply full blinding; there is no possibility of obtaining a speed-security tradeoff by applying partial blinding. This allows the hardening solution of blinding all constants to be simple and fully effective, relinquishing the need for any additional measures. One additional reason why the mechanisms above do not directly apply to the BPF JIT case is given by the position of the BPF engine. Since the BPF JIT engine is implemented inside the Linux kernel, any security solution must be integrated into the engine itself instead of using external libraries or post-processing mechanisms.

8 Conclusions

In this paper we presented two different approaches to successfully attack the BPF/JIT engine in the Linux kernel up to version 4.4. We demonstrated that randomization alone is insufficient to stop BPF JIT spray attacks, since it does not remove the attacker's ability to supply the attack payload using BPF constants. As a result of our attack, a robust constant blinding solution against BPF JIT spray attacks has been merged to the upstream Linux kernel, which fixes the actual root cause of the problem. More information about the attack can be obtained from our project page[11].

Acknowledgments. The authors would like to thank Daniel Borkmann for his helpful discussions about BPF/JIT, and his readiness and enthusiasms to make it more secure.

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

References

1. A detailed description of the Data Execution Prevention (DEP) feature. support.microsoft.com/en-us/kb/875352 (2016)
2. 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)
3. SECure COMPuting with filters. www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt (2016)
4. Athanasakis, M., et al.: The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In: NDSS (2015)
5. Bania, P.: JIT spraying and mitigations. arXiv preprint arXiv:1009.1038 (2010)
6. Blazakis, D.: Interpreter Exploitation: Pointer Inference and JIT Spraying. www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf (2010)
7. Borkmann, D.: 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)
8. Chen, P., Fang, Y., Mao, B., Xie, L.: JITDefender: A defense against JIT spraying attacks. In: IFIP. pp. 142–153 (2011)
9. Cook, C.: Status of the Kernel Self Protection Project. outflux.net/slides/2016/lss/kspp.pdf (2016)
10. Corbet, J.: A JIT for packet filters. lwn.net/Articles/437981/ (2012)
11. Corbet, J.: The kernel connection multiplexer. lwn.net/Articles/657999/ (2015)
12. Edge, J.: "Strong" stack protection for GCC. lwn.net/Articles/584225/ (2014)
13. Gorman, M.: Understanding the Linux virtual memory manager (2004)
14. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Librando: transparent code randomization for just-in-time compilers. In: CCS. pp. 993–1004 (2013)
15. Jang, Y., Lee, S., Ki, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. pp. 380–392 (2016)
16. Jangda, A., Mishra, M., Baudry, B.: libmask: Protecting Browser JIT Engines from the Devil in the Constants. In: PST (2016)
17. McAllister, K.: Attacking hardened Linux systems with kernel JIT spraying. mainisusuallyafunction.blogspot.de/2012/11/attacking-hardened-linux-systems-with.html (2012)
18. McCanne, S., Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: USENIX winter. vol. 46 (1993)
19. Mogul, J.: Efficient use of workstations for passive monitoring of local area networks, vol. 20. ACM (1990)
20. Mogul, J., Rashid, R., Accetta, M.: The packer filter: an efficient mechanism for user-level network code, vol. 21. ACM (1987)
21. Schulist, J., et al.: Linux Socket Filtering aka Berkeley Packet Filter (BPF). www.kernel.org/doc/Documentation/networking/filter.txt (2016)
22. Song, C., Zhang, C., Wang, T., Lee, W., Melski, D.: Exploiting and Protecting Dynamic Code Generation. In: NDSS (2015)
23. Starovoitov, A.: Tracing: attach eBPF programs to kprobes. lwn.net/Articles/636976/ (2015)
24. Team, P.: PaX address space layout randomization (ASLR) (2003)