

The Good, Bad and Compromisable Aspects of Linux eBPF

2022 discoveries of new privilege
escalation techniques

Matan Liber



Table of contents

03	Purpose
03	Executive summary
03	Does it apply to my organization?
03	Who should take the time to read this document?
04	Attack surface domain
04	Relevant MITRE ATT&CK TTPs
04	Recommended Mitigation
05	Taking it from the top with greater detail
05	What is eBPF
06	What makes eBPF such an interesting attack vector?
06	BPF - Key elements to Know
06	BPF maps
06	The BPF verifier and ALU sanitation
07	The culprit's abettor - improper input validation
07	Tech dive
08	Step 1: Getting an invalid register value
09	Step 2: Achieving heap out of bound read/write
09	ALU sanitation is still a blocker
13	BPF helpers
15	Step 3: Achieving arbitrary read/write
15	Kernel heap allocations
16	Bypassing freelist randomization protections in the kernel
18	Using bpf_structure overwrite to gain arbitrary read/write
18	Step 4: Elevating privileges by overwriting our process credentials
19	Recap and a future plan for mitigation
20	About the author
20	About Pentera
20	Thanks & credits
20	References

Purpose

Reading this paper will allow you to understand the eBPF mechanism and how a fairly small bug can lead to the compromise of the entire system.

Executive summary

Modern hacking techniques often use legitimate operating system tools for bad purposes. Such is the potential case with the common traffic monitoring Linux subsystem called eBPF.

eBPF is a two-way street that, if abused, allows the adversary direct access and privilege to the Linux kernel. Although eBPF does impose restrictions on the code running in it, some of them can be bypassed. This would result in the ability to run malicious code at the kernel level and achieve privilege escalation. In this paper we will review how it is done.

The fast-pass to disabling this privilege escalation technique is to set the configuration flag for this subsystem to prevent unauthorized privileged access and assure the environment has the latest kernel update.

For older kernels, a modern perimeter of access needs to be established to prevent and restrict possible unauthorized access.

Does it apply to my organization?

You might be prone to an attack that leverages eBPF bugs if you are running an Ubuntu workstation or server that was provisioned in the past 2 years.

Who should take the time to read this document?

SOC, Blue Teamers, DFIR and any other corporate function that manages the risk and response to attacks on Ubuntu devices.

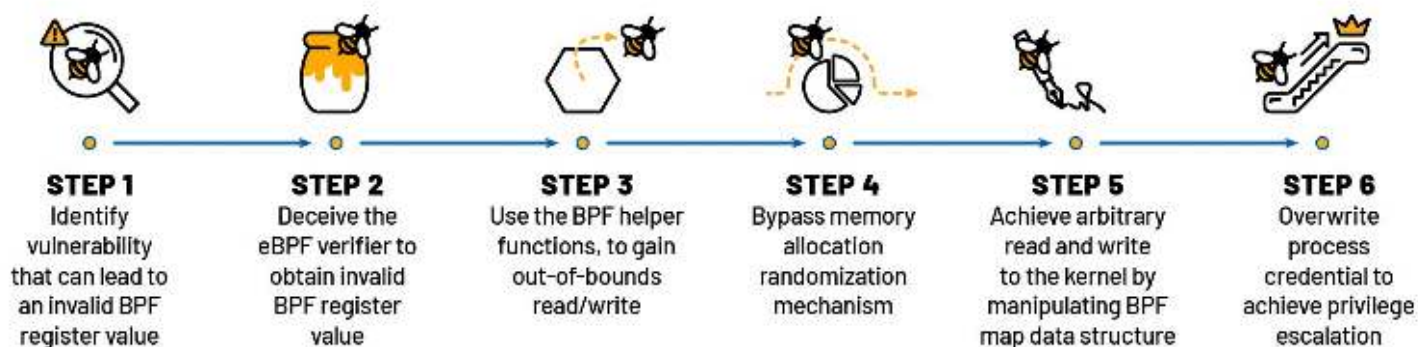
On the other side of the cyber spectrum, Red Teamers and pen-testers are encouraged to read this attack domain to enrich their attack engagements with hands-on tips from a senior security researcher.

Attack surface domain

Linux, mainly Ubuntu, but possibly any Linux machine where `unprivileged_bpf_disabled` flag is set to 0.

Relevant MITRE ATT&CK TTPs

Exploitation for Privilege Escalation T1068



Recommended Mitigation

Ensure that your Linux distro is configured not to allow unprivileged users to run eBPF programs. This means that the config parameter `'unprivileged_bpf_disabled'` should be set to 1 in the kernel to disable the possibility of running eBPF programs as an unprivileged user.

If the config parameter `'unprivileged_bpf_disabled'` in the kernel is set to 0, it will be possible to run eBPF programs as an unprivileged user and the attack vector described in this paper may be possible.

The faulty configuration is not the default in most Linux distributions. For example, it is properly disabled in Red Hat distributions. However, this is not always the case, and should regularly be monitored and enforced, to mitigate against malicious tampering and modification.

You can check the value of this kernel setting using the command:

```
cat /proc/sys/kernel/unprivileged_bpf_disabled
```

You can find the code for the POC on Github [here](#).

Taking it from the top with greater detail

Input validation vulnerabilities are no rare sight in Linux distributions, with the most common outcome being Privilege Escalation (PE). Just to name a few salient examples, we can recount [CVE-2020-8835](#), [CVE-2021-4204](#), [CVE-2021-20268](#), and most recently [CVE-2022-23222](#), all of which affect eBPF, an extended BPF JIT virtual machine in the Linux kernel.

In this paper, I will walk you through my thought process as I decipher the technical intricacies of a vulnerability report and develop a novel exploitation technique that circumvents randomization protections in the kernel.

What is eBPF

Before we delve deeper into Privilege Escalation in eBPF, let's take a step back to recap [what is the Berkeley Packet Filter \(BPF\)](#), the foundation for eBPF technology. BPF is a technology for operating systems that allows programs to analyze network traffic. It provides a raw interface to data link layers (i.e. Layer 2 connectivity), allowing a user space process to supply a filter program specifying which packets it wants to receive. BPF is available on most Unix-like operating systems, and eBPF is the extension for Linux and Microsoft Windows.

Essentially, an eBPF program is made of a series of special bytecode. An eBPF program can be written in a higher-level language and then compiled into the bytecode, or, it can be written as a set of x86 assembly-like instructions. eBPF programs have 11 registers including a stack pointer and an auxiliary pointer, a program counter and a 512 bytes stack.

eBPF is designed to safely and efficiently extend the capabilities of the kernel without requiring changes to kernel source code or load kernel modules. For this reason, eBPF programs, as you would expect, usually require high-level privileges to run.

What makes eBPF such an interesting attack vector?

When you consider that an eBPF program is essentially code provided by the user that runs with kernel-level privilege, the significance becomes self-evident. This sounds like everything an attacker could ever want – not that eBPF makes it easy for attackers.

eBPF certainly does a decent job imposing restrictions on the code, but that's exactly where eBPF vulnerabilities come into play, potentially allowing attackers to bypass restrictions and gain the ability to run malicious code at the kernel level. From there, the path to privilege escalation is surprisingly short.

BPF – Key elements to Know

BPF maps

A vital aspect of eBPF programs is the ability to share collected information, store state and store data larger than the 512 bytes provided by the stack. For this purpose, eBPF programs use eBPF maps. eBPF maps consist of key-value mappings and can be accessed from eBPF programs and applications in userspace via system calls.

The BPF verifier and ALU sanitation

The ability to run code at the kernel level is a very powerful tool, and when placed in the wrong hands, it can be abused. This is where the eBPF verifier comes in. The verifier has a crucial role in running eBPF programs: making sure they can't act maliciously. It does that by simulating the program's flow and checking many things, including tracking register values and types to name a few.

The eBPF register has 2 value types: pointers and scalars. The verifier is in charge of keeping track of the register's values and performing the following checks:

- Pointer bounds checking (Both during run time and before the program runs).
- Verifying that the stack's reads are preceded by stack writes. This is necessary because the BPF stack is not initialized to 0, which could lead to data leakage.
- Disallowing writing of pointers to the stack, again preventing pointer leaks.

The culprit's abettor - improper input validation

On January 13, 2022, a security researcher dubbed 'tr3e' posted on [Openwall](#) a discovery concerning an [improper input validation](#) in Linux Kernel eBPF. This vulnerability is the beginning of our journey to privilege escalation.

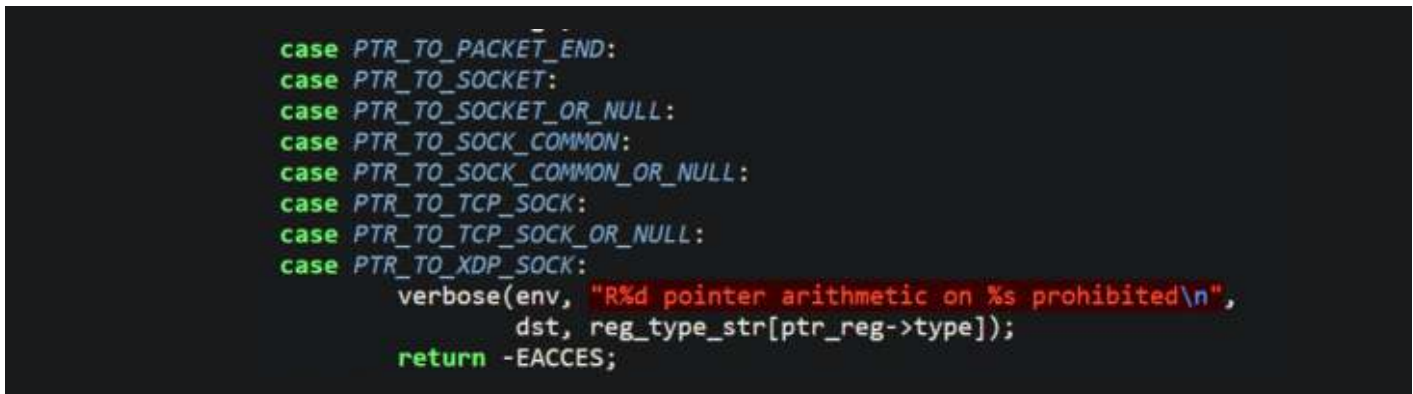
Tech dive

eBPF has several types of pointers, some of which have the phrase 'OR_NULL' in their names. Curiously enough, this is because some operations are rather unpredictable and might either yield pointers or fail at runtime and return 'NULL'. In other words, return values from these operations might or might not be usable - and this cannot be determined in advance.

For this reason, OR_NULL pointer types exist as an intermediate type, which can be checked at runtime against 0 to determine whether a pointer is 'NULL' or not. When checking against 0, two branches are created: in one the pointer type is without 'OR_NULL', and in the other, the register instead holds a scalar value of 0. For example, the pointer 'PTR_TO_MEM_OR_NULL' would change to 'PTR_TO_MEM'.

Pointer arithmetic should only be allowed when a pointer is known to be valid, but not null. The vulnerability discovered here is that pointer arithmetic is not disallowed for several OR_NULL pointer types. This results in the opportunity to trick the verifier into believing that no operation was performed on the pointer, when, in fact, its value was actually tampered with.

Once you know about it, the vulnerability is not difficult to spot directly in the code:
<https://git.kernel.org/>



```
case PTR_TO_PACKET_END:
case PTR_TO_SOCKET:
case PTR_TO_SOCKET_OR_NULL:
case PTR_TO SOCK_COMMON:
case PTR_TO SOCK_COMMON_OR_NULL:
case PTR_TO_TCP SOCK:
case PTR_TO_TCP SOCK_OR_NULL:
case PTR_TO_XDP SOCK:
    verbose(env, "R%d pointer arithmetic on %s prohibited\n",
            dst, reg_type_str[ptr_reg->type]);
    return -EACCES;
```

In this screenshot, we see that pointer arithmetic is prohibited on only some of the unpredictable pointers. One look at this table shows just how many of the unpredictable

pointer types are not properly excluded from pointer arithmetic checks:

OR_NULL` Pointers	Verifier Rules
PTR_TO_SOCKET_OR_NULL, PTR_TO SOCK_COMMON_OR_NULL PTR_TO_TCP SOCK_OR_NULL	Pointer arithmetic is prohibited
PTR_TO_MAP_VALUE_OR_NULL PTR_TO_BT ID_OR_NULL PTR_TO_MEM_OR_NULL PTR_TO_RDONLY_BUF_OR_NULL PTR_TO_RDWR_BUF_OR_NULL	Pointer arithmetic is not prohibited

Any of the `OR_NULL` pointers for which pointer arithmetic is permitted can be abused to trick the verifier into believing that no operation was performed on the pointer, when in fact its value was altered. This is where things start to get interesting. Let's see how we go from a "mere" input validation vulnerability to privilege escalation.

Step 1: Getting an invalid register value

The most basic aspect of our attack vector involves creating a situation where we have a register that the verifier thinks has a value 1, but actually has any value we choose.

Assuming we want to load the value X into the register, we do the following:

- Use the function `bpf_ringbuf_reserve` to generate a PTR_TO_MEM_OR_NULL in REG_0, making sure the actual value is NULL
- Copy REG_0 into another register
- Add x-1 into the other register. At this point, the verifier thinks that both registers either point to the same address as they did before this operation or contain null because of the vulnerability.
- NULL check REG_0
 - In the branch where REG_0 is not 0, we exit the program.
(We need to make sure this branch isn't taken.)
 - In the other branch, the verifier now thinks that both pointers contain 0, while in reality the second register contains x-1.
- Add 1 to the register.



SUCCESS!

Now the verifier thinks the register contains the value 1, when it actually contains X (where X is any value we want).

Step 2: Achieving heap out of bound read/write

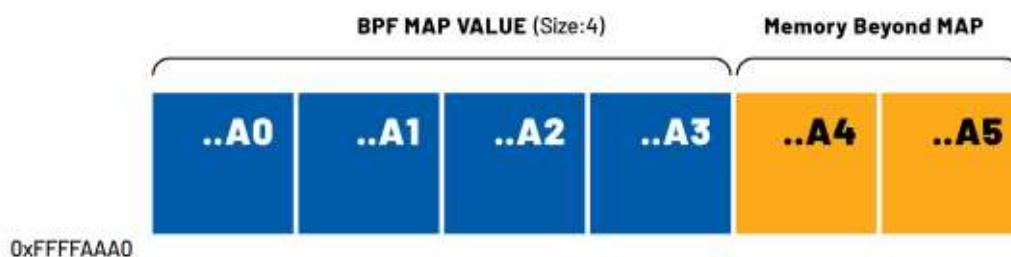
ALU sanitation is still a blocker

ALU sanitation is one of the mechanisms used to ensure that pointer arithmetic in BPF programs is not malicious.

Let's analyze why ALU sanitation is important.

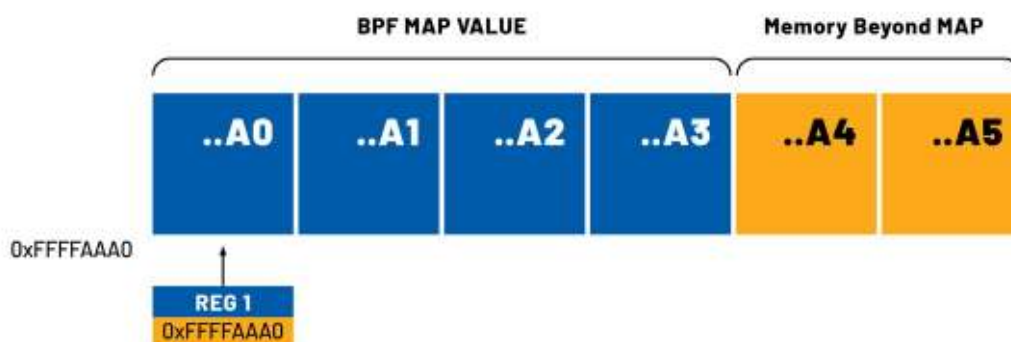
If pointer arithmetic was not validated at all, an attacker could do the following (*Keep in mind that the addresses and size in the following example are just for visualization and do not represent real values*):

First, create a map with values of size 4 in user-mode. This would create the following layout:



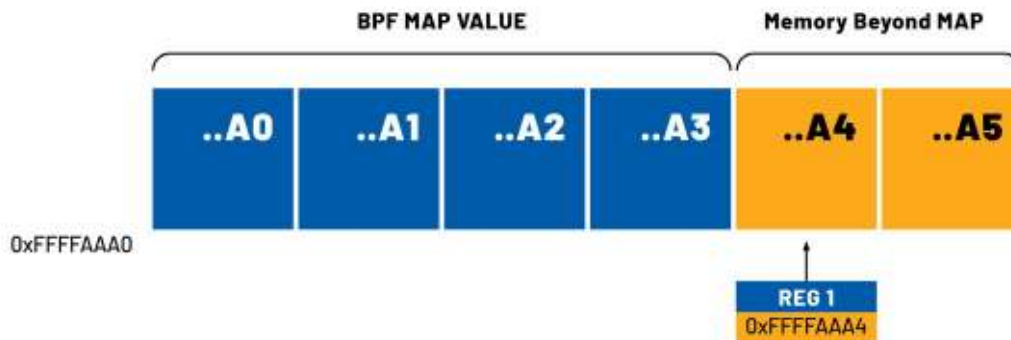
Then execute the following eBPF program:

1. Load a map value pointer into BPF_REG_1.

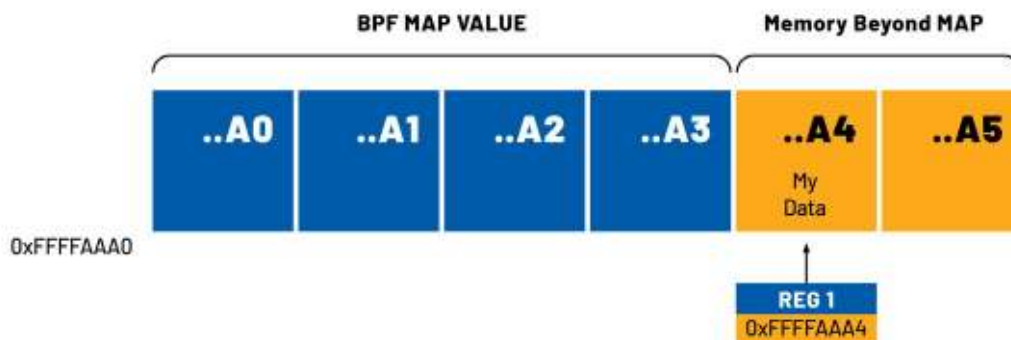


2. Add 4 to REG_1:

```
`BPF_ALU64_IMM(BPF_ADD, BPF_REG_1, 4)`
```



3. Write into the location REG_1 points to.

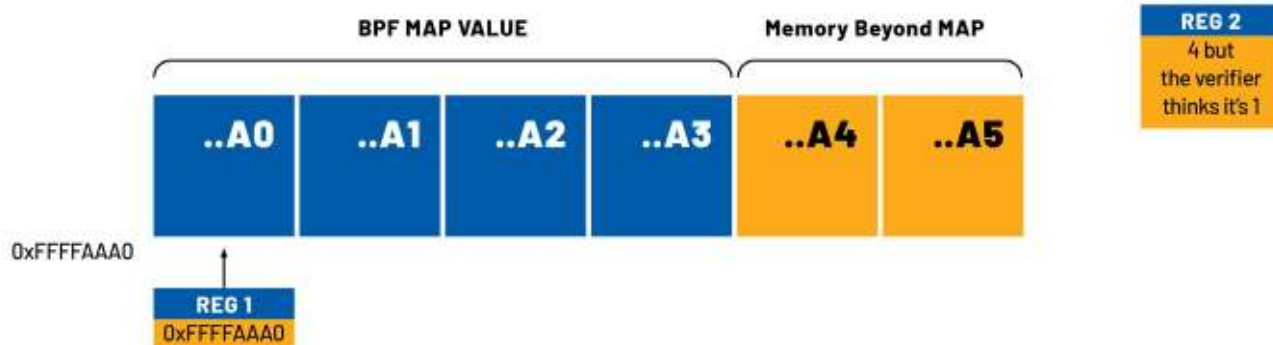


Here the verifier comes into play. It keeps tabs on the register and its boundaries and won't allow the program to run when it detects this out-of-bounds arithmetic.

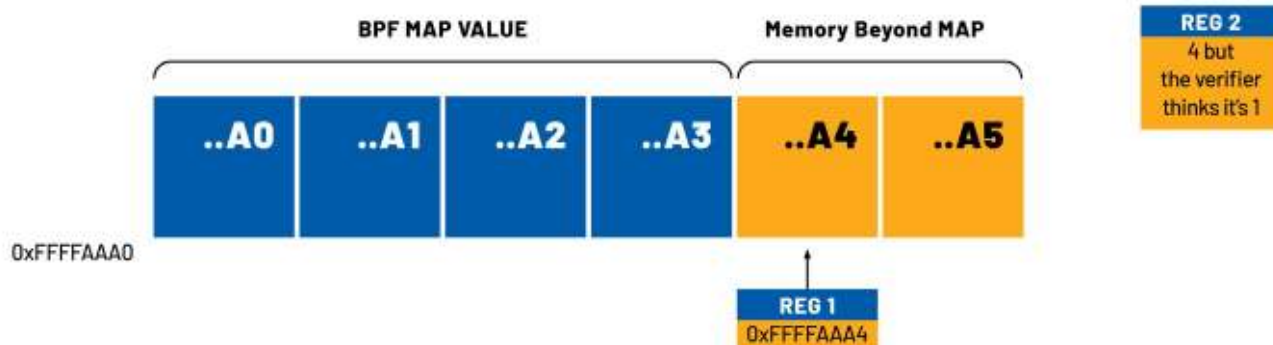
```
invalid access to map value, value_size=4 off=4 size=1
R1 min value is outside of the allowed memory range
R1 pointer arithmetic of map value goes out of range, prohibited for !root
```

Considering the primitive we achieved in the last step, you might think we could write out-of-bounds using pointer arithmetic with our invalid register in the following manner:

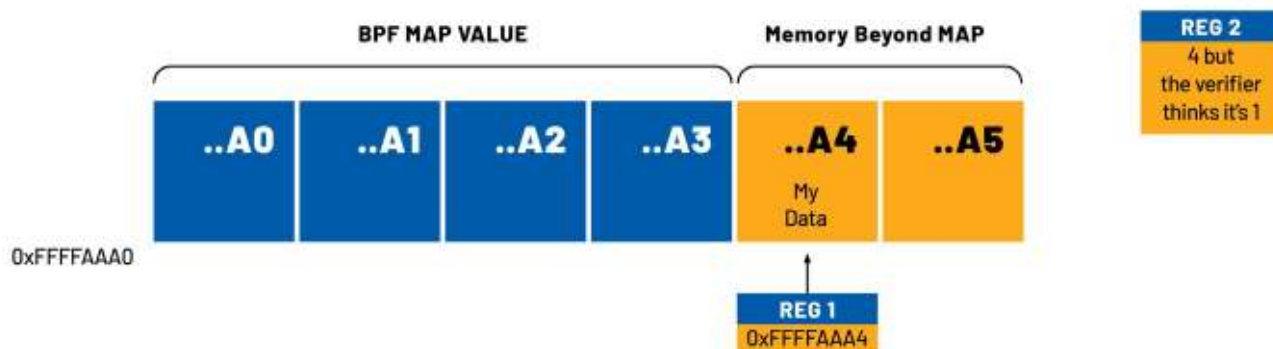
1. Use our primitive to get an invalid BPF_REG_2 that holds the value 4 but the verifier thinks holds the value 1.



2. Add REG_2 into REG_1:
`'BPF_ALU64_REG(BPF_ADD, BPF_REG_1, BPF_REG_2)'`



3. Write into the location this REG_1 points to.



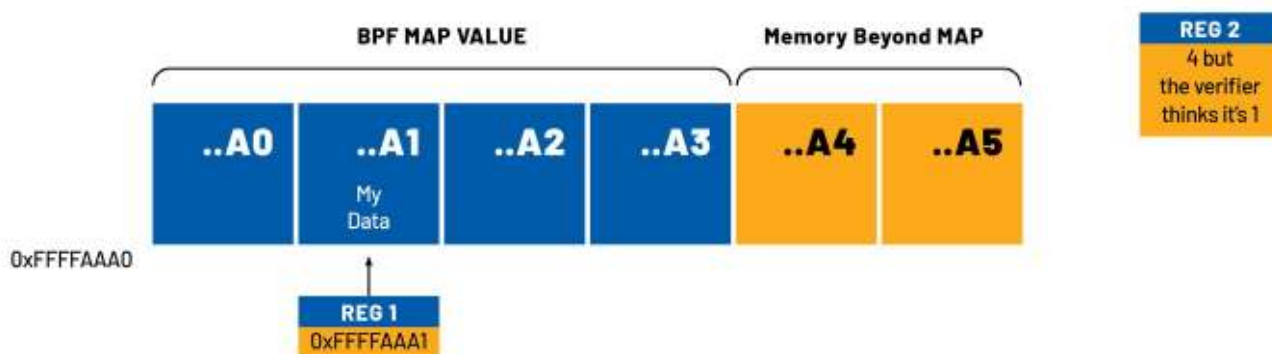
This way the verifier thinks the arithmetic operation is in-bounds.

Unfortunately, ALU sanitation will block us in this situation. Whenever the verifier sees an arithmetic operation between one register holding a pointer and another register holding a scalar of which the verifier knows the value it holds, it patches the operation in such a way that it would retain its effect without using the second register.

The verifier thinks BPF_REG_2 contains the value 1, so it will patch our instruction in step 2 to the equivalent of:

```
'BPF_ALU64_IMM(BPF_ADD, BPF_REG_1, 1)'
```

This renders our invalid register obsolete. So, in fact, our program will execute but we won't be able to write out-of-bounds.



Our next step is to figure out how to avoid this blocker and bypass ALU sanitation, which leads us to BPF helpers.

BPF helpers

To avoid our previous mishap, we use another feature of eBPF to our advantage – BPF helpers. BPF helpers are a set of APIs that BPF programs use for debugging, reading from packets, retrieving the time since the system was booted, and more.

Since our goal here is to achieve Local Privilege Escalation and we are starting out with a limited-privilege user, we can only execute certain types of BPF programs, as shown below:

```
if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
    type != BPF_PROG_TYPE_CGROUP_SKB &&
    !bpf_capable())
    return -EPERM;
```

The code snippet above is taken from `bpf_prog_load`. We see that as unprivileged users, `bpf_capable()` returns false. This means there are very few types we can run: `BPF_PROG_TYPE_SOCKET_FILTER` and `BPF_PROG_TYPE_CGROUP_SKB`. Anything else would return an error code.

The problem is that each program type is allowed to call a different subset of the available helpers. It took some digging, but eventually I found 2 suitable functions for use with the `BPF_PROG_TYPE_SOCKET_FILTER` program type:

- `bpf_skb_load_bytes`
- `bpf_ringbuf_output`

The first function, `bpf_skb_load_bytes`, is a function meant to read data from packets into memory. Let's have a look at its signature:

```
BPF_CALL_4(bpf_skb_load_bytes, const struct sk_buff *, skb, u32, offset,
            void *, to, u32, len)
```

As we can see, it receives:

- A pointer to `sk_buff` (which contains the packet data)
- An offset into the `sk_buff`
- A pointer to the destination memory
- The amount of bytes to copy



OOB WRITE SUCCESS!

By passing our invalid register as the length, we can achieve an out-of-bounds write. Just be careful not to pass a normal scalar value that is out-of-bounds as this will tip off the verifier and cause it to be disallowed.

The second function, `bpf_ringbuf_output`, is a function meant to copy data into a ring buffer that is used to communicate with the userspace from within an eBPF program.

Let's take a look at its signature:

```
BPF_CALL_4(bpf_ringbuf_output, struct bpf_map *, map, void *, data, u64, size,
           u64, flags)
```

As we can see, it receives:

- A pointer to `bpf_map`, that would be the map containing the ring buffer
- A pointer to the data to be copied
- The number of bytes to copy
- Flags



OOB READ SUCCESS!

By passing our invalid register as the size, we can achieve an out-of-bounds read.

We still need our invalid register to achieve the OOB read/write with the two functions.

Let's circle back to reiterate why we want the verifier to think our register has the value of 1 and not 0: If we try to copy 0 bytes, the verifier will not allow it, but it will not block us if we make it think we're copying 1 byte.

Step 3: Achieving arbitrary read/write

Now that we have OOB read and write to some pointer to memory, we need to find a good candidate that would serve as our memory pointer to overwrite to gain arbitrary read/write. BPF maps and their corresponding structure, `bpf_map`, are very good candidates.

Our first step is to allocate a map in user mode and then use a third BPF helper function, `bpf_map_lookup_elem`` within our bpf program, to get a map value pointer that was assigned to a key we passed as a parameter. Now we have a map value pointer we can freely pass to the previous helpers. However, now we encounter a new blocker.

Up until this point, we have achieved OOB read and write but we still have a problem to address: Using the aforementioned BPF helpers, `bpf_skb_load_bytes` and `bpf_ringbuf_output`, we can only overflow our map value pointers, but not underflow them.

This presents a problem since the `bpf_map` structure is located in memory **before** the map values, so we can't simply overwrite it.

Kernel heap allocations

To understand how we're going to circumvent the problem and overwrite our map structure, we first need to understand the basics of kernel heap allocations.

Allocations in the kernel heap eventually reside inside caches known as **kmem caches**. Kmem caches hold allocations of a whole range of sizes. There's a cache for each power of 2 starting with 8 and ending with 8192, along with 2 sizes that aren't powers of 2: 96 and 192.

Every allocation size is rounded up to the nearest size and a free object from the corresponding cache is allocated. For example, an allocation of size 560 would be rounded up to 1024 and a free object from the cache `'kmalloc-1K'` would be allocated.

Each cache is made up of slabs, which are typically 4K bytes in size and split into many objects according to their cache size. Whenever a cache runs out of free objects, it requests a new slab, and the allocation is given from the new slab. However, within each slab, the allocations are contiguous.

Here's the kicker: If we can get 2 maps of the same size allocated one after another in the same slab, we can overflow the first map into the second map and achieve our `bpf_map` structure overwrite!

Bypassing freelist randomization protections in the kernel

If only things were that easy. Simply allocating 2 maps won't cut it because freelist randomization, a mitigation introduced into the kernel, ensures that allocations from within the same slab are randomly arranged.

Let's say we have a slab with 4 free objects and we allocate 4 new objects from the same slab. If it weren't for freelist randomization, the objects would be allocated in a predictable order: 1 2 3 4. But with the mitigation in place, we could get 1 4 3 2 or 2 3 1 4, or any other order of allocation, so there is no way to know ahead of time that our allocations will be contiguous.

The issue might be clearer if visualized. For example, let's allocate two maps one after the other: bpf_map A followed by bpf_map B. Due to freelist randomization, we might get the following result after allocating the two maps:



Obviously, we can't just overflow map A into map B, because map A resides in memory after map B.

However, if we keep allocating more maps, eventually we will get the desired heap layout. To do this, we devise a probe to help us figure out the order of allocations. Every time we allocate a new map, we update it with a unique value to serve as our probe. We then use our BPF helper call to read out-of-bounds upstream from every map previously allocated to the location where the next map's value should be if the maps were allocated one after the other. We then check it against all the unique values we created. If it's still not found, we repeat the process and eventually luck will strike, providing us with the results we're after: evidence that we have successfully allocated two maps one after the other.

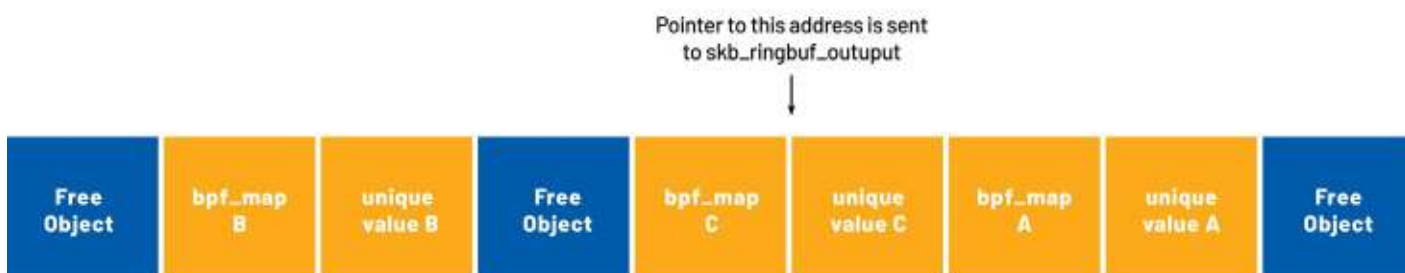
Getting back to our previous example, here is what the allocations would look like with the unique value as a probe:



Reading out-of-bounds from where unique value A resides, we read the contents of the next free object looking for our probe, the value we generated for unique value B, but unfortunately, we won't find it there. The same is true for reading out-of-bounds from map B.

True, in the above example, map B could be overflowed and eventually overwrite map A but this solution is too unstable. The 2 free objects in the middle could be corrupted in the process, causing the kernel to crash. While the original values could be re-written to avoid this problem, this solution is only partial. Consider what happens if, in the meantime, the kernel allocated a new object there? Again, we would overwrite it with different values and the kernel might crash.

There is one scenario where a map could be safely overflowed to overwrite another map. Observe the following situation:



Here, we finally got lucky. In this case, we will read out-of-bounds from map C and find unique value A exactly where we would expect to find it if map A were indeed allocated right after map C (in memory rather than chronologically).

So, there we have it. We have found a foolproof method to circumvent freelist randomization protections in the kernel and ensure that any out-of-bounds write will only overwrite map A. All we have to do is repeatedly allocate maps with a unique value to serve as a probe and read out-of-bounds until we find one of the unique values we generated in the expected spot, which will provide us with evidence that we got our target heap layout. That way we know that any out-of-bounds write will only overwrite map A.

Using bpf_structure overwrite to gain arbitrary read/write

To gain arbitrary read/write in the kernel, we need to manipulate some of the bpf_map structure members.

Arbitrary read is fairly trivial to achieve by overwriting the btf member of the bpf_map structure and then accessing it through userland by the bpf syscall.

Arbitrary write is a bit more complex to achieve. The exact details are described in [Manfred Paul's blog](#) but the point here is that the process of gaining arbitrary read is very much doable.

(For those of you who are curious about the exact details, I'll provide a short recap of the process: By overwriting the map_ops member, we can hijack the flow of many operations, though first we need to use our arbitrary read to make a copy of the original map_ops and then make the original pointer point to our fake ops table. Then we need to replace map_push_elem with map_get_next_key in our fake table and alter some other bpf_map members to bypass some checks within map_get_next_key.)

Step 4: Elevating privileges by overwriting our process credentials

Once we have arbitrary read and write, Privilege Escalation is around the bend.

Here's how we get there:

1. Locate 'init_pid_ns' (or use a symbol to get its address).
2. Go to init's task structure
3. Iterate init's task_struct list until we find a task_struct with the pid equal to our own process
4. Overwrite the uid, euid, and a few other members of our process cred structure with 0 to achieve Privilege Escalation!

The exact process is beautifully explained in [Manfred Paul's blog](#).

Recap and a future plan for mitigation

I hope you enjoyed diving into the depths of eBPF and following along the path of discovery to unfurl an input validation vulnerability and discover how it leads directly to Privilege Escalation.

Our journey began with a reported vulnerability in the BPF verifier affecting input validation, which allowed us to circumvent the verifier using an invalid register value. Next, we used BPF helper functions to manipulate the verifier and accomplish heap out-of-bounds read and write.

At this point, we had the ability to overflow, but not underflow our pointer. Since we still needed to gain arbitrary read/write in the kernel, we needed to manipulate some of the `bpf_map` structure members, which we achieved by developing a novel technique to bypass freelist randomization in kernel heap allocations. This allowed us to overwrite our process credentials and achieve our goal of escalating privileges in Linux!

My main conclusion as far as mitigating against these kinds of vulnerabilities can be divided into 2 categories:

1. Kernel developers - extend the principle of ALU sanitation to BPF helpers calls, since they too allow for access beyond the bounds of a pointer.
2. Network owners/security admins - validate your network continuously using automated validation solutions to verify if "minor bugs" in the eBPF mechanism can be leveraged for privilege escalation and asset compromise.

About the author



Matan Liber is a Cyber Attack Team Lead, Security Researcher and exploit developer at Pentera. Prior to joining Pentera, Matan served in a classified unit in the IDF, specializing in malware analysis, reverse engineering and IR.

About Pentera



Pentera is the category leader for Automated Security Validation, allowing every organization to test with ease the integrity of all cybersecurity layers, unfolding true, current security exposures at any moment, at any scale. Thousands of security professionals and service providers around the world use Pentera to guide remediation and close security gaps before they are exploited.

For more info visit: pentera.io

Thanks & Credits

I would like to give credit to two wonderful blogs I relied on heavily in my research:

- [Kernel Pwning with eBPF: a Love Story](#) by Valentina Palmiotti
- [CVE-2020-8835: LINUX KERNEL PRIVILEGE ESCALATION VIA IMPROPER EBPF PROGRAM VERIFICATION](#) by Manfred Paul

You can refer to these blogs for an in-depth explanation about eBPF and some of the ideas on how to exploit it.

References



[eBPF - Introduction, Tutorials & Community Resources](#)

The Good, Bad and Compromisable Aspects of Linux eBPF