Overflowing the Kernel Stack with BPF

Sai Roop Somaraju
Siddharth Chintamaneni
Dan Williams

<sairoop@vt.edu>
<sidchintamaneni@vt.edu>
<djwillia@vt.edu>
**BPF and kernel safety**

- BPF programs enable applications to extend the kernel's functionalities at runtime, all while ensuring stability and security.

- Guaranteed safety is made possible by the verifier engine which statically verifies BPF code.
BPF verifier and BPF runtime

• However, the verifier inherently relies on certain assumptions regarding the runtime execution environment, and these assumptions are essential to maintain safety.

• One such assumption is the availability of stack space to run the BPF program.
Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary
BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

- The stack space available to a BPF program is limited to 512 bytes.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.
When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

The stack space available to a BPF program is limited to 512 bytes.

When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.
BPF program interaction with the Kernel Stack

- When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

- The stack space available to a BPF program is limited to 512 bytes.

- When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.
When a verified BPF program executes from an attachment point, it typically inherits the existing kernel process stack.

The stack space available to a BPF program is limited to 512 bytes.

When a helper function or a kfunc is called from within the BPF program, it extends the same stack further into the kernel.
BPF-To-BPF calls

- BPF-to-BPF calls introduce a feature akin to function calls within BPF programs, leading to the creation of a new stack frame whenever a function call is initiated.
BPF-to-BPF calls introduce a feature akin to function calls within BPF programs, leading to the creation of a new stack frame whenever a function call is initiated.

- The stack space for each BPF program is still limited to a maximum of 512 bytes.
Tail calls allows a BPF program to call another BPF program and the caller BPF program will reuse the callee BPF programs stack frame. Each BPF tail call program is verified individually.
Tail calls enable a BPF program to call another BPF program, wherein the caller BPF program reuses the callee BPF program’s stack frame. Each BPF tail call program is verified individually.
Tail calls enable a BPF program to call another BPF program, wherein the caller BPF program reuses the callee BPF program's stack frame. Each BPF tail call program is verified individually.
If a tail call program is invoked from a BPF-to-BPF call function, the verifier restricts each BPF program's stack size to 256 bytes, as opposed to the standard 512 bytes.

At runtime, the Just-In-Time (JIT) compiler limits the number of tail calls to no more than 33 tailcalls.
Writing a sizeable BPF program of size 8 KB

- Base Kernel Stack
- BPF program #1 stack
- BPF program #2 stack
- BPF attachment
- subfunc_1()
- bpf2bpf call
- bpf tailcall
- func_1()
- bpf tailcall
- subfunc_1()
- func_2()
- bpf2bpf call
- bpf tailcall
- subfunc_2()
- bpf2bpf call
- bpf tailcall
- func_3()

- Writing a sizeable BPF program of size 8 KB

- max. 256 bytes
Writing a sizeable BPF program of size 8 KB
Writing a sizeable BPF program of size 8 KB

Each BPF program of *almost* 256 bytes * #33 tail calls
= ~8,448 bytes
BPF verifier assumptions about kernel's stack runtime

Therefore, here the verifier relies on two critical assumptions about the kernel’s runtime to restrict the depth of verified BPF programs:

1. Kernel stack will always have 8 KB of stack space available for a BPF program to run.
2. The total size of the BPF program's kernel stack and the stack for any helper functions it calls will be less than 8 KB.
BPF program attachment and its interaction with the stack

Stack overflow due to BPF program attachment

Stack overflow due to uncontrolled BPF program nesting

Discussion on Probable Solutions and Related Questions

Summary
At runtime, given the limited memory footprint of BPF programs and the controlled state of Kernel stack memory, one can assume that attachments are consistently innocuous.

Nonetheless, there have been cases reported in the file systems and networking communities where a significant amount of kernel stack memory was used in certain scenarios.

What if a BPF program is attached on such a kernel stack state?
To test our assumption, we considered XFS filesystem and ran a XFS test under certain memory constrained situation. It created >6KB of base stack.

System configuration: Intel x86_64 running on a VM using 1 core, 258 KB memory, 2GB swap memory.
Choosing an attachment function

- The most deepest functions of choice at the stack's top in XFS runs were associated with memory management and scheduling tasks, such as `list_lru_add()` and `update_load_avg()`.

- BPF dynamic attach mechanisms inheriting the bloated stack:
  - When dynamic tracing is active, `kprobe` optimizes by employing the same kernel stack instead of initiating a new interrupt stack.
  - `fentry` which uses bpf trampoline by design runs on the same kernel stack using dynamic tracing.
Design of the BPF program

- A BPF program is crafted as previously mentioned, utilizing 33 tail calls, resulting in more than 8KB of stack usage.

- Helper functions, which are not traced by either the verifier or the runtime environment, contribute to further stack growth on top of the BPF program.

- `bpf_get_stackid()` helper function is used in our case, which is called from the top/last tail call bpf program adding more stack space.
Overflowing a kernel stack using BPF program

- System call functions
- XFS stack
- Memory management functions, Scheduler functions, Block I/O functions, etc.

Stack growing downwards
Overflowing a kernel stack using BPF program

- BPF attach type functions
- XFS stack
- Memory management functions, Scheduler functions, Block I/O functions, etc.

stack growing downwards
Overflowing a kernel stack using BPF program
Overflowing a kernel stack using BPF program

stack growing downwards

system call functions

XFS stack

Memory management functions, Scheduler functions, Block I/O functions, etc.,

BPF attach type functions

BPF program #1 stack

BPF program #2 stack

BPF program #3 stack

BPF program #4 stack

BPF program #33 stack

Helper function stack

helper’s kernel stack
Overflowing a kernel stack using BPF program

stack growing downwards

system call functions

XFS stack

Memory management functions, Scheduler functions, Block I/O functions, etc.,

BPF attach type functions

BPF program #1 stack

BPF program #2 stack

BPF program #3 stack

BPF program #4 stack

BPF program #33 stack

Helper function stack

helper's kernel stack

stack overflow
Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary
BPF verifier assumptions about BPF programs nesting

- Verifier assumes that helper call stack usage is small.
BPF verifier assumptions about BPF programs nesting

- Verifier assumes that helper call stack usage is small.

- The new desire to nest multiple BPF programs is violating verifier’s assumption.
Tracepoints and Kprobe nesting checks for BPF programs

- Tracepoints/ Kprobes doesn’t allow nesting of multiple BPF programs.
- Inside `trace_call_bpf` function kernel checks if there is any active BPF program already executing on the same CPU.
- If this condition is true, the corresponding BPF program will not be executed.
What happens if the nesting checks are not implemented?

- If a BPF program calls a helper function
What happens if the nesting checks are not implemented?

- If a BPF program calls a helper function
- and another BPF program, attached to that helper function, calls the same helper, it can create an endless loop.
If a BPF program calls a helper function and another BPF program, attached to that helper function, calls the same helper, it can create an endless loop. This loop could cause the BPF program to run indefinitely, potentially leading to a system crash due to inheriting the same stack.

What happens if the nesting checks are not implemented?
Limitations of kprobes/ tracepoints approach

- However, when a BPF program is attached to a helper function or a function invoked within a helper function, the tracing events related to these interactions may not be captured.
What about fentry or trampoline nesting checks attachments?

- BPF Trampoline programs call __bpf_prog_enter_recur before executing BPF instructions. In this function, it essentially checks whether the same BPF program is currently executing on the CPU.
Limitations of BPF trampoline approach

- More than one BPF programs can run on a same CPU, which results in using the same stack.

- So by using nesting multiple BPF programs we can overflow the kernel stack.
In this test, an 8KB BPF program stack, akin to the previous one, is created and attached via a kprobe on the `__sys_socket()` function.

Adding another 8KB BPF stack through a bpf trampoline to the `bpf_get_stackid()` helper function or its kernel path leads to an overflow of the x86_64 Linux kernel stack.
Demo Video showing the stack overflow

Note: Running Linux version v.6.5.0 using QEMU with DYNAMIC_TRACING enabled.
Other possible potential attachments used with BPF trampoline

- Attached to hook 1
  - KPROBE Prog
  - Fentry Trampoline Prog
  - Tracepoint

- Attached to hook 2
  - Fentry Trampoline Prog
  - Fentry Trampoline Prog
  - KPROBE Prog
By using BPF programs attached with trampoline without relying on any tail calls and BPF-to-BPF functionality one can overflow the kernel stack.
Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- Summary
Requirements to address the problems we discussed

We came up with the following requirements to mitigate the problems caused by the implicit verifier assumptions about the stack state during kernel runtime

1. Kernel runtime should ensure and be able to accommodate the stack space required by the BPF programs to run,

2. and if there is a situation where it cannot allocate enough space for the BPF program, it should prevent it from running.
Probable Solution to Address P#1 Could be: Stack-Switching

For addressing the issue of stack overflow when attaching a BPF program to an unknown stack state
Probable Solution to Address P#1 Could be: Stack-Switching

For addressing the issue of stack overflow when attaching a BPF program to an unknown stack state

- The kernel can implement a mechanism to switch a BPF program to a new stack based on memory requirements.
Probable Solution to Address P#2 Could be: Stack-Switching and limit nesting

- The stack switching solution can also address the stack overflow issue that occurs with nesting.

- A new stack can be created according to the memory needs of nested programs, and the kernel should enforce a limit on the nesting depth.
Points for Discussion on probable solutions

In these two cases there is a chance that a BPF program might not get executed

1. If we request memory for the stack, there might be a chance that there is not enough space, and the kernel might prevent the BPF program from running.

2. By posing a limit on nesting, a perf BPF program might not get executed.
Open Question: What happens if a BPF program never gets executed

How does this impact critical BPF program extensions designed for purposes like security?

- Examples could be impacts of not running LSM BPF program/ Seccomp filter.
Open Question: Can an orchestration tool solve our problems?

Can BPF orchestrations tools like bpfd alert admins to monitor BPF programs so that we never run into stack overflow problems?
Agenda

- BPF program attachment and its interaction with the stack
- Stack overflow due to BPF program attachment
- Stack overflow due to uncontrolled BPF program nesting
- Discussion on Probable Solutions and Related Questions
- **Summary**
Summary

- It’s important to upheld the verifiers assumptions about stack during kernel runtime.
- Violations of these assumptions leads to stackoverflow issues.
- We showed two such cases in our presentation.
  1. Incorrect assumptions about availability of stack state.
  2. Uncontrolled nesting.
- Finally, we raised discussion points on probable solutions to mitigate these issues in the future and raised open questions.