Guardbee – Error Dectection Report

marcus lai

August 2025

1 Introduction

This report will summarize my work in summer of 2025 on the detection portion of the Guardbee project which aims to provide fault tolerance to the Linux Kernel despite incomplete error propagation. This report is meant to go more into the details and nuances of my work so that someone who's interested in taking over could have a better transition / onboarding experience. For further questions feel free to contact me at marcus41881653@gmail.com.

1.1 Related Prior Works

There are two works that are immediately related to detection with eBPF: PET and Vulshield. PET essentially does exactly what we are doing – error detection with eBPF. Vulshield on the other hand uses kernel modules instead of eBPF for error detection. Using eBPF gives us some static guarantees such as avoiding non-termination and blocking arbitrary memory writes. It also implements a dedicated safe data path, via *ebpf maps*, for data sharing between kernel and user space. However, Vulshield argues that kernel modules enables detection to be used on legacy kernels (ones before or around version 4, early 5 before eBPf was supported) which are still prevalent today. Of course, both approaches enjoy dynamic detector injection so that the kernel is not recompiled.

Note: It's perhaps worth nothing that this recompilation part is not entirely true since we need to use custom BPF functions for some of our templates. If the original kernel doesn't have these helpers we indeed still need to recompile.

Another advantage Vulshield has over PET is in better error coverage. This is an area that we hope to improve on over prior works. Vulshield claims that they have a working automation pipeline that can generate a detection module from an error report. However, this code is not yet public and so I have yet to test it. Their evaluation indicates that they were able to generate detectors for 31/32 samples automatically. PET also has ideas for automation but their pipeline is only semi-automated. Their pipelines leverages DWARF information for address translation (to know where to put eBPF program, converting from sanitized kernel addresses to unsanitized kernel addresses). They demonstrate these techniques on one sample in their repository. Comparatively, Vulshield's approach for automation seems trickier since it also requires some static analysis. Their author is responsive to emails so one of the next steps is definitely to ask for their entire code, and perform some testing. Indeed if their code works (good error cover + good automation) we might be able to use their thing. Or at the very least get some ideas from them.

Safety While eBPF does offer some extra security guarantees over kernel modules, we note that our implementation may not be much safer than Vulshield's approach after all. Specifically for use after free error, PET's implementation of kernel memory sweeping is entirely delegated to a custom BPF helper function that is called from the user space program. The rest of the program contains very simple logic that cannot be problematic. In this way, we are kind of "cheating" by wrapping potentially vulnerable code into custom helpers.

1.1.1 PET Repo

One component of my internship was to go over PET's code. This section covers some notes about their repository. Each error detection sample in the repository has 3 components to make note of. First is the ebpf program itself which is a C file with suffix bpf.c. These are located inside the kernel code (linux-5.15-vulns) in kernel/samples/bpf. Each .bpf.c program will also have an associated userspace .c program to load the ebpf program. The code to reproduce the error is in POCs. Any custom helper functions will be in kernel/trace/bpf_trace.c.

Note that PET changed many things in their kernel, mostly for triggering errors I believe. For more information refer to the Use after free section.

2 Error Template and Samples

Using Error Templates in eBPF programs to detect errors is the main novelty of PET. For more specifics about error templates for each error I refer the reader to the PET and Vulshield papers. This section will go over some new templates and implementation details that are worth nothing.

2.1 Slab OOB (new template)

Slab out of bounds is a little different from regular out of bounds access. Slab out of bounds happens when an accessed object is inside a slab, and the object is being accessed at an offset that is outside the range of the slab. What makes this a bit trickier is when an object is accessed, we don't actually know where the slab object begins (what the slab object address is). So we leverage slab information from the report to inject one eBPF program to detect the error. We notice that the slab allocator that allocates the slab is invariant across runs, and also that the slab size is invariant. We also notice that slabs are aligned to their own size. Thus, to find the beginning of the slab from the address accessed, we perform a mask of the VA to round down to the nearest, smaller, slab-size-aligned address. For example, if the slab size is 1024 (invariant), we would round down the accessed object to the nearest 1024-aligned address by masking.

Then, since the size of the slab is invariant, we know where the bounds of the slabs are. Using this information we can check if the access is indeed outside the bounds of the slab. A similar technique is used for Slab Use after free.

Note: we note that there are ways in the kernel to get the start of the slab. We can implement an eBPF function for it. This is perhaps better than address masking. However, it is potentially unsafe to give user space programs that functionality? Then they can know the slab object starts for any address.

2.2 Stack OOB (new template)

Stack out of bounds is a also little different from regular out of bounds access. Stack out of bounds happens when a reference to a *stack local variable* in a function frame is passed along the control flow to some location. This reference is then accessed outside its fixed bounds, giving the error. We leverage the report for our template, and inject 3 eBPF programs for detection. From the report, we use 2 pieces of information (which we perhaps need to verify). The first is the offset of the local variable from the stack frame of the function that contains it, the second is the size of the local variable. We note that both this information can be gathered within the report but should be checked. In the fl_set_key_cfm sample in the repo, this offset is off.

Once we have the size and offset of the local var in the stack frame, we inject the following programs:

1. A program after function set up. We put this right after all the push instructions in function prologue. We add the address of the local variable into an eBPF map, using the offset and the stack register rsp.

2. A kretprobe at the function itself. This probe is trigger when the function returns right before function epilogue. We use this program to perform map cleanup – we remove the local variable from the map.

3. A program at the access site. We check first 1) whether the object accessed exists in the eBPF map (so we have a stack variable reference) and 2) whether the access goes beyond the bounds of the access size.

2.3 User Memory Access (new template)

The repo has one example of this. The error is just when a user space virtual address is read or written to during kernel execution. So the template is just a check of whether the address read or written to is in the user space range.

2.4 Slab Use after free (new template)

Slab use after free is a variant of use after free. The different between regular and slab UAF is that in slab UAF, the allocation and deallocation sites are of the slabs, and not the object itself. I'm not too sure why they wouldn't have information about the object to generate a uaf report. The template for slab uaf is a small extension to the uaf template (we refer the reader to the use after free section for an overview of the regular uaf template). For slab uaf, we need to perform an extra masking of the accessed address with the size of the object (this technique was explained in Slab OOB).

2.5 Data Race

The problem with data race samples is that syzkaller does not make reproducers for them. PET also doesn't have a sample for data race. Our current reproducer of a data race sample was obtained by reaching out to Vulshield's author. She had written her own reproducer for the error, and she only has the one sample for data race too. With this sample, we demonstrated data race detection with eBPF. Note that a separate, specific QEMU command is needed to run the sample. In the qemu_source.sh script it's implemented by adding the tag -d.

2.6 Kernel Info Leak (new template, no working sample. Had trouble making one, they no longer run)

Kernel info leak is when an uninitized variable is copied from kernel space to user space. The report tells us explicitly which ones the uninit bytes are. For example, it could tell us that bytes 14-15 of the 24 bytes are uninit. Thus, the template is simply: 1) when var (local or heap) is created, record the value of the bytes, 2) at copy site, check if the val is still the same, if yes then kernel info leak. This is basically the same template as uninit val.

The problem I was having is that the kernels no longer run on my qemu on sample 1 and the reproducer also doesn't run anymore on sample 3. A future work is to wrap this up.

2.7 Failure of Samples

In this section we will cover a few reasons why samples might fail, to hopefully save some time when working on new samples. First, the reproducer. Often times syzkaller samples will *not* have reproducers. But our automation step handles that case and rejects the sample so we don't have to worry about that too much. It is also worth nothing that our current automation step to retrieve samples checks if the kernel that failed was compiled with clang (if not it fails) and if it has a valid disk image (if not it fails). These are **not** necessary to work on a sample. If the automation fails, check that there is a 1) .config file and 2) a reproducer C file. If yes, you can download these two, write your own .json file, copy in the Makefile and guardbee.c from another sample, and manually walk through the set up script (make prepare, make config, make build).

One reason Soo decided to check the .config files to see if the crashed kernels were compiled with clang was because she was having some trouble getting some gcc samples to work. Sometimes you can't find a good, working clang sample and you kind of have to get gcc samples to work (especially since we want to save 2024, 2025 samples for evaluation). To recreate the errors, it's best to use the same specific versions of the gcc-compiler, linker, and assembler. But for gcc samples, it's a real hassle to try to match the compiler version, whether locally or in docker (trust me). Note that if you use docker you have to run qemu in docker too. My advice for gcc samples is to not specify the gcc version. Just do CC=gcc. Don't worry if it's gcc-10, gcc-13, ... Just use the local gcc version. That has worked for me. If the error can't be triggered when compiling with another gcc version, then the error shouldn't technically be a "linux kernel error" anyway, but a compiler error.

I also wasted a lot of time compiling kernels and writing ebpf programs for samples that didn't have a working reproducer. So make sure that as soon as you compile the program that you try to run repro.sh. Also that the kernel itself works at all, because sometimes it just crashes right away. Only proceed after confirming >= 3 times that an error report is generated. Following PET's convention I copy and put the local error report directly at the bottom of my eBPF program. It could very much be the case that one reproducer can generate multiple error reports. I had a sample that I ended up discarding that could generate more than 8 reports. This ended becoming frustrating to work with since I couldn't tell why my detector wasn't working and I kept triggering other errors. If you see that you can generate a LOT of reports, I recommend moving on to another sample. If it's just 2, 3 error report versions, it should be fine to proceed.

3 Use After Free

Use after free occurs when an object that is allocated is freed but leaves a reference behind that is then reaccessed through a read or write. This error is a temporal error, as opposed to spacial like a read out of bounds, since at different times of the kernel execution, the same address could refer to different objects (as the virtual space is reallocated). The question is then how can we distinguish between a valid read and invalid read? Indeed, use after free detection is not simply a matter of checking whether what we are accessing is currently allocated, since accessing a new, valid object with a stale reference is still use after free. Currently, there's no clean way of detecting use after free in general. One popular recent approach is to do quarantine + sweep. I did a mini literature review during my internship in another overleaf.

3.1 Quarantine and Sweep

This is a solution that PET and Vulshield defaulted to. It's one that "works" in a broad sense. The method works in the sense that everything is doing what it's supposed to, as intended. But it doesn't work in the sense that there are small, hidden assumptions for the method to work in theory. This method is also an adhoc prevention of use after free instead of detection, which is perhaps doing a bit more than necessary.

The method works like so:

1. At the free site, intercept object to be freed and skip the free. To do this in eBPF we need to use the bpf_override_return helper. This is only possible if we instrument the kernel's free function. This is usually in mm/slub.c or mm/slab_common.c with ALLOW_OVERRIDE_RETURN.

2. At the access site, check if the accessed object is in quarantine. If it is, then we certainly have use after free since the object was meant to be already freed

3. Since the quarantined list of object could build up, we need a way to free it. To know an quarantined object is safe to free, the sweeper scans the kernel memory to check if a reference to the object still exists. We scan every word (8 byte aligned). If yes, it keeps the object in quarantine. Otherwise, it actually frees the object.

Note first that already by step one we need to be appending to kernel code. Thus, we may be taking away the "no-recompilation" benefit of our approach. The way PET and Vulshield has implemented the sweeper is without stop the world pauses. When scanning the kernel, to ensure the references don't propagated to scanned areas, we should really be stopping all kernel execution. Only by freezing the memory and scanning, can we be sure that a reference to the object doesn't exist. This however takes up too much time. The way to ensure the quarantined reference don't propagate remains an open problem in garbage collection.

SwiftSweeper is a technique that solves a partial problem: it prevents the case that an unscanned part of memory is reallocated to the scanned part so it is missed. It does this by essentially by setting all unscanned memory to read only and implementing a page fault handler. They also had to implement their own memory allocator to persist associated meta data. Overall, since this part isn't the main contribution of the project, we decide to use the existing approach of quarantine and sweep without stop the world. I reiterate however that without stop the world we cannot give guarantees that an object doesn't have a reference to it anymore in memory. It is also possible for the object to be accessed by a hidden pointer. For example, what if the reference is stored in a file? Or what if the reference is accessed by adding an offset to another address? These are limitations to the Q&S approach.

3.2 Attempted Optimizations

Since Quarantine and Sweeper is a little problematic in theory and is a solution with potential overhead (we don't want to be scanning the kernel, it takes 20-40 seconds), I spent a good part of my internship thinking about improvements to the approach. To this end I started with a literature review for Use After Free detection. Luckily, in our case, we are tackling a reduced problem. Namely, while existing approaches need to identify use after free for *every* object allocated, we only need to detect use after free for allocations made at a specific allocation site. This gives way to some interesting proof of concepts that didn't entirely work out but are worth noting. To see why an optimization could be possible, consider One-time Allocation to detect use after free. In this approach, each virtual address is allocated only once, and once a virtual address is freed, it will never be allocated again. In this way, it is easy to know if an address we are access is something we've freed already with no ambiguity. However, in this approach, it is observed that the Linux kernel runs out of virtual address space in around 3 days. Thus, they need to consider recycling policies which can be complicated. In our context, since we only need to keep track of a smaller subset of allocated objects, it may be possible that it would take weeks or months to exhaust the VAs, or even be impossible to run out of VAs for the allocation site if it is rarely reached. We did not think through whether this particular optimization is possible to implement.

Another idea we tried is ID tagging. In this approach we would associate each allocation with an identifier (an int) or tag and persist the identifier with the referenced object. Then upon access we check whether the identifiers match, if not we have use after free. In our context, at allocation site, we could add an entry to our ebpf map of the object address and tag. At deallocation site, we would invalidate this tag. Then at access site we first access the referenced object to get its identifier, then check if the object is associated with this identifier. If not we have UAF.

One way of doing this is putting the ID in the higher bits of the object's allocated address. This is possible in newer intel, with a feature called LAM. However, we were unable to get LAM to work on the lab computers. Another way of doing this is persisting this in-line with the object. For example, when we allocate at the allocation site, we can dynamically (with eBPF) change the size of allocation from n to n+4. Then we can write the int to the 4 bytes after the object. This idea was rejected because we believe this would mess up alignment

in the kernel. I also tried to do this, by implementing a similar eBPF helper as PET to change register contents, but I got a nullptr dereference error.

A final idea is to still persist it inline. What I noticed is that in every error report I've seen, all allocated objects are slab allocated. Slab objects typically have sizes that are powers of 2, so in the error reports we often see that objects are given more space than they need (size = 60 would be given a slab of size 64 for example). This is present in all samples I've looked at, and this slab allocated name and size of slab given are invariant in all reports. The idea is then to put the tag somewhere in the slab object. This way, kernel address alignment is not messed with and no allocation size is changed. But the group believes that this method may not work, so we did not end up trying to implement it. This series of blog posts gives helpful insight into the structure of each slabs and what it contains. A future direction for a small optimization is to see if we could put tags in the slab: https://blogs.oracle.com/linux/post/linux-slub-allocator-internals-and-debugging-1.

If we really try to implement this improvement, we note that id tagging and persisting should be done in eBPF since we need to perform a kernel memory write. Namely, we shouldn't let the user manage the IDs and write the IDs. They should simply get an API via BPF helpers.

3.3 Free and Access Race (new thing)

Use after free is sometimes cause by a race between the access and free. Sometimes the access goes first then the object is freed (valid access), while other times the object is freed then accessed (uaf). Sample uafw2 is such an example. The problem with this race is that the eBPF programs themselves race too. Running the sames multiple times demonstrates that at times we have use after free detection, while at times we detect a valid free (meanwhile in both cases the kernel crashes). This is a problem because it means that sometimes uaf could evade detection. Further testing and experimentation is needed to see that this is actually the case.

Perhaps another example is suafr1. In the trace log we see that the allocation, accesses, and freeing of the object that crashes the kernel. However,

Here we also propose a theoretical solution. First, we need a notion of "time". This could be in the form of actual time, CPU ticks, or jiffies. In any case we need to implement some kind of helper that gives us a notion of time (is this safe?). For the solution, we persist an additional ebpf map storing this notion of time, updated during access site. When the free site is triggered, on top of the regular program we also check if there is an access site entry recently. We do this by reading the map. We now obtain the current time and subtract. If the time is very small (below a certain threshold that we set), we should warn that there is a potentially use after free. The hopes of this solution is to catch the case that the access and free are so close that uaf is possible, and help discover cases when the eBPF programs make the use and free go out of order so that the uaf evades detection.

4 eBPF Helper Adder

The process to add eBPF helpers is automated. The script does two things 1) adds the ALLOWINJECTERROR after kfrees and 2) add custom eBPF helpers. I recommend reading through the make commands to find the relevant ones for this section. The usual workflow works by first making two copies all relevant files from the source kernel to a separate src/ and src_clean directory within the sample. The clean version is for restoring the original kernel files. The files that are copied as BPF function header files, bpf_trace.c, and slab files to insert ALLOWINJECTERROR at kfree. To identify kfree, the Python script looks for "void kfree(". It also does a check to see if ALLOWINJECTERROR is already present. If yes, it skips appending the line. Otherwise, it appends the line.

Adding eBPF helpers is trickier. There are 4 things that are needed to add BPF functions 1) function definition, 2) adding the function to the switch case, 3) adding the function signature to the header files, and 4) adding the function description to the list of BPF helper descriptions.

In the header file the script looks for the line that starts with "FN(unspec", we uses this as the "template" of the macro. It assumes the second param is always the index. It then replaces "unspec" with the function name and the index with the index of the last function + 1. The rest of the macro template is invariant. To find where to put the function description, it first finds the comment block by looking for the block with the line: "Start of BPF helper function descriptions:". It then appends the description to the end of it.

Note if a line with both "FN" and the helper function name is found in the file, the script assumes the helper function already exists, so skips adding it. If the function name is anywhere in the src file, we also assume the source code is already in bpf_trace.c so we skip the helper function. These skips are independent.

To know we are in the switch case, the following check is performed:

```
if (("struct bpf_func_proto *" in line
    and "bpf_tracing_func_proto" in line
    and "enum bpf_func_id func_id" in line) or
```

```
    ("struct bpf_func_proto *" in line
    and "bpf_tracing_func_proto" in tempfile[index+1]
    and "enum bpf_func_id func_id" in tempfile[index+1])):
```

This looks for

```
static const struct bpf_func_proto *
bpf_tracing_func_proto(enum bpf_func_id func_id, const struct bpf_prog *prog)
```

and

```
... struct bpf_func_proto * bpf_tracing_func_proto(enum bpf_func_id func_id...
```

which could both be the switch case. It then looks for the "default" case and adds the helper function cases right before it. Also, it adds the function implementation code itself right before the switch block.