Implement a Memory Checkpointing feature using eBPF

1. Introduction

Extended Berkeley Packet Filter (eBPF) revolutionizes kernel technology by empowering developers to inject custom code dynamically and altering kernel behavior. This breakthrough enables high-performance networking, observability, and security tools without modifying applications. Notable eBPF features include system-wide performance tracing, efficient networking, and the detection/prevention of malicious activity.

A "tracepoint probe" refer to a probe or hook placed at a tracepoint in software code. Tracepoints are predefined points in the code where specific events or actions are logged or monitored.

eBPF maps are a generic data structure for storing different types of data, which allows for data sharing between eBPF kernel programs. These maps can help programs store and retrieve information based on a variety of data structures.

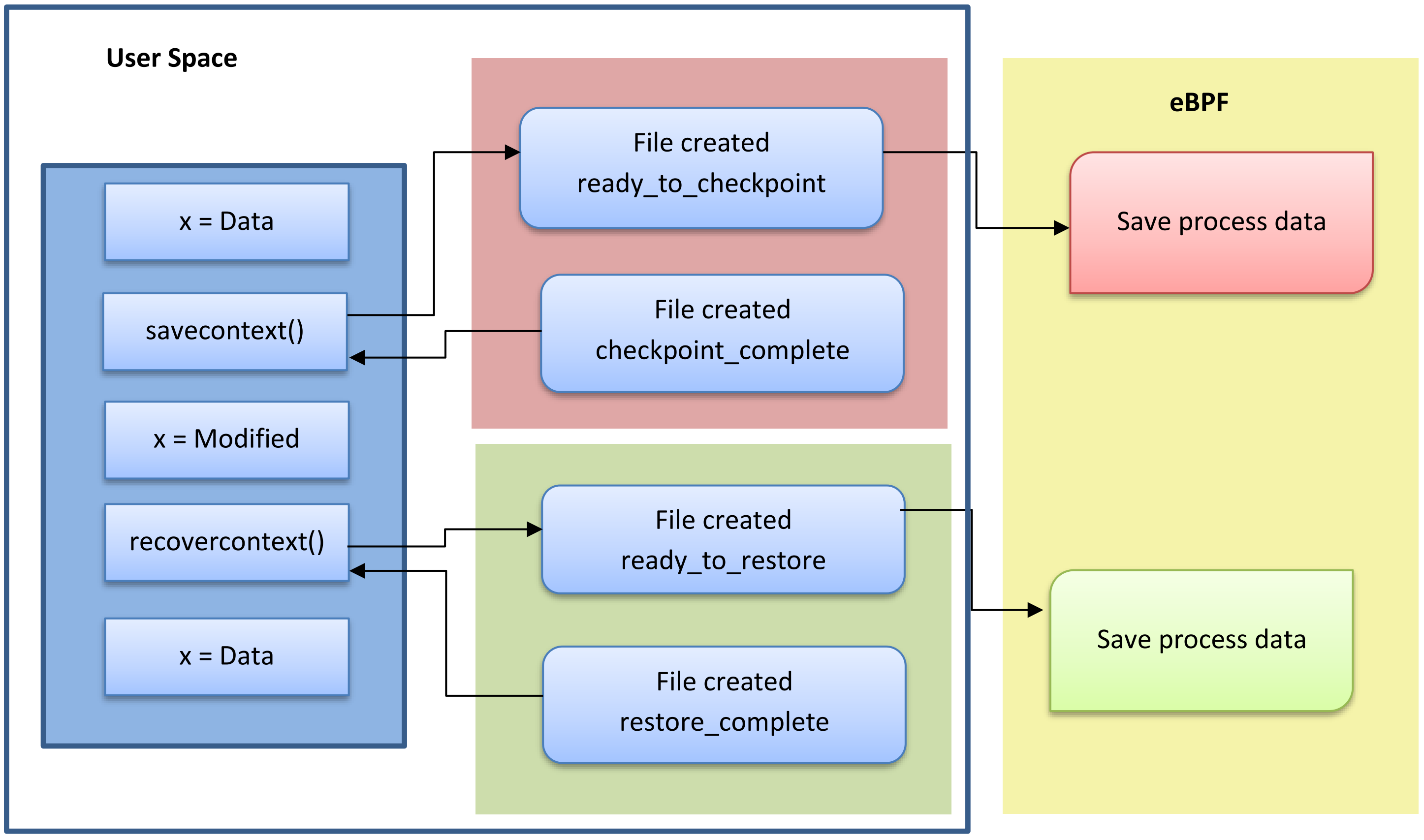
2. Machine Specifications

The system that we are working on has following specifications:

CPU	i5-9300H
Memory	8 GB
OS	Mint 20.03 Kernel 5.5

Table 1.1: Machine Specifications

3. Working Diagram


User Space
x = Data
savecontext()
x = Modified
recovercontext()
x = Data
File created ready_to_checkpoint
File created checkpoint_complete
File created ready_to_restore
File created restore_complete
eBPF
Save process data
Save process data

4. Preliminaries

We have established three distinct bpf hash maps, each serving specific functions within the context of the program. Following are the function of the hash maps:

- One BPF hash map, denoted as "data_map," serves as the repository for the actual process data.
- The second BPF hash map, referred to as "store_map," is responsible for storing metadata during the data storage process.
- The third BPF hash map, termed "restore_map," handles metadata during the data restoration process.

Additionally, an auxiliary BPF array, named "copy_map," has been implemented to facilitate the seamless transfer of data from the process memory to the hash maps.

5. Implementation

Storing Process Memory Data into Map:

To capture process memory data and store it into a map, we've set up a system that kicks into action at specific points during system calls. First off, we've established a function called TRACEPOINT_PROBE() at the sys_enter_openat syscall. When a file named "ready_to_checkpoint" is created by the "savecontext()" function, we identify the process ID of that particular process and retrieve its task_struct. From there, we extract the mm_struct, which contains the vm_area_struct. Now we have stored the information about vma into store_map, and tracepoint will return from there.

Additionally, we've defined another TRACEPOINT_PROBE() at the sys_enter_access syscall. In the "savecontext()" function, we keep trying to access the "/tmp/checkpoint_complete" file until it's created. This file serves as a signal indicating that all the anonymous memory of the process has been copied into our "data_map." Each attempt to access "checkpoint_complete" triggers the TRACEPOINT_PROBE() function. At each trigger, we copy one page of process memory into "data_map" with the assistance of metadata stored in "store_map." This process continues until the entire anonymous memory of the process is copied into "data_map," upon which the "checkpoint_complete" file is created.

Restoring Process Memory Data from Map:

To restore process memory data from the map, we've devised a similar mechanism. We've set up a TRACEPOINT_PROBE() at the sys_enter_openat syscall, which activates when a file named "ready_to_restore" is created by the "recovercontext()" function. Similar to the storing process, we identify the process ID, retrieve the task_struct, and obtain the mm_struct to access the vm_area_struct. This information is stored in our "restore_map," and the TRACEPOINT_PROBE() function concludes its execution.

In the "recovercontext()" function, we keep trying to access the "/tmp/restore_complete" file until it exists. This file signals the completion of copying process memory data from "data_map" back into the process memory. Similar to the storing process, each attempt to access "restore_complete" triggers the TRACEPOINT_PROBE() function. At each trigger, we copy one page from "data_map" into process memory using metadata from "restore_map" and update the process's metadata in "restore_map." This process continues until the entire process's anonymous memory is copied back into the process memory. Afterward, the "restore_complete" file is created, and the metadata in "restore_map" is deleted to ensure the "recovercontext()" function does not call the access syscall again.

6. Challenges overcame

Throughout the project, several challenges are encountered with the following solutions:

- Writing the script in BCC posed challenges in accessing the "vma_area iterator," a crucial component. Typically, in BPF, this access is straightforward, but it required an older kernel version (5.5) to function properly. Instead of converting all our code into BPF, we decided to downgrade the kernel to version 5.5 to resolve this issue.

- Initially, we tried to copy the entire vma_area region into the data map at once, but it didn't work out. So, we found a workaround: we kept calling a function until the copying was done. This meant we had to copy one page at a time into the map and restore one map at a time, but it got the job done.

- With the stack size limited to 512 B, moving data between process memory and the map was a slow process. To speed things up, we introduced a BPF array. This acted as a go-between, allowing us to first move data from process memory to the array, then from the array to the BPF map, and vice versa. It made the whole process a lot smoother, despite the small stack size limitation.