

SysArmor

ebpf based Intrusion Detection and Prevention Platform

Liam Wisehart

Shankaran Gnanashanmugam

Agenda

- 01 Problem Statement
- 02 SysArmor
- 03 Detection, Prevention & Response
- 04 Use Cases
- 05 Challenges

PROBLEM STATEMENT

In Meta's world, where billions of people connect every day, the risk of cyber threats is very high. Without a strong Intrusion Detection and Prevention System (IDPS), Meta's network and hosts could be attacked, leading to unauthorized access, data leaks, and system breakdowns.

In the past, we used SELinux and OSQuery for detection and enforcement. Due to performance issues we couldn't enable SELinux on critical prod environments and OSQuery was primarily snapshot based system.

The main problem is detecting and stopping advanced cyber threats in real time, considering the size and complexity of Meta's operations with low performance hit.

SYSARMOR

SysArmor is our Intrusion Detection Prevention System (IDPS) that provides enhanced system and network security detection events for critical infrastructure. SysArmor is also used to enforce mandatory access controls on the host. SysArmor uses various ebpf hooks to detect system/network events and LSM BPF for enforcement. The main differentiator between sysarmor and similar ebpf based security tools is that sysarmor evaluates its rules inside the ebpf program, instead of dispatching events to userspace logic

SYSARMOR

- [MITRE](#) ATT&CK framework for coverage
- 50+ bpf hooks
- Rule execution in ebpf code.
- Highly configurable
- Fast execution - optimized for high performance
- Least privilege – access is restricted by default
- Mandatory Access Controls using BPF LSM

DETECTION COVERAGE

1. Hardware
2. Networking
3. System calls
4. Privilege Escalation
5. Filesystem Modifications
6. Access Control
7. Process/Task Management
8. Resource Usage
9. Commodity Malware
 - a. Shellcode execution
 - b. Lateral movement/exploration

MANDATORY ACCESS CONTROLS

1. Network Access (Bind, Connect, Accept)
2. File operations
3. Execution of untrusted binaries
4. Unauthorized privilege escalations
5. Loading unauthorized kernel modules and ebpf programs

RESPONSE/REMEDIATION

1. File Acquisition
2. Log Acquisition
3. Actions on Endpoints
4. Network Endpoint Isolation
5. Targeted Network Connection Blocks
6. Quarantine malicious process and Binary
7. Remote Memory Collection
8. Stop process
9. Block process network access

USE CASE: MANDATORY ACCESS CONTROL FOR MOUNT/FILE ACCESS

Problem Statement: Access control for data disks that store sensitive data.

Requirements:

- Access is restricted by default.
- Only allowed processes will have access to these disks.
- Any tools/processes invoked by allowed processes will also inherit the access policy

Option to Configure Policies

- [Kernel device number](#) (major/minor)
- File/Directory Path

Solution:

- lsm/file_open

USE CASE: MANDATORY ACCESS CONTROL FOR MOUNT/FILE ACCESS

1. lsm/file_open
2. lsm/task_alloc
3. lsm/bprm_creds_for_exec
4. iter/task

USE CASE: MANDATORY ACCESS CONTROL FOR MOUNT/FILE ACCESS

- iter/task (Task Iterator)
 - Records executable information in task local storage for allowed processes.
- lsm/task_alloc
 - When Parent task has any task local storage then copy it to the new task.
- lsm/bprm_creds_for_exec
 - Records executable information in task local storage for allowed processes.
 - When Parent task has any task local storage then copy it to the new task.
- lsm/file_open
 - Quick exit for uninterested char devices operations
 - If the path matches the rules, default block unless the following rules pass
 - check if the process is allowed
 - Task storage has exe info and it is allowed.

USE CASE: MANDATORY ACCESS CONTROL FOR MOUNT/FILE ACCESS

```
SEC("iter/task")
int syar_lsm_fo_task_iter(struct bpf_iter__task* ctx) {
    struct task_struct* task = ctx->task;
    // code for task validation
    struct path* fpath = get_path_from_task(task);
    // code for fpath validation
    struct sysarmor_file_path* syar_path = get_file_path(fpath);
    // code for syar_path validation.
    u8* val = (u8*)bpf_map_lookup_elem(&syar_lsm_exe_policy, syar_path);
    // code for val validation.
    struct sysarmor_file_path* fo_val = bpf_task_storage_get(
        &syar_lsm_fo_task_allowed, task, 0, BPF_LOCAL_STORAGE_GET_F_CREATE);
    if (fo_val) {
        copy_file_path(syar_path, fo_val);
    }
    return 0;
}
```

```
// called for fork/clone syscalls
SEC("lsm/task_alloc")
int BPF_PROG(
    syar_lsm_task_alloc,
    struct task_struct* new_task,
    __u64 clone_flags) {
    struct task_struct* old_task = bpf_get_current_task_btf();
    struct sysarmor_file_path* parent_task_exe_path =
        bpf_task_storage_get(&syar_lsm_fo_task_allowed, old_task, 0, 0);
    // Parent process doesn't have a task storage. so we can ignore this
    // allocation.
    if (!parent_task_exe_path) {
        return 0;
    }
    // Parent process is in the allowed list. The child process gets all
    // the properties so we need to copy the parent's task storage.
    struct sysarmor_file_path* val = bpf_task_storage_get(
        &syar_lsm_fo_task_allowed, new_task, 0, BPF_LOCAL_STORAGE_GET_F_CREATE);
    // code to validate val
    copy_file_path(parent_task_exe_path, val);
    return 0;
}
```

ANATOMY OF A SYSARMOR DETECTION

- One or more ebpf probes that send detected events to userspace
- ebpf programs contain logic to filter allowed events in kernel space which dramatically improves performance. Some filtering criteria we have used:
 - Process name (ex “bash”)
 - Process binary file path (ex “/usr/bin/bash”)
 - Container user name (ex “liamwisehart”)
 - IP address
 - Port
 - Uid/gid
- Events are enriched with process and container info before being sent to userspace
- In some detections we perform additional userspace filtering or processing
- Events are then logged to various tables

Evaluating eBPF performance

- Operate on tiers with low tolerance for performance regression
- Meta has tooling to automatically track eBPF memory and CPU utilization (bpftax)
- In addition, we track the start and end time of a eBPF program, and record the elapsed time in a map when a eBPF program exits. We track the number of executions, as well as average + total execution time. We also track the time with/without producing an event since creating the event is frequently the most expensive step.
- Due to the fact that SysArmor does most of its processing in eBPF, CPU utilization of the daemon is negligible unless something unusual is occurring on the host.

USE CASE: Tracking TCP network access

- Bind()
 - Hook: kprobe/security_socket_open
 - LSM kernel flag not available on all hosts yet so we use a kprobe
 - Allowlist by service process name/binary path and port
- Connect()
 - Hook: cgroup/connect (will change to security_socket_connect)
 - Allowlist certain internal IP ranges and service process names/binary paths
- Accept()
 - Hook: kretprobe/inet_csk_accept
 - Not used on all prod tiers due to excessive resource consumption
 - Used to track traffic between development servers and other tiers

Due to the frequency of events most of these detections employ ebpf based deduplication (a map is used to track duplicate events)

USE CASE: Tracking UDP network access

- Sendmsg()
 - Hook: kprobe/udp_sendmsg
- Recvmsg()
 - Hook: kprobe/udp_recvmsg

Similar to TCP, we use a map to store recently sent destinations (or received senders) and only create events for new destinations/senders

Currently only used on certain tiers due to the number of events

USE CASE: Bash Commands

- Uretprobe in bash readline() function. Creates an event any time a bash command is entered in an interactive session.
- Has good performance compared to other methods of tracking interactive sessions.
- By exploring the process tree, allows identification of shellcode execution (ie bash has an unusual parent process).
- In order to track all bash commands on a host, we need to locate all copies of the bash binary. Each container contains its own copy of bash. SysArmor handles this in two ways
 - Searching in certain directories for bash binaries (the mount points for the containers running on the host)
 - Using a ebpf iterator to iterate all tasks on the system and identifying tasks with upid=1. These are the init processes for every container on the host. Then we can use /proc/<pid>/root/bin/bash to attach the uprobe.

USE CASE: Tagging Events with SSH Information

- For many types of event we would like to know the user that initiated the event, or if the event was initiated by a process not owned by ssh.
- We hook the `setuid()` system call. Ssh calls this system call once, after forking and starting to initiate a new session. When a new session is started, we record the pid.
 - We used to use a uprobe, but there was difficulty in coordinating the release of a binary with the right symbols.
- Ssh logs the new session details to syslog which includes the pid of the new session. We capture that log message and match it with the pid we recorded
- We maintain a map of pid -> process metadata, and store the SSH information there
 - Map entries updated on fork, exec, and exit

USE CASE: Python Scripts

- There is very poor visibility into what scripts are running on a host
- We place a uprobe into each copy of libpython on the host, hooking the function `PyRun_AnyFileExFlags`
 - This fires every time a script is run and provides the file name
- Aggregating the data tells us what scripts ran and how often
- Using our process info map (pid -> metadata) we can tag a process with the script that is running when the script is run. Then when the script does something detected in another detection (like bind to a port) we can use the metadata associated with that pid to identify the script. So instead of just seeing the process name “python”, we can get the actual responsible script.

USE CASE: Tracking file open outside of a container mount

- Many container escapes involve abusing shared filesystems like /proc or mounting the root filesystem in a container. By creating allowlist of all of the files and directories normally accessed by a container we can detect abnormal behavior.
- We track container creation with a kprobe on `create_new_namespaces`. This probe sends an event to userspace logic that queries the container engine (tupperware) for the container user name. This string is then associated with the mount namespace id of the container in a ebpf hashmap.
- The main logic is in a kprobe on `security_file_open`. When a file is opened, the processes' mount namespace id is used to lookup the container user name, which is then used to lookup a map of all the directories and files the container is supposed to be able to access.

CHALLENGE: Identifying Services

- It can be very difficult to tie a process or a container to an actual service identity for the purpose of creating a policy.
- Scripts are especially challenging (python, bash, ruby, etc)
- For processes:
 - Process name – can be changed easily
 - Process command – not always useful
 - Binary path - difficult to process in ebpf
 - Binary signature
- For containers:
 - Cgroup name - frequently contains randomness difficult to handle with ebpf
 - Namespace id - requires a lot of work to tie to container image/user name

CHALLENGE: Handling files and directories

- We would like to handle filesystem based policies in ebpf and avoid expensive userspace processing
 - This also enables us to use LSM to enforce the policy
- Extracting the path alone is expensive, processing them even more so
- Paths can be relative to some root, and it can be difficult to determine if a file is the file you are looking for in the constrained environment of ebpf. Some problems:
 - If a path contains randomness (like a uuid) is difficult to run a comparison
 - Exploring directory structures is difficult
 - Tying the root of a filesystem back to its container is difficult
- Limitations
 - Lack of regex/wildcarding
 - Limited instructions
 - Lack of data structures for representing paths effectively (we built our own)

CHALLENGE: Looking up file paths without using bpf_d_path

- Many ebpf hooks do not have access to bpf_d_path
- Sysarmor has logic to lookup a struct dentry that can be called from any hook.
- Each file name in the path is stored in reverse order. So /home/liamwisehart/foo becomes “foo\0liamwisehart\0home\0”. This has two advantages:
 - Since the first dentry is ‘foo’ this is the natural order of the data and it avoids a 2nd loop
 - This can be directly inserted into a ebpf hashmap as a unique key. When trying to match whether a file is in a directory. Sysarmor slides down the series of strings, trying to lookup each subsequent directory in a ebpf hashmap of allowed directories.
- We store the offset of each string in the vector for processing
 - Relative paths can be expressed by simply truncating the vector

Acknowledgements

- Kernel Team at Meta
- Network Platform Security Services Team



RESPONSE: Blocking a processes network access

- We want to be able to block the network access of a suspicious process, while leaving the rest of the system running. To do this we have several steps:
 - Run a ebpf task iterator to get the pid of each process we want to block, by process name or some other criteria
 - Run a ebpf task file iterator to iterate the files open by each pid. For sockets we record the source and destination ip address, port, and protocol used.
 - Run a cgroup_skb/egress and ingress program that drops packets that match the socket information we recorded
 - Iterators are rerun periodically to identify new sockets

USE CASE: Tracking Resource Consumption

- Goal is to prevent crashes due to fd or memory leaks
- Periodically run a task iter that checks the highest allocated fd and the heap size
- Configured with per-service profiles and a default threshold