SYSTEM CALL SANDBOXING

Jakub Jarosław Patałuch - J.J.Pataluch@student.tudelft.nl

Responsible Professor: Alex Voulimeneas

TU Delft

INTRODUCTION

System call sandboxing is a vital security technique that restricts applications to a **minimal** set of system calls, reducing the potential attack surface when compromised. Although significant advancements have been made in static system call policy generation, there is a pressing need for dynamic techniques that **adapt** to different execution **phases** of various applications like servers, media players, databases, etc. This project aims to come up with execution phases and analyze essential system calls for **PWD** and **NGINX**, then compare that with **auto-generated** policies by tools like **sysfilter** and **chestnut's Binalyzer**.

RESEARCH QUESTION

The main research questions are:

- What are the **essential** system calls required for the **correct operation** of selected applications (PWD and NGINX) across various execution **phases**?
- How do **static** system call filtering techniques compare to **dynamic** techniques in terms of **accuracy, security, and performance**?
- Can **dynamic** system call sandboxing adjust more effectively to the operational **context** of an application, thereby providing enhanced security without impacting system **performance**?

METHODOLOGY

- Devise multiple execution scenarios that may cause different syscall to get executed
- Run strace for the following scenarios
- Analyze syscalls triggered through execution phase, plot them and try to empirically find the execution phases
- Apply system call filters for the designed execution phases and gather results
- Gather results for the static analysis [1,4]
- Compare results

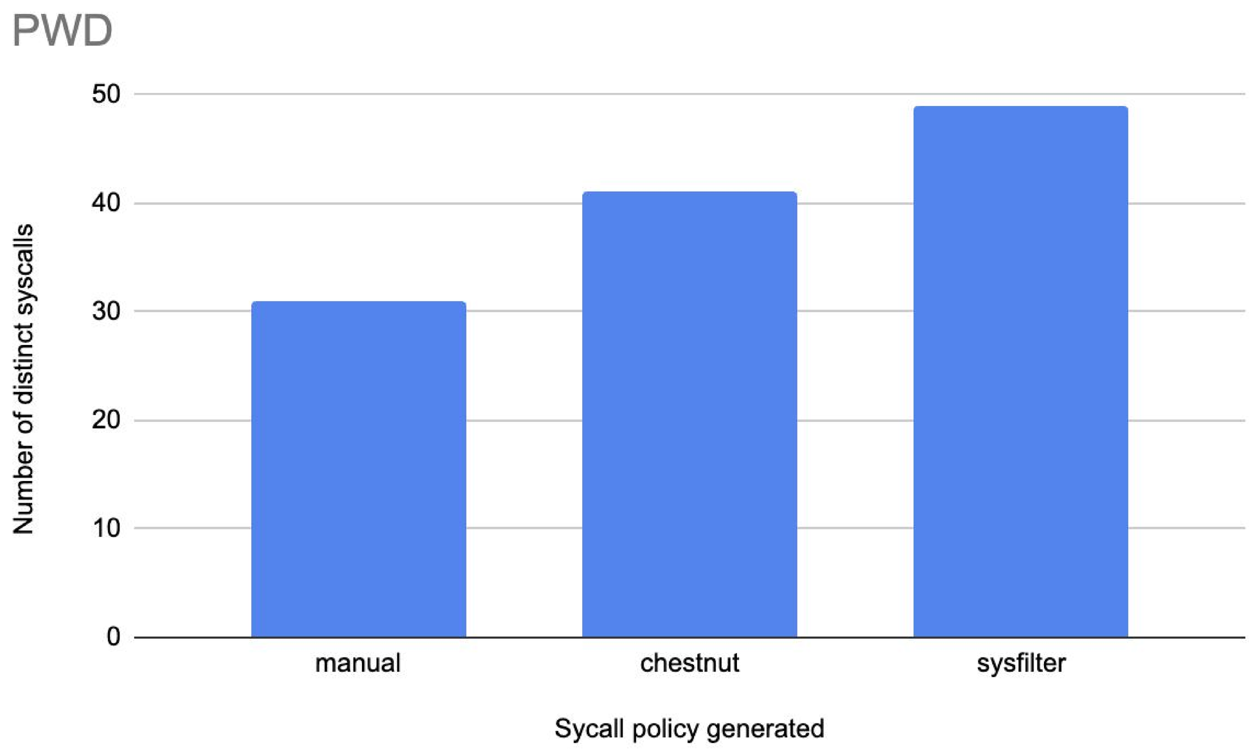
EXPERIMENT SETUP

- Design at least 6 different execution scenarios to cover possibly all syscalls that may be triggered for PWD and most common operational scenario for NGINX
- Unbiased experiment execution environment (Ubuntu 18.04LTS docker container)
- Clean runtime environment for each run (no interference with previous runs)
- No application crashes/critical errors are allowed (system call filter is invalid then)
- Single worker nginx for ease of strace output analysis

RESULTS

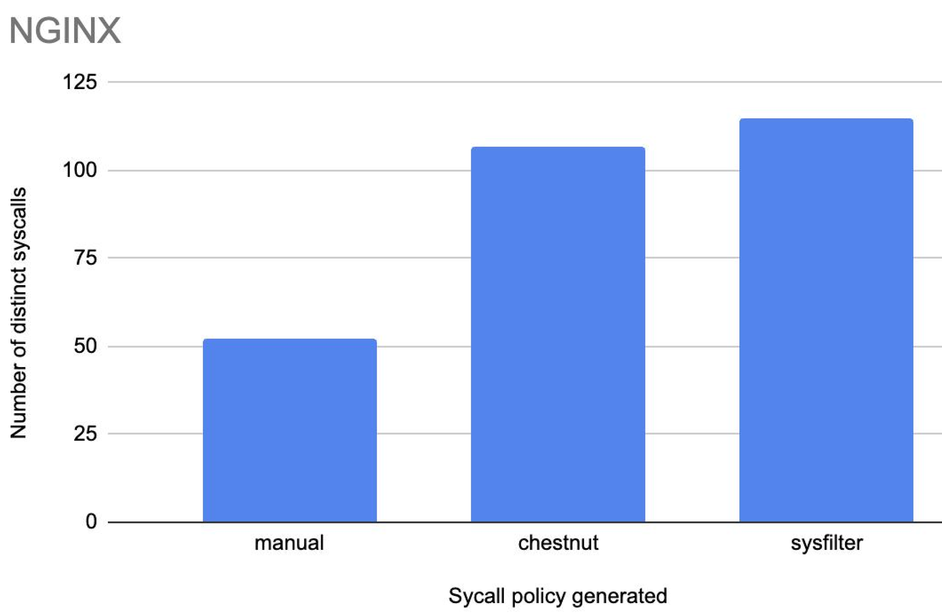
We managed to come up with following execution phases and amount of syscalls per policy for:

- PWD:



Execution Phase	System Calls Involved
Execution Initiation	execve, brk
Loading Shared Libraries	access, openat, fstat, mmap, close
Memory and Environment Setup	arch_prctl, mprotect, munmap, brk
Reading Configuration	getcwd, stat
Output Handling	fstat, write, close
Error Handling	write
Process Termination	close, exit_group

- NGINX:



Execution Phase	System Calls Involved
Execution Initiation	execve, brk, arch_prctl
Loading Shared Libraries	access, openat, read, fstat, mmap, close, mprotect
Reading Configuration Files	openat, fstat, pread64
Initializing Logging	openat, fstat, futex
Setting Up Worker Processes	clone, set_robust_list, getpid, close, setsid, umask, dup2, rt_sigprocmask, socketpair, ioctl, fcntl
Opening Necessary Files and Directories	openat, fstat, pread64, getdents64, close
Creating and Configuring Sockets	socket, setsockopt, ioctl, bind, listen
Setting Up Signal Handlers	rt_sigaction
Worker Process Initialization	setgid, setuid, prctl, rt_sigprocmask
Entering Event Loop	epoll_create, eventfd2, epoll_ctl, socketpair
Accepting and Serving Requests	epoll_wait, gettimeofday, accept4, recvfrom, stat, openat, fstat, writev, write, close, setsockopt

CONCLUSIONS

- **Tool Limitations Identified**: Comparison of manual and automated tools (sysfilter and chestnut) shows gaps on automated tools side like clear overapproximation and computational overhead.
- **Superior Manual Analysis**: Manual methods, though not scalable, offer more nuanced security policies (Especially phase-specific filtering), highlighting the need for advanced AI in automation.
- **Customization is Key**: Security measures must be adaptable to specific application behaviors; **a one-size-fits-all approach is ineffective.**

FUTURE WORK

- **Develop Advanced Hybrid Tools:** Create tools that combine static analysis and dynamic monitoring, enabling real-time adaptations to emerging security threats.
- **Reduce Computational Overhead:** Investigate methods to minimize latency and resource consumption in environments implementing dynamic sandboxing.
- **ML Integration:** Explore the use of AI and machine learning to automate and improve the precision of system call policy generation based on real-time data.
- **Cross-Platform Compatibility:** Develop sandboxing solutions that are effective across different operating systems and hardware architectures.
- **Longitudinal Studies:** Conduct long-term studies to evaluate the durability and effectiveness of hybrid sandboxing approaches under continuous operation.

REFERENCES

[1] Canella et al. "Automating Seccomp Filter Generation for Linux Applications". In CCSW 2021.
[2] Ghavamnia et al. "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction". In RAID 2020.
[3] Ghavamnia et al. "Temporal System Call Specialization for Attack Surface Reduction". In USENIX Security 2020.
[4] DeMarinis et al. "sysfilter: Automated System Call Filtering for Commodity Software". In RAID 2020.