System Call Sandboxing: Enhancing Security Through Analysis

Comparing Dynamic and Static System Call Analysis for Diff and SSH

Author
Duco de bruin
L.L.J.deBruin@student.tudelft.nl
Responsible Professor
Alexios Voulimeneas

01 Background

- Sandboxing for Software Security:
  - restrict access to system resources, limiting attack surface [1]
  - limit system calls (os to kernel)
- Static system call analysis:
  - inspecting the code and/or binary [2,3,4,5]
- Dynamic system call analysis:
  - evaluate execution upon realistic inputs [3]
- Execution phase seperation
  - Restrict system calls based on execution phase [6]
  - block unnecessary calls after initialization

02 Research Questions

- **How do dynamic and static analysis methods compare in identifying required system calls for applications, and how can execution phase separation further refine the system call sets?**
- Which system calls are required for diff and SSH, according to a custom dynamic approach and the static approach Sysfilter [5].
- Which phases can be identified in SSH and what effect does seperating the system calls for each phase have?

03 Methodology

Experiments will compare the resulting sets of the methods for Diff and SSH on x86-64.

- **Dynamic**
  1. Run Strace [7] on Diff and SSH
  2. Parse output into a set of syscalls with Sparse [8]
  3. Repeat steps 1 & 2 on multiple inputs (using script)
  4. Merge results using our tool [9]
- **Static**
  1. Run Sysfilter [5] on the binaries of Diff and SSH
  2. Parse syscall numbers to syscall names with our tool, using syscall list from Strace [10]
- **Execution phase seperation**
  1. Gather traces of SSH-client using Strace
  2. Stop execution before login, after login, after using connection
  3. Obtain syscalls per phase using Sparse

04 Results: Diff

- **Dynamic**: 24 found, 7% of all
- **Static** 59 found, 18% of all
- Dynamic is mostly subset of Static
  - Only new syscalls for Thread managements & access rights
- Static included many extra calls
  - Obviously wrong: Networking
  - possibly right: Signalling

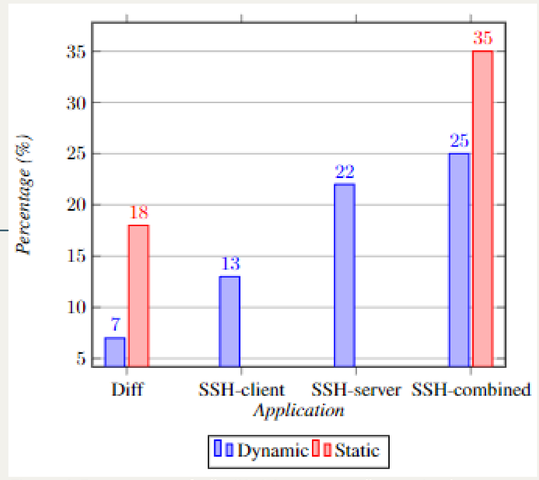

Figure 1: Percentage of all x86-64 system calls required per application, per analysis method.

05 Results: SSH

- **Dynamic** 83 found, 25% of all
- **Static** 111 found, 35% of all
- Possible to split dynamic analysis of SSH into 2 parts: **client** and **server**.
- Dynamic and static big differences
- In dynamic but not in static:
  - Process/thread management
  - I/O operations
- In static but not in dynamic:
  - Likely wrong: mkdir, gettimeofday
  - Possibly right: exit, listen

06 Results: Execution Phase Seperation

- 4 phases:
  - Initialization
  - Authentication
  - Session establishment
  - Working
- Initialization and authentication closely bound, so considered together.

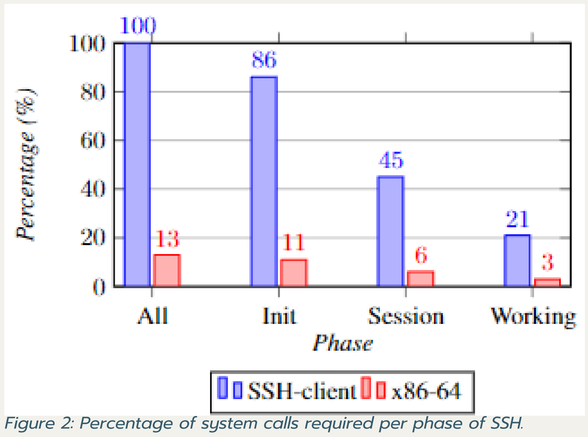

Figure 2: Percentage of system calls required per phase of SSH. Compared to either the number of required system calls for SSH-client or to all available in x86-64.

07 Limitations

- **Dynamic**
  - Hard to cover all execution paths
  - Requires manual labour and insights
  - Harder for background applications
- **Static**
  - Unused syscalls also found
  - Some larger applications hard to analyse, due to missing debug symbols
- **Solution:**
  - Combining both methods, as done with Chestnut [2]
  - p1 = static, p2 = dynamic

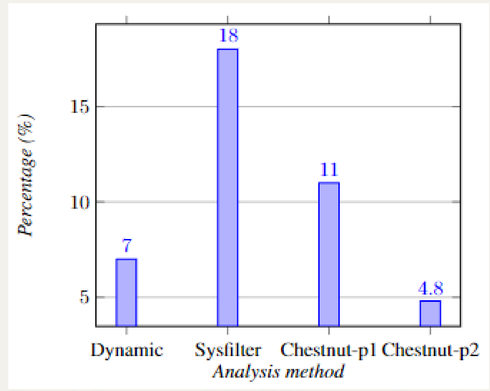

Figure 3: Percentage of all x86-64 system calls required for diff per analysis method

08 Conclusions

- Static and dynamic analysis both eliminate more than 60% of unnecessary system calls for both diff and SSH
- Static analysis covers every scenario but has large overhead
- Dynamic analysis might miss some edge cases but provides the tightest set
- So, there is a trade-off between maintaining full functionality and achieving maximum attack surface reduction
- A hybrid approach like Chestnut [2] mitigates the drawback of both method
- Execution phase seperation works well for enhancing syscall reduction for large applications
  - Working phase SSH only requires 20% of what the entire appliation would normally require

09 Future work

- Using fuzzing to get all paths covered [2]
- Comparing analysis on more applications
- Automating analysis & filter generation for custom method
- Automating phase seperation
- Investigate overhead of Sysfilter and potentially reduce it

References
[1] V. Prevelakis and D. Spinellis, "Sandboxing applications," in 2001 USENIX Annual Technical Conference (USENIX ATC 01), (Boston, MA), USENIX Association, June 2001.
[2] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in CCSW 2021, 2021
[3] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in RAID 2020, 2020
[4] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in USENIX Security 2020, 2020.
[5] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in RAID 2020, 2020.
[6] Zhang. et al., "Building dynamic system call sandbox with partial order analysis," in Proceedings of the ACM on Programming Languages, vol. 7, 2023
[7] M. Kerrisk, "man7." [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html. (accessed Apr. 28, 2024).
[8] D. de Bruin, "sparse." GitHub. [Online]. Available: https://github.com/DucodB/sparse. (accessed Apr. 28, 2024).
[9] D. de Bruin, "unique-file-merge." GitHub. [Online]. Available: https://github.com/DucodB/unique-file-merge. (accessed May 4, 2024).
[10] D. de Bruin, "syscall-numbertoname." GitHub. [Online]. Available: https://github.com/DucodB/syscall-numbertoname. (accessed May 27, 2024).