System Call Sandboxing

Comparing static and dynamic analysis and filter generation

Author: Petr Khartskhaev
(P.Khartskhaev@student.tudelft.nl)

TU Delft

Supervisor: Alexios Voulimeneas
(A.Voulimeneas@tudelft.nl)

Introduction

- **System calls (syscalls)** provide an interface for user-mode programs to request the operating system's kernel to execute an operation [1].
- The kernel has maximum permissions in the OS (ring 0) [1], so it is crucial potential malicious syscalls are prevented.
- Solution? **Use a filter**, a list marking allowed syscalls for each program.
- But **how** do we get that list?
  - **Statically**: Read the binary and construct list of syscalls used by the program [2].
    - Easier to execute
    - May be less accurate (more detected syscalls, less precise arguments – especially from library functions)
  - **Dynamically**: Run the program, see all possible scenarios, construct the list from trace.
    - Slower and more difficult to set up
    - Should be accurate, but may disallow valid syscalls if scenarios are non-comprehensive

Research Questions

1) How does the dynamic approach perform on programs such as *ls* and *Openbox*

2) How do outputs of different static approaches compare to an output from a dynamically aquired filter?

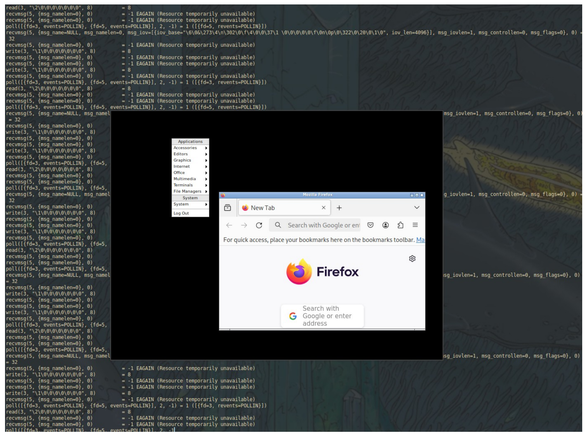
Dynamic Analysis

- Parse the file produced by the tracing tool into separate syscalls with arguments
- Combine the syscalls into a dictionary with a set of all used arguments
- Parse the arguments into addresses, integers, strings and structures
- Combine the arguments within each syscall to form a list of possible arguments

Process

1) Dynamically analysing small programs
   - Assembly *"Hello World!"* program without the C library
   - Dynamically linked C *"Hello World!"* program using the C library
   - The *ls* utility
2) Dynamically analysing a bigger program
   - *Openbox*
   - Two parts – startup and main loop
3) Statically analysing small programs
4) Statically analysing a bigger program

- For dynamic analysis:
  - Construct scenarios for each program such that (ideally) all execution branches are covered
  - Execute scenarios with *strace* running
  - Pass trace outputs to script
  - Run programs in *firejail* with syscall whitelist
- For static analysis:
  - Run *Chestnut*'s *Binalyzer* [2] tool on each program, then use the list in firejail
  - Run each program in *Callander* [3]
  - Compare produced lists



Results

Tool		"Hello World!" (Assembly)	"Hello World!" (C)	*ls*	Openbox
My Solution	Total	2	18	39	59
	Works	Yes	Yes	Yes	No[2]
Callander	Total	5	42	77	149
	Extra	3	30	46	94
	Missing	0	6	8	4
	Works	Yes	Yes	Yes	No[3]
Chestnut[1]	Total	2	154	266	287
	Extra	0	138	234	233
	Missing	0	2	7	5
	Works	Yes	Yes	No[4]	No

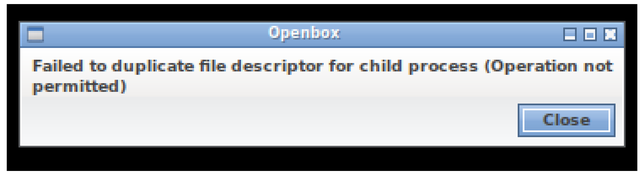
1) The code was modified to stop errors

2) Generally works, but cannot launch new applications

3) Launches, but works cannot interact with applications without errors

4) Lists all files and directories but no properties

2)



3)


```
How are you gentlemen? All your base are belong to us. (Openbox received signal 31)
How are you gentlemen? All your base are belong to us. (Openbox received signal 31)
How are you gentlemen? All your base are belong to us. (Openbox received signal 31)
Openbox-Message: Unable to find a valid menu file "/var/lib/openbox/debian-menu.xml"
How are you gentlemen? All your base are belong to us. (Openbox received signal 31)
```

4)

```
d????????? ? ? ? ?            ? .
d????????? ? ? ? ?            ? ..
d????????? ? ? ? ?            ? binalyzer
d????????? ? ? ? ?            ? callander
-????????? ? ? ? ?            ? callander_extraction.py
-????????? ? ? ? ?            ? docker2.trace
-????????? ? ? ? ?            ? docker.trace
d????????? ? ? ? ?            ? filter
-????????? ? ? ? ?            ? filter.py
d????????? ? ? ? ?            ? .git
-????????? ? ? ? ?            ? .gitignore
d????????? ? ? ? ?            ? ls
d????????? ? ? ? ?            ? openbox
d????????? ? ? ? ?            ? plan
-????????? ? ? ? ?            ? profile.json
d????????? ? ? ? ?            ? q
-????????? ? ? ? ?            ? README.MD
-????????? ? ? ? ?            ? seccomp.json
d????????? ? ? ? ?            ? static
d????????? ? ? ? ?            ? .venv
```

Conclusion

- For both relatively complex and simple programs, dynamic syscall analysis works well
- It detects on average around 45% fewer syscalls than tested static alternatives
- The larger a program is, the more difficult it is to construct comprehensive scenarios
- In future work, a more in-depth comparison should be performed featuring argument filtering and execution stage separation, as well as more static analysis alternatives

References

[1] Avi Silbershatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* Tenth Edition. John Wiley & Sons, Inc., 2018.

[2] Claudio Canella et al. *Automating Seccomp Filter Generation for Linux Applications.* 2020. arXiv: 2012.02554 **[cs.CR]**.

[3] Ryan Petrich. *"Linux Sandbending: Binding Program Behaviors without Binding Our- selves"*. In: Presented at All Day DevOps 2023, 2023. url: https://github.com/rpetrich/callander.