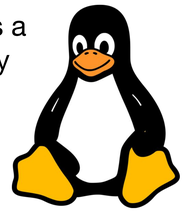
Logic-Powered Cyber Security: Automated SELinux Access Control Policy Verification

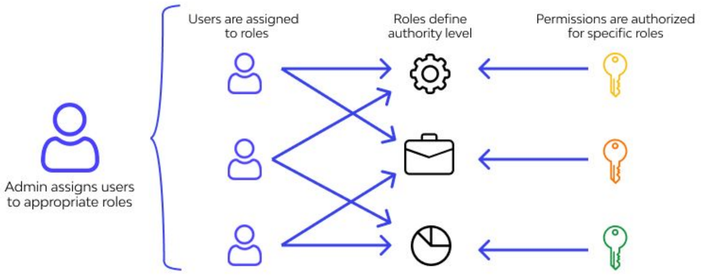
Divyam Pahuja, **Alvin Tang**, **Klim Tsoutsman**; Supervisor: AsPr **Alwen Tiu**

1. Introduction

Linux has been the most popular operating system for web servers, embedded devices, cloud computing and a wide range of applications in enterprise settings. Security of these systems is paramount for integrity, confidentiality and privacy.

Security-Enhanced Linux (**SELinux**), as a widely used Linux extension, is a security framework for Linux that provides fine-grained control of permissions to system resources by enforcing access control **policies** at the kernel level.





Permission to retrieve and modify system resources is managed with access control. Role-based access control (**RBAC**) considers multiple factors and conditions of those requesting access.

How to ensure the correct use of access control for system security without being tired of maintenance?

Problem:

1. Manual access control policy management is prone to error.
2. Tedious to **verify** that the policies achieve intended security specifications by system administrators.
   ➔ System vulnerable to **misconfigurations** resulting in loopholes.

Need:

A straightforward and efficient tool for automating the verification of SELinux RBAC policies.

Solution: satisfiable modulo theories (SMT) – assert logic consistency!

1. Linux mascot tux.png by Larry Ewing on Wikimedia Commons licensed with CC0. Available: https://commons.wikimedia.org/wiki/File:Linux_mascot_tux.png
2. Role Assignments by Laiye Inc. Available: https://documents.laiye.com/chatbot-private/en/docs/features/role_assignments
3. Android logo (2019-2023).svg by Google on Wikimedia Commons. Available: https://commons.wikimedia.org/wiki/File:Android_logo_(2019-2023).svg

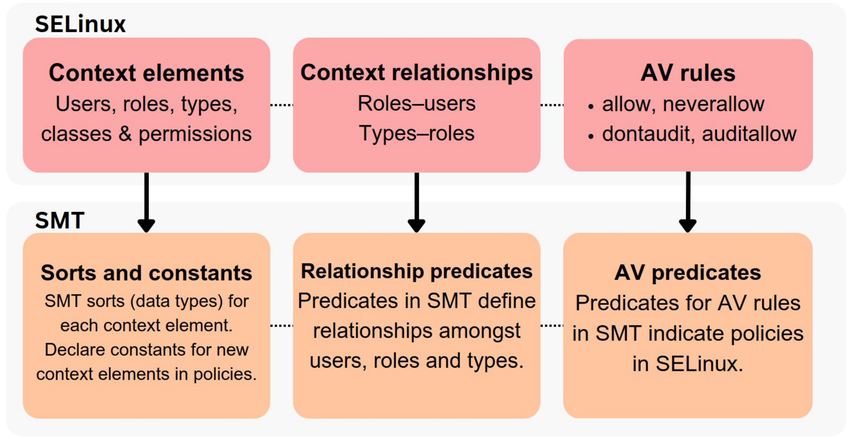
2. Solution

We develop an automated tool to verify RBAC policies using an expressive approach based on formal logic — **satisfiability modulo theories** (**SMT**). The solution involves two stages:

1. Formalising the RBAC design in SELinux in formal semantics
2. Implementing the tool with automated theorem proving (SMT solving).

Logic Formalisation of RBAC design in SELinux

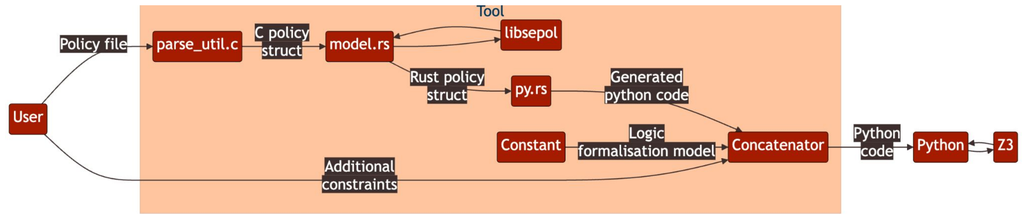
- RBAC design in SELinux consists of **context elements** as labels to both principals and resources namely *roles*, *types*, *classes* and *permissions*.
- Each principal (*e.g.*, user) is assigned roles; each role is assigned types.
- **Access vector** (**AV**) **rules** are policies determining if permissions are granted. AV rules match types assigned to principals with types assigned to resources.



Implementing the Automated Verifier

The verifying process involves three major steps.

1. Translate the SELinux policy file supplied by the user into SMT.
2. Accept security specifications as additional constraints in SMT from the user.
3. Use Z3 to check that the policy complies with the security specifications.



- We use SELinux C libraries (*e.g.*, libsepol) and Rust to implement the parser and translator. The concatenator then generates an executable script for the Z3 Python API.
- The SMT solver determines whether a set of logic statements are **satisfiable**. If they are not, the logic statements are **contradictory**, which implies that the SELinux policies are not consistent with specifications supplied by the system administrator.

3. Evaluation: Case Study

Do policy misconfigurations actually happen in real-world scenarios? **Yes!**

- Android 11 enables SELinux by default. In 2011, developers of the operating system created policies that were over-permissive.
- Adversaries could take advantage of this to gain privilege escalation and install malicious code to read and write data on mobile phones.

How the automated verifier could have identified the vulnerability to mitigate the risk?

1. Consider the AV rule: allow principals with the 'system app' type to write to the files assigned with the "APK data" type. *(Parsing)*

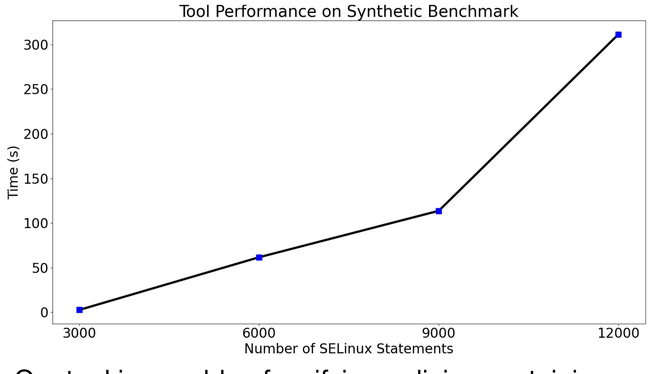
2. Android developers specify that only certain types have permission to files with the "APK data" type. *(Additional constraints)*

3. The "system app" type is not intended to gain permission. A logic statement in step 1 contradicts another in step 2. *(Theorem proving)*

The SMT solver output indicates it is "**unsatisfiable**" *i.e.*, a policy misconfiguration is **identified**!

4. Evaluation: Efficiency

We conducted **synthetic benchmarking** to evaluate the scalability and efficiency of the entire process.



Our tool is capable of verifying policies containing 12,000 context elements within 3.5 minutes.