Enhancing Secure Deployment with Ansible: A Focus on Least Privilege and Automation for Linux

Eddie Billoir, Romain Laborde, Ahmad Samer Wazan, Yves Rutschle, Abdelmalek Benzekri

Enhancing Secure Deployment with Ansible: A Focus on Least Privilege and Automation for Linux

Eddie Billoir
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Airbus Protect
Toulouse, France
eddie.billoir@airbus.com

Romain Laborde
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France
romain.laborde@irit.fr

Ahmad Samer Wazan
Zayed University
Dubai, United Arab Emirates
ahmad.wazan@zu.ac.ae

Abdelmalek Benzekri
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France
abdelmalek.benzekri@irit.fr

Yves Rütschlé
Airbus Protect
Blagnac, France
yves.rutschle@airbus.com

ABSTRACT

As organisations increasingly adopt Infrastructure as Code (IaC), ensuring secure deployment practices becomes paramount. Ansible is a well-known open-source and modular tool for automating IT management tasks. However, Ansible is subject to supply-chain attacks that can compromise all managed hosts. This article presents a semi-automated process that improves Ansible-based deployments to have fine-grained control on administrative privileges granted to Ansible tasks. We describe the integration of the RootAsRole framework to Ansible. Finally, we analyse the limit of the current implementation.

CCS CONCEPTS

• **Security and privacy** → *Access control*; *Operating systems security*.

KEYWORDS

Principle of Least privilege, Ansible, Infrastructure as Code, Security

ACM Reference Format:
Eddie Billoir, Romain Laborde, Ahmad Samer Wazan, Abdelmalek Benzekri, and Yves Rütschlé. 2024. Enhancing Secure Deployment with Ansible: A Focus on Least Privilege and Automation for Linux. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30–August 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3664476.3670929

1 INTRODUCTION

Infrastructure as Code (IaC) aims at automating infrastructure deployment based on practices from software development [10]. This approach makes the provision and change of IT infrastructures and their configuration faster using automatic, consistent and reusable procedures. In addition, IaC enhances transparency because the code of the infrastructure can be reviewed and audited. IaC also deeply modifies the IT administration processes since administrators can now apply an Agile approach such as Continuous Integration/Continuous Delivery to quickly adapt their IT infrastructure to new business requirements and recover from security incidents. Having such a controlled and repeatable change management process improves both speed and reliability.

Ansible is a well-known open-source, modular tool to automate IT management tasks such as software provisioning, configuration management, and application deployment functionality [4]. System administrators can describe repeatable IT management tasks in script files, called playbooks, that specify the sequence of actions to be applied to the managed hosts. It supports a large range of operating systems, cloud platforms, and network devices. it improves the predictability of IT tasks, keeping an idempotent behaviour, which means executing one time or multiple times a playbook will always result in the same state for the target managed hosts.

Ansible provides a large collection of ready-to-use content (plugins, modules, roles and playbooks) that simplifies the automation of common server tasks such as installing packages, creating and managing users, manipulating files and permissions, and managing services. One of the main strengths of Ansible is the ease with which these third-party features can be used to speed up deployment process implementation.

However, using third-party Ansible content implies that organisations must entrust administrative responsibilities to external entities. When administrative responsibilities are delegated to external entities, the security of the entire supply chain depends on the robustness of these third-party systems. If these dependencies are compromised through negligence or malice, the consequences can potentially lead to data breaches, service interruptions or even a complete infrastructure compromise. The recent supply chain attack which installed a backdoor on the 'xz' Linux utility in March 2024 is a good example of how systems can be exploited through a single dependency [7]. This backdoor was quietly implemented during a minor update of the software. To address these risks, the Zero-Trust Architecture [13] security model presents many security

ARES 2024, July 30–August 02, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3670929

practices and requirements to manage this supply chain effectively. One of its core requirements is the principle of least privilege.

The principle of least privilege (POLP) is an engineering process that involves understanding users' responsibilities to grant them only the minimum permissions required to accomplish their tasks using computer systems [14]. This principle applies to all users, especially those responsible for system administration, who often possess critical administrative privileges.

However, Ansible does not respect POLP properly for administrative tasks. Ansible escalates to full administrative rights using the sudo [16] command when a playbook requires some administrative privileges. Consequently, if a playbook which depends on third-party artefacts requests administrative privileges, the third party can have full administrative privileges on the managed hosts. For example, this can have dramatic consequences if the third-party artefact installs a backdoor. This issue is primarily due to design decisions in the underlying operating systems [5].

This article presents a semi-automated process that improves Ansible-based deployments to grant fine-grained administrative privileges to Ansible tasks. This process allows an administrator to know what administrative privileges are required by a playbook and the third-party artefacts. We also implemented a module to use RootAsRole [17] instead of sudo to execute Ansible tasks that require administrative privileges on the operating system. This helps System administrators minimise their administrative privileges and validate them at run time. Consequently, this limits the impact of a potential supply chain attack.

The rest of this article is structured as follows. Section 2 introduces the Ansible framework. It also gives a detailed example of a supply chain attack that can affect systems managed by Ansible. In section 3, we present the administrative privilege model of Linux and how the RootAsRole project can improve the management of Linux administrative privileges. Section 4 shows how the RootAsRole framework can be integrated with Ansible and discusses its security benefits. We discuss the potential and limitations of the presented work in section 5. Finally, we conclude this paper in section 6.

2 DEPLOYING CONFIGURATIONS USING ANSIBLE

In this section, we briefly introduce the main concepts of Ansible. Then, we present a use case to illustrate the supply chain attack problem.

2.1 Introduction to Ansible

An Ansible environment consists of one control node (the system on which Ansible is installed) and a set of managed hosts (the systems that are controlled by Ansible). An administrator writes playbooks that describe the desired state of the controlled hosts and every task to reach it. The administrator should also create an inventory that describes the list of managed nodes in a logically organized manner. Ansible doesn't need to install specific agents on the managed nodes. When the administrator executes a playbook, the control host generally communicates with managed nodes via SSH to transfer the configuration scripts to the managed hosts and execute them remotely. For example, in Figure 1, an Ansible control host deploys a new configuration on three managed hosts.

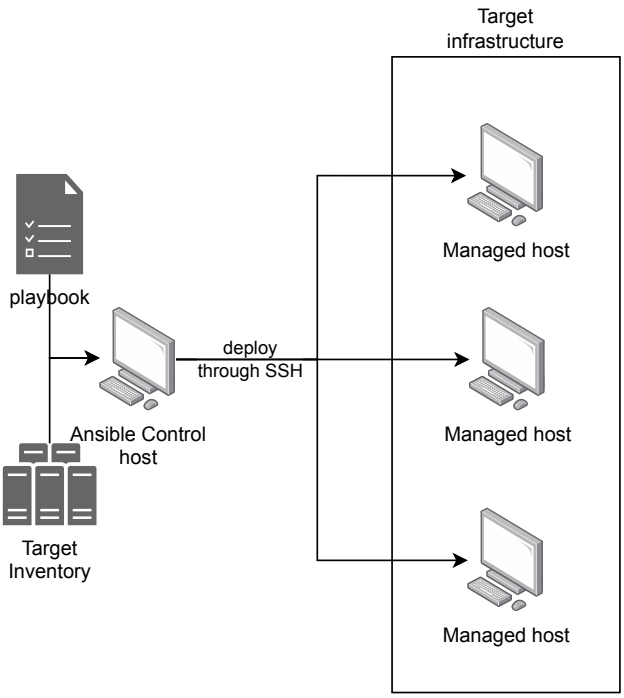

Figure 1: Example of an Ansible environment

Additionally, Ansible introduces the feature of Roles, which are reusable, self-contained units of automation artefact that bundle tasks, variables, templates, and configurations. Roles can be distributed directly or inside Ansible collections which are a distribution format for Ansible content that can include playbooks, roles, modules, and plugins to address a set of related use cases. For instance, a collection might automate the administration of a specific database or a specific server. Collections can be shared with the community and installed from distribution servers, such as Ansible Galaxy.

2.2 The supply chain attack example

Let's consider Alice, Dorine and Mallory. Alice is a system administrator who is responsible for the deployment process of an Apache2 webserver. Dorine is a web developer for the website content. Mallory is an Ansible Galaxy contributor; she created the role '`mallory.net_intput`' to simplify the configuration of the firewall policy on many Unix systems. To deploy the production infrastructure, Alice starts by writing a playbook presented in Figure 2. As she doesn't develop the website and isn't a network administrator, she decides to refer to Dorine to deploy the website content and to Mallory's community role to open the port on the firewall for the webserver. The playbook is described below :

(1) In line 2, she starts by describing the title of the playbook.

```yaml
---
- name: Ansible Playbook to Install and Setup Apache on Ubuntu
  hosts: webserver
  become: true
  roles:
    - mallory.net_input
  tasks:
    - name: Install latest version of Apache
      apt: name=apache2,iptables update_cache=yes state=latest
    - name: Create document root for domain configured in host variable
      file:
        path: "/var/www/{{ http_host }}"
        state: directory
        owner: www-data
        group: www-data
        mode: '0755'
    - name: Deploy website source code
      import_playbook:
        playbook: deploy-website.yml
    - name: Set up virtualHost
      template:
        src: "../templates/apache-template.conf.j2"
        dest: "/etc/apache2/sites-available/{{ http_conf }}"
    - name: Enable site
      command: a2ensite {{ http_conf }}
      notify: restart-apache
    - name: open port 80
      ansible.builtin.import_role:
        name: mallory.net_input
      vars:
        net_input_protocol: tcp
        net_input_port: 80
        net_input_state: accepting
  handlers:
    - name: restart-apache
      service:
        name: apache2
        state: restarted
```

Figure 2: Example of a playbook to install and configure an Apache2 server

(2) In line 3, she declares to grant every task administrative privileges on the managed hosts. By default, Ansible will use 'sudo' to elevate the privileges to root.
(3) In line 4, she defines on which managed hosts this playbook is being run.
(4) In lines 5-6, she imports the (dummy) 'mallory.net _input' Ansible Galaxy community role in the playbook tasks.
(5) In lines 8-9, she mandates the 'apt' module to enforce the 'apache2' package to be installed and in the 'latest' state.
(6) In lines 10-16, she mandates the 'file' module that '/var/www/ http_host ' path is a directory owned by 'www-data' with specific discretionary permissions.
(7) In lines 11-19, she delegates the website implementation deployment to the 'deploy-website.yml' playbook. In this example, this playbook is written by Dorine who is an internal web developer.
(8) In lines 20-23, she deploys the Apache2 configuration.
(9) In lines 24-26, she mandates that the Apache2 configuration is enabled by the webserver and notifies the "restart-apache" handler in line 41.
(10) In lines 27-33, she calls 'mallory.net_input' role in verifying that the firewall accepts incoming HTTP requests
(11) In lines 35-38, she declares a handler that restarts the apache2 software daemon. It is called in line 26 if the calling task is modifying the system state.

It is important to note in this example that Ansible will execute the 'mallory.net_input' community role and the 'deploy-website.yml' playbook on the managed host with all administrative privileges. Although using the role simplifies Alice's playbook implementation, it hides many tasks that are not directly visible. However, Alice is security-aware and should have already checked every task of every used dependency. Figure 3 presents the dependency graph and who can write on each dependency.

Once the playbook has been successfully tested on Alice's personal development infrastructure, let's introduce her deployment

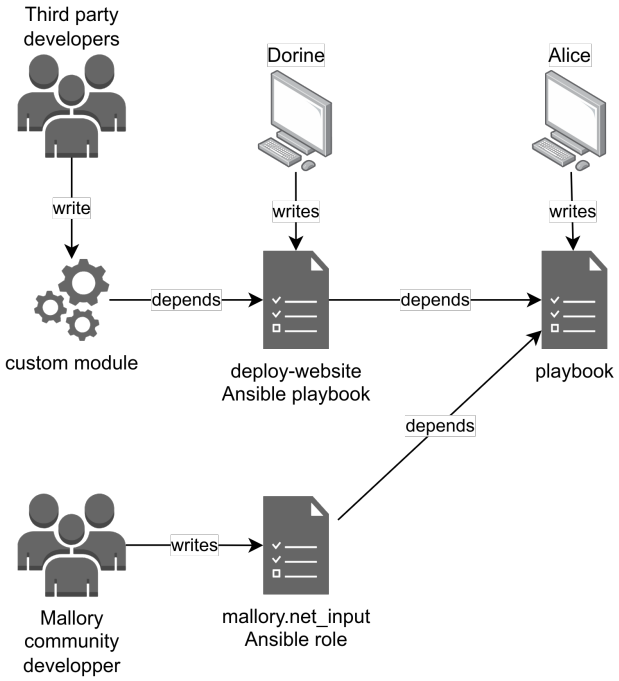

Figure 3: The playbook dependency graph

team: Bob and Charlie. Bob is responsible for testing the deployment process in a production-representative infrastructure. Charlie is responsible for deploying playbooks in the real production infrastructure. Alice asks Bob to test it on a production-representative infrastructure. If everything works properly, Bob tells Charlie that the playbook is ready for production deployment, and Charlie performs the deployment. This process is shown in Figure 4. Although this deployment process can represent a heavy workload, it ensures that the playbook process is maintained and knowledge is shared.

After multiple successful deployments on production, Mallory, the maintainer of the 'mallory.net_input' Ansible role, decides to perform a supply-chain attack by injecting a reverse shell into her tasks, i.e., opening a new port on the firewall and binding a full-privileged shell to it. As her current Ansible process is fully privileged, the malicious process can silently suppress security mechanisms, setting its own veiled monitoring system to avoid external anomaly detection. This way, Mallory can attack the production environment in the next deployments. The recent case of the 'xz' supply-chain attack is similar to our example [2].

We believe enforcing the principle of least privilege more precisely to ensure that every Ansible task is only granted administrative privileges for its objective will avoid such an attack.

3 ADMINISTRATIVE PRIVILEGE MANAGEMENT ON LINUX

In this section, we present the administrative privileges management on Linux systems. After introducing Linux capabilities, we list available tools for managing administrative privileges.

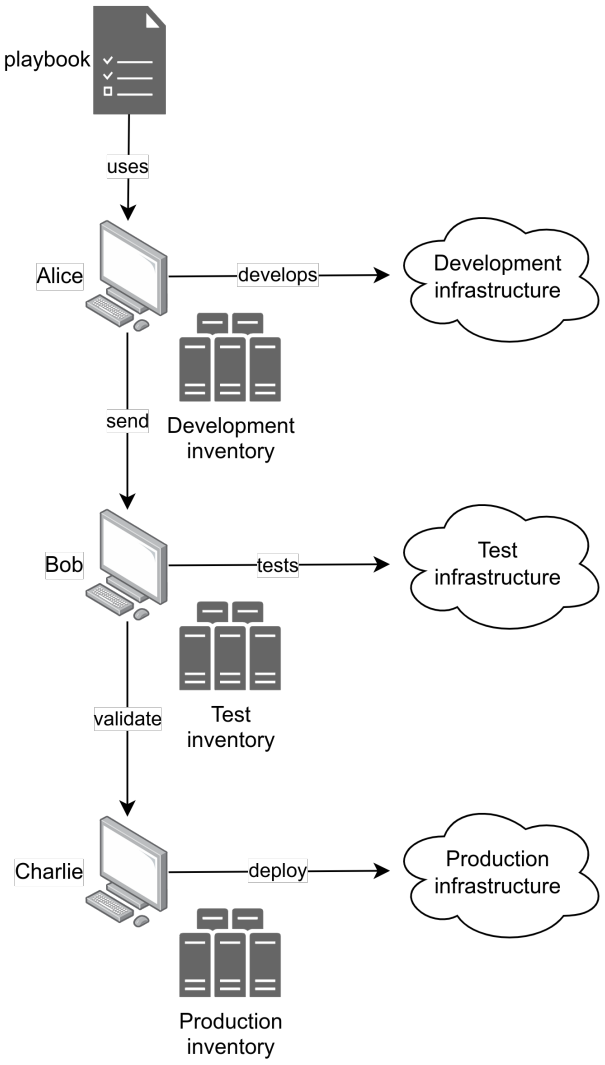

Figure 4: Project process to deploy in production example

3.1 Linux capabilities

Historically, Linux administration was based on the existence of a privileged user, the superuser or root, whose identifier value was 0. Checking whether a user's identifier was equal to zero was a basic way of checking whether a process could bypass access restrictions or security features. This approach to managing privileges on Linux is incorrect. It does not adhere to the principle of least privilege, as administrative tasks may have different objectives, such as managing network configuration or bypassing file access controls.

In 1998, Linux implemented the Linux Capabilities feature, which defines a set of privileges that can be independently granted to processes. This granular approach allows administrators to limit the scope of administrative powers. For example, CAP_SYS_TIME is a capability that is needed to set the system clock, and CAP_DAC_OVERRIDE allows bypassing the DAC file-system access control. However, since this feature exists, its adoption in the Linux community was difficult. Indeed, the initial version prerequisites the configuration of extended attributes in the file system to define how privileges are granted and inherited between processes. This prerequisite is unsuitable for many use cases because the file system configuration is static, and processes can be instantiated dynamically.

Since 2015, Linux has added the 'Ambient' feature to Linux Capabilities, which allows programs to define which privileges are inherited by parent processes independently from the file system. This new feature matches the dynamic behaviours of processes. However, this new feature doesn't significantly change its adoption rate. Indeed, many other technical reasons still make the feature unpractical, such as file-extended attributes drawbacks, its complete but complex documentation or even its lack of tools [18]. One major conceptual issue with Linux capabilities is that they are still coarse-grained, regrouping multiple use cases for one privilege.

3.2 The sudo command

'sudo' is the main utility used to assign administrative privileges on Linux. It is a command-line tool that allows a system administrator to delegate authority to give certain users (or groups of users) the ability to run some (or all) commands as root or another user while providing an audit trail of the commands and their arguments [16]. By default, when there is 'become: true' instruction in an Ansible playbook, Ansible uses the 'sudo' tool to switch from the current unprivileged user to the superuser. However, the 'sudo' command does not handle Linux capabilities, so when Ansible uses sudo, the process executed on the managed host always gains all Linux privileges for any task surrounded by the 'become: true' instruction.

3.3 SELinux

To apply the POLP even more fine-grained, SELinux is a security solution integrated into Linux Kernel as a Linux Security Module [15]. This solution is designed to enforce mandatory access controls (MAC) on processes and files, providing a robust mechanism for access control that overrides traditional discretionary access controls (DAC) [9, 12]. According to the implemented policy, SELinux restricts the specific kernel actions that processes are allowed to perform on system resources. Since SELinux controls kernel security access control, every right of SELinux is kernel-level, making it difficult to configure manually as it has many rights. To solve it, RedHat company propose to combine the 'auditd' program to automatically configure SELinux based on monitoring output [11]. Auditd is a Linux daemon that allows any administrator to audit system events on a Linux system. It monitors activities such as file accesses, logins, and other system-level behaviours. By integrating 'auditd' with SELinux, RedHat provides a powerful solution for managing SELinux security policies effectively. However, SELinux does not change the traditional Linux capabilities; it just adds further constraints [1]. Furthermore, RedHat documentation shows how to use privileged commands with the 'sudo' tool, which does not manage the Linux capabilities feature [11].

One important notion with capabilities is that they often change the behaviour of the kernel, so the second part of access control

conditions accesses another part of the kernel. For example, when executing 'fork()' with CAP_SYS_RESOURCE, the kernel bypasses the subprocess limit set on the system. SELinux could manage this behaviour by denying every CAP_SYS_RESOURCE capability request for the concerned process. As SELinux currently recommends 'sudo' in documentation, this approach is not the right way to manage the case, as it first grants the privilege with 'sudo' and then denies it with SELinux [14]. The proper solution is to manage Linux capabilities.

3.4 The RootAsRole project

The RootAsRole project provides new security mechanisms for controlling Linux capabilities at the user level. Its main features are similar to sudo, except it applies a Role-based access control model [8] and manages Linux capabilities. It includes three utility commands 'sr', 'chsr' and 'capable'. The 'sr' command [6] (abbreviation for "Switch Role") allows the execution of commands using only the minimum required capabilities. With the 'chsr' (abbreviation for "CHange Switch Role") tool, an administrator can specify in RootAsRole policies administrative tasks which consist of a Linux command, including constraints on the parameters, the set of Linux capabilities granted for this specific task and optionally, the uid and/or the gid to use when executing the command. The tasks are associated with roles which are assigned to users. Unlike the sudo command, changing the uid/gid is not necessary for executing a command with administrative privileges. Determining which Linux capabilities are required for a specific command line can be complicated. To solve this issue, RootAsRole proposes the 'capable' tool, which automatically identifies the Linux capabilities requested by a program for a specific use case, simplifying its configuration.

4 MANAGING LINUX CAPABILITIES FOR ANSIBLE TASKS

In this section, we outline a new method to enforce the principle of least privilege by limiting the capabilities granted to Ansible tasks using the RootAsRole framework. Our proposal involves two aspects: 1) integrating RootAsRole in the Ansible framework, 2) integrating the principle of least privilege in a devops process.

Firstly, we modified the Ansible dependency to 'sudo' for executing privilege operations on the managed host to the RootAsRole 'sr' command which allows us to precisely control the Linux capabilities granted to the process. Fortunately, Ansible is highly modular and allows the development of its own 'become' module to replace the default one [3]. Therefore, we developed an extension to include 'sr' as a new method available in the Ansible playbook when module 'become' is called. Based on this new module, we could integrate RootAsRole in both the Ansible playbook specification and verification process, as well as the Ansible playbook execution process.

The source code of the proof-of-concept presented in this section is available at this address: https://anonymous.4open.science/r/RootAsAnsible-641F.

```
1.  ---
2.  - name: Ansible Playbook to Install and Setup Apache on Ubuntu
3.    hosts: webserver
4.    become: true
5.    become_method: sr
6.    gather_facts: false
7.    roles:
8.      - mallory.net_input
9.    tasks:
10.     - name: Install latest version of Apache
11.       become_flags: "-r deploy_apache -t install_apache2"
12.       apt: name=apache2,iptables update_cache=yes state=latest
13.     - name: Create document root for domain configured in host variable
14.       become_flags: "-r deploy_apache -t create_document_root"
15.       file:
16.         path: "/var/www/{{ http_host }}"
17.         state: directory
18.         owner: www-data
19.         group: www-data
20.         mode: '0755'
21.     - name: Deploy website source code
22.       become_flags: "-r deploy_apache -t deploy_website"
23.       import_playbook:
24.         playbook: deploy-website.yml
25.     - name: Set up virtualHost
26.       become_flags: "-r deploy_apache -t setup_virtualhost"
27.       template:
28.         src: "../templates/apache-template.conf.j2"
29.         dest: "/etc/apache2/sites-available/{{ http_conf }}"
30.     - name: Enable site
31.       become_flags: "-r deploy_apache -t enable_site"
32.       command: a2ensite {{ http_conf }}
33.       notify: restart-apache
34.     - name: open port 80
35.       become_flags: "-r deploy_apache -t open_port_80"
36.       ansible.builtin.import_role:
37.         name: mallory.net_input
38.       vars:
39.         net_input_protocol: tcp
40.         net_input_port: 80
41.         net_input_state: accepting
42.   handlers:
43.     - name: restart-apache
44.       become_flags: "-r deploy_apache -t restart_apache"
45.       service:
46.         name: apache2
47.         state: restarted
```

Figure 5: Modified playbook to add RootAsRole become method, role and tasks per ansible task

4.1 Integrating RootAsRole and Ansible

Now that RootAsRole is integrated into Ansible, Alice can benefit from the new features. Alice will slightly modify her playbook to become the one in Figure 5. The additions are as follows:

(1) In line 5, the utility command used to escalate privileges is changed to the RootAsRole 'sr' command. This module was specially developed for this work.
(2) In line 6, Alice disables system information gathering. By default, Ansible gathers system information for every module. However, it is not needed for the proof-of-concept purpose.
(3) In lines 11, 14, 22, 26, 31, 35, and 44, Alice mandates RootAsRole to use the '`deploy_apache`' role and a specific RootAsRole task. There will be one RootAsRole task for each Ansible task in the playbook.

Alice (or someone else) should also write the RootAsRole policy. A RootAsRole policy specifies a set of roles. Each role consists of the set of actors who can play this role and the set of tasks that can be performed by these actors according to this role. Finally, a task defines the set of commands and the granted Linux capabilities. RootAsRole includes the utility 'capable' to help the RootAsRole policy creator determine the Linux capabilities required by a command. Then, Alice can test every task in her playbook with the 'capable' module to write her RootAsRole policy.

Figure 6 is the sample of the RootAsRole policy dedicated to the Mallory's role usage. The full policy is available at https://anonymous.4open.science/r/RootAsAnsible-641F. This policy sample should be interpreted as follows:

(1) Line 12 declares the "deploy_apache" role.

```
{
  "version": "3.0.0-alpha.4",
  "storage": {
    "method": "json",
    "settings": {
      "immutable": false,
      "path": "/etc/security/rootasrole.json"
    }
  },
  "roles": [
    {
      "name": "deploy_apache",
      "actors": [
        {
          "type": "group",
          "groups": [
            "ansible"
          ]
        }
      ],
      "tasks": [
        {
          "name": "open port 80",
          "purpose": "Open port 80 for Apache",
          "cred": {
            "capabilities": {
              "default": "none",
              "add": [
                "CAP_NET_ADMIN",
                "CAP_NET_RAW",
                "CAP_DAC_READ_SEARCH"
              ]
            }
          },
          "commands": {
            "default": "all"
          }
        },
        // TRUNCATED FOR BREVITY
      ]
    }
  ]
}
```

Figure 6: Sample RootAsRole policy associated with the playbook

(2) Line 14-19 assign users who are members of the Linux group "ansible" to the RootAsRole "deploy_apache" role.
(3) Line 22-38 defines a task named "open port 80" that allows all commands with only required privileges for modifying the firewall without changing the user, i.e. Linux capabilities CAP_NET_ADMIN, CAP_NET_RAW, and CAP_DAC_READ_SEARCH.

4.2 Security Benefits

With this new security measure enforced, Mallory's supply chain attack becomes more complex because she cannot access the full set of administrative privileges, and OS security mechanisms remain active to detect or avoid the attack. For instance, in her network task, Mallory only needs CAP_NET_ADMIN, CAP_NET_RAW, and CAP_DAC_READ_SEARCH privileges as a regular user. Even if Mallory manages to create a reverse shell, she still does not elevate to full root privileges. These privileges allow her to configure the system network effectively. However, holding these privileges prevents her from altering the entire configuration or controlling the monitoring systems.

Consequently, she is compelled to seek alternative routes to access the system, which remains under surveillance and provides only limited access. Furthermore, Mallory cannot gain additional privileges in her reverse shell due to a feature in Linux capabilities used by RootAsRole called the Bounding set. This security feature is the set that defines which privileges exist for the process. Removing them from this set avoids privilege escalation, regardless of the method used. Thus, RootAsRole prevents any sub-process from acquiring more privileges than those specified in its Bounding set, effectively blocking Mallory from escalating her privileges within the shell.

However, Mallory's supply-chain attack cannot be completely avoided. Indeed, if its process is not containerized, she can still intercept network-monitoring packets. To solve this, we recommend creating a new policy module with SELinux only to allow firewall rules to be inspected and an incoming HTTP network packet rule to be added [11]. By combining RootAsRole and SELinux policies, administrators can enforce POLP for administration tasks more efficiently using best practices and avoid most supply-chain attacks.

Alice can now transmit her playbook with the associated RootAsRole policy to Bob for testing. Bob integrates Alice's playbook into his testing playbook, which deploys the RootAsRole policy on the testing infrastructure before importing her playbook. Once the deployment is successful in many cases, Bob can send his playbook to Charlie. Charlie verifies that the inventory matches well with Bob's one before deploying it.

Once these verifications and fixes are done and the playbook is still functioning, Bob can send the final Alice's playbook and its fixed RootAsRole policy to Charlie for updating or deployment purposes. Then Charlie deploys the new RootAsRole policy if needed and executes Alice's playbook.

5 DISCUSSION & FUTURE WORKS

We discuss in this section the limitations of the current solution and present future works.

5.1 Automated generation of RootAsRole policies based on Ansible playbooks

Alice can identify the required privileges for each playbook task with the RootAsRole 'capable' utility. Currently, this process is manual and time-consuming. To obtain all the required privileges for a use case, Alice must start by executing 'capable' for the first Ansible task in the playbook without any administrative privilege. The 'capable' utility outputs a first set of privileges required by the task, and then Alice needs to grant these privileges (or those pertain) to the tested command and repeat the process until the task succeeds. She needs to execute this process for each task in her playbook.

This process could be automated with Ansible. Indeed, we could imagine developing a new module that automates this process. This module could execute Alice's playbook, surrounding each task with the 'capable' tool, and determining the required privileges based on the execution in a testing environment. This module should create a RootAsRole policy and a modified playbook per the generated role in the policy. Of course, the generated policy has to be validated by the administrator before being deployed.

```
sr /bin/sh -c ".tmp/temporaryfile.sh"
```

Figure 7: Command-line executed by Ansible for a task

5.2 Adapting RootAsRole to Ansible specifities

The RootAsRole policy written in Figure 6 does not respect POLP completely. Indeed, as specified in line 36, the task restricts the Linux capabilities but it allows any commands for the task. This issue is due to an Ansible limitation. Indeed, when deploying a task, Ansible connects through SSH, copies some script files to a temporary folder on the managed host, and executes them. These files could be bash scripts, Python programs, or any executable program. Then, Ansible prepares a command line to execute the file; if the task requires it, Ansible calls a 'become' module to escalate privileges. For example, the final executed command is structured with sr as shown in Figure 7.

As a potential solution, RootAsRole can check the integrity of the program using a hash. However, in this case, the checked program is the interpreter (here it is '/bin/sh') but not the Ansible script. This first issue requires a new RootAsRole plugin specifically designed for Ansible deployments.

Another challenge with hash checking is that administrators must compute the hash for each deployed Ansible script, covering a wide range of usage scenarios, and then repeat the permission deployment process for each update to Ansible. Since we aim at avoiding supply-chain attacks, which typically foothold in a software update, implementing a new RootAsRole policy every time Ansible is updated proves inefficient. Furthermore, if the RootAsRole policy generation is automated, it could inadvertently grant access to the supply-chain attack within the RootAsRole policy. This challenge still needs to be addressed.

6 CONCLUSION

The adoption of Infrastructure as Code (IaC) is transforming IT operations, streamlining deployment processes and strengthening system resilience. While IaC accelerates provisioning and configuration tasks with its automated, consistent, and auditable procedures, Ansible empowers administrators with precise orchestration capabilities across diverse environments.

However, the reliance on third-party Ansible content underscores the need to address supply-chain risks, particularly tasks that require administrative privileges. In this article, we present a first attempt to implementing the principle of least privileges in the Ansible framework by integrating RootAsRole. Our work enables security administrators to control administrative privileges granted to Ansible tasks.

For future work, we need to automate the process for generating RootAsRole policies to decrease the workload and apply POLP more finely in deployments, marking an ongoing evolution of robust IT infrastructures. We will also investigate further RootAsRole improvements to adapt to Ansible deployments

REFERENCES

[1] Anderson. 2021. Answer to "How to Add a Capability to SELinux Custom Role?".
[2] (@AndresFreundTec@mastodon.social) AndresFreundTec. 2024. I Accidentally Found a Security Issue While Benchmarking Postgres Changes.If You Run Debian Testing, Unstable or Some Other More "bleeding.…
[3] Ansible project. 2024. Become Plugins — Ansible Community Documentation. https://docs.ansible.com/ansible/latest/plugins/become.html#become-plugin-list.
[4] Ansible project. 2024. Homepage | Ansible Collaborative. https://www.ansible.com/.
[5] Eddie Billoir, Romain Laborde, Ahmad Samer Wazan, Yves Rütschlé, and Abdelmalek Benzekri. 2023. Implementing the Principle of Least Privilege Using Linux Capabilities: Challenges and Perspectives. In *2023 7th Cyber Security in Networking Conference (CSNet)*. IEEE, Montreal, QC, Canada, 130–136. https://doi.org/10.1109/CSNet59123.2023.10339753
[6] Eddie Billoir, Ahmad Samer Wazan, Romain Laborde, and Benzekri Abdelmalek. 2024. LeChatP/RootAsRole.
[7] Evan Boehs. 2024. *Everything I know about the XZ backdoor.* Boehs, Evan. https://boehs.org/node/everything-i-know-about-the-xz-backdoor
[8] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.* 4, 3 (aug 2001), 224–274. https://doi.org/10.1145/501978.501980
[9] Carl E. Landwehr. 1981. Formal Models for Computer Security. *Comput. Surveys* 13, 3 (Sept. 1981), 247–278. https://doi.org/10.1145/356850.356852
[10] K Moris. 2021. Infrastructure as Code Dynamic Systems for the Cloud Age.
[11] Red Hat. 2024. Using SELinux Red Hat Enterprise Linux 8 | Red Hat Customer Portal. https://access.redhat.com/documentation/fr-fr/red_hat_enterprise_linux/8/html-single/using_selinux/index.
[12] Red Hat. 2024. What Is SELinux? https://www.redhat.com/en/topics/linux/what-is-selinux.
[13] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. 2020. Zero Trust Architecture. https://doi.org/10.6028/NIST.SP.800-207
[14] J.H. Saltzer and M.D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (Sept. 1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939
[15] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.
[16] Sudo. 2023. Sudo-Project/Sudo. Sudo Project.
[17] Ahmad Samer Wazan, David W Chadwick, Remi Venant, Eddie Billoir, Romain Laborde, Liza Ahmad, and Mustafa Kaiiali. 2022. RootAsRole: a security module to manage the administrative privileges for Linux. *Computers & Security* pre-proof, 102983 (2022), 24.
[18] Ahmad Samer Wazan, David W. Chadwick, Remi Venant, Romain Laborde, and Abdelmalek Benzekri. 2021. RootAsRole: Towards a Secure Alternative to Sudo/Su Commands for Home Users and SME Administrators. In *ICT Systems Security and Privacy Protection*, Audun Jøsang, Lynn Futcher, and Janne Hagen (Eds.). Vol. 625. Springer International Publishing, Cham, 196–209. https://doi.org/10.1007/978-3-030-78120-0_13