The pkexec Privilege Escalation Mechanism in Linux: A User-Space Implementation Analysis

Executive Summary

The requirement for a root password when invoking pkexec from a lower-privileged account in contemporary Linux distributions is a sophisticated, multi-layered implementation primarily residing within **user space**. pkexec functions as a client to the **Polkit authorization framework**, requesting elevated privileges for specific operations. Polkit, in turn, delegates the actual password verification to the **Pluggable Authentication Modules (PAM)** system. While the SUID bit on the pkexec binary is indispensable for its initial elevation to communicate with the privileged polkitd daemon, the intricate logic governing authorization decisions and authentication prompts is executed entirely in userland. This architectural design facilitates flexible, fine-grained control over privilege escalation without necessitating modifications to the kernel for policy enforcement.

Introduction to pkexec and Privilege Escalation

The Purpose of pkexec

pkexec is a command-line utility designed to allow an authorized user to execute a program as another user. In most common scenarios, if no specific username is provided, the program is executed with the privileges of the administrative superuser, root.[1] This functionality positions pkexec as a tool akin to sudo, yet it is fundamentally distinct in its design philosophy. Unlike sudo, which can grant broad permissions to execute any command as root, pkexec is tightly

integrated with the Polkit authorization framework. This integration enables a more granular approach to privilege management, allowing system administrators to define precise policies for specific actions rather than conferring blanket administrative capabilities.[2] This fine-grained control is particularly valuable in modern graphical desktop environments and for system services that require elevated privileges for very specific tasks.

Understanding SUID Binaries in Linux

The SUID (Set User ID) bit is a special file permission in Linux that plays a critical role in privilege escalation. When this bit is set on an executable file, the program runs with the effective user ID of the file's owner, regardless of the user who executes it.[4] For pkexec, which is typically owned by root, the SUID bit ensures that it executes with root privileges even when invoked by a standard, unprivileged user.[4]

This initial elevation of privilege is a foundational element of pkexec's operation. It provides pkexec with the necessary permissions to securely initiate communication with privileged system daemons, such as polkitd, which also run with elevated privileges (typically as root).[3] Without the SUID bit, an unprivileged pkexec process would lack the authority to establish this secure communication channel and request authorization checks from a privileged system service.

It is important to understand that the SUID bit functions as a gateway, not the ultimate gatekeeper, in this privilege escalation process. The kernel's role is to facilitate this initial change in effective user ID. However, the complex logic that determines *why* a password is required, *which* password, and *how* it is verified, is not handled by the kernel. Instead, these policy decisions are offloaded to user-space components. This architectural separation ensures that the low-level mechanism of privilege elevation (a kernel function) is distinct from the high-level policy enforcement for authorization and authentication (handled in user space). This design promotes modularity, flexibility, and enhanced security by distributing responsibilities across different layers of the operating system.

Polkit: The Authorization Framework

Polkit's Core Architecture: polkitd Daemon, Authentication Agents, and

D-Bus

Polkit serves as a robust authorization manager, providing an Application Programming Interface (API) that allows privileged programs (referred to as "mechanisms") to offer services to unprivileged programs (known as "clients" or "subjects") via inter-process communication (IPC).[5] The central component of this framework is polkitd, a privileged background service that acts as the "Authority" for all authorization checks. polkitd operates entirely within user space, making it a key userland component in the privilege management stack.[2]

Communication between pkexec (the client requesting a privileged action) and polkitd (the authority deciding on that action) is securely managed through **D-Bus**. D-Bus is a sophisticated user-space IPC system that facilitates secure message passing and credential checking between processes.[3] This secure communication channel is vital for maintaining the integrity of authorization requests.

When polkitd determines that a specific action requires user authentication, it interacts with an **Authentication Agent**. These agents are typically provided by the user's graphical desktop environment (e.g., GNOME, KDE) and run within the unprivileged user context.[2] The primary function of the authentication agent is to display the password prompt to the user and securely relay the entered credentials back to polkitd for verification.[2] In scenarios where a graphical agent is unavailable, such as within an SSH session or a text-mode console, pkexec is capable of registering its own textual authentication agent (pkttyagent) to handle the credential input.[1]

The distributed nature of Polkit's architecture, comprising a central daemon, session-specific authentication agents, and D-Bus for communication, offers significant advantages in terms of security, flexibility, and modularity. By separating the authorization authority (polkitd) from the user interface for authentication (the authentication agent) and the client (pkexec), Polkit enhances system security. Sensitive operations, such as handling user credentials during a password prompt, are managed by a dedicated, often isolated, authentication agent. This design reduces the attack surface on the core polkitd daemon. Furthermore, this modularity allows different desktop environments to provide their own authentication agents, ensuring a consistent and integrated user experience across various graphical interfaces.[5] The underlying policy engine (polkitd) remains independent of the specific user interface, making the system easier to develop, debug, and maintain, as each component has a clearly defined responsibility.

Authorization Flow: Actions (.policy files) and Rules (.rules files)

Polkit's behavior is meticulously controlled by a set of configuration files that define "actions"

and "authorization rules".[2] This layered configuration approach allows for both default behaviors and highly customizable overrides.

Actions (.policy files): These are XML-formatted files typically found in the /usr/share/polkit-1/actions/ directory.[2] Each .policy file defines one or more specific actions that can be subjected to authorization (e.g., org.freedesktop.policykit.exec is the default action for pkexec [1]). These files include human-readable descriptions of the action, messages to be displayed during authentication prompts, and, crucially, **default authorization settings**.[5] These defaults are specified using elements such as allow_any, allow_inactive, and allow_active, which dictate the authorization behavior for clients in various session states. Values for these elements can include no (not authorized), yes (authorized without authentication), auth_self (authentication by session owner required), auth_admin (authentication by an administrative user required), auth_self_keep, or auth_admin_keep (authorization retained for a brief period after successful authentication).[5] For pkexec, the default setting is typically auth_admin, which mandates the password of an administrative user.[1]

Rules (.rules files): These are JavaScript files located in two primary directories: /usr/share/polkit-1/rules.d/ for system-provided rules (often from packages) and /etc/polkit-1/rules.d/ for local, administrator-defined configurations.[2] These JavaScript rules provide a mechanism for more complex logic and are capable of **overriding the default authorization settings** established in the .policy files.[2] They allow administrators to define specific conditions—such as group membership, user identity, or session type—under which an action might require a different level of authentication, or even no authentication at all.[2] Rules files are processed in lexicographical order based on their filenames, meaning a file named 00-custom.rules would be evaluated before 60-default.rules.[2]

The combination of .policy and .rules files forms a powerful policy-rule hierarchy that enables granular control over system privileges. The .policy files establish a secure baseline for each action, often defaulting to auth_admin for privileged operations like pkexec.[1] This "secure-by-default" approach is typically set by application developers. The .rules files then provide a flexible override mechanism, allowing system administrators or distribution maintainers to customize this behavior without altering the original application-provided .policy files.[2] This is particularly beneficial for:

- **Distribution-Specific Behavior:** It explains why different Linux distributions may exhibit varying default password behaviors for pkexec or sudo.[14] Distributions can achieve this by adding or modifying JavaScript rules in their /etc/polkit-1/rules.d/ directory.[2]
- **Local Customization:** Organizations can tailor security policies precisely to their unique operational requirements.[2]
- **Maintainability:** Local changes are preserved across package updates, preventing configuration drift.

Furthermore, the use of JavaScript for rules enables evaluation based on dynamic context, such

as whether a user is in an active graphical session or connected remotely via SSH. This allows for more sophisticated authorization decisions than what static XML definitions alone could provide.[2]

The following table details the common values used in Polkit policy files for defining authorization defaults:

Value	Description	Behavior for pkexec
no	The user is not authorized to perform the action. No authentication is required or possible.	pkexec will exit with an authorization error (return value 127) without prompting for a password.[1]
yes	The user is authorized to perform the action without any authentication.	pkexec will execute the command immediately without prompting for a password.[12]
auth_self	Authentication is required, but the user does not need to be an administrative user. The user's own password is sufficient.	pkexec will prompt for the password of the user who invoked the command.[5]
auth_admin	Authentication as an administrative user is required. This typically means the root password or the password of a user in an administrative group (e.g., wheel, sudo).	pkexec will prompt for the password of an administrative user, often the root password, depending on PAM configuration.[5]
auth_self_keep	Similar to auth_self, but the authorization is retained for a brief period (e.g., 5 minutes) after successful authentication, similar to sudo's timeout.[5]	pkexec will prompt for the user's password. Subsequent pkexec calls within the timeout period for the same action may not require a password.[12]

auth_admin_keep	Similar to auth_admin, but the authorization is retained for a brief period (e.g., 5 minutes) after successful authentication.[5]	pkexec will prompt for an administrative user's password. Subsequent pkexec calls within the timeout period for the same action may not require a password.[12]

PAM: The Authentication Backbone

Overview of Pluggable Authentication Modules (PAM)

PAM, or Pluggable Authentication Modules, is a highly modular framework within Linux that provides a standardized Application Programming Interface (API) for applications to handle various authentication-related requests. These requests include user authentication, account management, password changes, and session management.[15] A key strength of PAM lies in its ability to abstract the underlying authentication mechanisms. This means that applications can utilize diverse authentication backends—such as local Unix password databases (pam_unix), Kerberos (pam_krb5), or LDAP directories (pam_ldap)—without requiring the application itself to possess intimate knowledge of these backend implementations.[15]

PAM configurations are defined in service-specific files, typically located in the /etc/pam.d/ directory.[15] Each of these files specifies a "stack" of PAM modules that the framework will process sequentially for a given service. This modularity allows system administrators to customize authentication policies by simply adding, removing, or reordering modules within these configuration files.

How Polkit Leverages PAM for Authentication

While Polkit's primary role is *authorization*—determining *if* a privileged action is permissible based on defined policies—it offloads the actual *authentication* process (i.e., verifying the user's credentials, typically a password) to the PAM system.[5] This clear division of labor is a

fundamental aspect of the system's security architecture.

The libpolkit-agent-1 library, which is used by Polkit authentication agents, provides an abstraction layer that interfaces with the native authentication system. The pam(8) man page is explicitly cited as an example of such a native system.[5] This library facilitates the registration and communication with the Polkit D-Bus service, allowing the authentication agent to present prompts and relay credentials.

Polkit maintains its own dedicated PAM service file, commonly found at /etc/pam.d/polkit-1.[16] This file specifies the precise sequence of PAM modules that Polkit will invoke to authenticate users when a password prompt is triggered. For instance, it frequently includes modules like pam_unix.so for local password verification against /etc/passwd and /etc/shadow, and potentially pam_wheel.so for checking administrative group memberships.[15]

When a Polkit authentication agent displays a password prompt to the user, it is effectively initiating a PAM "conversation." This conversation involves PAM sending challenges to the application (in this case, the authentication agent), and the application responding with user input.[15] PAM then processes these credentials through its configured stack of modules, verifying them against the relevant backends.

The architectural decision to decouple authorization from authentication is a cornerstone of robust security design. This separation offers several significant advantages:

1. **Flexibility in Authentication Backends:** By relying on PAM, Polkit can seamlessly support a diverse array of authentication methods, including local passwords, LDAP, Kerberos, smart cards, and even biometric authentication, without needing to implement or understand the intricacies of each method internally.[15] This is particularly beneficial for enterprise environments that often utilize heterogeneous authentication infrastructures.
2. **Clear Separation of Concerns:** Polkit concentrates solely on enforcing *policy* (who can perform what action, under what conditions), while PAM is dedicated to *credential verification*. This distinct separation enhances system maintainability, simplifies security auditing, and reduces complexity.
3. **Enhanced Security Hardening:** Any potential vulnerabilities discovered within a specific authentication backend are confined to that particular PAM module. This design limits the potential impact of such vulnerabilities, preventing them from compromising the entire authorization framework.

The specific PAM modules included in /etc/pam.d/polkit-1 [17] and their defined order directly influence the authentication process. For example, if a distribution configures its PAM stack to prioritize pam_rootok.so (a module that might allow root to authenticate without a password under specific conditions) or pam_wheel.so [16] (which checks for membership in the wheel group), the behavior of the password prompt can change considerably. This explains why some distributions, such as SUSE, might default to requiring the root password for sudo or pkexec operations, while others like Fedora or Ubuntu might prompt for the user's password.[14] This

variability underscores that the "root password" requirement is not an inherent, hardcoded behavior of pkexec but rather a configurable policy choice implemented through the flexible PAM framework.

User Space vs. Kernel Space Implementation

Defining User Space and Kernel Space in Linux

Modern computer operating systems, including Linux, employ virtual memory to establish distinct address spaces known as **user space** and **kernel space**.[7] This fundamental separation is crucial for memory protection and hardware protection, safeguarding the system from malicious or erroneous software behavior.

Kernel space is a highly privileged memory region exclusively reserved for the operating system kernel, its extensions, and most device drivers. Code executing in kernel space operates at the highest privilege level (e.g., CPU ring 0 on x86 architectures) and possesses direct, unrestricted access to the system's hardware resources.[7]

Conversely, **user space** (often referred to as userland) is the memory area where application software, standard libraries (such as glibc and PAM), system daemons (like systemd, polkitd, and sshd), and user shells execute. Each user-space process typically runs within its own isolated virtual memory space. This isolation is the foundation for memory protection and privilege separation, preventing one application from directly accessing or corrupting the memory of another.[7] User-space code communicates with the kernel and requests privileged operations exclusively through well-defined **system calls**.[8]

The pkexec and Polkit Stack: A User Space Paradigm

The entire pkexec authorization and authentication stack, from the pkexec binary itself to the polkitd daemon and the various authentication agents, operates predominantly within **user space**.[2]

- pkexec is fundamentally a user-space application.[1]

- polkitd, the central authority for authorization, is a system daemon that runs as a user-space process.[2]
- Authentication agents, responsible for displaying password prompts, are also user-space processes, often integrated with graphical desktop environments.[2]
- PAM, the Pluggable Authentication Modules framework, is implemented as a user-space library.[15]

The communication between these various components—for instance, pkexec initiating a request to polkitd, or polkitd interacting with an authentication agent—is facilitated by D-Bus. D-Bus is an Inter-Process Communication (IPC) mechanism that functions entirely within user space, enabling secure and structured message exchange between applications and system services.[3] This architectural design confirms that the core decision-making logic for pkexec's password prompt, encompassing both authorization by Polkit and authentication by PAM, resides exclusively in user space.

The Role of the SUID Bit in User Space Privilege Elevation

Despite the fact that the authorization and authentication logic is implemented in user space, the pkexec binary itself is a SUID program.[4] This means that when an unprivileged user executes pkexec, the kernel temporarily elevates the process's effective user ID to that of the file owner, which is typically root.

This kernel-level enforcement of the SUID bit is crucial because it grants pkexec (now running with root privileges) the necessary authority to securely initiate communication with the polkitd daemon. Since polkitd also runs with elevated privileges, this initial elevation of pkexec ensures that the D-Bus communication channel can be established and trusted.[3] Without this mechanism, an unprivileged pkexec would be unable to reliably request authorization from a privileged system service.

The kernel's role in this specific scenario is thus limited but essential: it enforces the SUID bit, allowing pkexec to change its effective user ID to root. Beyond this initial privilege change, the kernel primarily provides the fundamental operating system services—such as process scheduling, memory management, file system access, and system calls—that user-space components like pkexec, polkitd, and PAM libraries rely upon. The complex logic of *why* a password is needed, *which* password is required, and *how* that password is verified is explicitly handled in user space. This demonstrates a clear and effective separation of concerns between low-level privilege enforcement (a kernel function) and high-level security policy and authentication (implemented in user space).

Configuring pkexec and Polkit Behavior

Default Password Requirements Across Distributions

The behavior of pkexec concerning password prompts is not universally uniform across all Linux distributions; variations can be observed.[14] For instance, while most distributions, including Fedora, typically configure sudo and pkexec to request the user's own password by default, certain distributions, such as SUSE and its derivatives, might default to requiring the root password for sudo operations.[14] This divergence in default behavior is largely attributable to the specific configurations of Polkit and its interaction with PAM.

The auth_admin setting, commonly found as a default in Polkit policies [5], typically signifies that an administrative user's password is required. The precise definition of "administrative user" is often determined by the underlying PAM configuration. This might involve PAM modules checking for membership in specific administrative groups (e.g., wheel or sudo groups via pam_wheel.so [16]) or directly authenticating against the root account.

Customizing Authorization Rules for pkexec

System administrators possess considerable flexibility in customizing pkexec's behavior by modifying Polkit's authorization rules. This customization is primarily achieved by creating or editing JavaScript .rules files within the /etc/polkit-1/rules.d/ directory.[2] These locally defined rules are designed to take precedence over the system-provided default rules, allowing for precise policy adjustments.[2]

Examples of how pkexec's behavior can be customized include:

- **Requiring the Root Password for sudo and pkexec:** While sudo can be configured via the sudoers file (e.g., using defaults rootpw [14]), pkexec's behavior is adjusted through Polkit rules. An administrator might create a JavaScript rule that explicitly sets the authorization requirement for pkexec actions to auth_admin or even auth_admin_keep, overriding any less restrictive defaults.[5]
- **Allowing Specific Groups Password-Less Execution:** Rules can be crafted to permit

members of designated groups (ee.g., a "superuser" or "owner" group) to execute certain pkexec actions without a password prompt. This is accomplished by defining specific authorization rules in /etc/polkit-1/rules.d/ that check for group membership and then set the authorization result to yes or auth_self_keep.[2]

- **Enabling Graphical (X11) Applications:** By default, pkexec does not allow running X11 applications due to environment variable stripping.[1] To enable this, a specific annotation, org.freedesktop.policykit.exec.allow_gui, must be set to a non-empty value (e.g., true) in the relevant Polkit action policy file.[1]

The ability to customize pkexec's behavior through JavaScript rules highlights a high degree of administrative control and enables dynamic policy enforcement. This moves beyond static permissions to a system where access can be granted or denied based on a variety of dynamic contexts. These contexts can include:

1. **User/Group Membership:** Policies can be tailored to grant privileges only to specific users or members of particular groups.[2]
2. **Session Type:** The system can differentiate authorization requirements based on whether a user is in an active local graphical session, an inactive local session, or a remote session (e.g., via SSH).[5]
3. **Specific Action Parameters:** Although pkexec itself does not validate arguments passed to the program, Polkit rules could potentially check annotations related to specific action parameters.[1]

This inherent flexibility empowers system administrators to fine-tune security policies to align with specific organizational needs, thereby striking a balance between stringent security measures and practical usability. This explains why pkexec's default behavior can be overridden to suit the unique requirements of diverse computing environments.[13]

Security Considerations and Best Practices

Environment Variables and X11 Applications

For security reasons, pkexec by default executes programs in a "minimal known and safe environment".[1] This measure is designed to mitigate potential code injection vulnerabilities that could arise from malicious environment variables, such as LD_LIBRARY_PATH. A direct consequence of this security posture is that pkexec will not, by default, permit the execution of

X11 (graphical) applications as another user. This limitation occurs because essential environment variables like $DISPLAY and $XAUTHORITY, which are critical for X11 applications to function, are not set in this minimal environment.[1]

To enable pkexec to run X11 applications, the org.freedesktop.policykit.exec.allow_gui annotation must be explicitly set to a non-empty value in the corresponding Polkit action policy file.[1] However, this configuration is generally discouraged due to the security implications it introduces and is primarily intended for compatibility with legacy programs.[1]

This design choice reflects a fundamental principle in security engineering: balancing security with usability. By default, pkexec prioritizes a highly restricted environment to minimize potential attack vectors. The option to allow_gui exists to accommodate specific use cases and maintain compatibility, but it comes with a clear warning, indicating that this expands the potential attack surface (e.g., through vulnerabilities in X11 itself). This illustrates the constant tension and necessary trade-offs between strict security enforcement and practical system usability in software design.

Distinction from Mandatory Access Control (MAC) Frameworks (SELinux/AppArmor)

It is critical to differentiate the pkexec/Polkit/PAM stack from Mandatory Access Control (MAC) frameworks such as SELinux and AppArmor.[19] While pkexec, Polkit, and PAM collectively manage **authorization** (who is permitted to perform an action) and **authentication** (verifying a user's identity), MAC frameworks provide an additional, **kernel-level** layer of **access control**.[19]

SELinux and AppArmor enforce policies that dictate *what a process can do* (e.g., access specific files, interact with particular network ports) *after* that process has already been authenticated and authorized. These MAC frameworks operate independently of the authentication flow itself.[19] SELinux employs a label-based security model, assigning security contexts to all system objects (files, processes, directories), while AppArmor utilizes a path-based model, applying security profiles directly to individual applications.[19] Both are implemented as Linux Security Modules (LSMs) within the kernel, providing a robust, system-wide enforcement mechanism.[19]

This clear delineation of security layers is important for understanding the overall security posture of a Linux system. By explicitly stating that SELinux and AppArmor are not directly involved in the password prompt or the authentication flow, it clarifies that the core mechanism for pkexec's password requirement is indeed located in user space. Instead, MAC frameworks serve as complementary security layers, providing defense-in-depth by restricting the actions of

even authorized and authenticated processes. They are distinct in their function from the pkexec/Polkit/PAM stack, which focuses on the initial authorization and authentication for privilege escalation.

Conclusion

The requirement for a root password when calling pkexec from a lower-privileged account in modern Linux distributions is a sophisticated, multi-layered implementation that resides predominantly in **user space**. The process is initiated by the pkexec utility, which, as a SUID binary, gains temporary root privileges to securely communicate with system services.

The core of this mechanism is the **Polkit authorization framework**. Polkit, through its polkitd daemon and user-session-specific authentication agents, evaluates intricate authorization rules defined in XML .policy files (for defaults) and JavaScript .rules files (for overrides and dynamic policy enforcement). When these rules mandate authentication (typically auth_admin), Polkit delegates the actual credential verification to the **Pluggable Authentication Modules (PAM)** system. PAM, in turn, utilizes its own highly configurable stack of modules (defined in /etc/pam.d/polkit-1) to interact with various authentication backends, verifying the user's password.

The kernel's role in this entire process is limited but crucial: it facilitates the initial privilege elevation of pkexec by enforcing the SUID bit. Beyond this, all complex authorization logic, the display of authentication prompts, and the handling of user credentials occur within user space, with D-Bus serving as the secure inter-process communication backbone.

This modular, user-space design offers significant advantages for system administrators, providing unparalleled flexibility to customize privilege escalation policies based on user, group, session context, and specific actions. This architecture represents a powerful and adaptable component of modern Linux security, allowing for fine-grained control that balances security requirements with operational usability. A thorough understanding of this architecture is indispensable for effective system administration, troubleshooting, and security hardening in contemporary Linux environments.

Works cited

1. pkexec: polkit Reference Manual - Freedesktop.org, accessed June 22, 2025, https://polkit.pages.freedesktop.org/polkit/pkexec.1.html
2. The Polkit authentication framework | Security and Hardening Guide | openSUSE Leap 15.6, accessed June 22, 2025, https://doc.opensuse.org/documentation/leap/security/html/book-security/cha-security-polkit.html

3. Privilege escalation with polkit: How to get root on Linux with a seven-year-old bug, accessed June 22, 2025, https://github.blog/security/vulnerability-research/privilege-escalation-polkit-root-on-linux-with-bug/
4. [SOLVED] pkexec from Polkit allows any user to gain root privs - Arch Linux Forums, accessed June 22, 2025, https://bbs.archlinux.org/viewtopic.php?id=303523
5. polkit - Authorization Framework - Ubuntu Manpage, accessed June 22, 2025, https://manpages.ubuntu.com/manpages/jammy/man8/polkit.8.html
6. polkit Reference Manual - Freedesktop.org, accessed June 22, 2025, https://www.freedesktop.org/software/polkit/docs/latest/polkit.8.html
7. User space and kernel space - Wikipedia, accessed June 22, 2025, https://en.wikipedia.org/wiki/User_space_and_kernel_space
8. What is the difference between user-space and kernel-space program/application? - Unix & Linux Stack Exchange, accessed June 22, 2025, https://unix.stackexchange.com/questions/796127/what-is-the-difference-between-user-space-and-kernel-space-program-application
9. Security and Hardening Guide | The Polkit authentication framework - SUSE Documentation, accessed June 22, 2025, https://documentation.suse.com/sles/15-SP5/html/SLES-all/cha-security-polkit.html
10. DBus and Polkit Introduction - Blog, accessed June 22, 2025, https://u1f383.github.io/linux/2025/05/25/dbus-and-polkit-introduction.html
11. Writing polkit Authentication Agents - Freedesktop.org, accessed June 22, 2025, https://www.freedesktop.org/software/polkit/docs/latest/polkit-agents.html
12. command line - How to configure pkexec? - Ask Ubuntu, accessed June 22, 2025, https://askubuntu.com/questions/287845/how-to-configure-pkexec
13. MX Linux (and other Debian distros) Getting rid of the password prompt for sudo and pkexec, accessed June 22, 2025, https://forum.puppylinux.com/viewtopic.php?t=10396
14. Password for sudo and pkexec - Fedora Discussion, accessed June 22, 2025, https://discussion.fedoraproject.org/t/password-for-sudo-and-pkexec/79143
15. How do programs use PAM for Authentication? : r/linux - Reddit, accessed June 22, 2025, https://www.reddit.com/r/linux/comments/1bxvcbq/how_do_programs_use_pam_for_authentication/
16. Configure PAM to only use fingerprint authentication when lid is open - Arch Linux Forums, accessed June 22, 2025, https://bbs.archlinux.org/viewtopic.php?id=292000
17. Polkit-122 - Linux From Scratch!, accessed June 22, 2025, https://www.linuxfromscratch.org/~ken/inkscape-python-deps/blfs-book-sysv/postlfs/polkit.html
18. PAM Integration - Yocto Project Wiki, accessed June 22, 2025, https://wiki.yoctoproject.org/wiki/PAM_Integration
19. AppArmor vs SELinux: Compare the Differences in Linux Security, accessed June

22, 2025, https://tuxcare.com/blog/selinux-vs-apparmor/
20. Securing Linux with SELinux (or AppArmor) - LinuxBlog.io, accessed June 22, 2025, https://linuxblog.io/securing-linux-selinux-apparmor/