Linux Security Modules and the Current Threat Landscape (2025)

Researcher: Gilberto G. – on *A*xiom*A*dvanced Project: Open Tech Research and Experimental Systems axiomadvanced.com

Research Initiated: October 02, 2025

Introduction and Background

Linux has become a cornerstone of modern computing across servers, desktops, containers, and even critical infrastructure. With this ubiquity comes a pressing need for strong security mechanisms. By design, Linux traditionally used Discretionary Access Control (DAC) – where owners control file permissions – but DAC alone proved insufficient against sophisticated threats like trojan horses and exploits[1]. Over the past two decades, the Linux community and researchers have responded by developing **Linux Security Modules (LSMs)**, a kernel framework for enforcing security policies beyond DAC. Introduced in the early 2000s, the LSM interface allows pluggable security modules to mediate operations (file access, process execution, etc.) through hooks in the kernel. This enabled the integration of Mandatory Access Control (MAC) systems such as **Security-Enhanced Linux (SELinux)** and others into the mainline kernel. SELinux, initially released by the U.S. NSA, was merged into Linux and quickly became a flagship LSM, supported in multiple distributions by the mid-2000s[2]. Around the same time, alternative MAC LSMs like **AppArmor**, **Smack**, and **TOMOYO** were developed to offer different security models. These security enhancements were driven by the recognition that kernel-level controls are needed to contain threats that can bypass or abuse DAC. As Chen, Li, and Mao (2017) observe, DAC mechanisms are vulnerable to malware and exploitable bugs, whereas MAC-enabled systems can potentially prevent certain attacks – for example, limiting an attacker's ability to install rootkits or persist on a host[3][4].

In parallel, the threat landscape has evolved significantly. Today’s attacks on Linux systems commonly include **local privilege escalation** exploits, **file tampering/persistence** mechanisms, and stealthy **kernel rootkits**. These are especially relevant in scenarios ranging from multi-

tenant containers in the cloud to general-purpose desktop systems and industrial control systems. This research report provides an in-depth review of Linux's security modules (both mature and experimental) and examines how they address current popular attack vectors. We focus on technical details of mainline LSMs (with SELinux as a prime example) as well as new developments like eBPF-based security. I will also survey the prevailing threats – privilege escalation, unauthorized file modification, and rootkits – highlighting how Linux's security features and recent academic research tackle these challenges. The goal is to offer a cutting-edge overview (circa late 2025) of the Linux security ecosystem, including capabilities, usage contexts (containers, desktops, critical infrastructure), and references to academic literature and CVE data to ground the discussion.

Table of Contents

Linux Security Modules: Mainline Mechanisms

Linux Security Modules (LSMs) provide a generalized framework for the kernel to invoke security checks. When certain operations occur (opening a file, executing a program, etc.), the kernel calls LSM hook functions; an active module can then approve or deny the action based on its policy. This design, upstream since Linux 2.6, allowed integrating multiple security models without hard-coding them into the kernel[1]. Below we outline the major LSMs that have been incorporated into the Linux kernel, as well as their roles and adoption:

- **Security-Enhanced Linux (SELinux):** SELinux is the most widely deployed LSM and implements a form of mandatory access control known as **Type Enforcement** (with optional multi-level security). Under SELinux, every process and object (file, socket, etc.) has a *security context* label, and a central policy rules which labels (types) can access which. SELinux policies are extremely granular, enabling the principle of least privilege for each daemon or user action[5][6]. In practice, SELinux is enabled by default in Red Hat Enterprise Linux (RHEL) and Fedora (using a *targeted* policy that confines core services)[7]. It's also used on Android devices to sandbox apps. When running in **enforcing mode**, SELinux can outright block unauthorized actions that would be allowed by DAC, greatly reducing the post-exploit capabilities of malware. For example, if an Apache web server process is compromised, SELinux policy can prevent it from reading files outside web content directories or executing unexpected binaries, thereby containing the damage. SELinux's comprehensive control comes at the cost of complexity – the policies have thousands of rules, and misconfiguration can disrupt system functionality. Nonetheless, its granularity is unmatched; SELinux can even mediate kernel interfaces like /proc and /sys, and it supports security contexts over network objects and Linux capabilities. SELinux is a **label-based** MAC; it requires files and processes to be labeled with types, and administrators must maintain consistent labeling (often assisted by tools and default policies). Despite a learning curve, it's considered a cornerstone of Linux hardening[8][5]. As of 2025, SELinux remains actively maintained and is seeing use in container environments (e.g. Docker on Fedora/RHEL uses SELinux to isolate containers via the "svirt" system) and even in virtualization (sVirt for VM image separation)[9]. Academic analysis (Chen et al., 2017) has explored the "quality of protection" SELinux

offers; for instance, how effectively a given SELinux policy can prevent specific attack paths like installing a rootkit or leaving behind a trojan[10][11]. Those studies underscore that SELinux's strength ultimately depends on the policy in use – a strict policy can thwart many lateral movements, whereas a permissive or misconfigured policy might not add much security[12][13].

- **AppArmor:** AppArmor is another major LSM, included by default in Ubuntu, SUSE, and others. It also implements MAC, but uses a **pathname-based** approach: policies refer to file paths rather than labels. AppArmor profiles confine programs by specifying which files (paths) they can read, write, or execute, as well as other restrictions (network access, capabilities, etc.). AppArmor is often considered easier to manage for administrators unfamiliar with SELinux – one can craft a profile by listing allowed resources for a service. Ubuntu has leveraged AppArmor to sandbox user applications (e.g., the Snap packaging system uses AppArmor and seccomp to confine snaps)[14][15]. For example, a PDF reader installed as a Snap on Ubuntu will run under a tailored AppArmor profile that limits it to only read user's Documents but not system files, and to use only certain syscalls (thanks to seccomp). AppArmor's path-based model means it cannot distinguish two files with the same path in different mount contexts, and it trusts the filesystem path name (which has some corner cases like hard links). Nevertheless, it provides significant containment and has a simpler learning curve. Comparatively, SELinux offers more granular control (distinguishing objects by label even if paths change) but requires more upfront policy development[16][17]. Both SELinux and AppArmor are actively used; some distros choose one or the other. Notably, AppArmor can be easier to use in development or on desktops (where users might set custom profiles for apps), whereas SELinux's robust enforcement is often seen in enterprise and server settings.

- **Smack (Simplified Mandatory Access Control Kernel):** Smack is a lighter-weight MAC LSM that, like SELinux, uses labels on objects, but with a simpler rule set defined in terms of label "rules" (typically a few default labels like "System" and "User"). It was integrated in Linux around 2008. Smack is used in certain embedded Linux systems – for instance, it was the default MAC in Intel's Yocto project and in Tizen (a Linux-based OS for devices). Smack's philosophy is to be straightforward: each subject and object has a

Smack label, and a simple set of rules (stored in smackfs) decides access (often defaulting to "only same label can access same label, unless rules allow"). This simplicity makes Smack suitable for appliances or systems that don't need the full complexity of SELinux. In critical infrastructure devices or automotive systems, Smack provides isolation between processes (e.g., a process labeled as "EngineControl" can be prevented from accessing "Infotainment" data). While not as widely adopted as SELinux/AppArmor, Smack's presence in the mainline kernel and its usage in industry (e.g., embedded IVI systems) make it a relevant security module.

- **TOMOYO Linux:** TOMOYO is another LSM (mainlined in 2009) focused on path-based access control, somewhat like AppArmor but with an emphasis on ease of policy generation. TOMOYO can automatically learn a program's behavior and generate a policy for it. It uses domain transitions based on program execution history. Although TOMOYO has a niche user base (primarily developed in Japan and used in some Asian Linux distributions), it demonstrates the variety of approaches to MAC available under the LSM framework. TOMOYO's learning mode can be useful on general-purpose systems to gradually build a policy that fits the system's normal usage. However, outside of specialized circles, TOMOYO is less commonly deployed than the above modules.

- **Yama:** Yama is a simpler LSM (upstream since Linux 3.4) that isn't a full MAC system but rather a set of specific security restrictions – primarily it can restrict the use of `ptrace` on the system. By default on many Linux distributions, Yama is enabled to disallow one process from ptrace-attaching to another unless certain criteria are met (like being parent/child or explicitly authorized)[18]. This prevents a common attack technique where a malicious process debugs another process to read its memory or inject code. Yama also includes optional restrictions on creating hard links to files one doesn't own (to mitigate some TOCTOU attacks). Yama is an example of a *minor LSM* that can be stacked alongside a major one – e.g., you can have SELinux (major LSM) and Yama (secondary) active together, thanks to the Linux kernel's support for stacking multiple LSMs.

- **Integrity Measurement Architecture (IMA) and EVM:** These are LSM components focused on file integrity rather than access control. **IMA** can measure (hash) files upon

load or execution and can enforce policies like “for this file, require it matches a known good hash or a signature before allowing execution.” **EVM** (Extended Verification Module) works with IMA to ensure that file metadata (like security labels or hashes stored as extended attributes) haven’t been tampered with. In effect, IMA/EVM provide a way to detect or prevent unauthorized modifications to important files – a file tampering defense at the OS level. For instance, on a high-security system, IMA-appraisal can be set to refuse execution of any binary that isn’t signed by a trusted key, or to refuse access to configuration files if their hash doesn’t match what was recorded at boot. This is highly relevant for critical infrastructure and servers: even if an attacker gains access, if they alter a protected file (say a critical system binary), IMA can block it from being used. IMA and EVM are part of the kernel’s arsenal against file tampering and persistence, although they require careful setup (e.g., maintaining a database of hashes or keys). They are commonly used in conjunction with secure boot and SELinux for a multilayered security approach. For example, a secure Linux appliance might use secure boot to ensure the kernel is trusted, IMA/EVM to ensure binaries and scripts haven’t been altered, and SELinux to enforce that processes only do what they’re supposed to.

- **Kernel Lockdown:** The *Lockdown* LSM (merged in Linux 5.4, 2019) is a bit unusual – it’s not a full security policy by itself, but a mode that restricts even the **root** user from performing certain dangerous actions on the running kernel. Lockdown was introduced to strengthen the separation between user space (even UID 0) and the kernel when certain conditions are met (such as UEFI Secure Boot being enabled)[19][20]. When **Lockdown mode** is active (it can be set to “integrity” or “confidentiality” levels), the kernel will refuse operations that could allow tampering with the kernel or reading sensitive data from it. For example, Lockdown disables access to */dev/mem*, */dev/kmem*, and other memory interfaces[20]. It forbids writing to Model-Specific Registers (MSRs) which could alter CPU state[21], blocks use of kernel debuggers or any facility that could modify running kernel code, and requires that any kernel modules loaded must be digitally signed[22]. Even kexec of a new kernel image is disallowed unless the image is signed[23]. In essence, Lockdown is a safety catch: it closes off avenues that root could traditionally use to compromise the kernel (accidentally or maliciously). Many distributions now enable Lockdown automatically when Secure Boot is enabled, to

prevent an attacker who gains root on a machine from then loading unsigned malicious kernel code. This feature directly complicates kernel-level rootkits (discussed later), since commonly, rootkits might try to load a malicious module or write to kernel memory – Lockdown blocks those techniques[22][24]. Administrators of high-security systems are encouraged to keep Lockdown mode on (it can be turned off via a boot parameter if truly needed for debugging, but that weakens security).

- **Other Mainline LSMs:** Over the years, Linux has gained a few more built-in security modules for niche purposes. For example, **LoadPin** is an LSM that ensures kernel modules are only loaded from a designated filesystem (to prevent an attacker from loading a module from e.g. an untrusted USB drive). **SafeSetID** is an LSM that restricts which user IDs a process can setuid() to (important in multi-user systems or container environments to prevent user ID abuse). These smaller LSMs often complement the bigger MAC frameworks. Starting with Linux 5.x, the kernel allows multiple LSMs to be enabled simultaneously (with some caveats); typically one *major* LSM like SELinux or AppArmor, plus any number of *secondary* LSMs like Yama, SafeSetID, LoadPin, and Landlock can run together. This stacking provides defense in depth – e.g., an Ubuntu system might run AppArmor for general policy confinement, Yama to restrict ptrace, and LoadPin to enforce module loading policy, all at once.

LSM Configuration and Use: On most Linux distributions, the choice of LSM and its configuration is made by the vendor: e.g., RHEL/Fedora use SELinux enforcing by default, Ubuntu/Debian use AppArmor (with SELinux available as an option, but not default), and some embedded or special systems use Smack. The kernel boot parameter `lsm=` can be used to specify which modules to initialize and their order. A global `/sys` interface shows which LSMs are active. System administrators should be aware of the active LSM, since it will affect system behavior and logs. In practice, a common workflow is: **leave the default LSM enabled**, and if customizing, use distribution-provided tools to adjust policies (for SELinux, using `semodule` or booleans; for AppArmor, editing profile files in `/etc/apparmor.d/`). Disabling an LSM (like setting `selinux=0`) is generally discouraged unless one has a specific reason, as it removes an important layer of defense. Modern software is increasingly developed with LSMs in mind – for example,

container platforms and sandboxed applications often ship with the appropriate AppArmor/SELinux profiles or policy snippets to integrate into the system's security policy.

Emerging and Experimental Security Modules

Linux's security is a moving target, and in recent years new experimental modules and features have appeared, often inspired by research or the needs of containerized and cloud-native environments. Below are some of the cutting-edge developments in 2024–2025 regarding Linux security modules:

- **Landlock LSM:** Landlock is a novel security module introduced in Linux 5.13 (2021) that enables *unprivileged sandboxing*. Designed by Mickaël Salaün, Landlock allows even a non-root process to create a security sandbox restricting its future actions. This is a departure from traditional LSMs, which generally require root privileges to configure policies. Landlock focuses on a subset of controls, mainly file system access rights. A process can use the Landlock API to willingly drop access to certain files/directories (e.g., "this process and its children can only read/write within ~/Downloads and nowhere else"). Because it's stackable, Landlock's restrictions apply in addition to the global security policy (SELinux/AppArmor)[25][26]. The use case is to sandbox applications in a finer-grained way – for instance, a browser could sandbox its tab processes to only allow network and temp-file access, or a package manager could ensure a plugin only touches certain paths. Landlock is still evolving; since its merge, more access types have been added (like network socket control is a planned addition). **Notably, a significant vulnerability in Landlock was discovered in 2024**: it was found that via a combination of system calls (`fork()` and a specific `keyctl()` operation), a process could escape its Landlock-imposed restrictions[27][28]. Essentially, Landlock was not tracking credential changes in one code path, allowing an attacker to regain full access. This hole (present since Landlock's introduction) was patched in Linux 6.11 in mid-2024 once discovered by Jann Horn[29][30]. The incident underlines that new security mechanisms can have teething issues. With that fix applied, Landlock is again considered a useful (though still limited) sandboxing tool. By 2025, adoption of Landlock is nascent – developers must explicitly use its API in programs. It's an area to watch as more software may start self-confining via Landlock for defense in depth.

- **eBPF and KRSI (BPF-LSM):** Perhaps the most significant recent advancement is the ability to write security policies in eBPF. eBPF (extended Berkeley Packet Filter) is a technology that allows safe, sandboxed programs to run inside the kernel, originally used for networking and tracing. The **Kernel Runtime Security Instrumentation (KRSI)** project, upstreamed around 2019–2020, introduced hooks to attach eBPF programs to LSM events[31][32]. In practice, this is called **BPF-LSM**. It means one can dynamically load an eBPF program that runs whenever a security-sensitive event happens (file open, exec, etc.), and that program can make a decision (allow or deny, or just log). This essentially enables *writing custom security modules without rebuilding the kernel*. By 2022, major distributions began shipping kernels with BPF-LSM enabled by default[33][34]. Google and others pushed this technology to allow more flexible security policies that can adapt at runtime. One concrete use is **KubeArmor**, an open-source tool that applies security policies in Kubernetes clusters. Initially, KubeArmor leveraged AppArmor/SELinux to enforce policies on containers, but it found that managing SELinux in highly dynamic Kubernetes environments was challenging (especially since not all clusters have the same LSM, and writing SELinux policies requires expertise)[35]. The breakthrough was when BPF-LSM support became widespread – by end of 2022, over 80% of Linux cloud distros had BPF-LSM available[33][34] – KubeArmor switched to primarily use eBPF for enforcement. The BPF-LSM approach compiles user-specified rules into eBPF programs that execute at kernel hooks[36]. This not only eases deployment (no need to manage SELinux policies on each node manually), but also allows stacking – BPF-LSM can enforce alongside SELinux/AppArmor, providing multiple layers of defense[37]. For example, a host could keep SELinux enforcing for base system isolation, but KubeArmor injects a BPF program to further restrict particular container behaviors (say, block access to certain file patterns that SELinux policy might have allowed). The performance overhead of BPF-LSM is minimal (eBPF programs are JIT-compiled in the kernel), and it offers unparalleled flexibility – policies can be updated on the fly. We are effectively seeing the beginning of a new paradigm: **Policy as Code** for Linux security. Instead of only static policies written in SELinux's policy language or AppArmor profiles, security teams can write higher-level rules (even in a language like C or using frameworks) that get translated to eBPF and enforced kernel-

side. Cisco's acquisition of Isovalent (a company behind eBPF technologies like Cilium) in 2023 highlighted industry interest in this area[38][39]. One must note, however, that eBPF itself must be treated with caution – if an attacker gains the ability to load arbitrary eBPF, that is dangerous (hence by default only root with the right capabilities can load eBPF code). But in controlled fashion, eBPF is empowering defenders. Research and industry projects (like **KRSI/BPF-LSM**, **Cilium** for networking, **Tracee** for runtime detection) are leveraging it for security monitoring and enforcement. BPF-LSM is likely to complement or even eventually *replace some uses of traditional LSMs*, especially in cloud/container scenarios[37][40]. For now, it coexists: e.g., Kubernetes 1.25+ allows setting AppArmor/SELinux profiles on pods *or* using a tool like Cilium or KubeArmor to enforce rules via eBPF.

- **Security Namespaces (Experimental):** A notable line of research (Sun et al., 2018) tackled the limitation that containers all share the single global LSM policy. In a multi-tenant environment, one might wish each container to have its own security module policy – for instance, Container A could have a tailored SELinux policy distinct from Container B. This isn't possible by default, since SELinux and AppArmor were designed as system-global. Sun et al. proposed **security namespaces**, essentially providing separate LSM instances per container (with proper isolation so a container-admin cannot affect the host or others). They implemented a prototype for namespacing IMA (integrity) and SELinux policies for Docker containers[41][42]. The challenge here is to route security decisions to the right policy namespace. Their system would determine which container(s) are affected by an operation and consult the appropriate policy, ensuring one container's policy decision can't compromise another[43][44]. The reported overhead was low (under 0.7% added latency on system calls)[45]. While this work has not (yet) made it into mainline Linux, it has influenced how people think about container security. Some of its concepts have been partially realized through other means – for example, Kubernetes now has the idea of a "security context" for pods, and with KRSI one could imagine each container loading an eBPF policy specific to it. Another related project is **MSpace (Multi-Security Namespace)**, which extends these ideas; the goal is to let containers or pods enforce their own MAC policies without needing privileges on the host. These experimental directions show the future: making Linux security multi-tenant

aware, so that cloud providers could allow their customers (container users) to define policies that apply only within their container, safely.

- **Third-Party Hardening Patches:** Outside of official kernels, there have been long-running projects like **Grsecurity/PaX** (which historically provided an enhanced security patchset including an LSM called **GRKERNSEC** with RBAC controls, plus a suite of exploit hardening features). While Grsecurity is not in mainline and is now commercial, it pioneered techniques like aggressive memory protections (PaX) and fine-grained ACLs for kernel objects. Some of its ideas (e.g., hardened memory allocator, additional LSM hooks) have over time trickled into Linux or inspired equivalents in the Kernel Self Protection Project (KSPP). Administrators of extremely high-threat environments sometimes run custom kernels with such patches, but one must weigh the maintenance cost and compatibility issues. In the context of this report, we focus on upstream solutions, but it's worth noting the existence of such experimental forks which often inform mainstream development.

- **Ongoing Enhancements:** The landscape continues to change. As of 2025, efforts are underway to improve LSM "stacking" such that multiple major LSMs could operate fully in parallel (currently, you cannot run SELinux and AppArmor at the same time for full policies – you must choose one at boot, though stacking minor modules is supported). There's also discussion about making LSMs more dynamic (loading/unloading policies at runtime more easily). Another area is **synergy with hardware security** – for instance, using TPMs (Trusted Platform Modules) with IMA to remote attest the integrity of a system, which is crucial for critical infrastructure that demands trust verification. The academic and open-source communities are actively publishing research on strengthening the Linux kernel against attacks; some focus on control-flow integrity (CFI) in kernel, others on detecting anomalies via machine learning. We will touch on some of these in the threats section.

In summary, Linux today has a rich toolkit of security modules. The mainline LSMs like SELinux and AppArmor are mature and widely used to enforce strong access controls. Newer modules like Landlock and BPF-LSM address emerging needs (unprivileged sandboxing and cloud-native dynamic policy enforcement). Together, these mechanisms create a layered defense.

However, the efficacy of each layer depends on proper configuration and keeping pace with kernel updates – as seen, even security modules can have bugs (as with Landlock's 2024 fix) and need timely patching. Next, we turn to how these modules and features help mitigate the most pressing categories of attacks in the current threat landscape: privilege escalation, file tampering, and rootkits.

Security in Different Environments

Before analyzing specific attack types, it's useful to consider how Linux security modules are applied in various contexts:

- **Containerization Environments:** Containers pose a unique challenge because they amplify the principle of least privilege across potentially thousands of ephemeral instances. Out of the box, containers rely on *namespace isolation* (each container has its own view of the filesystem, process list, etc.) and *cgroups* (resource limits). However, these alone don't restrict what a process *inside* the container can do to the kernel – if it's running as root, it could still perform privileged calls unless constrained. Thus, container runtimes employ LSMs and other kernel features to harden containers. On a host with SELinux enabled, each container can be assigned a distinct SELinux context; Docker and Podman on Fedora/RHEL do this using the **sVirt** integration, so that even if a container process breaks out of its namespace, SELinux policy will prevent it from accessing host files or other containers' files (by label separation)[9]. On Ubuntu, Docker defaults to an AppArmor profile ("docker-default") for containers, which confines container processes' capabilities and accessible filesystem paths. Additionally, **seccomp** (Secure Computing mode) is used to filter syscalls: Docker has a default seccomp filter that blocks dangerous system calls (like `kexec_load`, `perf_event_open`, etc.), reducing kernel attack surface from inside containers. **Linux capabilities** are also dropped – by default containers run with a limited set of capabilities (excluding most potent ones like CAP_SYS_ADMIN, which is effectively "root power"). All these combined mean a container is far less powerful than a raw root process on the host. That said, misconfigurations (like running a container with `--privileged` flag, which disables these protections) or kernel vulnerabilities can undermine the isolation. A famous container escape was **CVE-2019-5736**, a runC vulnerability: a malicious container could overwrite the host's `runc` binary and thus execute code on the

host as root[46][47]. This was not a kernel bug, but a flaw in the container runtime. Still, such an attack often requires some level of privileges inside the container (in that case, the attacker needed to run as root in the container). Using SELinux/AppArmor can mitigate the impact – for instance, AppArmor profiles could potentially have prevented writing to the runc binary's path, or SELinux could have labeled the runtime binary in a way that the container process (running with container_type label) couldn't modify it. After CVE-2019-5736, Docker added even more sanity checks and the community emphasized not running untrusted containers as root. Container orchestrators like Kubernetes also integrate with LSMs: one can specify AppArmor profiles for pods, SELinux context for pods, and seccomp profiles. The NSA's **Kubernetes Hardening Guide (2022)** explicitly recommends using these features: *"Hardening applications against exploitation using security services such as SELinux®, AppArmor®, and seccomp"*[48]. Additionally, new tools emerged to make this easier – **KubeArmor** (mentioned above) or **Kyverno** can auto-generate policies, and Open Policy Agent (OPA) Gatekeeper can enforce certain security contexts (like no privileged pods). On the container host side, it's also advisable to enable Yama (to stop ptrace between containers), and to use IMA to ensure container images/binaries aren't tampered (some projects measure container images at load time). Lastly, **user namespaces** provide an extra layer: root inside a user namespace can be mapped to a non-root UID outside, so even if container "root" breaks out, it has no real root privileges on the host. This is used by rootless Docker/Podman and is a strong mitigation against kernel vulnerabilities leading to root, though not all daemons support running completely rootless in all scenarios.

- **General-Purpose Desktops:** Traditional desktop Linux users have not always been as strict with security modules – indeed many would simply disable SELinux if it caused trouble. However, the landscape is changing with the advent of sandboxed apps and an increasing awareness of ransomware and other threats targeting Linux. Ubuntu's use of AppArmor is a case in point: every graphical application shipped as a **Snap package** on Ubuntu runs under an AppArmor profile that confines its file and system access[49][50]. For example, a malicious game downloaded as a snap would be confined from reading /etc/shadow or the user's SSH keys by the snap's AppArmor profile and seccomp filter,

significantly limiting damage if that app were trojanized. Flatpak, another popular app sandboxing mechanism (used on Fedora, etc.), uses Linux namespaces extensively and can integrate with SELinux (on Fedora, Flatpak uses SELinux label sandboxing when available) or just rely on seccomp and filesystem namespacing to limit access. Modern Linux desktop environments also use *Portal* APIs to mediate access (so an app can't directly read files unless via a user-approved dialog). Under the hood, these rely on kernel features: e.g., seccomp to intercept syscalls and notify userland (in Flatpak portals). For the average desktop user, these improvements mean the Linux system is less likely to be completely owned by one compromised application. Still, social engineering and user execution of malware remain a risk, and many desktop users run as uid 1000 with full sudo rights, which is not ideal. Here, AppArmor or SELinux can enforce some policy for user sessions – for example, Fedora has an SELinux boolean to confine user home directories so that a compromised system service can't read user files. Ubuntu (and other Debian-based distros) historically had an "unconfined" profile for the user session, but certain high-risk apps (browsers, etc.) might have sandboxing through other means (Chrome and Firefox use their own sandbox layers with namespaces and seccomp, and can integrate with AppArmor if present). Another emerging desktop threat is **ransomware** targeting Linux. An attacker who runs a malicious binary under the user account could attempt to encrypt all the user's files. LSMs can help here: with AppArmor, one could enforce that, say, the Firefox process is not allowed to modify files under ~/Documents (since a web browser shouldn't normally encrypt your documents). Some Linux users are experimenting with using **AppArmor proactively for ransomware protection** – by setting most directories as read-only for untrusted processes. Such use requires manual tuning and is not widespread yet. On the distribution front, there's a push to make security modules more seamless: Fedora's Workstation edition comes with SELinux enabled (targeted policy) which mostly confines network-facing daemons, not user apps, but that still helps. Ubuntu's approach with AppArmor on snaps is bringing MAC to everyday applications. As the Linux desktop market grows (e.g., with more people using it or Chromebooks which run Linux apps in containers), we can expect more granular use of LSMs to protect user data.

- **Critical Infrastructure and Servers:** In critical environments like industrial control systems (ICS), energy grid, healthcare systems, etc., the stakes are high. Many of these systems run on Linux (or some UNIX-like OS) under the hood. A notable trend in such sectors is adopting **"secure by default" Linux distributions** (like RHEL with SELinux, or Ubuntu Core with snap confinement) and following guidelines like CIS benchmarks or regulatory frameworks (e.g., IEC 62443 for industrial security) that often include enabling MAC. For instance, in a power plant control server running Linux, enabling SELinux in enforcing mode can be crucial – it can ensure that even if an attacker exploits a vulnerability in the SCADA software, they cannot, for example, tamper with system binaries or start unexpected network connections. Government and industry guidelines increasingly mention these Linux protections. The U.S. Cybersecurity & Infrastructure Security Agency (CISA) has released advisories urging organizations to keep systems updated and leverage built-in OS access controls. Even something as simple as limiting which user can sudo or use certain capabilities can make a difference in critical systems – but MAC policies go further by limiting even root's actions according to policy. A historical barrier in ICS was that vendors often shipped devices with SELinux *disabled*, fearing it would interfere with their application (or using older kernels where it wasn't available). This is slowly changing as newer devices and OS versions come in. **Case study:** In 2021, a water treatment facility attack in Florida was an example where an attacker tried to manipulate control software. That system was running an outdated Windows, but hypothetically, on Linux, an LSM could have helped if, say, the water control software was only allowed by policy to open certain device files and not, for example, spawn a shell or modify arbitrary files. Similarly, on telecom routers running Linux, enabling modules like **IMA/EVM** ensures that firmware or config files aren't altered by an attacker without detection. Some critical kernels also use **Lockdown mode** (especially when Secure Boot is in play) to prevent low-level tampering. We also see interest in **Integrity Monitoring** – where a central system monitors the hashes of critical files on various hosts (like tripwire, but extended); LSMs like IMA can assist by maintaining a runtime measurement list. In summary, critical infrastructure Linux systems benefit from the same technologies but often at higher strictness: SELinux in full enforcing with custom policies, AppArmor profiles for each service, lockdown enabled,

IMA requiring signed binaries, etc. The trade-off is that it requires expertise to configure and maintain. From a threat perspective, these systems face targeted attacks (often by advanced threat actors) aiming for rootkits or persistent footholds. Therefore, layering all possible defenses is warranted. Encouragingly, many modern ICS-focused Linux variants (like **Fedora IoT, Ubuntu Core, AWS Greengrass**) do come with such security features on. The hope is that even if an attacker finds a zero-day in an ICS application, the surrounding LSM policy and kernel hardening will contain the blast radius and possibly log the attempt.

Having set the stage for how Linux security modules are deployed, we now delve into the specific classes of attacks and how they are mitigated (or in some cases, insufficiently mitigated) by these security features. We will incorporate relevant real-world CVEs and academic insights for each threat category.

Privilege Escalation Attacks

Privilege escalation refers to an attacker obtaining higher privileges than they initially have on a system. On Linux, this typically means a normal user process gaining root-level access (vertical escalation), or a confined process breaking out of its restrictions. Privilege escalation is a critical step in many attack chains, allowing an adversary to take full control of a system or container. The Linux kernel and OS are frequently the target of local privilege escalation (LPE) exploits, and recent years have seen a surge of disclosed vulnerabilities. In fact, the number of Linux CVEs (many of which enable privilege escalation) skyrocketed after the Linux kernel started its own CVE enumeration process – by late 2024 the kernel was assigning around *55 CVEs per week*, compared to historically ~20 per quarter, indicating many more bugs (including LPE flaws) being discovered and reported[51][52]. For example, **CVE-2024-1086** is a use-after-free bug in the kernel's netfilter subsystem disclosed in early 2024 that allows a local user to escalate to root privileges[53]. It was serious enough to be listed in CISA's Known Exploited Vulnerabilities Catalog shortly after disclosure, as exploits appeared in the wild[54]. Another notorious example was **"Dirty Pipe" (CVE-2022-0847)** – a flaw in the Linux kernel's pipe mechanism that let an unprivileged user write to arbitrary read-only files, which could be

exploited to inject code into root processes or modify sensitive files to gain root[55]. These exemplify the kinds of vulnerabilities attackers love for privilege escalation.

How do Linux security modules help mitigate privilege escalation? It's important to clarify that LSMs like SELinux or AppArmor do not *magically fix kernel bugs* – a memory corruption in kernel space, once exploited, can allow an attacker to execute arbitrary code in the kernel, thereby bypassing LSM checks entirely (since LSM hooks rely on the kernel functioning correctly). However, MAC policies can still limit what an *unprivileged exploit* can do *before* it gains kernel control, and in some cases can contain a compromised service even if root is gained.

1. **Confining services to reduce impact:** Suppose an attacker exploits a web application running as user "www-data". By default, that might let them read/write any file that user can, but not give root. Often though, the next step is to exploit a kernel bug or misconfiguration to get root. If that web service was running under a strict SELinux or AppArmor profile, even if the attacker hasn't gotten root yet, they might find their path to escalation constrained. For example, many kernel LPE exploits need certain conditions: maybe the ability to load a module, or to open some device file, or create specific filesystem types. A good policy might disallow the web service from doing those unusual activities. **Chen et al. (2017)** in their analysis note that a MAC system's benefit is in preventing an attacker from reaching the kernel or other critical assets in the first place[4] [10]. If the MAC policy blocks a step of the attack (say opening a special file needed to trigger a bug), the escalation fails. SELinux targeted policy, for instance, blocks network daemons from using the *kexec* system call, from ptracing other processes, and from reading/writing many important files. These restrictions can incidentally foil certain exploit techniques. AppArmor profiles often include denying access to /proc/kcore or debug interfaces for confined apps.

2. **Limiting root's power (post-exploit):** Consider an attacker who *does* manage to get root (UID 0) in user space – perhaps by exploiting a SUID binary or misconfigured Docker socket. On a traditionally configured Linux system, root can do virtually anything. But with LSMs, *root can be restricted too.* SELinux can be configured with *MLS (Multi-Level Security)* or role-based rules that even root processes are confined to a domain. For example, on an SELinux system, the "init" system and its services run in an *unconfined_t*

domain by default (full root privileges), but one can choose to confine even those. Some high-security setups run most services, even as root, in confined domains that lack access to most of the filesystem. Thus, if an attacker becomes root by breaking a service, they might still be stuck in a limited SELinux domain that cannot, say, read /etc/shadow or modify system binaries. A famous instance of this concept is Android: on Android, **every app is technically running as Linux uid 10000+** and if an app somehow gets kernel code execution and becomes UID 0, it *still* runs in the SELinux domain u:r:kernel:s0 which is very constrained (Android's SELinux policy is designed to prevent even a compromised root from certain actions, relying on the assumption that if the kernel isn't fully compromised the policies hold). This is an extreme scenario, but it shows MAC can make privilege escalation less immediately useful to an attacker. Additionally, **Lockdown mode** (as described) ensures that even root in the init namespace cannot directly modify the running kernel or load unsigned modules[22]. So an attacker with root rights but facing Lockdown might not be able to implant a kernel rootkit via typical means. They would be forced to find a way to *disable* lockdown or find a kernel vulnerability to break it (which is an additional hurdle).

3. **Kernel Self-Protection:** Outside LSMs, the kernel has been hardened with features that make exploitation harder – these are worth noting in the context of priv escalation. **KASLR (Address Space Layout Randomization)**, **stack canaries, SMEP/SMAP on CPUs, and CFI (Control-Flow Integrity)** are examples of features that make it more challenging for an attacker's code to run successfully in kernel space[56][57]. These aren't LSMs, but complementary. Modern kernels have many of these on by default. For instance, if an attacker exploits a kernel bug to get code execution, SMEP might prevent them from directly jumping to a userland shellcode (forcing them to do ROP), and CFI (if enabled) might catch unusual control flow. These mechanisms, combined with LSM policies that restrict access to the most powerful interfaces, collectively raise the bar for privilege escalation.

4. **Patching and CVE Response:** The best mitigation is of course to patch known privilege escalation bugs promptly. With the explosion of kernel CVEs in 2024[51], administrators should pay close attention to Linux kernel updates. Many high-profile LPEs like Dirty

Pipe (2022) and "Sequoia" (CVE-2021-33909, a filesystem bug) were quickly patched in the kernel, but systems that lag on updates remain vulnerable. In enterprise, tools like live patching (kpatch, ksplice) are being used to deploy fixes without downtime. Some Linux distributions have also moved to more frequent security updates in response to the high CVE counts (e.g., Ubuntu's 20.04 and later kernels are getting regular updates from the upstream stable tree which includes many CVE fixes). An interesting statistic from 2025: one report noted a **967% increase in reported Linux vulnerabilities from 2023 to 2024** (313 in 2023 vs. 3,329 in 2024)[58] – though this is largely due to the new CVE counting method, it highlights that many potential escalation paths are being discovered and must be addressed.

In practice, **privilege escalation attacks remain one of the hardest challenges** because a single kernel flaw can undermine many layers of defense. LSMs significantly help by limiting what unprivileged processes can do (potentially preventing some exploits) and by limiting the power of even a successful root exploit. But they are not foolproof – a kernel exploit typically can disable LSM checks entirely if it gains arbitrary write in kernel. This is why system design should include multiple layers: strong LSM policies, kernel hardening, container isolation, and monitoring. Monitoring is crucial: if an attacker tries and fails to do something (like an SELinux denial for an unexpected access), the logs might tip off administrators to an ongoing attack. Many breaches have been detected by unusual AVC (SELinux denial) logs or AppArmor complaints that, upon investigation, revealed malicious activity.

Example CVEs and LSM interplay:

- *CVE-2016-5195 (Dirty COW)* – a famous priv-esc where a race condition in copy-on-write allowed writing to read-only root-owned mappings. SELinux would not directly stop this (since it was a kernel bug in memory management), but systems with SELinux did provide one advantage: on Fedora, for instance, many services run as unprivileged users without sudo rights, so an attacker had to first get on the system as some user. Assuming they did, Dirty COW gave root. Post-exploit, if Lockdown had been available then, it could have prevented loading a rootkit module.

- *CVE-2022-0847 (Dirty Pipe)* – as noted, allowed writing to any file that could be opened for reading. An attacker could use it to append to /etc/passwd or replace a setuid binary to later get root[55]. Here, a combination of traditional DAC and MAC might help: DAC could make sensitive files non-world-readable (so an unprivileged user can't even open them to exploit Dirty Pipe, in many distros /etc/shadow is not readable, but /etc/passwd is world-readable though not highly sensitive since it has no hashes). SELinux in targeted policy doesn't forbid reading most files for unconfined user, so it wouldn't stop the initial step. However, if an attacker was a confined service user (like nginx_t domain in SELinux), that domain might not have access to /etc/passwd at all by policy – thus the exploit couldn't open the file to write to it, blocking the priv esc. This illustrates that a tight MAC policy can sometimes fortuitously block an exploit's requirements.

To summarize, privilege escalation threats are rampant, but Linux's security mechanisms do make a difference. Systems employing MAC, dropping capabilities, using seccomp, and enabling kernel hardening are significantly more resistant. Attackers often go after the "low-hanging fruit" – misconfigured systems or those missing these layers. Even in exploit development forums, attackers note that a certain technique "won't work if SELinux is in enforcing mode" or "needs CAP_SYS_ADMIN in the container, so hope it's privileged." By not giving those advantages to the attacker, we force them to find more complex ways in, hopefully deterring all but the most advanced adversaries. In high-security contexts, one should operate under the assumption that privilege escalation *will* happen and focus on **containment and detection** – LSMs are key for containment, and audit logs (plus tools like Linux Audit or SIEM integrations) are key for detection when something abnormal happens.

File Tampering and Integrity Threats

Once attackers gain a foothold on a system, they often seek to **tamper with files** – whether to implant backdoors, deface websites, modify configurations, or install persistent malware. File tampering is also a method for privilege escalation (as seen with Dirty Pipe allowing writes to setuid binaries). Ensuring the integrity of system files and sensitive data is thus a fundamental aspect of system security.

Linux file permissions (DAC) provide the first line of defense: e.g., only root can modify system binaries in /usr/bin, and config files in /etc are typically only writable by root or specific admins. However, if an attacker becomes root, DAC is no obstacle. This is where LSMs and related features come in to provide *additional* protection of file integrity:

- **Mandatory Access Control on files:** SELinux policies can specify exactly which processes (domains) are allowed to modify certain files or directories. For instance, the SELinux policy might say that only the package manager process (rpm_t domain) can modify files under /usr/bin, while a web server process (httpd_t) cannot even if it runs as root. This prevents a compromised web server from altering binaries. In general, SELinux's type enforcement often separates the system into *write domains* and *read-only domains*. Many domains have no permission to write to system executables, configuration directories, or user home directories unless explicitly allowed. This mitigates file tampering for persistence – an attacker who compromises, say, a database service might attempt to edit /etc/crontab to add a malicious job or replace an SSH authorized_keys file to maintain access. If the policy denies the database service domain from writing to /etc or /home, those actions are blocked and logged. AppArmor similarly can mark certain paths as read-only or completely inaccessible for confined processes. An example: an AppArmor profile for nginx can specify deny /etc/passwd w, meaning even if nginx (or something running in its context) tries to open /etc/passwd for write, it will be denied. Thus, MAC can maintain integrity of key system files against illegitimate writes.

- **Immutable and append-only flags:** Outside LSMs, Linux also has filesystem attributes (chattr +i) to make files immutable. Some admins set critical files like /etc/passwd or sensitive binaries immutable so that they cannot be altered (except by rebooting into single-user mode or so). However, this is not widely used in practice for system files because system updates need to modify them. Instead, the trend is to use **read-only root filesystems** or verification mechanisms. Container deployments often have an immutable infrastructure concept: containers typically don't persist changes to the base image (except in a writable layer). Even so, if an attacker breaks into a container and tampers with files, they might only be tampering with that container's view (which could be ephemeral). For hosts, technologies like **dm-verity** can ensure a partition's contents are

read-only and match a known hash tree. Android uses this for system partitions. While not an LSM, it's part of the file integrity story.

- **Integrity Measurement Architecture (IMA):** As introduced, IMA can operate in different modes – one is measurement (just logging hashes of accessed files), another is appraisal (enforcing that hashes match expected values). In enforcement mode, IMA prevents execution (or reading, depending on policy) of files that are not verified. For example, a server might maintain a list of approved checksums for all executables; if malware somehow drops a new binary or alters one, IMA can detect that and stop it from running[24][59]. In practice, maintaining such a list is complex, so often IMA is used in conjunction with digital signatures: important files are signed by a key, and the kernel (with IMA and EVM) will only allow access if the signature in the extended attribute is valid. This essentially extends code signing to user-space files on Linux. A concrete adoption of this is in some Linux-based IoT appliances that require a high level of trust – the vendor signs all binaries, and the device won't run code unless it's properly signed (the keys are loaded into the kernel's keyring as part of trust provisioning). Another usage is recording the measurement of files to a TPM (Trusted Platform Module) so that a remote system can attest whether the files have been altered. For example, a critical infrastructure gateway could, on boot and runtime, measure its critical files and later prove to an auditor that no unauthorized changes occurred (if measurements match expected values). This defends against stealthy file tampering.

- **EVM (Extended Verification Module):** EVM works hand-in-hand with IMA by protecting the security extended attributes themselves (so an attacker cannot simply unset the "hash" or "signature" attribute on a file to bypass IMA). EVM can seal these attributes with an HMAC tied to a hardware trust (like TPM). In short, EVM ensures the integrity of metadata (including SELinux labels, IMA hashes, etc.), making it extremely difficult for an attacker to cover their tracks or tamper with security labels without detection.

- **Audit and Monitoring:** Even with preventive controls, it's valuable to detect tampering. Linux Audit subsystem can watch file access events. For instance, one could configure audit rules to alert when certain files are modified or even attempted to be modified. If an

attacker tries to edit /etc/ssh/sshd_config, an alert could be triggered. Some modern host intrusion detection systems (HIDS) for Linux, like OSSEC or Wazuh, utilize audit or inotify to detect changes to critical files. While this is more about detection than prevention, it is a complementary strategy. From an academic perspective, there are papers on filesystem integrity and anomaly detection that explore monitoring file system calls for suspicious patterns (like a process suddenly writing to many binaries might indicate malware).

Examples of File Tampering & Defense:

- **Persistent malware via init scripts or cron:** A classic persistence method is adding a malicious script in /etc/init.d or a cronjob in /etc/cron.d. On a system with proper MAC, the only processes that should write there are those involved in system installation or configuration (typically package managers or administrators in a certain role). SELinux policy would have a type for init scripts, and only allow certain domains (e.g., rpm_t or an admin user domain) to create files with that type. A malware running as apache (httpd_t) would be prevented from dropping a script in /etc/init.d due to type enforcement. If an attacker did get root and tried to do it manually while in a restricted role/domain, that could also be caught depending on policy.

- **Unauthorized data exfiltration or encryption (ransomware):** Attackers might not only tamper to install backdoors, but also to exfiltrate or encrypt data (which involves reading many files and rewriting them). LSMs can mitigate this in some configurations. There have been discussions in the community about using AppArmor to stop unknown processes from mass-reading personal files. For instance, one could apply a default AppArmor profile to user-launched applications that forbids access to files unless the user explicitly grants it (this is somewhat what sandbox frameworks do under the hood). If ransomware were running under such a profile, it might not be able to open all ~/Documents files to encrypt them. Admittedly, on a typical Linux without such restrictions, if a user runs a ransomware binary, it inherits the user's DAC permissions and can read their files. This is an area where more fine-grained user-space policies can help. Projects like Firejail or Qubes OS attempt to isolate user applications to limit damage from these scenarios.

- **CVE-2021-33909 (“Sequoia”):** This vulnerability allowed a local user to exploit a quirk in the filesystem (specifically in /proc/) to increase the size of an inode and write beyond what should be allowed, leading to privilege escalation. It was essentially a file tampering at the kernel FS level. LSMs wouldn’t directly stop this (again, being a kernel flaw), but interestingly, certain hardened kernel configs or mount options (like restricting user access to certain proc files) could mitigate it. Mounting /proc with hidepid can prevent unprivileged users from seeing others’ process info, which might close off some avenues. While not directly about writing files, it illustrates that kernel file interfaces can be an attack surface; hence why IMA can even cover things like checking integrity of certain pseudo-files, and why Lockdown disables some access to kernel memory via files.

In recent years, **supply chain security** has also become a concern – attackers tampering with software before it even reaches the user (e.g., inserting malware into a Linux package or repository). LSMs on the end-user system won’t prevent that initial tampering, but they might catch unusual behavior from a trojanized package. For instance, if someone managed to slip malicious code into the vi editor and you installed it, an integrity system could detect the binary doesn’t match the official signature (if using signed packages or IMA signatures). Or SELinux might log if vi suddenly tries to open network sockets (which normally it wouldn’t, hinting at something fishy). Thus, integrity enforcement paired with behavior restriction helps counter supply chain risks at runtime.

To enforce file integrity robustly, organizations often use a combination of **LSM policies, file integrity tools, and version control**. Critical config files might be kept under version control (etckeeper, for example) to detect diffs. Some organizations deploy **tripwire/AIDE** regularly to scan for unauthorized changes – these tools essentially compute hashes of files and compare to a baseline. They overlap with what IMA does, though IMA is kernel-enforced in real-time, whereas tripwire is typically run periodically in userland.

From an academic viewpoint, there are interesting developments like filesystems that can inherently prevent tampering (e.g., Verity filesystems, or FS-verity which is a feature that allows certain files to be marked as verifiable by a hash tree). **FS-verity**, added to Linux 5.4, allows individual files (like an executable or a dataset) to be made read-only and associated with a Merkle tree of hashes – the kernel will verify each read against the hash tree. This is used by

Android for APK verification and by some container systems to ensure image files aren't altered. It's yet another tool in the integrity toolbox. Although FS-verity is not an LSM, it works alongside LSMs (LSM could enforce that a given file must have verity enabled, etc.).

In conclusion, Linux provides strong mechanisms to guard against file tampering. LSMs enforce who/what can modify files at a granular level, and integrity subsystems ensure that any modification (authorized or not) doesn't go unnoticed or unapproved. When properly configured, these can even stop a root user (or an attacker who became root) from stealthily altering critical files. An attacker might then resort to disabling these protections (which itself would likely be noticed if they are active). One challenge is that not all systems enable these features due to performance or complexity concerns. There's often a balance between convenience and integrity: e.g., enabling IMA-appraisal can slightly slow down file access and requires maintaining signatures, so not everyone does it. However, for high-value systems, the slight overhead is justified by the significant gain in security.

Kernel Rootkits and Stealth Attacks

Perhaps the most insidious threat to a Linux system is a **kernel rootkit** – malware that resides in the kernel to hide itself and provide continued privileged access to an attacker. Once an attacker installs a rootkit in the kernel, they can subvert almost any aspect of the system: they can hide processes and files from regular listings, intercept network packets, keystrokes, and even render security tools blind. Rootkits have a long history in the Linux world (e.g., the LKM rootkits like adore, knark from the early 2000s). Modern rootkits can be even more sophisticated, sometimes using zero-day exploits or novel techniques (like hooking the Linux eBPF engine, as observed in some recent malware).

How rootkits operate: A rootkit typically either loads as a malicious kernel module or patches the running kernel (via /dev/mem, /dev/kmem, or by modifying in-memory structures) to insert its hooks. Once inside the kernel, it may hook system call tables, VFS operations, or other critical function pointers to conceal the presence of certain processes or files, or to backdoor authentication (e.g., always grant root if a certain password is used), etc.[60][61]. Some kernel rootkits don't even need to hide files – they simply live entirely in kernel memory. Others might

hide on disk by modifying filesystem data structures (there have been rootkits that hide in reserved disk blocks outside normal files).

LSMs and rootkits: Linux Security Modules can play both offensive and defensive roles with rootkits:

- On the *defensive* side, some LSM features explicitly try to prevent rootkits from getting into the kernel in the first place. The aforementioned **Lockdown mode** is one: it closes common injection vectors (no writing to /dev/mem or /dev/kmem, no using iopl to get I/O port access, no unsigning modules, etc.)[20][22]. Lockdown essentially treats the root user as potentially hostile, which is exactly the mindset needed to thwart rootkits (since a rootkit often is deployed by someone who already briefly had root access). Another defensive LSM is the module signing enforcement – when configured, the kernel won't load any module that isn't signed by a trusted key. This means an attacker can't simply compile a malicious kernel module and insmod it, unless they also compromise the key or find a way to bypass verification (which is non-trivial if secure boot is on and the key is from UEFI or so). SELinux and AppArmor can also indirectly help: while they don't usually prevent the loading of modules (that's outside their normal policy scope), they can restrict *user-space actions that lead to rootkit installation*. For example, a typical way to load a rootkit is using the `insmod` or `modprobe` command. On a system like RHEL with SELinux, the `insmod` program and module syscalls are reserved to certain domains. A confined service running as httpd_t cannot load a module because that domain won't have the CAP_SYS_MODULE capability or access to the module files. Even if the attacker is root in a shell, if they are in an SELinux role that is not allowed to load modules (SELinux can tie capabilities to roles/types), they might be prevented. Usually, however, if an attacker gets a *real* root shell (unconfined_t on a targeted policy), they could load modules. This is why strategies like Lockdown and module signing are crucial – they operate even when the attacker is UID 0. In general, hardening recommendations for Linux servers now often include **disabling unused interfaces** like preventing the loading of any new modules on a running system (set `/proc/sys/kernel/modules_disabled` to 1 after boot, if your system supports that and you don't need to load modules at runtime).

That's a simple but effective step: it means even if attacker gets root, they can't load a new LKM rootkit because module loading is globally turned off (until next reboot).

- On the *offensive/detection* side, interestingly, LSM hooks can be used to detect anomalies that rootkits cause. Some research projects and tools instrument LSM hooks to catch suspicious behavior. For instance, a rootkit might try to hide a process by filtering it out of /proc reads – an anomaly detection system could notice that the getdents (directory read) hook for /proc is returning inconsistent results. In fact, one academic approach used timing of system calls to detect rootkits: the presence of additional instructions (from rootkit hooks) can slightly increase the execution time of syscalls[62][63]. By monitoring for such timing anomalies with high precision (even using eBPF timers), researchers have shown it's possible to detect certain rootkits at runtime[64][63]. Another method is *cross-view detection*: comparing the kernel's view of things vs. an external view. For example, a userland tool queries the kernel for a list of processes, and also scans memory for process data structures; if something appears in one but not the other, a rootkit might be hiding it[65]. LSMs can assist by providing hooks to implement such cross-checks in kernel space too.

However, as Stühn et al. (2024) conclude in their survey, **existing rootkit detection mechanisms are often insufficient** – no single solution catches all rootkits, and many require prior knowledge of the rootkit's behavior[66]. Traditional signature-based detection fails on new rootkits[67]. This is an active area of research. It's essentially an arms race: as detection improves, rootkits evolve to be stealthier (e.g., doing less obvious hooking, or living in firmware, etc.).

Real-world rootkit examples: One recent example (2022) was the **"Syslogk" rootkit**, which hid itself by hooking kernel functions responsible for listing modules and processes. It could be remotely activated via special packets (triggering it to unhide or perform actions). Another is **"HiddenWasp"** (2019) – a user-space and kernel-space hybrid malware for Linux that installed a rootkit to hide the userland components. In 2022, researchers also uncovered **"Symbiote"**, which was not a standalone rootkit but rather a parasite that injects into all running processes via LD_PRELOAD and uses eBPF to hide network traffic – a reminder that not all stealth malware is in the kernel; user-mode rootkits exist too. For user-mode, LSMs like SELinux can sometimes

catch those because, for example, Symbiote opened /proc/net in weird ways to hide its connections – if an AppArmor profile had strict rules on what an Apache process can read, it might flag the unusual file access.

Mitigations summary for rootkits:

- **Prevent initial insertion:** Use secure boot + kernel module signing + lockdown so that injecting code into kernel is extremely difficult. Disable interfaces like old /dev/kmem (which is already off by default on modern kernels), restrict /dev/mem via lockdown or by not running in legacy mode. Keep the system patched so known priv escalation paths (which could be used to insert rootkits) are closed. Use LSM policies to make it hard for an attacker to even execute the steps to load a rootkit (e.g., don't allow arbitrary writing to /lib/modules or executing insmod from a non-admin domain).

- **Detection and response:** Employ kernel modules or eBPF programs that periodically check kernel integrity – for example, verify the syscall table hasn't been altered (some security modules do this internally; SELinux and others protect their own hooks, but not all kernel). There's a thing called integrity_check that one can enable to have the kernel self-check certain pointers, though it's limited. The **Linux Kernel Integrity subsystem** (LKRG by Openwall) is an out-of-tree module that attempts to detect rootkits by hashing important structures and monitoring for changes. It's not an LSM, but it's a notable defense layer some use.

- **Recovery:** If a rootkit is suspected, one often needs to reboot into a known-clean state (with a clean kernel) because once kernel is tainted, you can't fully trust any output. But rebooting an important system might be costly. Thus the best approach is prevention and early detection. Proper logging (with remote log collection) can help; a rootkit might try to suppress logs, but if logs are shipped out in real-time (e.g., via auditd or an agent), the footprints might already be out. Some rootkits try to prevent this by also hiding their malicious activities from audit or turning off auditd, which again can be caught if one is monitoring the monitors.

In terms of **academic literature**, besides detection approaches, there have been proposals for making the kernel *formally verified or using microkernel approaches* to eliminate rootkits – but

those are far from mainstream Linux. Another interesting angle is hypervisor-based protection: running Linux under a hypervisor and using the hypervisor to monitor the guest kernel for integrity (Virtual Machine Introspection). Projects like Xen's altp2m (alternate EPT views) can be used to trap writes to certain kernel areas, effectively preventing unauthorized modifications. Microsoft's HyperGuard for Windows does something akin to that. For Linux, that's not commonly used in production outside of some cloud provider techniques. But it's a possible future for critical Linux systems – a lightweight hypervisor whose sole job is to ensure the kernel isn't being tampered with, while the LSM enforces policies inside.

To wrap up, kernel rootkits represent a scenario where, if successful, an attacker can neutralize many security measures. Thus, a defense-in-depth approach is needed: **prevent**, **detect**, and **limit**. LSMs like SELinux/AppArmor can *limit* what an attacker can do to reach the point of installing a rootkit (e.g., preventing actions that lead to kernel exploit or module loading). Features like Lockdown *prevent* certain rootkit installation vectors outright. And although not primarily their design, LSM hooks have been utilized in research to *detect* rootkit behavior. Stühn et al. (2024) emphasize combining methods for detection – anomaly detection, signature scanning, cross-view – to improve coverage[66]. There is no silver bullet; even in 2025, defending against rootkits is challenging, but modern Linux has a much stronger posture against them than it did a decade ago. For example, the common old tactic of using /dev/kmem to write to kernel memory is now blocked; loading a rogue module requires either a stolen key or a disabled verification; many distros compile out the legacy /proc/syscalls that allowed easy hooking. Attackers have correspondingly shifted to more indirect methods or to userland persistence (because kernel attacks are harder). As defenders, continuously updating kernel security features and policies is critical – one reason why running latest kernels (if possible) is recommended for security: new mitigations and LSM improvements are continually added.

Conclusion

Linux's security landscape in 2025 is robust yet continually challenged by evolving threats. **Linux Security Modules** like SELinux and AppArmor have matured into effective mandatory access control systems widely deployed in enterprise servers, cloud containers, and even endpoints. They provide fine-grained control over what processes can do, significantly reducing the attack surface and containing compromises. Meanwhile, experimental modules such as

Landlock and BPF-LSM (KRSI) represent the next generation of flexibility and unprivileged security, aiming to sandbox applications and leverage the power of eBPF for dynamic policy enforcement. On the kernel-hardening front, features like Lockdown mode, module signing, and integrity verification (IMA/EVM, FS-verity) directly tackle the techniques attackers used in the past for rootkits and unauthorized changes.

The current threat environment highlights **privilege escalation**, **file tampering**, and **rootkits** as primary concerns on Linux. We have seen that no single defense is sufficient: instead, layers of defense work in concert. For privilege escalation, ensuring least privilege by default (through capabilities and LSM confinement) means even if a vulnerability exists, the opportunities for exploitation or impact are narrowed. The dramatic increase in Linux vulnerability disclosures in 2024[58] is a double-edged sword – on one hand it indicates more bugs to fix, but on the other it means more attention to Linux security and faster patching cycles. LSMs act as a safety net when not all bugs are known or fixed: for instance, a strong SELinux or AppArmor policy might block an exploit's actions even if the kernel bug is still present, buying time until a patch is applied.

For file tampering, Linux provides the tools to enforce an almost immutable runtime environment – MAC policies to restrict who can change what, and cryptographic integrity checks to detect any unauthorized changes. The principle of "trust but verify" is well implemented by IMA: even if an attacker slips past access controls, any illicit modification can be caught and stopped if the system is configured to require signatures or known hashes. As systems become more critical, such as in infrastructure and IoT, we expect increased adoption of these integrity mechanisms (and indeed regulatory standards may mandate them).

Rootkits remain a formidable menace, but the window for installing kernel rootkits has been substantially narrowed by modern defenses. It is no longer trivial for an attacker, even with root access, to plant a persistent kernel module on an updated, well-configured Linux system – they would have to bypass Secure Boot, find a signing key or exploit a kernel flaw to disable protections. That raises the bar to nation-state level adversaries in many cases. Instead, attackers often resort to user-space malware, which, while dangerous, is easier to clean up and contain than a kernel rootkit. The ongoing research in anomaly detection (like timing analysis with eBPF probes[62]) and other innovative rootkit detection strategies gives hope that even sophisticated

stealth techniques can be uncovered. However, as the research literature suggests, a combination of methods and deep knowledge is required to catch all rootkits[66].

In the big picture, the Linux security model today embodies **defense-in-depth**: from compile-time and boot-time safeguards (PIE, canaries, secure boot) to runtime mandatory policies (LSMs) to post-compromise detection (audit logs, tripwire, etc.). It's crucial for administrators and organizations to keep all these layers active and up-to-date. The trends indicate even more integration between these layers – for example, future Linux versions might tie integrity verification with LSM decisions more tightly, or use hardware features (like Intel CET shadow stacks, ARM pointer authentication) to further thwart exploits, all under the umbrella of kernel self-protection[56][68].

One should not forget usability and performance: security improvements must be balanced with system functionality. The fact that SELinux and others are now default in major distros and generally stay enabled is a testament to the community's effort in making policies workable out-of-the-box. In academic terms, there's also interest in the *usability of security modules* – a 2016 study compared how sysadmins manage SELinux vs AppArmor and found that simpler models can lead to fewer mistakes (hence why tools like *SEinux troubleshooter* exist to assist admins) [69][70]. We can expect future efforts to simplify policy writing (perhaps through AI suggestions or more learning-based policy generation as TOMOYO attempted).

Finally, Linux security does not operate in a vacuum. It's part of an ecosystem: CVE databases, security mailing lists (like oss-security) and incident reports inform how we adjust policies. For example, when a new exploit technique is publicized, one might see a quick patch to AppArmor or SELinux templates to counter it (like blocking new syscall vectors). It's a community effort bridging practitioners and researchers. The inclusion of academic research – from formalizing MAC policy comparisons[71][10] to surveying container threats – is crucial in guiding next steps. As of this research (October 2025), Linux stands as a relatively secure platform when its security features are utilized to the fullest. The key is vigilance and proper configuration: a hardened Linux system with updated kernel, appropriate LSM policies, and integrity checking forms a very tough target, even against cutting-edge threats.

Sources:

- Chen, H., Li, N., & Mao, Z. (2017). *Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems*. **NDSS Symposium 2017**[72][73].
- Stühn, B., et al. (2024). *Prevalent mechanisms for rootkit detection are insufficient*. **Journal of Cybersecurity**, 2024. (Summary of findings in[66]).
- Sun, Y., et al. (2018). *Security Namespace: Making Linux Security Frameworks Available to Containers*. **27th USENIX Security Symposium**[41][42].
- Red Hat, Inc. (2025). *SELinux and RHEL: A technical exploration of security hardening*. **Red Hat Blog** (February 10, 2025)[5][74].
- Larabel, M. (2024). *Linux's Landlock Sandboxed Apps Could Remove Restrictions On Itself*. **Phoronix News** (July 28, 2024)[27][29].
- AccuKnox, Inc. (2023). *Enhancing Runtime Security With eBPF/BPF-LSM*. **AccuKnox Blog** (Oct 2023)[33][34].
- Cardillo, A., Serper, A., & Sahu, S. (2024). *Active Exploitation Observed for Linux Kernel Privilege Escalation (CVE-2024-1086)*. **CrowdStrike Blog**[54][75].
- NVD. (2024). *CVE-2024-1086 Detail*. **National Vulnerability Database**[53].
- NVD. (2022). *CVE-2022-0847 Detail ("Dirty Pipe")*. **National Vulnerability Database**[55].
- Codenotary. (2025). *Linux Vulnerability Surge of 2024: Strategic Responses for IT Leaders*. **Codenotary Blog** (Jun 3, 2025)[58][76].
- Linux Kernel manual page. (2025). *kernel_lockdown(7)*. **man7.org**[20][22].
- Day, B. (2024). *Enhancing Linux Kernel Security: Lockdown Mode and Self-Protection Insights*. **linuxsecurity.com**[24][56].

---

[1] [2] [3] [4] [10] [11] [12] [13] [18] [71] [72] [73] ndss-symposium.org

https://www.ndss-symposium.org/wp-content/uploads/2017/09/Chen.pdf

[5] [6] [7] [8] [9] [69] [70] [74] SELinux and RHEL: A technical exploration of security hardening

https://www.redhat.com/en/blog/selinux-and-rhel-technical-exploration-security-hardening

[14] [15] Security policies | Snapcraft documentation

https://snapcraft.io/docs/security-policies

[16] Exploring Linux Security Modules: SELinux vs AppArmor vs TOMOYO

https://linuxsecurity.com/news/security-projects/linux-security-modules-lsm-selinux-vs-apparmor-vs-tomoyo

[17] The Results of a Usability Study Comparing SELinux, AppArmor ...

https://www.researchgate.net/publication/220593634_Empowering_End_Users_to_Confine_Their_Own_Applications_The_Results_of_a_Usability_Study_Comparing_SELinux_AppArmor_and_FBAC-LSM

[19] [20] [21] [22] [23] kernel_lockdown(7) - Linux manual page

https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html

[24] [56] [57] [59] [68] Strengthening Linux Kernel Security Using Lockdown And Self-Protection

https://linuxsecurity.com/howtos/learn-tips-and-tricks/lockdown-mode-kernel-self-protection

[25] [26] Landlock: Unprivileged Sandboxing — Landlock documentation

https://landlock.io/

[27] [28] [29] [30] Linux's Landlock Sandboxed Apps Could Remove Restrictions On Itself - Phoronix

https://www.phoronix.com/news/Landlock-Restrictions-Bug-Fixed

[31] KRSI — the other BPF security module - LWN.net

https://lwn.net/Articles/808048/

[32] Kernel Runtime Security Instrumentation LSM+BPF=KRSI - YouTube

https://www.youtube.com/watch?v=vxoRxbGy2IU

[33] [34] [35] [36] [37] [38] [39] [40] [48] Enhancing Runtime Security With EBPF/BPF-LSM: Impact Of Cisco's Acquisition Of Isovalent

https://accuknox.com/blog/runtime-security-ebpf-bpf-lsm

[41] [42] [43] [44] [45] cs.ucr.edu

http://www.cs.ucr.edu/~trentj/papers/usenix18.pdf

[46] CVE-2019-5736 Detail - NVD

https://nvd.nist.gov/vuln/detail/cve-2019-5736

[47] Mitigating High Severity RunC Vulnerability (CVE-2019-5736)

https://www.aquasec.com/blog/runc-vulnerability-cve-2019-5736/

[49] Security and sandboxing - Ubuntu Core documentation

https://documentation.ubuntu.com/core/explanation/security-and-sandboxing/

[50] Snap on F41 not works as expected - Fedora Discussion

https://discussion.fedoraproject.org/t/snap-on-f41-not-works-as-expected/135720

[51] [52] The Great Kernel CVE Flood of 2024

https://tuxcare.com/blog/the-great-kernel-cve-flood-of-2024/

[53] NVD - cve-2024-1086

https://nvd.nist.gov/vuln/detail/cve-2024-1086

[54] [75] Active Exploitation Observed for Linux Kernel Privilege Escalation Vulnerability (CVE-2024-1086)

https://www.crowdstrike.com/en-us/blog/active-exploitation-linux-kernel-privilege-escalation-vulnerability/

[55] NVD - cve-2022-0847

https://nvd.nist.gov/vuln/detail/cve-2022-0847

[58] [76] Linux Vulnerability Surge of 2024: Strategic Responses for IT Leaders

https://codenotary.com/blog/linux-vulnerability-surge-of-2024-strategic-responses-for-it-leaders

[60] [61] [62] [63] [64] [65] [66] [67] Trace of the Times: Rootkit Detection through Temporal Anomalies in Kernel Activity

https://arxiv.org/html/2503.02402v1