OSDI 26

USENIX ASSOCIATION

USEC: A User-Requirement-Driven Mandatory Access Control Framework for Linux

Yu Jiang, Wenhuan Liu, Fuchen Ma, Yuheng Shen, Yuanliang Chen
Lei Zhang, He Li, Quan Zhang, Chijin Zhou

Tsinghua University, UnionTech Software Technology, East China Normal University







Research Background

Why Kernel Access Control Matters

Modern operating systems are widely deployed in enterprises, clouds, desktops, and critical infrastructure.

Kernels mediate access to files, devices, IPC endpoints, and network sockets for all applications.

Fine-grained Mandatory Access Control (MAC) is essential for containing compromised applications and enforcing least privilege.

Key Research Gap

Existing MAC frameworks such as SELinux are powerful, but difficult to configure, expensive to enforce broadly, and prone to compatibility failures. As a result, strong kernel protection is often disabled in real deployments.

Why Existing MAC Fails in Production

Complex Policy Configuration

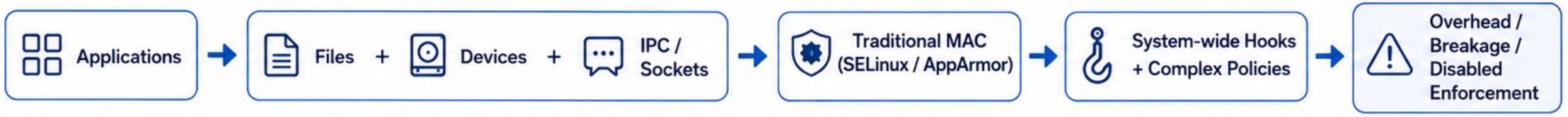
Administrators must maintain thousands of low-level rules, labels, and constraints, making policies hard to understand, audit, and evolve.

High Runtime Overhead

Always-on MAC checks along file, IPC, process, and socket paths introduce visible overhead on real workloads.

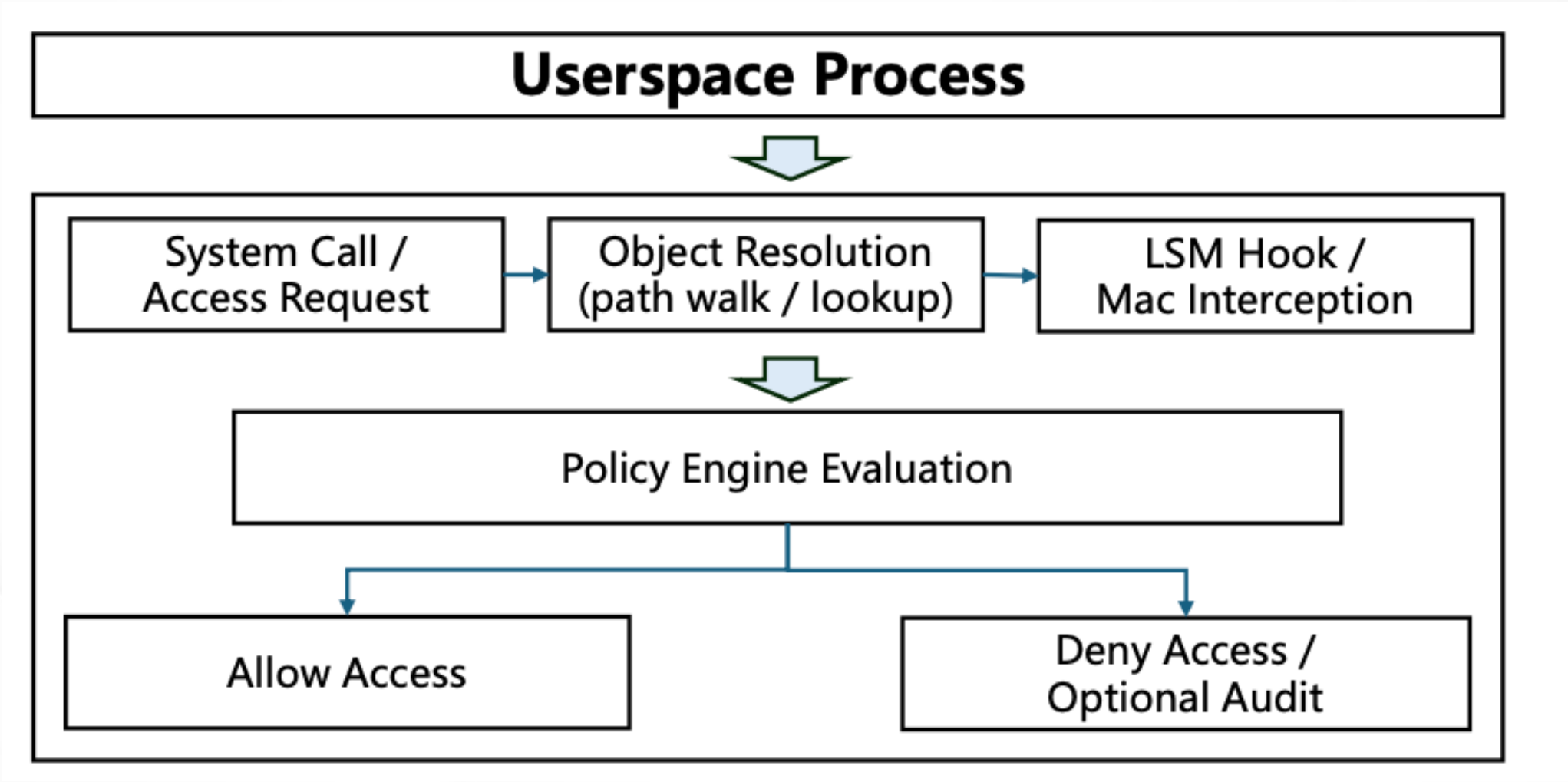
Fragile Compatibility

Strict path- and type-based policies can silently break legitimate applications, installers, and services in production.



USEC is motivated by the need to make strong kernel MAC practical in production Linux systems through resource-centric policy, demand-driven enforcement, and compatibility-aware design.

Research Background



A Basic runtime workflow of MAC enforcement

How to Make Effective MAC Deployable?

Protect what matters, enforce only when needed.

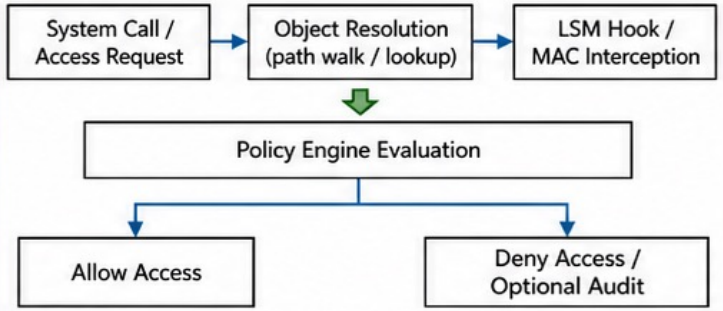


Motivating Example – Protecting an Endpoint Security Agent

1 Security Goals

- Protect agent runtime data under `/opt/agent/data/`
- Allow install / upgrade of the agent binary
- Preserve audit-only access to logs under `/var/log/agent/`
- Keep three-way separation: system / security / `audit admin`

2 Basic MAC Runtime Workflow



3 SELinux Policy Construction

Many low-level rules

Types & Labels

```
agent_t
agent_exec_t
agent_data_t
agent_log_t
file_contexts …
```

Roles & Constraints

```
sysadmin_u / secadmin_u / audit_u
allow rules
role mappings
constraints …
```

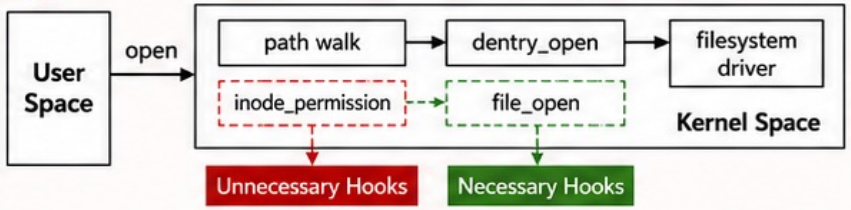
4 Deployment Pain

policy bloat / silent denials / compatibility failures

? Why the Problem Happens

- SELinux is process/domain-centric, not resource-centric.
- Administrators must coordinate types, labels, roles, and constraints.
- Small distro or path changes require edits across multiple policy files.

5 Overhead Under MAC Enforcement



- Even a simple file-open request triggers multiple MAC checks.
- Under coarse-grained policy, only the final decision may matter.
- Broad, always-on mediation adds cost on hot paths.

Connection to USEC

- Express policy directly over protected resources.
- Retain only capability-relevant hooks.
- Reduce enforcement cost while improving deployability.

Key Insight: Strong MAC should match how vendors think — protect a small set of critical resources, and enforce only on relevant kernel paths.

Threat Model

Protected Resources

- Agent runtime data under "/opt/agent/data/"
- Agent binary installation / upgrade path
- Audit-only log access under "/var/log/agent/"
- Principle of three-way separation among system admin, security admin, and audit admin.

Desired Operations

- ✓ Read (by authorized roles)
- ✓ Write (to designated paths)
- ✓ Install / Upgrade
- ✓ Audit-only Read

Security Objective

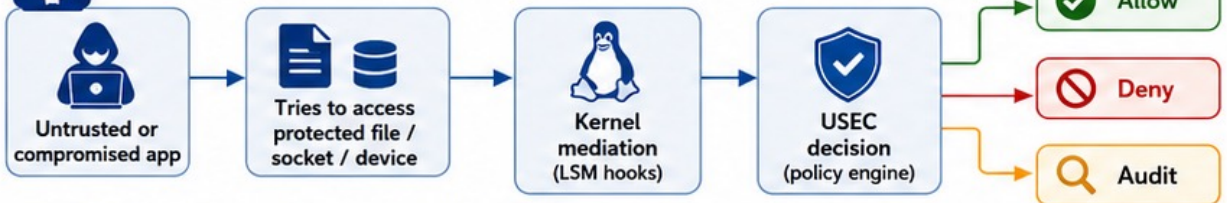
- Protect a small set of critical kernel-managed resources.
- Contain compromised or buggy user-space applications.
- Enforce least privilege and separation of duties.
- Preserve deployability on real production Linux systems.

User requirements → Protected resources → Selective enforcement

Out of Scope / Assumptions

- Trusted Linux kernel and LSM framework.
- Correctly installed USEC policy.
- Not targeting kernel-exploit defenses.
- Not solving all system-wide MAC policies; focus is selected critical resources.
- Unmentioned resources fall back to DAC and optional audit.

Threat Model in Practice



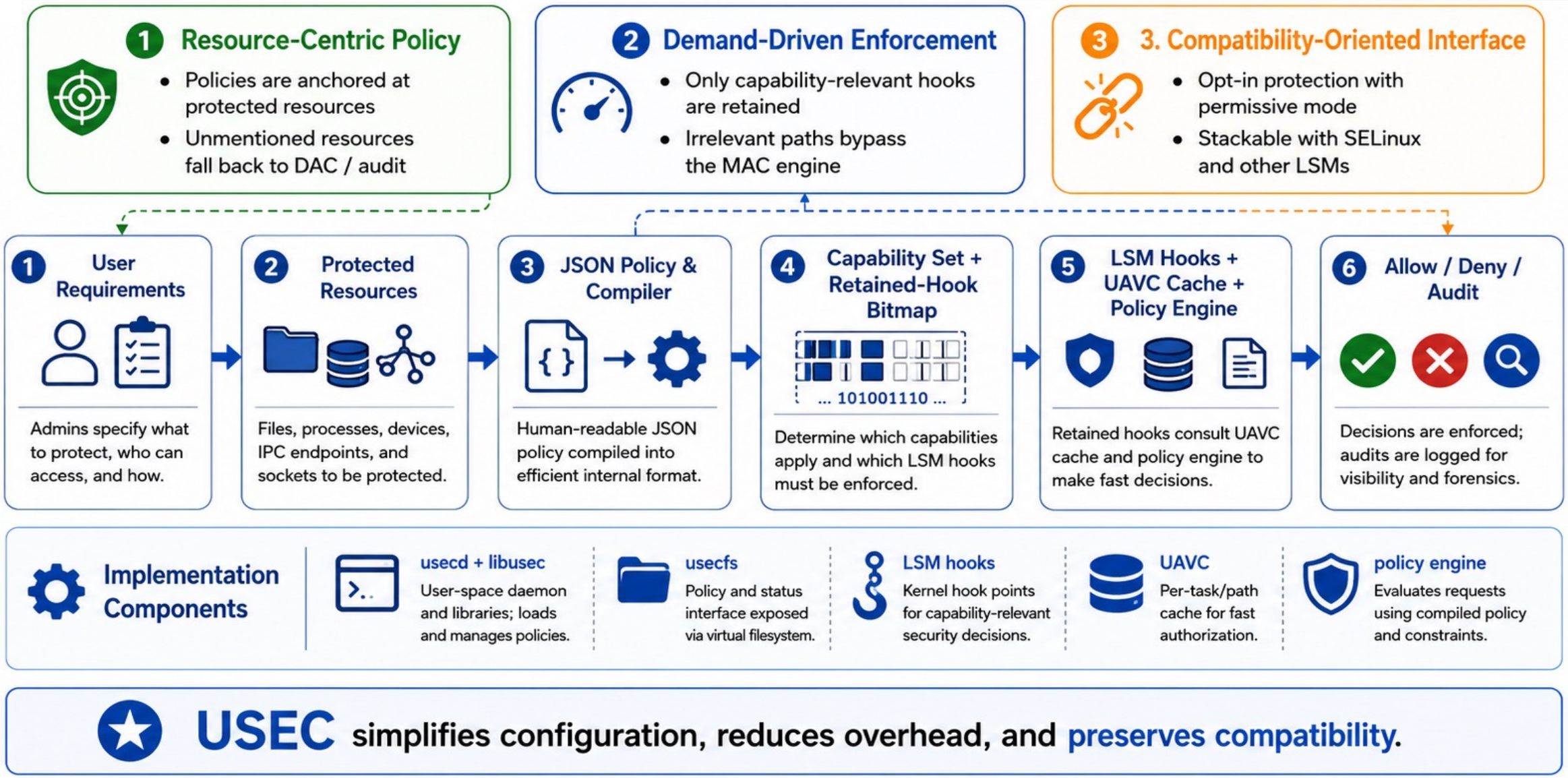
- Attackers abuse ordinary application privileges.
- They should not tamper with sensitive agent data.
- They should not bypass duty separation for admin roles.

Why USEC's Scope Matters

- Vendors usually care about a narrow set of high-value resources, not every object in the system.
- Resource-centric protection matches how operators reason about security.
- Selective scope is the basis for lower overhead and better compatibility.
- This scope directly motivates resource-centric policy and demand-driven enforcement.

Key Insight: USEC is designed for practical strong protection of a small, explicit set of critical resources — not blanket mediation of the entire system.

Our Solution – USEC Overview



Our Solution – Resource-Centric Policy Model

Traditional MAC

Process / Domain-Centric

Process Domains → Object Types → Allow Rules → Many Low-Level Dependencies

Hard to maintain, audit, and evolve at scale.

USEC

Resource-Centric

Resource Descriptor → Principals + Operations → Direct Rule Lookup

Simple, explicit, and aligned with how systems are built.

1. Rules are anchored at resource descriptors such as file paths, device nodes, mount points, and socket endpoints.

2. Policies specify which principals may read, write, or execute each resource.

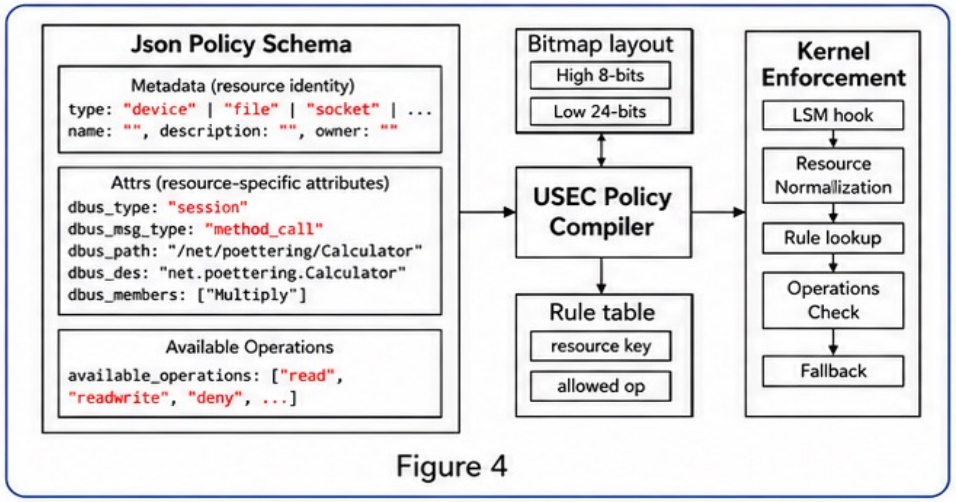


Figure 4

3. Only explicitly declared protected resources are matched by USEC.

4. If no rule matches, access falls back to DAC and optional audit.

```
{
  "name": "camera",
  "subtype": "device",
  "operations": ["read", "readwrite", "deny"]
}
```

Key Insight: Think from resources, not domains.

Our Solution – Demand-Driven MAC Enforcement

USEC compiles compact capability intent into a selective enforcement pipeline.

1 Capability Declaration
Vendors specify security-critical capabilities.

2 Capability-to-Resource Mapping
Identify kernel objects and transitions that realize each capability.

3 Resource-to-Hook Resolution
Resolve capability-relevant hook set from mapped resources.

4 Selective Enforcement Pipeline
Only capability-relevant hooks are retained and invoked.

Demand-Driven MAC Enforcement
Near-zero overhead on irrelevant paths

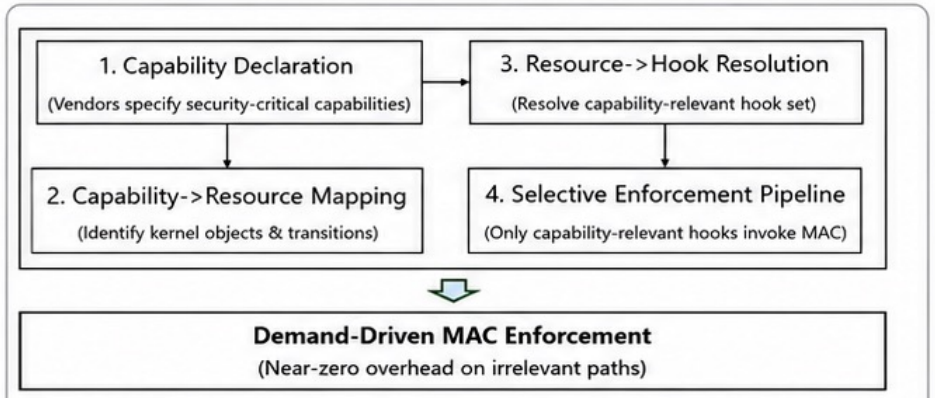


Figure 5: The Basic Pipeline of USEC's MAC Enforcement.

How it works at runtime

- Each installed hook checks a retained-hook bitmap.
- If the bit is unset, it returns immediately.
- Otherwise USEC invokes `check_usec_perm()`, UAVC, and the policy database.

Example retained hooks for file-write integrity

- `file_open`
- `file_permission`
- `mmap_file`
- `inode_setattr`
- `inode_link`
- `inode_rename`

Key Takeaway: Enforcement cost scales with the declared protection scope, not the full kernel mediation surface.

Our Solution – Compatibility-Oriented Security Interface

Figure 6: Workflow of USEC's Security Interface

System Exec → Runtime Hook → Hook Registration

1	**Userspace application**	An application in userspace wants to access a resource (e.g., file, socket).
2	**System call**	The application issues a system call to request the operation.
3	**Kernel interface**	The kernel receives the syscall and performs initial processing.
4	**LSM**	Control reaches the LSM hook (through the security hook point).
5	**Runtime dispatcher**	USEC's dispatcher looks up the hook mapping table for this hook.
6	**Hook mapping table**	If a matching entry exists, obtain the registered hook chain for the hook.
7	**Hook chain**	Execute the hook chain in order to collect policy decisions.
8	**Security execution**	Aggregate decisions and return the result to the kernel / caller.

Coexistence with SELinux

- USEC is opt-in and supports permissive mode.
- USEC is fully **stackable** with SELinux and other LSMs.
- USEC keeps policy and state separate from SELinux.
- Separate runtime state includes usec_state, upolicydb, usec_uavc, and usidtab.

SELinux Component	↔	USEC Counterpart
selinux_state	↔	usec_state
policydb	↔	upolicydb
selinux_avc	↔	usec_uavc
sidtab	↔	usidtab

Operational Benefit

Enables incremental deployment on top of existing SELinux configurations without entangling internal state.

Key Takeaway: USEC inserts a lightweight security interface at the LSM hook point, dispatches to registered hooks via a mapping table, and makes decisions without modifying kernel behavior or SELinux internals.

Our Solution – Compatibility-Oriented Security Interface

SELinux	USEC
struct selinux_state	struct usec_state
struct selinux_avc	struct usec_uavc
struct policydb	struct upolicydb
struct selinux_policy	struct usec_policy
struct sidtab	struct usidtab
struct av_decision	struct uav_decision
selinux_avc_init()	usec_avc_init()
avc_has_perm_noaudit()	check_usec_perm_noaudit()
avc_has_perm()	check_usec_perm()

How USEC’s policy/AVC layer mirrors SELinux

Implementation Details

USEC as a Linux Security Extension

- Integrates with the Linux Security Module (LSM) framework.
- Provides a lightweight, capability-driven access-control path.
- Policies are loaded dynamically from user space without modifying applications.

Five Core Components

1. usecd + libusec
user-space daemon and libraries; generate and manage policies.

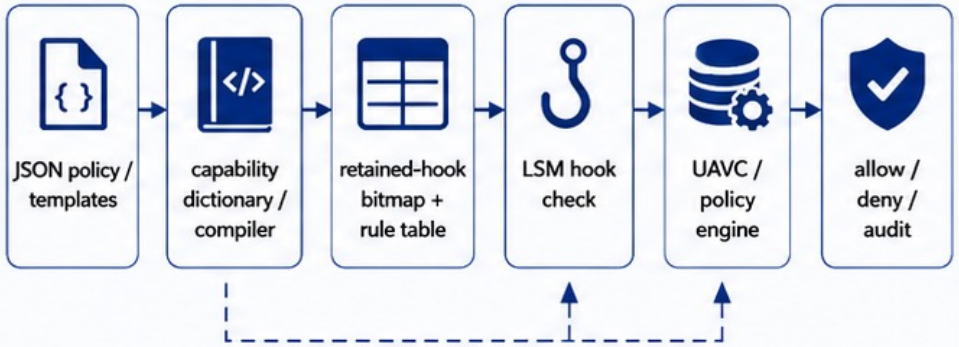
2. usecfs
virtual filesystem interface for policy and status transfer.

3. LSM hooks
mediation points for files, process control, IPC, and sockets.

4. UAVC
USEC-specific AVC cache for fast authorization decisions.

5. Policy engine
evaluates requests against compiled capability policies.

Compile-Time to Runtime Path



Only retained hooks enter the MAC path.

Deployment Properties

- Implemented as a loadable LSM module.
- Compatible with LSM stacking.
- Reuses existing kernel security infrastructure where possible.
- Stackable with SELinux while keeping USEC state separate.
- Policies can be loaded and updated dynamically.

Implementation Size

19,223	63,189	82,412
lines of kernel code added	lines in user-space components	lines total

Automation vs. Manual Effort

- Policy loading, hook dispatch, cache lookup, and enforcement are handled automatically.
- The main manual effort is maintaining capability-to-hook bindings for kernel versions and configurations.
- This localization keeps updates lightweight.

Key Takeaway: USEC translates high-level security intent into an efficient kernel enforcement path while remaining practical to deploy and maintain.

Evaluation Setup

Research Questions

1 Q1: Does USEC reduce policy complexity?

Can USEC express the same policies with fewer lines and easier maintenance than SELinux?

2 Q2: Does USEC lower runtime overhead?

Does USEC incur less performance degradation than SELinux compared to an LSM-disabled baseline?

3 Q3: Does USEC preserve compatibility and support deployment?

Can USEC mitigate SELinux-induced failures and reuse SELinux policy fragments in practice?

Compared Systems

SELinux
Mainstream full-coverage MAC baseline.

USEC
Proposed framework.

AppArmor
Path-oriented practical baseline.

LSM Disabled
Baseline without MAC overhead.

Each configuration is run for 10 trials; we report **average, standard deviation,** and **relative degradation** where applicable.

Evaluation Scenarios

A. Policy Complexity Reduction:
- Camera access control.
- ATSPI / D-Bus interface control.
- Compare policy size, expressiveness, and maintainability.

B. Runtime Overhead:
- UnixBench.
- Filebench.
- Nginx throughput / latency.
- Compare degradation versus LSM-disabled baseline.

C. Compatibility:
- Mitigating SELinux-induced failures.
- Matching SELinux protection and compatibility.
- Reusing SELinux policy fragments inside USEC.

What We Measure

- Policy lines / complexity.
- Throughput degradation.
- Latency overhead.
- Compatibility / deployability outcomes.

Key Takeaway: The evaluation is designed to test the full claim of the paper — simpler configuration, lower overhead, and better compatibility.

Evaluation I – Policy Complexity Reduction

Under equivalent requirements, USEC policies contain up to **10× fewer** lines of policy code.

Case A: Camera Access Control

SELinux (Traditional)

Multiple files, attributes, labels, interfaces

Minimal working configuration exceeds **300 lines**

USEC

2 JSON documents

Fewer than **20 lines**

USEC – Camera Resource (*resource.json*)

```
{
   "name": "camera",
   "type": "object",
   "subtype": "device",
   "attrs": { "devtype": "camera" },
   "available_operations": ["read", "readwrite", "deny"],
   "version": "1.0"
}
```

USEC – Policy Grant (*policy.json*)

```
{
   "policies": [{
     "subject": { "name": "com.example.app" },
     "objects": [{ "object": "camera", "operations": ["read"] }]
   }]
}
```

Case B: ATSPI / DBus Interface Control

SELinux (Traditional)

DBus methods decomposed into types, attributes, and dbus class rules

Scattered across multiple policy constructs that must be coordinated

USEC

Interface + members modeled directly as first-class resources

Single, cohesive policy that matches developer mental model

USEC – AT-SPI Write Interface (*object.json*)

```
{
   "name": "ATSPI-WRITE",
   "type": "object",
   "subtype": "dbus",
   "attrs": {
     "dbus_interface": "org.a11y.atspi.EditableText",
     "dbus_members": ["InsertText", "CutText", "DeleteText"]
   },
   "operations": ["read", "readwrite", "deny"]
}
```

USEC – Policy Grant (*policy.json*)

```
{
   "policies": [{
     "subject": { "name": "dbus-send" },
     "objects": [{ "object": "ATSPI-WRITE", "operations": ["read"] }]
   }]
}
```

Key Takeaway: USEC expresses intent directly at the level developers reason about.

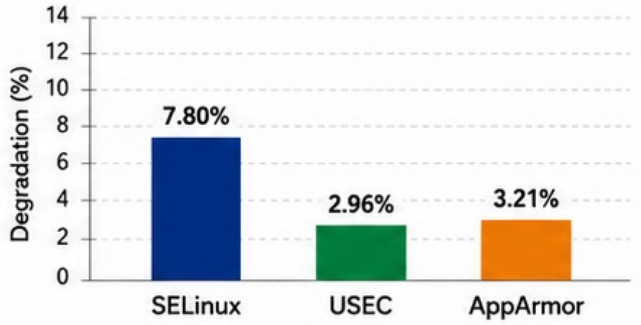
Evaluation II – Runtime Overhead

Performance degradation vs. LSM-disabled baseline (lower is better)

1 UnixBench

Overall system benchmark

Performance degradation vs baseline

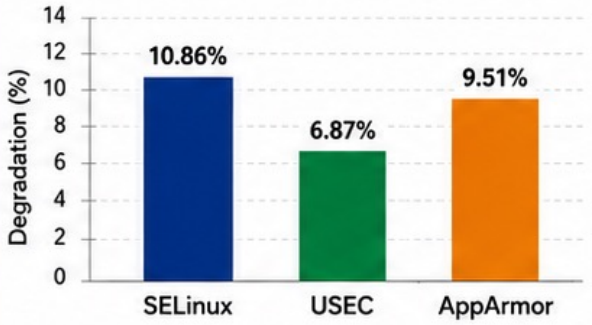


SELinux 7.80% | USEC 2.96% | AppArmor 3.21%

2 Filebench

Filesystem & I/O intensive

Performance degradation vs baseline

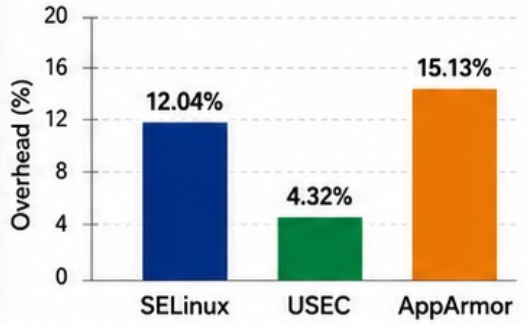


SELinux 10.86% | USEC 6.87% | AppArmor 9.51%

Nginx (Throughput)

Web server workload

Throughput overhead vs baseline



SELinux 12.04% | USEC 4.32% | AppArmor 15.13%

Key Takeaway

USEC remains consistently closer to the LSM-disabled baseline across CPU, filesystem, and web-server workloads.

Nginx Latency (Lower is Better)

P50 latency:
3.569 ms USEC vs 3.710 ms SELinux

P99 latency:
12.140 ms USEC vs 12.990 ms SELinux

UnixBench Results (Table 3)

Performance Degradation vs Baseline (LSM Disabled)			
SELinux	USEC	AppArmor	LSM Disabled
7.80%	2.96%	3.21%	0% (Baseline)

Filebench Results (Table 4)

Performance Degradation vs Baseline (LSM Disabled)			
SELinux	USEC	AppArmor	LSM Disabled
10.86%	6.87%	9.51%	0% (Baseline)

Nginx Results (Table 5)

Throughput Overhead vs Baseline				Latency (ms) — Lower is Better	
SELinux	USEC	AppArmor	LSM Disabled	P50	P99
12.04%	4.32%	15.13%	0% (Baseline)	3.710	12.990
				3.569	12.140

SELinux
USEC
AppArmor
LSM Disabled (Baseline)

Across CPU, filesystem, and web-server workloads, **USEC shows the lowest overhead,** staying much closer to the **LSM-disabled baseline.**

Up to **60%** lower overhead than SELinux in CPU-bound (UnixBench) workloads.

~37% lower overhead than SELinux in filesystem (Filebench) workloads.

~64% lower throughput overhead than SELinux in web-server (Nginx) workloads.

Evaluation III - Compatibility, Deployment, and Takeaways

1 Compatibility Cases

Case 1: Mitigating SELinux-induced failures

- USEC uses templates and D-Bus-based configuration.
- Protects selected resources without breaking applications.

```
policy_template_t g_policy_template[MAX_ID_TYPE] = {
    [APPID_TYPE] = {
    .policy_template = "type deepin_app_@n_t;\n"
        "deepin app domain_set(deepin_app_@n_t)\n"
        "domain unkillable(deepin_app_@n_t)\n"
        "type deepin_app_file_t;\n"
        "deepin_app_private_file_type_set(\
        "deepin_app_@n_ file_t)\n"
        "allow deepin_app_@n_t
         deepin_app_@n_file_t:dir
          file_class_set ~{ relabelfrom relabelto };\n"
        "deepin_access_all_sensitive_resource(deepin_app_@n)\n\n",
    .template_escape_cnt = 0
},
```

Figure 11: Template → SELinux fragment

Case 2: Matching SELinux protection and compatibility

- A subset of SELinux policy can be loaded into USEC.
- USEC preserves protection while keeping runtime state separate.

```
# Put USEC in permissive mode during migration
$ busctl call \
org.deepin.PernissionManager \
/org.deepin/PermissionManager/PolicyManager \
org.deepin.PermissionManager.PolicyManager \
DisableSecurity

# Copy the SELinux policy fragment into USEC's policy set
$ cp /etc/selinux/targeted/policy/usec_test.pp \
/etc/usec/policy/usec_test.pp

# Load policy into USEC
$ usec-policy-load /etc/usec/policy/usec_test.pp

# Re-enable USEC enforcement via the system security manager
$ busctl call \
org.deepin.PermissionManager \
/org.deepin.PermissionManager/PolicyManager \
org.deepin.PermissionManager.PolicyManager \\
EnableSecurity
```

Figure 12: SELinux fragment → USEC migration

2 Real-World Impact

- Adopted by **210+** security vendors
- Deployed on **8,000,000+** enterprise endpoints
- Used in finance, energy, and transportation
- Strong kernel protection can be practical at scale

3 Lessons Learned

- Hook retention must be **kernel-aware** and re-instantiated for the target environment.
- Deployability depends on shrinking the **enforcement surface** in practice.

Key Insight: Strong MAC becomes deployable when protection is resource-centric, demand-driven, and compatibility-aware.

OSDI 26

USENIX ASSOCIATION

Thank You

Yu Jiang, Wenhuan Liu, Fuchen Ma, Yuheng Shen, Yuanliang Chen
Lei Zhang, He Li, Quan Zhang, Chijin Zhou





