Easy Anti-Cheat (EAC) Technical Playbook

Payson - github.com/paysonism

A professional whitepaper on the architecture, detection layers, and defensive engineering strategies of Epic Games' Easy Anti-Cheat (EAC).

1. Introduction

Easy Anti-Cheat (EAC) is a leading anti-cheat solution developed by Epic Games and widely used across popular titles such as Fortnite. EAC protects online ecosystems from cheating through a layered defense model: client-side modules, kernel-level monitoring, and server-side analytics. This playbook is intended to serve as a professional, technical reference for engineers, security researchers, and developers working on cheat prevention.

2. Core Components of EAC

- Client Module: Integrated SDK that runs within the game client to perform integrity checks and telemetry collection.
- Kernel Driver: Runs at ring-0 to detect tampering, unauthorized memory access, and suspicious kernel activity.
- Backend & Analytics: Cloud infrastructure that aggregates telemetry, applies ML-based models, and enforces bans.
- Game Developer Integration: Developers can define server-authoritative invariants and integrate EAC APIs into game workflows.

Sidebar: What is a Kernel Driver?

A kernel driver is low-level software that runs with high privileges in the operating system. EAC uses a kernel driver to monitor memory, detect hidden processes, and block unauthorized system calls. While powerful, kernel drivers must be carefully maintained for compatibility and security.

3. Detection Strategies in EAC

- Integrity Verification: Hashes and checksums of binaries, assets, and modules are validated to ensure no tampering.
- Runtime Monitoring: Memory regions and process tables are scanned for injected modules or suspicious activity.
- Driver Vetting: EAC detects unauthorized or blacklisted kernel drivers attempting to manipulate memory.
- Behavioral Analytics: Player actions are profiled, and statistical models flag impossible or anomalous activity.
- Heuristic Detection: EAC applies rulesets to detect known cheat tools, debuggers, or suspicious I/O patterns.

4. Technical Previews

Example: File Integrity Check (pseudo-code)

if hash(file_binary) != expected_hash: flag_violation("Client binary mismatch detected")

Example: Server Authoritative Reconciliation

if client_position not within server_calculated_position ± tolerance: flag_violation("Suspicious movement - possible speedhack")

Example: Unauthorized Driver Detection

for driver in loaded_drivers: if driver.signature not in trusted_list: flag_violation("Unrecognized kernel driver")

5. Current Challenges & Limitations

- Compatibility: Kernel driver conflicts with Windows HVCI and Secure Boot.
- Advanced Evasion: Hypervisor-level cheats and VM-based exploits remain difficult to detect.
- Machine Learning Drift: Models require constant retraining to avoid false positives.
- Rapid Cheat Development: Developers must maintain agile CI/CD pipelines for rule updates.

6. Defensive Recommendations

- Adopt layered defenses across kernel, user-mode, and server environments.
- Prioritize server-authoritative logic for all critical game state variables.
- Deploy deception strategies such as honeypot entities to identify cheat clients.
- Automate signature and heuristic updates with continuous integration pipelines.
- Align kernel driver behavior with OS vendor security frameworks for reduced compatibility risk.
- Conduct frequent red-team engagements simulating advanced cheats.

7. Immediate Next Steps

• Map out and enforce server authoritative invariants for core gameplay systems.
• Audit telemetry pipelines for data quality, coverage, and labeling accuracy.
• Expand compatibility testing for kernel drivers across multiple Windows versions.
• Deploy deception-based trap entities to monitor cheat interactions.
• Schedule targeted red-team exercises to assess hypervisor-level attack vectors.