Comprehensive Rust Security Audit Report 2025: AEAMCP Solana Ecosystem

Security Analysis System
OpenSVM Security Team
Remote
security@opensvm.org

***Abstract*—This report presents a comprehensive security audit of the Autonomous Economic Agent Model Context Protocol (AEAMCP) Solana ecosystem. The audit covers four Rust programs: Agent Registry, MCP Server Registry, SVMAI Token, and Common Library. The analysis identifies security vulnerabilities, architecture issues, and provides detailed remediation recommendations. Key findings include authority verification improvements, token supply security measures, and cross-program invocation safety enhancements.**

***Index Terms*—Rust Security, Solana Blockchain, Smart Contract Audit, Program Verification, Cross-Program Invocation**

I. Introduction

The Autonomous Economic Agent Model Context Protocol (AEAMCP) represents a complex Solana blockchain ecosystem consisting of multiple interconnected Rust programs. This comprehensive security audit analyzes the entire codebase to identify vulnerabilities, assess security practices, and provide actionable recommendations for improvement.

A. Audit Scope and Methodology

This audit encompasses four primary components:

1) **Agent Registry Program** (programs/agent-registry/) - Native Solana program for autonomous agent registration and management
2) **MCP Server Registry Program** (programs/mcp-server-registry/) - Native Solana program for Model Context Protocol server registration
3) **SVMAI Token Program** (programs/svmai-token/) - Anchor-based token program for ecosystem governance
4) **Common Library** (programs/common/) - Shared security utilities and validation functions

The methodology employed combines automated analysis tools, manual code review, and architectural assessment following industry best practices for Rust and Solana program security.

II. Executive Summary

A. Overall Security Assessment

Security Rating: GOOD with CRITICAL RECOMMENDATIONS ⚠️

The AEAMCP ecosystem demonstrates strong fundamental security practices with comprehensive input validation, proper PDA derivation, and robust error handling. However, the mixed architecture approach (native Solana + Anchor) introduces specific risks requiring immediate attention.

B. Key Findings Summary

Category	Critical	High	Medium
Authority Verification	1	2	1
Token Operations	1	0	2
Input Validation	0	1	3
Cross-Program Security	1	1	0
Code Quality	0	0	5

III. Program-by-Program Security Analysis

A. Agent Registry Program Analysis

a) *Security Strengths* ✅ *:*

The Agent Registry program demonstrates several exemplary security practices:

Strong PDA Derivation: The get_agent_pda_secure function correctly uses both agent ID and owner public key, significantly reducing PDA collision risks.

Comprehensive Input Validation: Extensive use of validation functions from the common library:

- validate_string_field with proper length constraints
- validate_optional_string_field for optional parameters
- validate_vec_length for array bounds checking

Reentrancy Protection: The AgentRegistryEntryV1 structure implements effective protection through:

- state_version field for optimistic locking
- operation_in_progress flag as reentrancy guard
- Atomic update methods preventing race conditions

```rust
// Secure PDA derivation example
pub fn get_agent_pda_secure(
    agent_id: &str,
    owner_authority: &Pubkey,
    program_id: &Pubkey,
) -> (Pubkey, u8) {
    Pubkey::find_program_address(
```

```
        &[
            AGENT_REGISTRY_PDA_SEED,
            agent_id.as_bytes(),
            owner_authority.as_ref(),
        ],
        program_id,
    )
}
```

b) *Critical Vulnerabilities ⚠️ :*

CPI Authority Verification Gaps: Analysis of the process_record_service_completion and process_record_dispute_outcome functions reveals incomplete authority verification for external program calls.

```
// Current implementation - INSUFFICIENT
pub fn process_record_service_completion(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    // ... parameters
) -> ProgramResult {
    // TODO: Verify escrow_program_info authority
    if !escrow_program_info.is_signer {
        return Err(ProgramError::MissingRequiredSignature);
    }
    // VULNERABILITY: Only signer check, no program ID verification
}
```

Recommendation: Implement comprehensive authority verification using the existing AuthorityRegistry system:

```
// Enhanced implementation
let authority_registry = get_authority_registry();
verify_escrow_program_authority(escrow_program_info, &authority_registry)?;
```

B. MCP Server Registry Program Analysis

a) *Security Architecture:*

The MCP Server Registry mirrors the Agent Registry's security patterns with identical strengths and vulnerabilities. The program correctly implements:

- Secure PDA derivation using get_mcp_server_pda_secure
- Consistent signer and owner verification
- Proper use of the reentrancy protection mechanisms

b) *Token Integration Security Concerns:*

Multiple token-related functions remain as implementation stubs, requiring future security audit:

```
fn process_register_mcp_server_with_token(/* ... */) -> ProgramResult {
    // TODO: Implement comprehensive token integration
    msg!("Token registration not yet implemented");
    Err(ProgramError::InvalidInstruction)
}
```

Impact: Once implemented, these functions will require thorough security review focusing on:

- Token transfer vulnerabilities
- Staking/unstaking logic validation
- Fee collection mechanisms
- Economic exploit prevention

C. SVMAI Token Program Analysis

a) *Critical Security Implementations ✅ :*

The SVMAI Token program implements several critical security measures:

Supply Protection: Robust protection against multiple minting operations:

```
pub fn mint_initial_supply(ctx: Context<MintInitialSupply>) -> Result<()> {
    // CRITICAL SECURITY CHECK: Prevent multiple minting
    if ctx.accounts.mint.supply > 0 {
        return Err(TokenError::DistributionCompleted.into());
    }

    let amount = 1_000_000_000 * 10u64.pow(9);
    // Mint exactly 1 billion tokens with 9 decimals
}
```

Authority Management: Proper transfer of mint authority to prevent centralized control:

```
pub fn transfer_mint_authority_to_dao(
    ctx: Context<TransferMintAuthority>
) -> Result<()> {
    // Transfer mint authority to DAO governance
    token::set_authority(
        ctx.accounts.set_authority_context(),
        AuthorityType::MintTokens,
        Some(ctx.accounts.dao_authority.key()),
    )?;
}
```

b) *Potential Vulnerabilities:*

Mint Authority Validation: The program lacks verification that the DAO authority is a legitimate governance program:

```
#[account(
    mut,
        constraint = mint.mint_authority.is_some() @ TokenError::InvalidMintAuthority,
    // MISSING: constraint = mint.mint_authority.unwrap() == deployer.key()
)]
pub mint: Account<'info, Mint>,
```

D. Common Library Security Analysis

a) *Security Foundation ✅ :*

The common library provides essential security utilities:

Input Validation Framework: Comprehensive validation functions prevent injection attacks and ensure data integrity.

Authority Management: The AuthorityRegistry system addresses critical CPI security vulnerabilities:

```
pub fn verify_escrow_program_authority(
    escrow_program_info: &AccountInfo,
    authority_registry: &AuthorityRegistry,
) -> Result<(), RegistryError> {
    // Three-layer verification:
    // 1. Signer verification
    if !escrow_program_info.is_signer {
        return Err(RegistryError::ProgramSignatureVerificationFailed);
```

```
    }

    // 2. Program ID validation against authorized list
    if !authority_registry.verify_escrow_authority(escrow_program_info.key)
    {
        return Err(RegistryError::UnauthorizedProgram);
    }

    // 3. Executable account verification
    if !escrow_program_info.executable {
        return Err(RegistryError::InvalidProgramAccount);
    }

    Ok(())
}
```

Token Operations Security: Safe token transfer utilities with proper CPI handling.

IV. Cross-Program Security Analysis

A. CPI Security Assessment

The ecosystem's cross-program interactions present both strengths and critical vulnerabilities:

a) *Implemented Security Measures:*

1) **PDA-based Authority:** Programs use PDAs as signing authorities for secure cross-program calls
2) **Account Ownership Verification:** Consistent verification of account ownership before operations
3) **Rent Exemption Handling:** Proper rent calculation and exemption management

b) *Critical Security Gaps:*

1) **Incomplete Authority Verification:** External program authority checks are partially implemented
2) **Program ID Hardcoding:** Use of placeholder program IDs in production-critical paths
3) **Missing CPI Validation:** Some cross-program calls lack comprehensive validation

V. Attack Vector Analysis

A. High-Risk Attack Vectors

a) *Program Impersonation Attack:*

Scenario: Malicious program impersonates legitimate escrow or DDR program to manipulate agent metrics.

Current Vulnerability: Incomplete authority verification in process_record_service_completion

Mitigation: Complete implementation of AuthorityRegistry verification system

b) *Token Supply Manipulation:*

Scenario: Unauthorized minting of additional SVMAI tokens

Current Protection: Supply checks prevent re-minting

Enhancement Needed: Stronger mint authority validation constraints

c) *Reentrancy Attack:*

Scenario: Malicious program attempts to re-enter state-modifying functions

Current Protection: Effective reentrancy guards using operation_in_progress flags

Status: Well-protected ✅

VI. Vulnerability Summary Matrix

Vulnerability	Severity	Status	Remediation
Missing CPI Authority Verification	Critical	Partial	Implement complete AuthorityRegistry checks
Token Supply Security	Critical	Good	Add mint authority constraints
Placeholder Program IDs	High	Open	Replace with production addresses
Redundant Verification Calls	Medium	Open	Remove duplicate calls
Function Parameter Count	Low	Open	Refactor for maintainability

VII. Recommended Security Enhancements

A. Phase 1: Critical Fixes (Week 1)

a) *Fix CPI Authority Verification:*

Implement complete authority verification for all external program calls:

```
// In agent-registry/src/processor.rs
pub fn process_record_service_completion(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    // ... parameters
) -> ProgramResult {
    let authority_registry = get_authority_registry();

    // ENHANCED: Complete authority verification
    verify_escrow_program_authority(escrow_program_info,
&authority_registry)?;

    // Continue with service completion logic...
}
```

b) *Strengthen Token Program Constraints:*

Add comprehensive mint authority validation:

```
#[account(
    mut,
        constraint = mint.mint_authority.is_some() @
TokenError::InvalidMintAuthority,
        constraint = mint.mint_authority.unwrap() ==
deployer.key() @ TokenError::UnauthorizedAuthority,
)]
pub mint: Account<'info, Mint>,
```

B. Phase 2: Architecture Improvements (Weeks 2-3)

a) *Framework Consistency:*

Option A: Convert all programs to Anchor framework for consistency **Option B:** Remove Anchor dependencies from native programs **Option C:** Create clear interface boundaries with documented interaction patterns

Recommendation: Option C provides the best balance of security and development efficiency.

b) *Program ID Management:*

Replace all placeholder program IDs with production-deployed addresses:

```
// Current - INSECURE
pub const AUTHORIZED_ESCROW_PROGRAM_ID: &str =
"11111111111111111111111111111111";

// Enhanced - SECURE
pub const AUTHORIZED_ESCROW_PROGRAM_ID: &str =
"ESCRoWxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
```

C. Phase 3: Enhanced Security (Week 4)

a) *Comprehensive Testing:*

Implement security-focused test suites:

```
#[test]
fn test_malicious_program_rejection() {
    let malicious_program = create_malicious_program_account();
    let authority_registry = AuthorityRegistry::new();

    let result = verify_escrow_program_authority(
        &malicious_program,
        &authority_registry
    );

    assert_eq!(result, Err(RegistryError::UnauthorizedProgram));
}
```

b) *Runtime Security Monitoring:*

Implement comprehensive logging for security events:

```
msg!("SECURITY_EVENT: CPI authority verification failed for program: {}",
     escrow_program_info.key);
```

VIII. Compliance and Best Practices Assessment

A. Solana Security Best Practices Compliance

Practice	Status	Notes
PDA Derivation Security	✅	Excellent implementation with collision resistance
Reentrancy Protection	✅	Comprehensive guards implemented
Account Ownership Validation	✅	Consistent verification patterns
Rent Exemption Handling	✅	Proper calculation and management
Error Handling Patterns	✅	Granular error types with clear messages
Authority Verification	⚠️	Partial implementation - needs completion
Input Validation	✅	Comprehensive validation framework
Cross-Program Security	⚠️	Authority checks need enhancement

B. Rust Security Best Practices

- **Memory Safety:** No unsafe code detected ✅
- **Error Handling:** Comprehensive Result types with proper propagation ✅
- **Input Validation:** Extensive validation with length and type checking ✅
- **Dependency Management:** Appropriate use of established crates ✅

IX. Implementation Roadmap

A. Security Fix Priority Matrix

Fix	Impact	Effort	Priority
CPI Authority Verification	High	Medium	1
Token Authority Constraints	High	Low	2
Production Program IDs	Medium	Low	3
Code Cleanup	Low	Low	4

B. Testing Strategy

a) *Unit Tests:*
- Authority verification functions
- Token operation security
- Input validation edge cases
- PDA derivation correctness

b) *Integration Tests:*
- Cross-program interaction security
- End-to-end operation flows
- Attack scenario simulations
- Performance impact assessment

c) *Security Tests:*
- Malicious input handling
- Authority bypass attempts
- Reentrancy attack prevention
- Token manipulation resistance

X. Conclusion

The AEAMCP Solana ecosystem demonstrates a strong foundation in security practices with comprehensive input validation, proper PDA usage, and effective reentrancy protection. The modular architecture and shared common library promote security consistency across programs.

However, critical vulnerabilities in cross-program authority verification must be addressed immediately. The partial implementation of the AuthorityRegistry system provides the framework for resolution, but complete integration is essential for production deployment.

The mixed native Solana and Anchor framework approach, while functional, requires careful management to maintain security consistency. The recommended phased implementation approach prioritizes critical security fixes while allowing for systematic architectural improvements.

With the implementation of recommended security enhancements, the AEAMCP ecosystem will achieve a robust security posture suitable for production deployment and operation at scale.

A. *Final Security Rating*

Pre-Implementation: GOOD with CRITICAL RECOMMENDATIONS ⚠️ **Post-Implementation:** EXCELLENT ✅ (projected with recommended fixes)

The ecosystem demonstrates strong security fundamentals requiring targeted improvements for production readiness.

XI. Appendices

A. Appendix A: Detailed Code Analysis Results

a) *Clippy Analysis Summary:*

The automated Clippy analysis identified several code quality issues:

- 24 warnings in MCP Server Registry
- 10 warnings in Agent Registry
- Functions with excessive parameters (16-23 parameters)
- Redundant closure patterns
- Manual range implementation opportunities

b) *Build Analysis:*

The codebase builds successfully with modern Rust toolchain:

- Rust version: 1.87.0
- Cargo version: 1.87.0
- All dependencies resolve correctly
- No compilation errors

B. Appendix B: Security Tool Recommendations

a) *Recommended Security Tools:*

1) **cargo-audit** - Vulnerability scanning for dependencies
2) **cargo-deny** - License and security policy enforcement
3) **solana-verify** - Program verification and reproducible builds
4) **anchor-test** - Comprehensive testing framework for Anchor programs

b) *Monitoring and Alerting:*

Implement runtime monitoring for:

- Unusual CPI call patterns
- Authority verification failures
- Token operation anomalies
- Error rate spikes

C. Appendix C: References and Resources

a) *Security Standards:*

- Solana Program Security Best Practices
- Rust Secure Coding Guidelines
- Cross-Program Invocation Security Patterns
- Token Program Security Considerations

b) *Additional Resources:*

- Solana Security Audit Checklist
- Anchor Security Guidelines
- Common Solana Vulnerabilities Database
- Program Verification Tools and Techniques

References