Security Audit Report: MockFrost

NiM Security Team

August 11, 2025

1 Executive Summary

This document presents the findings of a security audit conducted on the MockFrost repository, a mock implementation of the BlockFrost API for testing Cardano smart contracts. The audit focused on identifying potential security vulnerabilities, with particular attention to transaction execution, resource management, and data integrity.

2 Scope

The audit covered:

- Transaction submission and validation
- Script evaluation mechanisms
- Session management
- Resource allocation and limits
- Input validation
- Access control mechanisms

3 Critical Findings

3.1 Resource Exhaustion Vulnerabilities

3.1.1 Memory Management in Script Evaluation

Severity: High

The script evaluation mechanism in `tx_tools.py` contains several critical vulnerabilities related to memory management that could lead to denial of service:

Unbounded Script Cache The `@cache` decorator on `uplc_unflat` creates an unbounded memory leak:

- Each unique script processed is permanently stored in memory (`tx_tools.py:338`):

  ```
  @cache
  def uplc_unflat(script: bytes):
      return uplc.unflatten(script)
  ```

- The cache has no size limit or eviction policy
- Malicious users could submit many unique scripts to exhaust server memory through `submit_a_transaction` (`server.py:307`)
- The cache persists across sessions, allowing cumulative memory growth

Unconstrained Script Size The system lacks input size validation in `submit_a_transaction` (`server.py:307-318`):

```
@app.post("/{session_id}/api/v0/tx/submit")
def submit_a_transaction(
    session_id: uuid.UUID,
    transaction: Annotated[bytes, Body(media_type="application/cbor")],
) -> str:
    return get_session(session_id).chain_state.transaction_submit_raw(
        transaction, return_type="json"
    )
```

Key issues:

- No maximum size limit is enforced on individual script inputs
- Large scripts can be submitted that consume excessive memory during parsing
- The CBOR decoder used for script deserialization lacks size limits
- No validation of script complexity or nesting depth in `evaluate_script` (`tx_tools.py:346-366`):

  ```
  def evaluate_script(script_invocation: ScriptInvocation):
      uplc_program = uplc_unflat(script_invocation.script)
      args = [script_invocation.redeemer.data, script_invocation.script_context]
      if script_invocation.datum is not None:
          args.insert(0, script_invocation.datum)
      args = [uplc_plutus_data(a) for a in args]
      allowed_cpu_steps = script_invocation.redeemer.ex_units.steps
      allowed_mem_steps = script_invocation.redeemer.ex_units.mem
  ```

```
res = uplc.eval(
    uplc.tools.apply(uplc_program, *args),
    budget=uplc.cost_model.Budget(allowed_cpu_steps, allowed_mem_steps),
)
```

Uncontrolled Memory Allocation Memory usage during script evaluation has incomplete controls:

- While `allowed_mem_steps` limits execution memory, initial loading is unbounded
- Script parsing and preparation can consume arbitrary amounts of memory
- No overall process memory monitoring or limits
- Memory allocated for intermediate data structures is not tracked

Recommendations:

- Replace `@cache` with a size-limited LRU cache implementation
- Add strict size limits for script inputs (e.g., 1MB maximum)
- Implement memory usage monitoring during script loading phases

Implementation Example

```
from functools import lru_cache
from typing import Optional
import resource

# Set process memory limit
def set_memory_limit(max_bytes: int):
    resource.setrlimit(
        resource.RLIMIT_AS,
        (max_bytes, max_bytes)
    )

# Replace unlimited cache with size-limited version
@lru_cache(maxsize=1000)
def uplc_unflat(script: bytes) -> Optional[object]:
    if len(script) > 1_000_000:  # 1MB limit
        raise ValueError("Script too large")
    return uplc.unflatten(script)
```

Status Resolved.

3.1.2 Session Management

Severity: High

The session management system in `server.py` contains critical security vulnerabilities that could lead to resource exhaustion and potential system compromise:

Unbounded Session Creation The current implementation allows unrestricted session creation:

- No rate limiting on session creation requests
- Malicious users can create thousands of sessions to exhaust server resources

Session Persistence Issues Sessions are managed with poor lifecycle control:

- Sessions are stored indefinitely in the `SESSIONS` dictionary
- No session expiration mechanism
- No automatic cleanup of inactive sessions
- Memory usage grows linearly with the number of abandoned sessions

State Management Vulnerabilities Each session maintains significant state:

- Chain state (`MockFrostApi`) contains growing UTxO sets
- Transaction history accumulates without bounds
- No limits on state size per session
- Potential for memory exhaustion through state manipulation

Recommendations:

- Add rate limiting for session creation
- Add session expiration after a fixed timeout (e.g., 1 hour)
- Create a background task to clean up expired sessions
- Add a monitor to limit size of the chain state

Implementation Example

```
from datetime import datetime, timedelta
from typing import Dict, Optional
import uuid
from dataclasses import dataclass
from collections import defaultdict

@dataclass
class SessionLimits:
    max_sessions_per_ip: int = 5
    session_timeout: timedelta = timedelta(hours=1)
    max_state_size: int = 1_000_000  # 1MB

class SessionManager:
    def __init__(self, limits: SessionLimits):
        self.sessions: Dict[uuid.UUID, Session] = {}
        self.ip_sessions: Dict[str, set] = defaultdict(set)
        self.limits = limits

    def create_session(self, ip_address: str) -> Optional[uuid.UUID]:
        # Check IP session limit
        if len(self.ip_sessions[ip_address]) >= self.limits.max_sessions_per_ip:
            raise ValueError("Session limit exceeded for IP")

        # Create new session
        session_id = uuid.uuid4()
        session = Session(
            chain_state=MockFrostApi(),
            creation_time=datetime.now(),
            last_access_time=datetime.now()
        )

        self.sessions[session_id] = session
        self.ip_sessions[ip_address].add(session_id)
        return session_id

    def cleanup_expired(self):
        now = datetime.now()
        expired = [
            sid for sid, session in self.sessions.items()
            if (now - session.last_access_time) > self.limits.session_timeout
        ]
        for sid in expired:
            self.delete_session(sid)
```

Status Resolved.

4 Medium Severity Findings

4.1 State Management Issues

4.1.1 UTxO State Management

Severity: Medium

The UTxO state management system in `mock.py` contains several security concerns that could affect system stability and reliability:

UTxO Set Growth The UTxO management lacks proper bounds:

- Unbounded growth of the UTxO set over time
- No cleanup mechanism for spent or expired UTxOs
- Memory usage grows linearly with UTxO set size

Concurrency Issues The UTxO state updates lack proper synchronization:

- `SESSION` object is not shared between parallel workers of the server, limiting the amount of workers to 1
- No atomic operations for UTxO state updates
- Potential race conditions during parallel transaction processing
- Missing locks for concurrent access to shared state
- Possible inconsistencies in UTxO set during high load

Reference Validation UTxO reference handling has security gaps:

- Insufficient validation of UTxO existence before use
- Missing checks for UTxO ownership
- Incomplete validation of UTxO data integrity
- Potential for reference manipulation attacks

Recommendations:

- Add proper locking mechanisms for state updates
- Enhance UTxO reference validation
- Add UTxO cleanup mechanisms

Implementation Example

```
from dataclasses import dataclass
from typing import Dict, Set
from threading import Lock
from pycardano import TransactionInput, TransactionOutput, Address

class UTxOManager:
    def __init__(self, limits: UTxOLimits):
        self.limits = limits
        self.utxos: Dict[TransactionInput, TransactionOutput] = {}
        self.address_utxos: Dict[Address, Set[TransactionInput]] = {}
        self.lock = Lock()

    def add_utxo(self, tx_input: TransactionInput,
                 tx_output: TransactionOutput) -> bool:
        with self.lock:
            # Validate and add UTxO
            if tx_input in self.utxos:
                raise ValueError("UTxO already exists")

            self.utxos[tx_input] = tx_output
            addr_utxos.add(tx_input)
            self.address_utxos[tx_output.address] = addr_utxos
            return True

    def remove_utxo(self, tx_input: TransactionInput) -> bool:
        with self.lock:
            if tx_input not in self.utxos:
                return False

            tx_output = self.utxos[tx_input]
            addr_utxos = self.address_utxos[tx_output.address]
            addr_utxos.remove(tx_input)

            if not addr_utxos:
                del self.address_utxos[tx_output.address]

            del self.utxos[tx_input]
            return True
```

Status Resolved.

5 Low Severity Findings

5.1 Error Handling

5.1.1 Exception Management

Severity: Low

The error handling system throughout the codebase has several areas that could be improved to enhance security and reliability:

Exception Information Exposure Current error handling may leak sensitive information:

- Raw exception messages are returned to users
- Stack traces may be exposed in error responses
- Internal system paths and configurations visible in errors
- Database/storage errors may reveal implementation details

Inconsistent Error Handling Error handling lacks standardization:

- Different error formats across endpoints
- Inconsistent error status codes
- Missing error categorization
- Incomplete error documentation

Logging Deficiencies The logging system has several gaps:

- Missing logs for critical security events
- Inconsistent log levels
- No structured logging format
- Insufficient error context in logs

Recommendations:

- Implement standardized error responses
- Add proper error sanitization
- Enhance logging coverage
- Create error categorization system

Implementation Example

```
from enum import Enum
from typing import Optional, Dict, Any
from dataclasses import dataclass
import logging
import traceback

class ErrorCategory(Enum):
    VALIDATION = "validation"
    SECURITY = "security"
    SYSTEM = "system"
    RESOURCE = "resource"

@dataclass
class ErrorResponse:
    error_code: str
    message: str
    category: ErrorCategory
    details: Optional[Dict[str, Any]] = None

class ErrorHandler:
    def __init__(self):
        self.logger = logging.getLogger("security")

    def handle_error(self,
                     exception: Exception,
                     category: ErrorCategory) -> ErrorResponse:
        # Generate error code
        error_code = f"{category.value}_{exception.__class__.__name__}"

        # Log error with context
        self.logger.error(
            "Error occurred",
            extra={
                "error_code": error_code,
                "error_type": exception.__class__.__name__,
                "stack_trace": traceback.format_exc()
            }
        )

        # Create sanitized response
        return ErrorResponse(
            error_code=error_code,
            message=self.sanitize_error_message(str(exception)),
            category=category
```

```
        )

    def sanitize_error_message(self, message: str) -> str:
        # Remove sensitive information
        sanitized = message.replace(
            "/workspace", "[REDACTED]"
        ).replace(
            "/home", "[REDACTED]"
        )

        # Limit length
        if len(sanitized) > 200:
            sanitized = sanitized[:197] + "..."

        return sanitized
```

Status Acknowledged.

5.2 Input Sanitization

Severity: Low

The input validation and sanitization mechanisms have several areas that need improvement:

Address Validation Address handling lacks comprehensive validation:

- Incomplete validation of address format
- Missing checks for address network compatibility
- No validation of address encoding
- Potential for address manipulation attacks

Script Parameter Validation Script parameter handling needs enhancement:

- Insufficient validation of script parameters
- Missing type checks for complex parameters
- No validation of parameter ranges
- Potential for parameter injection attacks

Numeric Input Validation Numeric input handling lacks proper bounds checking:

- Missing range validation for numeric inputs
- No overflow checks for arithmetic operations
- Insufficient precision handling
- Potential for numeric overflow attacks

Recommendations:

- Implement comprehensive address validation
- Add proper parameter type checking
- Enhance numeric input validation
- Create input sanitization utilities

Implementation Example

```
from dataclasses import dataclass
from typing import Optional, Union, Any
from decimal import Decimal
import re

@dataclass
class ValidationLimits:
    max_string_length: int = 1000
    max_integer_value: int = 1_000_000_000
    max_decimal_places: int = 6

class InputValidator:
    def __init__(self, limits: ValidationLimits):
        self.limits = limits
        self.address_pattern = re.compile(
            r'^addr1[a-zA-Z0-9]{53}$'
        )

    def validate_address(self, address: str) -> bool:
        if not isinstance(address, str):
            raise ValueError("Address must be a string")

        if len(address) > self.limits.max_string_length:
            raise ValueError("Address too long")

        if not self.address_pattern.match(address):
```

```
            raise ValueError("Invalid address format")

        return True

    def validate_numeric(
        self,
        value: Union[int, float, Decimal],
        min_value: Optional[Union[int, float]] = None,
        max_value: Optional[Union[int, float]] = None
    ) -> bool:
        if isinstance(value, float):
            # Check decimal places
            str_value = str(value)
            if '.' in str_value:
                decimals = len(str_value.split('.')[1])
                if decimals > self.limits.max_decimal_places:
                    raise ValueError("Too many decimal places")

        if isinstance(value, int):
            if abs(value) > self.limits.max_integer_value:
                raise ValueError("Integer value too large")

        if min_value is not None and value < min_value:
            raise ValueError(f"Value below minimum: {min_value}")

        if max_value is not None and value > max_value:
            raise ValueError(f"Value above maximum: {max_value}")

        return True

    def sanitize_string(
        self,
        value: str,
        allow_html: bool = False
    ) -> str:
        if not isinstance(value, str):
            raise ValueError("Input must be a string")

        if len(value) > self.limits.max_string_length:
            raise ValueError("String too long")

        # Remove control characters
        value = ''.join(char for char in value
                        if ord(char) >= 32)

        if not allow_html:
```

```
        # Basic HTML escaping
        value = value.replace(
            "&", "&amp;"
        ).replace(
            "<", "&lt;"
        ).replace(
            ">", "&gt;"
        )

    return value
```

Status Acknowledged.

6 Conclusion

While MockFrost provides a valuable testing environment for Cardano smart contracts, several security improvements are needed to ensure safe and reliable operation. The most critical issues revolve around resource management and concurrency. Implementing the recommended fixes will significantly improve the security posture of the system.