PhantomID Network System

Prototype Notice and Abuse Prevention Disclaimer

This system is currently a prototype implementation. While the core daemonized identity tree and Zero-Knowledge Proof (ZKP) verification mechanisms are functional, certain trust policies are still under active development. Specifically, when a user logs in through a child account, they are not intended to maintain cryptographic control over any subordinate child nodes after a parent account is deleted or invalidated. The "Orphaned Cryptographic State" mechanism ensures that child nodes regain autonomy and cannot be held "hostage" by malicious child account operators. This protects the integrity of the identity tree and prevents abuse scenarios where compromised subtrees could persist unauthorized control.

Key Prototype Limitations:

- Trust policy enforcement is still being refined
- Network synchronization between multiple daemon instances requires additional testing
- Post-quantum cryptographic migration is planned but not yet implemented
- Production deployment should include additional audit logging and monitoring capabilities

Overview

PhantomID is a daemon-based cryptographic identity management system that maintains a live, self-healing hierarchical tree of anonymous accounts. Unlike traditional PKI or blockchain-based identity systems, PhantomID operates as a persistent service that continuously validates parent-child relationships using Zero-Knowledge Proofs (ZKPs), ensuring network integrity without exposing sensitive identity information.

The system implements a novel approach to distributed identity management where nodes can survive parent deletion through an "Orphaned Cryptographic State" mechanism, allowing for network resilience and autonomous trust reconstruction.

Architecture

PhantomID employs a daemon-server architecture built around several core components:

```
phantomid-with-tree/
├── bin/              # Compiled binaries and shared libraries
├── obj/              # Intermediate build artifacts
├── main.c            # Daemon initialization and signal handling
├── network.c/.h      # TCP server and client connection management
├── phantomid.c/.h    # Core identity tree and ZKP implementation
├── Makefile*         # Build configurations for Unix/Windows
└── config/           # Runtime configuration files
```

Core Components

- **Phantom Daemon Process**: A persistent service that maintains the identity tree state and performs continuous cryptographic verification

- **Hierarchical Identity Tree**: A dynamic tree structure where each node represents a cryptographic identity with parent-child relationships
- **ZKP Verification Engine**: Schnorr-based zero-knowledge proof system for identity validation
- **Network Protocol Layer**: TCP-based communication for identity operations and tree synchronization
- **Orphan State Manager**: Handles node isolation and reparenting when tree integrity is compromised

Daemonized Process and ZKP-based Tree Maintenance

Root Account Initialization

Upon system startup, PhantomID creates a **Root Account** that serves as the cryptographic anchor for the entire identity tree. This root account:

- Generates a cryptographically secure seed using OpenSSL's RAND_bytes
- Derives a 64-character hexadecimal identity using SHA-256
- Establishes itself as the sole administrative node with `is_root=true` and `is_admin=true`
- Begins continuous self-verification using ZKP protocols

Live Tree Verification

The daemon continuously performs the following verification operations:

1. **Parent-Child Link Validation**: Each node's relationship to its parent is verified using Schnorr identification protocols
2. **Identity Freshness Checks**: Account expiration times are monitored and expired accounts are flagged for removal
3. **Cryptographic Integrity**: Hash chains and derived keys are validated against stored commitments
4. **Network Consensus**: When multiple daemon instances operate, they synchronize tree state using authenticated messages

ZKP Protocol Implementation

The system implements a modified Schnorr protocol for identity proofs:

```
// Commitment phase
t = g^r mod p

// Challenge from verifier
c ∈ Z_q (random)

// Response calculation
s = r + c * x_A mod q

// Verification equation
g^s ?= t * y_A^c mod p
```

Where `x_A` is the prover's private key, `y_A` is the public key, and the protocol ensures zero-knowledge disclosure.

Orphan Handling

Orphaned Cryptographic State

When a parent node is deleted or becomes cryptographically invalid, child nodes enter an **Orphaned Cryptographic State**. In this state:

- **Identity Preservation**: The node retains its cryptographic identity and can still prove possession of its private key
- **Relationship Severance**: All parent-child links are marked as invalid but not destroyed
- **Verification Capability**: The node can still participate in ZKP challenges and generate valid proofs
- **Administrative Isolation and Anti-Abuse**: The node loses all inherited administrative privileges from its deleted parent and cannot retain cryptographic control over subordinate nodes. Orphaned nodes regain autonomy to prevent abuse scenarios where child nodes could be held "hostage" by a malicious operator.

Recovery Mechanisms

Orphaned nodes have three primary recovery paths:

1. Isolation Mode

- Node continues operating independently
- Maintains internal cryptographic state
- Can service local authentication requests
- Cannot participate in broader network trust relationships

2. Subtree Formation

- Multiple orphaned nodes can form new trust relationships
- Mutual ZKP verification establishes lateral trust
- One node may be elected as a new subtree root through consensus
- Formed subtrees operate as independent cryptographic domains

3. Reparenting Negotiation

- Orphaned nodes can request adoption by existing valid nodes
- Requires explicit cryptographic proof exchange
- New parent must verify orphan's cryptographic validity
- Adoption is recorded with timestamps and audit trails

Network Healing Protocol

The daemon implements an autonomous healing protocol:

```
1. Detect parent node failure/deletion
2. Transition affected children to orphaned state
3. Broadcast orphan status to network participants
4. Initiate recovery protocol based on configured policy
```

```
5. Verify new relationships using fresh ZKP exchanges
6. Update tree structure and resume normal operations
```

Build and Run Instructions

Prerequisites

Linux/Unix Systems:

```
sudo apt-get install build-essential libssl-dev pkg-config
# or
sudo dnf install gcc openssl-devel pkgconfig
```

Windows Systems:

```
winget install ShiningLight.OpenSSL.Dev
# Install MinGW-w64 toolchain
```

Compilation

```
# Unix/Linux
make clean && make

# Windows
mingw32-make -f Makefile.win clean
mingw32-make -f Makefile.win
```

Running the Daemon

```
# Start daemon on default port 8888
./phantomid

# Custom configuration
./phantomid -p 9999 -v -d

# Available options:
# -p, --port PORT    Server port (default: 8888)
# -v, --verbose      Enable detailed logging
# -d, --debug        Debug mode with tree traversal output
# -h, --help         Display usage information
```

Client Operations

```
# Connect to daemon
nc localhost 8888

# Available commands:
create [parent_id]     # Create new account
delete <id>           # Delete account (triggers orphan handling)
msg <from> <to> <msg> # Send authenticated message
list [bfs|dfs]        # Display tree structure
help                  # Command reference
```

Security Model

Cryptographic Foundations

- **Hash Function**: SHA-256 for identity derivation, SHA-512 for HMAC operations
- **Random Generation**: OpenSSL RAND_bytes for cryptographically secure entropy
- **Key Derivation**: HMAC-based derived keys for purpose-specific identities
- **ZKP Protocol**: Schnorr identification with 256-bit security level

Thread Safety and Memory Protection

- **Mutex Protection**: All tree operations are protected by pthread mutexes
- **Memory Management**: Secure allocation with overflow detection and cleanup
- **Constant-Time Operations**: Verification operations resist timing attacks
- **Resource Isolation**: Client connections are isolated with separate state management

Attack Resistance

- **Malicious Node Deletion**: Orphan handling prevents cascade failures
- **Network Partition**: Subtrees can operate independently and merge when connectivity restores
- **Replay Attacks**: Timestamps and nonces prevent message replay
- **Timing Attacks**: Constant-time comparison functions for all cryptographic operations

Quantum Considerations

While primarily based on hash functions (which provide some quantum resistance), the current implementation uses discrete logarithm-based ZKPs. Future versions will incorporate post-quantum cryptographic primitives for full quantum resistance.

References and Papers

Formal Specifications

- Formal Proof of Zero-Knowledge Protocol and HMAC-based Derived Key Security
- Phantom Encoder Design Pattern for Zero-Knowledge Systems

Related Implementations

- **Python Phantom Encoder**: Available as part of the Node-Zero library

- **C Implementation**: This repository (PhantomID daemon)
- **Project Repository**: https://github.com/obinexuscomputing/phantomid-poc/

Cryptographic References

1. Schnorr, C.P. (1991). "Efficient signature generation by smart cards." Journal of Cryptology, 4(3), 161-174.
2. Goldwasser, S., Micali, S., & Rackoff, C. (1989). "The knowledge complexity of interactive proof systems." SIAM Journal on Computing, 18(1), 186-208.
3. Krawczyk, H., Bellare, M., & Canetti, R. (1997). "HMAC: Keyed-hashing for message authentication." RFC 2104.
