```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::system_program;
use solana_program_test::*;
use solana_sdk::{
    signature::{Keypair, Signer},
    transaction::Transaction,
};

// ==================== ANCHOR PROGRAM TESTS ====================

#[cfg(test)]
mod anchor_tests {
    use super::*;

    #[tokio::test]
    async fn test_create_ephemeral_vault() {
        let program_id = Pubkey::new_unique();
        let mut program_test = ProgramTest::new(
            "ephemeral_vault",
            program_id,
            processor!(ephemeral_vault::entry),
        );

        let parent = Keypair::new();
        program_test.add_account(
            parent.pubkey(),
            Account {
                lamports: 10_000_000_000,
                ..Account::default()
            },
        );

        let (mut banks_client, payer, recent_blockhash) = program_test.start().await;

        // Derive vault PDA
        let (vault_pda, _bump) = Pubkey::find_program_address(
            &[b"vault", parent.pubkey().as_ref()],
            &program_id,
        );

        // Create vault instruction
        let session_duration = 3600i64; // 1 hour
        let approved_amount = 1_000_000u64;

        let ix = create_vault_instruction(
            &program_id,
            &parent.pubkey(),
            &vault_pda,
```

```
        session_duration,
        approved_amount,
    );

    let mut transaction = Transaction::new_with_payer(&[ix], Some(&payer.pubkey()));
    transaction.sign(&[&payer, &parent], recent_blockhash);

    banks_client.process_transaction(transaction).await.unwrap();

    // Verify vault account
    let vault_account = banks_client.get_account(vault_pda).await.unwrap().unwrap();
    assert!(vault_account.data.len() > 0);
}

#[tokio::test]
async fn test_approve_delegate() {
    // Setup
    let program_id = Pubkey::new_unique();
    let mut program_test = ProgramTest::new(
        "ephemeral_vault",
        program_id,
        processor!(ephemeral_vault::entry),
    );

    let parent = Keypair::new();
    let ephemeral = Keypair::new();

    // Add accounts with balance
    program_test.add_account(
        parent.pubkey(),
        Account {
            lamports: 10_000_000_000,
            ..Account::default()
        },
    );

    let (mut banks_client, payer, recent_blockhash) = program_test.start().await;

    // First create vault
    let (vault_pda, _) = Pubkey::find_program_address(
        &[b"vault", parent.pubkey().as_ref()],
        &program_id,
    );

    // Create vault
    // ... (similar to above)

    // Now approve delegate
```

```
    let ix = approve_delegate_instruction(
        &program_id,
        &parent.pubkey(),
        &vault_pda,
        &ephemeral.pubkey(),
    );

    let mut transaction = Transaction::new_with_payer(&[ix], Some(&payer.pubkey()));
    transaction.sign(&[&payer, &parent], recent_blockhash);

    banks_client.process_transaction(transaction).await.unwrap();

    // Verify delegation
    let (delegation_pda, _) = Pubkey::find_program_address(
        &[b"delegation", vault_pda.as_ref()],
        &program_id,
    );

    let delegation_account = banks_client
        .get_account(delegation_pda)
        .await
        .unwrap()
        .unwrap();
    assert!(delegation_account.data.len() > 0);
}

#[tokio::test]
async fn test_auto_deposit_for_trade() {
    // Test auto-deposit functionality
    let program_id = Pubkey::new_unique();
    let mut program_test = ProgramTest::new(
        "ephemeral_vault",
        program_id,
        processor!(ephemeral_vault::entry),
    );

    let parent = Keypair::new();
    program_test.add_account(
        parent.pubkey(),
        Account {
            lamports: 100_000_000_000,
            ..Account::default()
        },
    );

    let (mut banks_client, payer, recent_blockhash) = program_test.start().await;

    let (vault_pda, _) = Pubkey::find_program_address(
```

```
        &[b"vault", parent.pubkey().as_ref()],
        &program_id,
    );

    // Create vault first
    // ...

    // Deposit
    let deposit_amount = 5_000_000u64; // 0.005 SOL
    let ix = auto_deposit_instruction(
        &program_id,
        &parent.pubkey(),
        &vault_pda,
        deposit_amount,
    );

    let mut transaction = Transaction::new_with_payer(&[ix], Some(&payer.pubkey()));
    transaction.sign(&[&payer, &parent], recent_blockhash);

    let result = banks_client.process_transaction(transaction).await;
    assert!(result.is_ok());

    // Verify vault balance increased
    let vault_account = banks_client.get_account(vault_pda).await.unwrap().unwrap();
    assert!(vault_account.lamports >= deposit_amount);
}

#[tokio::test]
async fn test_execute_trade_unauthorized() {
    // Test that unauthorized wallet cannot execute trade
    let program_id = Pubkey::new_unique();
    let mut program_test = ProgramTest::new(
        "ephemeral_vault",
        program_id,
        processor!(ephemeral_vault::entry),
    );

    let parent = Keypair::new();
    let unauthorized = Keypair::new();
    let ephemeral = Keypair::new();

    program_test.add_account(
        parent.pubkey(),
        Account {
            lamports: 10_000_000_000,
            ..Account::default()
        },
    );
```

```
    program_test.add_account(
        unauthorized.pubkey(),
        Account {
            lamports: 10_000_000_000,
            ..Account::default()
        },
    );

    let (mut banks_client, payer, recent_blockhash) = program_test.start().await;

    // Create vault and approve ephemeral (not unauthorized)
    // ...

    // Try to execute trade with unauthorized wallet
    let (vault_pda, _) = Pubkey::find_program_address(
        &[b"vault", parent.pubkey().as_ref()],
        &program_id,
    );

    let ix = execute_trade_instruction(
        &program_id,
        &unauthorized.pubkey(), // Wrong wallet
        &vault_pda,
        1, // trade_id
        5000, // fee
    );

    let mut transaction = Transaction::new_with_payer(&[ix], Some(&payer.pubkey()));
    transaction.sign(&[&payer, &unauthorized], recent_blockhash);

    let result = banks_client.process_transaction(transaction).await;
    assert!(result.is_err()); // Should fail
}

#[tokio::test]
async fn test_revoke_access() {
    // Test revocation functionality
    let program_id = Pubkey::new_unique();
    let mut program_test = ProgramTest::new(
        "ephemeral_vault",
        program_id,
        processor!(ephemeral_vault::entry),
    );

    let parent = Keypair::new();
    let ephemeral = Keypair::new();
```

```
    program_test.add_account(
        parent.pubkey(),
        Account {
            lamports: 10_000_000_000,
            ..Account::default()
        },
    );

    let (mut banks_client, payer, recent_blockhash) = program_test.start().await;

    // Create vault, approve delegate, deposit
    // ...

    let parent_balance_before = banks_client
        .get_balance(parent.pubkey())
        .await
        .unwrap();

    // Revoke
    let (vault_pda, _) = Pubkey::find_program_address(
        &[b"vault", parent.pubkey().as_ref()],
        &program_id,
    );

    let ix = revoke_access_instruction(&program_id, &parent.pubkey(), &vault_pda);

    let mut transaction = Transaction::new_with_payer(&[ix], Some(&payer.pubkey()));
    transaction.sign(&[&payer, &parent], recent_blockhash);

    banks_client.process_transaction(transaction).await.unwrap();

    // Verify funds returned
    let parent_balance_after = banks_client
        .get_balance(parent.pubkey())
        .await
        .unwrap();

    assert!(parent_balance_after >= parent_balance_before);
}

#[tokio::test]
async fn test_cleanup_expired_vault() {
    // Test cleanup of expired vault
    // Advance clock past expiry, verify cleanup works
    // Verify cleanup reward is paid
}

#[tokio::test]
```

```
async fn test_session_expiry() {
    // Test that operations fail after session expires
}

#[tokio::test]
async fn test_excessive_deposit_rejected() {
    // Test that deposits over limit are rejected
}

#[tokio::test]
async fn test_deposit_limit_enforcement() {
    // Test total deposit limit (100 SOL)
}

// Helper functions to build instructions
fn create_vault_instruction(
    program_id: &Pubkey,
    parent: &Pubkey,
    vault: &Pubkey,
    duration: i64,
    amount: u64,
) -> solana_sdk::instruction::Instruction {
    // Build instruction
    unimplemented!()
}

fn approve_delegate_instruction(
    program_id: &Pubkey,
    parent: &Pubkey,
    vault: &Pubkey,
    delegate: &Pubkey,
) -> solana_sdk::instruction::Instruction {
    unimplemented!()
}

fn auto_deposit_instruction(
    program_id: &Pubkey,
    parent: &Pubkey,
    vault: &Pubkey,
    amount: u64,
) -> solana_sdk::instruction::Instruction {
    unimplemented!()
}

fn execute_trade_instruction(
    program_id: &Pubkey,
    ephemeral: &Pubkey,
    vault: &Pubkey,
```

```
        trade_id: u64,
        fee: u64,
    ) -> solana_sdk::instruction::Instruction {
        unimplemented!()
    }

    fn revoke_access_instruction(
        program_id: &Pubkey,
        parent: &Pubkey,
        vault: &Pubkey,
    ) -> solana_sdk::instruction::Instruction {
        unimplemented!()
    }
}

// ==================== BACKEND SERVICE TESTS ====================

#[cfg(test)]
mod backend_tests {
    use super::*;

    #[tokio::test]
    async fn test_session_manager_create_session() {
        let db_pool = setup_test_database().await;
        let encryption_key = [0u8; 32];
        let session_manager = SessionManager::new(db_pool.clone(), encryption_key);

        let parent_wallet = Keypair::new().pubkey();
        let session = session_manager
            .create_session(parent_wallet, 3600, 1_000_000)
            .await
            .unwrap();

        assert_eq!(session.parent_wallet, parent_wallet);
        assert!(session.is_active);
        assert!(session.expires_at > chrono::Utc::now().timestamp());

        cleanup_test_database(db_pool).await;
    }

    #[tokio::test]
    async fn test_session_manager_revoke_session() {
        let db_pool = setup_test_database().await;
        let encryption_key = [0u8; 32];
        let session_manager = SessionManager::new(db_pool.clone(), encryption_key);

        let parent_wallet = Keypair::new().pubkey();
        let session = session_manager
```

```
        .create_session(parent_wallet, 3600, 1_000_000)
        .await
        .unwrap();

    session_manager
        .revoke_session(&session.session_id)
        .await
        .unwrap();

    // Verify session is inactive
    let result = session_manager.get_session(&session.session_id).await;
    assert!(result.is_ok());

    cleanup_test_database(db_pool).await;
}

#[tokio::test]
async fn test_auto_deposit_calculator() {
    let calculator = AutoDepositCalculator::new();

    let amount = calculator.calculate_deposit_amount(10);
    assert!(amount > 0);
    assert!(amount <= 10_000_000); // Max 0.01 SOL

    let should_top_up = calculator.should_top_up(1_000, 50);
    assert!(should_top_up);

    let top_up = calculator.calculate_top_up_amount(5_000_000, 10);
    assert!(top_up >= 0);
}

#[tokio::test]
async fn test_vault_monitor_alerts() {
    let db_pool = setup_test_database().await;
    let monitor = VaultMonitor::new(db_pool.clone());

    // Create test session with low balance
    // ...

    let alerts = monitor.monitor_all_vaults().await.unwrap();
    // Verify appropriate alerts generated

    cleanup_test_database(db_pool).await;
}

#[tokio::test]
async fn test_cleanup_expired_sessions() {
    let db_pool = setup_test_database().await;
```

```
        let encryption_key = [0u8; 32];
        let session_manager = SessionManager::new(db_pool.clone(), encryption_key);

        // Create expired session
        let parent_wallet = Keypair::new().pubkey();
        let session = session_manager
            .create_session(parent_wallet, -1, 1_000_000) // Already expired
            .await
            .unwrap();

        let expired = session_manager.cleanup_expired_sessions().await.unwrap();
        assert!(expired.contains(&session.session_id));

        cleanup_test_database(db_pool).await;
    }

    #[tokio::test]
    async fn test_keypair_encryption_decryption() {
        let encryption_key = [0u8; 32];
        let session_manager = SessionManager::new(
            setup_test_database().await,
            encryption_key,
        );

        let original = Keypair::new();
        let encrypted = session_manager.encrypt_keypair(&original).unwrap();
        let decrypted = session_manager.decrypt_keypair(&encrypted).unwrap();

        assert_eq!(original.pubkey(), decrypted.pubkey());
    }

    // Helper functions
    async fn setup_test_database() -> PgPool {
        // Create test database connection
        unimplemented!()
    }

    async fn cleanup_test_database(pool: PgPool) {
        // Clean up test data
        pool.close().await;
    }
}

// ==================== INTEGRATION TESTS ====================

#[cfg(test)]
mod integration_tests {
    use super::*;
```

```
    #[tokio::test]
    async fn test_full_session_lifecycle() {
        // 1. Create session
        // 2. Approve delegation
        // 3. Auto-deposit
        // 4. Execute trades
        // 5. Revoke
        // Verify funds returned correctly
    }

    #[tokio::test]
    async fn test_concurrent_sessions() {
        // Create multiple sessions for same user
        // Verify isolation
    }

    #[tokio::test]
    async fn test_session_expiry_auto_cleanup() {
        // Create session
        // Wait for expiry
        // Verify automatic cleanup
    }

    #[tokio::test]
    async fn test_emergency_revocation() {
        // Test immediate revocation works
        // Verify pending transactions handled
    }

    #[tokio::test]
    async fn test_rate_limiting() {
        // Test rate limits enforced
    }
}

// ==================== SECURITY TESTS ====================

#[cfg(test)]
mod security_tests {
    use super::*;

    #[tokio::test]
    async fn test_unauthorized_trade_execution() {
        // Verify non-delegate cannot execute trades
    }

    #[tokio::test]
```

```
    async fn test_parent_wallet_verification() {
        // Verify only parent can revoke
    }

    #[tokio::test]
    async fn test_session_hijacking_prevention() {
        // Test session token security
    }

    #[tokio::test]
    async fn test_spending_limit_enforcement() {
        // Test approved amount limits enforced
    }

    #[tokio::test]
    async fn test_expired_session_rejection() {
        // Verify expired sessions cannot trade
    }

    #[tokio::test]
    async fn test_double_spend_prevention() {
        // Test vault balance tracking prevents double-spend
    }

    #[tokio::test]
    async fn test_keypair_storage_security() {
        // Verify encrypted storage
        // Test key cannot be extracted
    }
}

// ==================== PERFORMANCE TESTS ====================

#[cfg(test)]
mod performance_tests {
    use super::*;

    #[tokio::test]
    async fn test_session_creation_performance() {
        // Verify < 500ms
        let start = std::time::Instant::now();
        // Create session
        let duration = start.elapsed();
        assert!(duration.as_millis() < 500);
    }

    #[tokio::test]
    async fn test_transaction_signing_performance() {
```

```
        // Verify < 50ms
    }

    #[tokio::test]
    async fn test_concurrent_session_handling() {
        // Test 1000+ concurrent sessions
    }

    #[tokio::test]
    async fn test_database_query_performance() {
        // Verify queries optimized
    }
}

// ==================== EDGE CASE TESTS ====================

#[cfg(test)]
mod edge_case_tests {
    use super::*;

    #[tokio::test]
    async fn test_zero_balance_cleanup() {
        // Test cleanup with no remaining funds
    }

    #[tokio::test]
    async fn test_maximum_session_duration() {
        // Test 24 hour max
    }

    #[tokio::test]
    async fn test_minimum_deposit() {
        // Test very small deposits
    }

    #[tokio::test]
    async fn test_rapid_revoke_after_create() {
        // Test immediate revocation
    }

    #[tokio::test]
    async fn test_multiple_cleanup_attempts() {
        // Test cleanup idempotency
    }
}

// Test utilities
mod test_utils {
```

```
    use super::*;

    pub fn generate_test_keypair() -> Keypair {
        Keypair::new()
    }

    pub async fn create_funded_account(
        banks_client: &mut BanksClient,
        payer: &Keypair,
        lamports: u64,
    ) -> Keypair {
        let account = Keypair::new();
        // Fund account
        account
    }

    pub async fn wait_for_confirmation(
        banks_client: &mut BanksClient,
        signature: &str,
    ) {
        // Wait for transaction confirmation
    }
}
```