Security Test Cases for Solana Wagering Smart Contract

Overview

This document contains comprehensive test cases to validate the security fixes and identify vulnerabilities in the wagering smart contract system.

Critical Vulnerability Test Cases

1. Authorization Bypass Attack

Test Case: Unauthorized Fund Distribution

```
describe("Authorization Bypass Attack", () => {
  it("Should prevent unauthorized fund distribution", async () => {
    // Setup: Create a game session with legitimate game server
    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // Create game session with legitimate server
    await program.methods
      .createGameSession(sessionId, betAmount, { winnerTakesAllOneVsOne: {} })
      .accounts({ gameServer: legitimateGameServer.publicKey })
      .signers([legitimateGameServer])
      .rpc();

    // Join players
    await joinPlayers(sessionId, [user1, user2]);

    // Attack: Try to distribute winnings with malicious server
    const maliciousServer = Keypair.generate();

    try {
      await program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: maliciousServer.publicKey })
        .remainingAccounts([...winnerAccounts])
        .signers([maliciousServer])
        .rpc();

      assert.fail("Should have failed with unauthorized access");
    } catch (error) {
      assert.include(error.toString(), "UnauthorizedDistribution");
    }
  });
});
```

2. Integer Overflow Attack

Test Case: Arithmetic Overflow in Payouts

```
describe("Integer Overflow Attack", () => {
  it("Should prevent arithmetic overflow in payouts", async () => {
    const sessionId = generateSessionId();
    const maxBet = new BN("18446744073709551615"); // Max u64

    // Create game session with maximum bet
    await program.methods
      .createGameSession(sessionId, maxBet, { payToSpawnOneVsOne: {} })
      .accounts({ gameServer: gameServer.publicKey })
      .signers([gameServer])
      .rpc();

    // Join players and simulate high kill counts
    await joinPlayers(sessionId, [user1, user2]);

    // Simulate kills that would cause overflow
    for (let i = 0; i < 1000; i++) {
      await program.methods
        .recordKill(sessionId, 0, user1.publicKey, 1, user2.publicKey)
        .accounts({ gameServer: gameServer.publicKey })
        .signers([gameServer])
        .rpc();
    }

    // Attempt distribution - should handle overflow gracefully
    try {
      await program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: gameServer.publicKey })
        .remainingAccounts([...playerAccounts])
        .signers([gameServer])
        .rpc();
    } catch (error) {
      // Should either succeed with capped values or fail gracefully
      assert.include(error.toString(), "ArithmeticOverflow");
    }
  });
});
```

3. Race Condition Attack

Test Case: Concurrent Team Joining

```
describe("Race Condition Attack", () => {
  it("Should prevent duplicate slot assignment", async () => {
    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // Create 1v1 game session
    await program.methods
      .createGameSession(sessionId, betAmount, { winnerTakesAllOneVsOne: {} })
      .accounts({ gameServer: gameServer.publicKey })
      .signers([gameServer])
      .rpc();

    // Simulate concurrent joining to same team
    const joinPromises = [
      program.methods
        .joinUser(sessionId, 0)
        .accounts({
          user: user1.publicKey,
          gameServer: gameServer.publicKey,
          userTokenAccount: user1TokenAccount,
        })
        .signers([user1])
        .rpc(),

      program.methods
        .joinUser(sessionId, 0)
        .accounts({
          user: user2.publicKey,
          gameServer: gameServer.publicKey,
          userTokenAccount: user2TokenAccount,
        })
        .signers([user2])
        .rpc()
    ];

    const results = await Promise.allSettled(joinPromises);

    // Only one should succeed
    const successCount = results.filter(r => r.status === 'fulfilled').length;
    assert.equal(successCount, 1, "Only one player should be able to join team 0");
  });
});
```

High Severity Test Cases

4. Reentrancy Attack

Test Case: Reentrancy During Token Transfer

```
describe("Reentrancy Attack", () => {
  it("Should prevent reentrancy during token transfers", async () => {
    // This test would require a malicious token program
    // that calls back into the wagering program during transfer

    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // Setup game session
    await createGameSession(sessionId, betAmount);
    await joinPlayers(sessionId, [user1, user2]);

    // Attempt to trigger reentrancy during payout
    // (This would require a custom malicious token program)
    try {
      await program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: gameServer.publicKey })
        .remainingAccounts([...winnerAccounts])
        .signers([gameServer])
        .rpc();

      // Verify state consistency after potential reentrancy
      const gameSession = await program.account.gameSession.fetch(gameSessionPda);
      assert.equal(gameSession.status, GameStatus.Completed);
    } catch (error) {
      // Should fail gracefully if reentrancy is detected
      assert.include(error.toString(), "AlreadyProcessing");
    }
  });
});
```

5. Input Validation Attack

Test Case: Malicious Input Handling

```
describe("Input Validation Attack", () => {
  it("Should reject malicious session IDs", async () => {
    const maliciousSessionId = "A".repeat(1000); // Very long session ID

    try {
      await program.methods
        .createGameSession(maliciousSessionId, new BN(100000000), { winnerTakesAllOneVsOne: {} })
        .accounts({ gameServer: gameServer.publicKey })
        .signers([gameServer])
        .rpc();

      assert.fail("Should have rejected malicious session ID");
    } catch (error) {
      assert.include(error.toString(), "InvalidSessionId");
    }
  });

  it("Should reject invalid team numbers", async () => {
    const sessionId = generateSessionId();
    await createGameSession(sessionId, new BN(100000000));

    try {
      await program.methods
        .joinUser(sessionId, 99) // Invalid team number
        .accounts({
          user: user1.publicKey,
          gameServer: gameServer.publicKey,
          userTokenAccount: user1TokenAccount,
        })
        .signers([user1])
        .rpc();

      assert.fail("Should have rejected invalid team number");
    } catch (error) {
      assert.include(error.toString(), "InvalidTeamSelection");
    }
  });
});
```

Medium Severity Test Cases

6. DoS Attack via Large Remaining Accounts

Test Case: Compute Limit Exhaustion

```
describe("DoS Attack via Large Remaining Accounts", () => {
  it("Should limit number of remaining accounts", async () => {
    const sessionId = generateSessionId();
    await createGameSession(sessionId, new BN(100000000));
    await joinPlayers(sessionId, [user1, user2]);

    // Create excessive number of accounts
    const excessiveAccounts = Array(1000).fill(null).map(() => ({
      pubkey: Keypair.generate().publicKey,
      isSigner: false,
      isWritable: true,
    }));

    try {
      await program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: gameServer.publicKey })
        .remainingAccounts(excessiveAccounts)
        .signers([gameServer])
        .rpc();

      assert.fail("Should have rejected excessive accounts");
    } catch (error) {
      // Should fail due to compute limit or account validation
      assert.include(error.toString(), "InvalidRemainingAccounts");
    }
  });
});
```

7. Fake Kill Recording

Test Case: Game Integrity Violation

```
describe("Fake Kill Recording", () => {
  it("Should validate kill authenticity", async () => {
    const sessionId = generateSessionId();
    await createGameSession(sessionId, new BN(100000000));
    await joinPlayers(sessionId, [user1, user2]);

    // Record fake kills (player killing themselves)
    try {
      await program.methods
        .recordKill(sessionId, 0, user1.publicKey, 0, user1.publicKey)
        .accounts({ gameServer: gameServer.publicKey })
        .signers([gameServer])
        .rpc();

      // Should either reject or have additional validation
      const gameSession = await program.account.gameSession.fetch(gameSessionPda);
      // Verify kill was not recorded or was marked as invalid
    } catch (error) {
      assert.include(error.toString(), "InvalidKill");
    }
  });
});
```

Edge Case Test Cases

8. Boundary Value Testing

Test Case: Zero and Maximum Values

```
describe("Boundary Value Testing", () => {
  it("Should handle zero bet amounts", async () => {
    const sessionId = generateSessionId();

    try {
      await program.methods
        .createGameSession(sessionId, new BN(0), { winnerTakesAllOneVsOne: {} })
        .accounts({ gameServer: gameServer.publicKey })
        .signers([gameServer])
        .rpc();

      assert.fail("Should have rejected zero bet amount");
    } catch (error) {
      assert.include(error.toString(), "InvalidBetAmount");
    }
  });

  it("Should handle maximum team capacity", async () => {
    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // Create 5v5 game
    await program.methods
      .createGameSession(sessionId, betAmount, { winnerTakesAllFiveVsFive: {} })
      .accounts({ gameServer: gameServer.publicKey })
      .signers([gameServer])
      .rpc();

    // Try to join 6th player to team
    const players = Array(6).fill(null).map(() => Keypair.generate());
    await joinPlayers(sessionId, players.slice(0, 5)); // Join 5 players

    try {
      await program.methods
        .joinUser(sessionId, 0)
        .accounts({
          user: players[5].publicKey,
          gameServer: gameServer.publicKey,
          userTokenAccount: await getTokenAccount(players[5].publicKey),
        })
        .signers([players[5]])
        .rpc();

      assert.fail("Should have rejected 6th player");
    } catch (error) {
      assert.include(error.toString(), "TeamIsFull");
    }
  });
});
```

9. State Transition Testing

Test Case: Invalid State Transitions

```
describe("State Transition Testing", () => {
  it("Should prevent invalid state transitions", async () => {
    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // Create game session (WaitingForPlayers)
    await createGameSession(sessionId, betAmount);

    // Try to distribute winnings before game starts
    try {
      await program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: gameServer.publicKey })
        .remainingAccounts([...winnerAccounts])
        .signers([gameServer])
        .rpc();

      assert.fail("Should have rejected distribution in WaitingForPlayers state");
    } catch (error) {
      assert.include(error.toString(), "InvalidGameState");
    }
  });
});
```

Integration Test Cases

10. End-to-End Security Testing

Test Case: Complete Attack Scenario

```
describe("Complete Attack Scenario", () => {
  it("Should prevent coordinated attack", async () => {
    // Simulate a coordinated attack combining multiple vulnerabilities
    const sessionId = generateSessionId();
    const betAmount = new BN(100000000);

    // 1. Create game session
    await createGameSession(sessionId, betAmount);

    // 2. Join players
    await joinPlayers(sessionId, [user1, user2]);

    // 3. Attempt to exploit multiple vulnerabilities simultaneously
    const attackPromises = [
      // Unauthorized distribution attempt
      program.methods
        .distributeWinnings(sessionId, 0)
        .accounts({ gameServer: Keypair.generate().publicKey })
        .remainingAccounts([...winnerAccounts])
        .signers([Keypair.generate()])
        .rpc(),

      // Race condition attempt
      program.methods
        .joinUser(sessionId, 0)
        .accounts({
          user: Keypair.generate().publicKey,
          gameServer: gameServer.publicKey,
          userTokenAccount: await getTokenAccount(Keypair.generate().publicKey),
        })
        .signers([Keypair.generate()])
        .rpc(),

      // Invalid input attempt
      program.methods
        .recordKill("invalid_session", 0, user1.publicKey, 1, user2.publicKey)
        .accounts({ gameServer: gameServer.publicKey })
        .signers([gameServer])
        .rpc()
    ];

    const results = await Promise.allSettled(attackPromises);

    // All attacks should fail
    results.forEach((result, index) => {
      if (result.status === 'fulfilled') {
        assert.fail(`Attack ${index} should have failed`);
      }
    });
  });
});
```

Test Execution Instructions

Prerequisites

1. Set up test environment with Anchor framework
2. Deploy the smart contract to testnet
3. Create test keypairs and token accounts
4. Fund test accounts with SOL and SPL tokens

Running Tests

```
# Run all security tests
anchor test --skip-local-validator

# Run specific test suite
anchor test --skip-local-validator --grep "Authorization Bypass"

# Run with verbose output
anchor test --skip-local-validator --verbose
```

Expected Results

- All critical vulnerability tests should fail (demonstrating vulnerabilities)
- After fixes, all tests should pass
- Any test that passes before fixes indicates a security vulnerability

Continuous Security Testing

Automated Security Checks

1. Run security tests in CI/CD pipeline
2. Implement fuzzing for input validation
3. Add static analysis tools (cargo-audit, clippy)
4. Monitor for new vulnerabilities in dependencies

Monitoring and Alerting

1. Set up alerts for failed security tests
2. Monitor transaction logs for suspicious patterns
3. Implement rate limiting for critical functions
4. Add circuit breakers for anomalous behavior

Note: These test cases should be run both before and after implementing security fixes to validate that vulnerabilities are properly addressed.