QuillAudits

AUDIT REPORT

November 2025

For

SCOR™

Table of Content

Executive Summary 04

Number of Security Issues per Severity 06

Summary of Issues 07

Checked Vulnerabilities 08

Techniques and Methods 10

Types of Severity 12

Types of Issues 13

Severity Matrix 14

High Severity Issues 15

1. Missing Wallet Addresses in Cancel Signature Verification 15

Medium Severity Issues 17

2. Missing Replay Protection and Expiration in Release Signature 17

3. Missing GameId Zero Validation Causes Permanent Fund Lock 18

4. getReservationDetails returns details only for expired reservations (logic inverted) 21

Low Severity Issues 22

5. BattleWallet is incompatible with fee-on-transfer (tax) ERC-20 tokens 22

6. _validateWallet can be spoofed — any contract can pretend to be a BattleWallet by returning the factory address from staticcall("factory()") 23

7. Decreasing Reservation TTL Delays Release of Newer Expired Reservations 24

Functional Tests 25

Automated Tests 25

Threat Model 26

Closing Summary & Disclaimer 39

Executive Summary

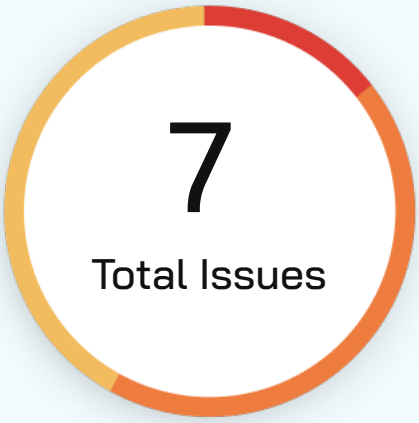
Project Name	SCOR
Project Type	Wallet
Project URL	https://scor.io
Overview	SCOR Battle Wallet is a wagering protocol that lets two players stake ETH or ERC-20 tokens against each other in on-chain "battles." Each player controls a BattleWallet, a lightweight upgradeable proxy deployed deterministically by the BattleWalletFactory. Players deposit funds into their wallets, and the factory coordinates game sessions (called Reservations) between pairs of wallets. Reservations lock funds until they are either settled—when a winner and loser are declared—or expire after a set TTL. The factory verifies off-chain EIP-712 signatures from an approver to relay reserve, settle, cancel, and releaseExpired actions, ensuring both sides receive identical instructions without directly interacting. Upon settlement, the loser's wallet pays the winner and a small fee to a designated fee wallet. Players can withdraw any unreserved balance anytime, and expired reservations automatically unlock. The architecture separates custody (wallets) from coordination (factory) while keeping all funds fully non-custodial.
Audit Scope	The scope of this Audit was to analyze the Sweet Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/sweet-io-org/evm-battle-wallet-public
Contracts in Scope	BattleWallet.sol , BattleWalletFactory.sol, BattleWalletProxy.sol
Branch	Master
Commit Hash	293843c67be5077a1cdaaa143fa4d0949250d490
Language	Solidity
Blockchain	Base Chain

Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	7th November 2025
Updated Code Received	10th November 2025
Review 2	18th November 2025
Fixed In	https://github.com/sweet-io-org/evm-battle-wallet-public/commit/221128b1f3ed9567b20caff2cf9d50c27fe87a4a

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0%)
High	1 (14%)
Medium	3 (43%)
Low	3 (43%)
Informational	0 (0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**1**	**3**	**3**	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Missing Wallet Addresses in Cancel Signature Verification	High	Resolved
2	Missing Replay Protection and Expiration in Release Signature	Medium	Resolved
3	Missing GameId Zero Validation Causes Permanent Fund Lock	Medium	Resolved
4	getReservationDetails returns details only for expired reservations (logic inverted)	Medium	Resolved
5	BattleWallet is incompatible with fee-on-transfer (tax) ERC-20 tokens	Low	Resolved
6	validateWallet can be spoofed — any contract can pretend to be a BattleWallet by returning the factory address from staticcall("factory()")	Low	Resolved
7	Decreasing Reservation TTL Delays Release of Newer Expired Reservations	Low	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Missing Wallet Addresses in Cancel Signature Verification(unconfirm)

Resolved

Path

BattleWalletFactory.sol

Function Name

`relayCancel()`

Description

The relayCancel function in BattleWalletFactory.sol fails to include wallet addresses in its signature verification process. This critical oversight allows attackers to replay cancel signatures across different wallet pairs that share the same gameId, potentially canceling legitimate reservations without authorization.

```
function relayCancel(
    address walletOne,
    address walletTwo,
    uint64 gameId,
    bytes calldata cancelSignature
) external {
    _verifyCancelSignature(gameId, cancelSignature);
    BattleWallet firstWallet = _validateWallet(walletOne);
    BattleWallet secondWallet = _validateWallet(walletTwo);

    firstWallet.cancel(gameId);
    secondWallet.cancel(gameId);
}

```

```
function _verifyCancelSignature(uint64 gameId, bytes calldata signature) private view {
    bytes32 structHash = keccak256(abi.encode(CANCEL_TYPEHASH, gameId, address(this)));
    bytes32 digest = _hashTypedDataV4(structHash);
    address signer = ECDSA.recover(digest, signature);
    if (signer != approver) revert InvalidSignature();
}

```

The signature verification mechanism is incomplete. When the approver signs a cancellation request, the signature should cryptographically bind the cancellation to specific wallet addresses. However, the current implementation only includes the gameId and factory address in the signature payload, omitting the critical walletOne and walletTwo parameters that are passed to the function.

In the BattleWallet system, multiple wallet pairs can independently create reservations with the same gameId. The gameId is not globally unique across all wallets - it's simply an identifier for a specific game between two players. This design means:
1. WalletA and WalletB might have a reservation with gameId = 100
2. WalletC and WalletD might also have a completely separate reservation with gameId = 100
3. These are two distinct games that happen to share the same ID value

Since the cancel signature only verifies the gameId (not which wallets are involved), a signature intended for one pair can be maliciously reused on any other pair with the same gameId.

Impact

High. Legitimate games can be disrupted at any time

Likelihood

High. Very feasible and zero cost

POC

Step 1: Legitimate Cancellation

- Player using WalletA and WalletB requests cancellation of gameId=100
- Approver signs: Sign(gameId=100, factory=0x...)
- Transaction: relayCancel(WalletA, WalletB, 100, signature)
- Result: Game cancelled successfully

Step 2: Malicious Replay
- Attacker observes the cancellation signature from Step 1
- Attacker identifies WalletC and WalletD also have gameId=100
- Attacker calls: relayCancel(WalletC, WalletD, 100, signature)
- Signature verification passes (only checks gameId=100)
- Result: WalletC and WalletD's game is cancelled WITHOUT authorization

Recommendation

Include Wallet Addresses in cancel Signature

Medium Severity Issues

Missing Replay Protection and Expiration in Release Signature

Resolved

Path

BattleWalletFactory.sol

Function Name

`relayReleaseExpired(), _verifyReleaseSignature()`

Description

The relayReleaseExpired function in BattleWalletFactory.sol lacks critical security measures to prevent signature replay attacks and does not include expiration timestamps. This allows malicious actors to reuse valid release signatures initially desired by the owner indefinitely.

The release expired mechanism is designed to allow authorized cleanup of expired reservations from wallet contracts. However, the signature verification only validates that the signature was created by the approver for a specific wallet address - it does not prevent the signature from being reused multiple times or restrict when the signature can be used. This means Attackers can use stale signature no longer desired by the wallet owner to force cleanup even when not desired

Impact

Attackers can use stale signature no longer desired by the wallet owner to force cleanup even when not desired

Likelihood

High

Recommendation

Add a nonce to prevents replay attacks and an expiration to limit the signature validity window

Missing GameId Zero Validation Causes Permanent Fund Lock

Resolved

Path

BattleWalletFactory.sol

Function Name

`_reserve()`

Description

The _reserve function in BattleWallet.sol does not validate that request.gameId is non-zero before creating a reservation. Sincegameld = 0 is used internally as a sentinel value to mark the end of the linked list structure, reservations created with gameld = 0 can never be properly released or cleaned up, resulting in peased or cleaned up, resulting in permanent lock of reserved funds.

```
function _reserve(ReserveRequest calldata request, uint64 expiration) internal {
    if (request.amount == 0) revert InvalidAmount();
    if (request.factory != factory) revert InvalidFactory();
    if (request.feeBasisPoints > 2500) revert InvalidFeeBasisPoints();
    if (request.feeBasisPoints > 0 && request.feeWallet == address(0)) revert ZeroAddress();
    //  MISSING: if (request.gameId == 0) revert InvalidGameId();

    (address opponent, bool isPlayerOne) = _identifyOpponent(request.player1, request.player2);
    uint64 providedNonce = isPlayerOne ? request.noncePlayer1 : request.noncePlayer2;
    if (providedNonce != nextNonce) revert InvalidNonce();

    if (!(expiration > block.timestamp)) revert ExpiredInPast();

    _releaseExpiredInternal();
    if (reservations[request.gameId].exists) revert GameExists();

    // ... reservation creation continues ...

    reservations[request.gameId] = Reservation({
        amount: request.amount,
        opponent: opponent,
        isToken: request.isToken,
        feeWallet: request.feeWallet,
        feeBasisPoints: request.feeBasisPoints,
        exists: true,
        active: true,
        expiration: expiration,
        nextGameId: 0  //  This is normal for the last element
    });

    if (firstGameId == 0) {
        firstGameId = request.gameId;  //  If gameId=0, firstGameId becomes 0
        lastGameId = request.gameId;   //  If gameId=0, lastGameId becomes 0
    } else {
        Reservation storage lastRes = reservations[lastGameId];
        lastRes.nextGameId = request.gameId;  //  If gameId=0, points to sentinel
        lastGameId = request.gameId;
    }
}
```

Critical Problem in _releaseExpiredInternal():

```
function _releaseExpiredInternal() private {
    uint64 currGameId = firstGameId;
    if (currGameId == 0) {
        return;  //  If firstGameId is 0, function exits immediately!
    }

    uint256 currentTime = block.timestamp;
    uint256 totalEth = totalReservedEth;
    uint256 totalToken = totalReservedToken;

    uint64 nextGameId;
    while (currGameId != 0) {  // Loop condition: stop when currGameId == 0
        Reservation storage res = reservations[currGameId];
        if (currentTime < res.expiration) {
            break;
        }

        if (res.active) {
            if (res.isToken) {
                if (res.amount > totalToken) revert AmountGreaterThanReserved();
                totalToken -= res.amount;
            } else {
                if (res.amount > totalEth) revert AmountGreaterThanReserved();
                totalEth -= res.amount;
            }
        }

        nextGameId = res.nextGameId;
        if (nextGameId == 0) {
            lastGameId = 0;  //  End of list marker
        }

        delete reservations[currGameId];
        currGameId = nextGameId;
    }
    // ... state updates ...
}
```

Let's consider these attack scenarios:

Legitimate User Flow:
1. User A and User B create a game with gameId=0 (by mistake or poor frontend)
2. Both wallets reserve funds: 100 ETH each
3. totalReservedEth increases by 100 for each wallet
4. Game expires after 1 hour
5. releaseExpired() is called
6. _releaseExpiredInternal() exits immediately (firstGameId == 0)
7. Reservation never deleted, funds never released
8. 100 ETH permanently locked in each wallet

Financial Impact:
- 200 ETH total locked across both wallets
- No way to recover without contract upgrade
- Users cannot withdraw their own funds

Scenario 2: Malicious Fund Locking
Attacker Strategy:
1. Attacker creates two wallets (or uses one wallet + victim)
2. Initiates reservation with gameId=0
3. Attacker wallet locks 0.01 ETH (minimal amount)
4. Victim wallet has 1000 ETH, locks 0.01 ETH
5. Reservation expires
6. Victim's 0.01 ETH is permanently locked

Impact

Funds lock

Likelihood

Medium to High depending upon the fact if gameId = 0 is allowed or not

Recommendation

Consider adding this validation

getReservationDetails returns details only for expired reservations (logic inverted)

Resolved

Description

BattleWallet.getReservationDetails(uint64 gameId) is intended to return the details of an active reservation (amount, opponent, expiration, token flag) when the reservation exists and has not yet expired. The current implementation contains an inverted time check which causes the function to return details only when the reservation is expired.

```
1  function getReservationDetails(uint64 gameId)
2        external
3        view
4        returns (uint256 amount, address opponent, uint64 expire, bool isToken, bool found)
5    {
6        Reservation storage res = reservations[gameId];
7        if (!res.exists || !res.active) {
8            return (0, address(0), 0, false, false);
9        }
10       if (!(res.expiration < block.timestamp)) {
11           return (0, address(0), 0, false, false);
12       }
13       return (res.amount, res.opponent, res.expiration, res.isToken, true);
14   }
```

The if (!(res.expiration < block.timestamp)) condition is true when res.expiration >= block.timestamp — i.e. not expired — causing the function to return (0,0,0,false,false) for valid, active reservations. Only when res.expiration < block.timestamp (the reservation is expired) does the function fall through to return the stored details.

This is almost certainly the opposite of the intended behavior: callers expect an active reservation's metadata while it's valid, not only after it has expired.

Impact

Incorrect UI / UX: Wallet front-ends, explorers, dApps, and off-chain services that call getReservationDetails will receive found == false for active reservations, and may show users that no reservation exists when one does. This leads to confusing or wrong displays (e.g., showing funds available when they are reserved).

Integration bugs: Other contracts or off-chain services that rely on this view for logic (e.g., to decide whether to attempt a settlement, to present a confirmation step, or to compute available balance) may behave incorrectly.

Likelihood

High — The function is public and will almost certainly be used by UIs and integrators. The inverted logic is deterministic and present in deployed code; anyone calling the function will observe the wrong behavior. Because it is a view-only problem, it is easy for integrators to hit in production and is likely to be noticed via out-of-sync UI behavior.

Recommendation

Replace the inverted time check so the function only returns details for active and unexpired reservations.

Low Severity Issues

BattleWallet is incompatible with fee-on-transfer (tax) ERC-20 tokens

Resolved

Path

BattleWallet.sol

Description

The BattleWallet contract assumes standard ERC-20 behavior: transfers do not burn or deduct fees and token.balanceOf(address(this)) reflects actual tokens available. Several code paths rely on that assumption:

_reserve() checks available tokens using token.balanceOf(address(this)) and compares against totalReservedToken before incrementing totalReservedToken.

settleForLoser() and _distributeTokenWinnings() perform token.safeTransfer(winner, payout) and token.safeTransfer(feeWallet, fee) expecting recipients to receive the exact amounts sent.

withdrawToken() uses token.balanceOf(address(this)) - totalReservedToken to compute available tokens.

Fee-on-transfer (tax) tokens — e.g., reflective/tax tokens that deduct a percentage on each transfer — break these assumptions. When such tokens are used:

Transfers out of the wallet (payouts) are further reduced by the token's fee on transfer, so winners and feeWallets receive less than expected.

Balance-based invariant checks (e.g., tokenBalance < totalReservedToken) can fail unexpectedly if fees reduce contract balances, causing InsufficientFunds reverts and stuck flows.

The contract cannot detect or compensate for the portion taken as fee and therefore can underpay winners or leave reservations unresolved.

Because BattleWallet mixes direct balance checks and raw token transfers, fee-on-transfer tokens will lead to incorrect accounting, settlement failures, and potentially stuck funds or unfair payouts.

Impact

Low

Likelihood

Low

Recommendation

Consider caching the balanceOf() before and after and save the balance

_validateWallet can be spoofed — any contract can pretend to be a BattleWallet by returning the factory address from staticcall("factory()")

Resolved

Path
BattleWalletFactory.sol

Function Name
`_validateWallet()`

Description
BattleWalletFactory._validateWallet(address wallet) attempts to verify that a given address is a BattleWallet controlled by this factory by doing a staticcall

This check is insufficient. Any arbitrary contract can implement a factory() view function that returns the factory's address (i.e., return address(0x...factory...);). If such a contract is supplied as wallet to relay functions (relayReserve, relaySettle,relayCancel, relayReleaseExpired), _validateWallet will accept it as a valid BattleWallet and the factory will call reserve, settleForWinner, settleForLoser, cancel, or releaseExpired on that contract.

Because nothing else in _validateWallet verifies the actual runtime code or that the contract is a proper BattleWallet proxy, this lets an attacker or a mistaken caller cause the factory to call arbitrary contract code under the assumption it's a wallet. That can produce reverts, unexpected side-effects, gas consumption, or worse.

Impact
High

Likelihood
The weakness is trivial to exploit if an attacker or a misconfigured caller is able to pass arbitrary addresses into the factory's relay functions (which accept wallet addresses from callers).

Recommendation
Verify deployed proxy bytecode (extcodehash) against expected proxy runtime bytecode or maintain the mapping of the deployed wallets

Decreasing Reservation TTL Delays Release of Newer Expired Reservations

Resolved

Path

BattleWalletFactory.sol

Function Name

`setReservationTtl()`

Description

The setReservationTtl function in BattleWalletFactory.sol allows the admin to change the time-to-live (TTL)for new reservations. When the TTLis decreased, newer reservations may expire before older ones, but cannot be released until all preceding reservations in the linked list have expired first. This creates operational inefficiency and delayed fund availability.

Reservations in BattleWallet are stored in a linked list ordered by creation time (not expiration time). The _releaseExpiredInternal() function traverses this list sequentially and stops when it encounters the first non-expired reservation. If the admin decreases the TTL,newer reservations created with the shorter TTL will expire earlier than older reservations created with the longer TTL,but they cannot be released until all older reservations expire first.

Delayed Fund Availability:
- Funds remain locked beyond their intended expiration time
- Maximum delay equals the difference in TTLs

User Experience:
- Unexpected delays in receiving winnings or refunds
- Confusion about why expired games aren't released

Maximum Delay Calculation:

Max Delay = Old_TTL - New_TTL
Example: Old TTL = 7200s (2 hours), New TTL = 1800s (30 min)
Max Delay = 7200 - 1800 = 5400 seconds (90 minutes)

Impact

Low

Likelihood

Low

Recommendation

Option 1 : Document the behavior
Option 2 : Only allow ttl increase
Option 3 : Add a grace period warning (event emission)

Functional Tests

Some of the tests performed are mentioned below:

- **Successful reservation and fund locking between two wallets**
- **Settlement correctly transfers winnings and fee to fee wallet**
- **Factory relays revert if either wallet validation fails**
- **Reservation signature verification enforces approver authenticity**
- **Expired reservations are automatically released and funds unlocked**

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
BattleWallet.sol, BattleWalletFactory.sol	deployBattleWallet(address walletOwner)	Purpose: deterministically deploy a BattleWallet proxy for walletOwner and initialize it. **Inputs** • walletOwner — fully controlled by caller; must be an EOA. Constraints: should not be zero address; Create2 salt derived from owner prevents double deploys. **Impact** • Creates a wallet that will hold user funds and trust the factory. A broken deploy can allocate the wrong owner, register fake wallets, or leave implementation uninitialized. **Intended branches / coverage** • Successful deploy for new owner: proxy created; initialize(owner, token) called; BattleWalletDeployed emitted; registry updated. • predictBattleWalletAddress matches actual address (deterministic). • Revert on walletOwner == address(0) or on duplicate deploy (salt collision).

Contract	Function	Threats
		Test coverage • Deploy works and returns expected address. • initialize called and getOwnerAndFactory() on proxy returns correct values. • Duplicate deployment for same owner reverts. • Predicted address equals deployed address. **Negative tests** • Revert when walletOwner == address(0). • Ensure attacker can't front-run to take ownership — simulate pre-deploy? • Ensure implementation cannot be initialized independently to hijack proxies (implementation hygiene).
	relayReserve(ReserveRequest request, bytes reserveSignature, bytes playerOneApproval, bytes playerTwoApproval)	Purpose: Relay a factory-signed reservation to both participating wallets; applies uniform TTL and ensures both sides accept the same request.

Contract	Function	Threats
		Inputs • request (gameId, amount, player1, player2, isToken, nonces, feeWallet, feeBasisPoints, factory) — partly controlled off-chain by approver and players. • reserveSignature — approver-signed EIP-712 signature (factory domain). • playerOneApproval, playerTwoApproval — optional wallet-owner approvals (wallet-domain signatures) if requireApproval is enabled. **Impact** • Locks tokens/ETH in two wallets and increments totalReserved*. Mistakes here can freeze funds or lock incorrect amounts. **Intended branches / coverage** • Valid signature and wallet approvals: both walletOne.reserve(...) and walletTwo.reserve(...) succeed (atomic relay). • Missing player approvals is allowed if wallets do not require approval. • TTL applied uniformly; expiration computed in factory before relay. • Revert when reserveSignature invalid, or when _validateWallet fails for either wallet.

Contract	Function	Threats
		Test coverage • Happy path: approver signs valid request; both wallets accept; Reserved events emitted on both wallets; totals increased. • Case where a wallet requires approval: supply owner signatures and confirm acceptance. • Case where one wallet rejects (insufficient funds, bad nonce) — full transaction reverts and no side effects. • Ensure expiration is block.timestamp + reservationTtl and stored identically on both wallets. **Negative tests** • Invalid approver signature → revert. • Supply mismatched factory in request → revert. • One wallet has insufficient balance → revert and ensure no partial side effects. • Parallel submissions using same nonce: one succeeds, others revert. • Attempt with gameId == 0 (if you implement check) should revert. • Supply malicious fake wallet (see _validateWallet spoofing) — ensure reject after hardening.

Contract	Function	Threats
	reserve(ReserveRequest request, uint64 expiration, bytes walletApproval)	Purpose: Record a reservation in the wallet (linked-list append), verify owner-approval if required, and increment totalReserved*. **Inputs** • request (includes nonces) — controlled by factory/ approver flow; expiration computed by factory; walletApproval optional owner signature. **Impact** • Changes wallet accounting: increases totalReservedToken/ totalReservedEth, sets reservation mapping and linked list. Bugs can cause incorrect totals or corrupt list. **Intended branches / coverage** • requireApproval == true: verify _verifyReserveApprovalSignature. • Check request.amount > 0, feeBasisPoints bounds, nonce equals nextNonce, and expiration > block.timestamp. • Token reserve path: check tokenSet, token.balanceOf - totalReservedToken >= amount. • ETH reserve path: check address(this).balance - totalReservedEth >= amount. • Append to the end of linked list; increment nextNonce.

Contract	Function	Threats
		Test coverage • Happy path for ETH and ERC-20 reservations. • Nonce increments and prevents replay. • Linked-list first-element append and multi-element append. • totals reflect new reservation. **Negative tests** • Zero amount should revert (currently coded). • Invalid nonce → revert. Expiration <= now → revert. • Reserve when token not set but isToken==true → revert. • Token with insufficient balance → revert. • Duplicate gameId → revert GameExists. • feeBasisPoints > 2500 → revert.
	relaySettle(SettlementRequest request, bytes settlementSignature) — Factory & settleForLoser / settleForWinner	Purpose: Relay a factory/approver-signed settlement that marks a game settled and triggers payouts (winner receives payout and fee goes to fee wallet).

Contract	Function	Threats
		Inputs • request (gameId, winner, loser, factory) and settlementSignature (approver-signed). Factory validates signature then calls wallets: loser settleForLoser, winner settleForWinner. **Impact** • Moves funds (ETH or tokens) from loser wallet to winner and feeWallet; can cause over/under pay, reentrancy, or double-settlement bugs. **Intended branches / coverage** • _verifySettlementSignature verifies approver signed it. • settleForLoser must verify reservation exists active and not expired; perform distribution (_distribute*) and decrement totalReserved*. • settleForWinner sets active=false and decrements totals; event emitted. • Atomic behavior through factory: if any wallet call reverts, whole relay reverts.

Contract	Function	Threats
		Test coverage • Happy path: both wallets called and amounts move correctly; fees calculated correctly. • Losing wallet path with token payout and fee transfer to feeWallet. • Reject settlement when reservation not found / inactive / expired (for loser path). • Ensure totals adjusted correctly and events emitted. **Negative tests** • Attempt replay of same settlement twice → should revert (GameNotFound or inactive). • Attempt settle after reservation expired → • settleForLoser should revert with ReservationExpired. • Reentrancy attack: winner is contract with fallback — ensure nonReentrant added and changes order safe. • Settlement signature invalid → revert in factory.
	relayCancel(address walletOne, address walletTwo, uint64 gameId, bytes cancelSignature) — Factory & cancel(uint64 gameId)	Purpose: Cancel an active reservation on both wallets (e.g., mutual cancellation or approver-driven).

Contract	Function	Threats
		Inputs • walletOne, walletTwo, gameId, cancelSignature (approver signed). **Impact** • Mark reservation inactive and decrement totals. Improper cancel logic can leave tombstones or double-counting. **Intended branches / coverage** • _verifyCancelSignature confirms approver signature. • Each wallet must check reservation exists & active; decrement totalReserved* and set active=false. • Deletion/unlinking behavior (unlink or leave tombstone) — tests must assert contract behavior. **Test coverage** • Happy path: both wallets cancel; ReservationCancelled emitted; totals decreased. • If one wallet call reverts, entire relay reverts (atomic). • Confirm linked list state after cancel (if you implement unlink).

Contract	Function	Threats
		Negative tests • Cancel on nonexistent game → revert. Cancel on already inactive → • revert. Cancel leaving tombstones • and later gas exhaustion in cleanup — test for accumulation and ensure batch cleanup handles it. Purpose: Trigger cleanup of expired reservations in a wallet (release funds back to owner). **Inputs** • walletAddress, signature (approver-signed) — currently lacks nonce/expiry (must be tested/mitigated). **Impact** • Significant liveness/safety risk: improper cleanup logic + signature replayability can lead to stuck funds, griefing, or unexpected state transitions.

Contract	Function	Threats
		Intended branches / coverage • Factory verifies signature (and after patch: verifies nonce + expiry). • Wallet should iterate and delete expired reservations, adjust totals carefully; batch limit to avoid gas OOM. • For releaseExpiredBatch(maxIters), ensure partial progress is possible and idempotent. **Test coverage** • Happy path: expired head nodes cleaned and totals decreased. • Batch operation: large expired list cleaned across multiple batched calls. • Signature replay: after implementing nonce/expires, ensure repeated signature fails. **Negative tests** • Simulate TTL changes creating out-of-order expirations and ensure batch scan removes all expired nodes. • Ensure releaseExpired cannot be abused to remove active reservations. • Test replay of old signature (should revert after nonce/expiry fix).

Contract	Function	Threats
	relayCancel(address walletOne, address walletTwo, uint64 gameId, bytes cancelSignature) — Factory & cancel(uint64 gameId)	Purpose: Cancel an active reservation on both wallets (e.g., mutual cancellation or approver-driven). **Inputs** • amount — user controlled; contract checks availability via address(this).balance - totalReservedEth or token.balanceOf - totalReservedToken. **Impact** • Critical funds outflow function; incorrect checks can allow draining reserved funds or block legitimate withdrawals due to inflated totals. **Intended branches / coverage** • _releaseExpiredInternal() (or batch) called before • computing available funds. Check amount != 0, check available >= amount. • Transfer net amount and emit EthWithdrawn / TokensWithdrawn. • Reentrancy guard via nonReentrant.

Contract	Function	Threats
		Test coverage • Owner withdraws allowed amount when no reservations exist. • Withdraw fails when trying to withdraw more than available. • Withdraw cannot remove reserved funds (attempts revert). • After releaseExpired, previously reserved funds become withdrawable. **Negative tests** • Attempt withdraw of zero → revert (InvalidAmount). • Simulate token drain external to contract (token with transferFrom exploited) and ensure withdrawals revert due to tokenBalance < totalReservedToken. • Reentrancy attempt via malicious fallback — ensure nonReentrant prevents issues.

Closing Summary

In this report, we have considered the security of Sweet Smart Contract. We performed our audit according to the procedure described above.

No critical issues in Sweet Smart Contract, 2 High, 3 medium 2 Lows were found,the sweet team resolved all the issues mentioned above

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

November 2025

For

SCOR™

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com