ZKSECURITY

Audit of Renegade's Solidity Contracts and Malleable Matches

Date: May 7th, 2025

Introduction

On April 21st 2025, zkSecurity started a security audit of Renegade's updated circuits and contracts. The code was shared via public GitHub repositories, accompanied by written specifications for both the circuits and smart contracts. The audit lasted two weeks with one consultant.

This is zkSecurity's third audit collaboration with Renegade. The first audit reviewed circuits and Stylus contracts in general, while the second audit was focused on a new feature, atomic settlement. A detailed description of the system can be found in those two reports.

Scope

The Renegade team froze the following branches in the three affected repos for this audit:

1. renegade/zksecurity-audit-4-21-25
2. renegade-contracts/zksecurity-audit-4-21-25
3. renegade-solidity-contracts/zksecurity-audit-4-21-25

The audit, performed on the branches listed above, covers several updates to the Renegade platform:

- **Solidity contracts**. Stylus (Rust) contracts were rewritten in Solidity to support deployment to more chains. We reviewed the `renegade-solidity-contracts` repo, covering all methods of the `Darkpool` contract and its subcomponents.
- **Malleable matches**. A new darkpool capability was added: Performing an atomic match-settle trade, where the exact match size does not have to be decided at proof creation time, but only when calling the onchain contract. The circuit only specifies bounds on the trade size.
  We reviewed the new circuit and the new Stylus contract method, as well as the Solidity method covered by the first point.
- **Optimization: In-Circuit Full Wallet Commitments**. Previously, circuits only computed a commitment of private wallet shares, which was combined with public shares onchain to produce the full wallet commitment. To save gas consumed by onchain Poseidon-hashing, the computation of full commitments in some methods was moved down into the circuit. For other methods, an alternative that expects the full commitment in the proof statement was added next to the existing variant.
  We reviewed changes to the affected circuits and methods, covered by the following commit ranges in the respective repos: renegade, renegade-contracts. We also reviewed updated Merkle insertion methods.
- **Optimization: Poseidon gadget**. The Poseidon gadget was rewritten to make more efficient use of jellyfish's PLONK layout, collapsing the entire round function into a single row per state element.
  We reviewed the new Poseidon gadget in its entirety.

Summary

The code in all three repositories was found to be well-organized, well-tested, thoroughly commented, and of high quality. The Renegade team was highly responsive and provided valuable context throughout the process.

We did not find any issue in the new Poseidon gadget or with in-circuit full wallet commitments. In the new malleable-match circuit, a single soundness bug was identified, deep inside the low-level gadgets it depends on.

Although not explicitly part of the audit scope, we re-reviewed Renegade's "proof-linking" relation used in several contract interactions to bundle proofs from different parties. No further issue of note was found in this effort.

Most of our findings relate to the new Solidity contracts. Apart from several specific issues, we added a summary finding to document a few best practices that we think could benefit the codebase going forward. Given Renegade's goal of supporting tokens as permissionlessly as possible, we recommend maintaining a comprehensive database of known ERC20 token quirks and their potential security implications. As the protocol expands its token support, we advise implementing a continuous monitoring and testing process to ensure that these token-specific behaviors don't introduce new attack vectors or vulnerabilities into the protocol's core functionality. The following overview is a great starting point.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	zk_gadgets/comparators.rs	Underconstrained Comparison Can Be Exploited to Drain Funds	**High**
#01	Darkpool.sol	Admin Functions Are Inaccessible If the Darkpool Is Paused	**Medium**
#02	ExternalTransfers.sol	Fee-on-Transfer Tokens Cause Underfunded Deposits	**Medium**
#03	ExternalTransfers.sol	Unchecked ERC20 Transfer Return Values Can Cause Silent Token Transfer Failures	**Medium**
#04	WalletOperations.sol	Direct Usage of `ecrecover` Permits Malleable Signatures	**Low**
#05	Darkpool.sol	Lack of Two-Step Process for Ownership Transfer	**Informational**
#06	IWETH9.sol	WETH Interface Differs Between Chains	**Informational**
#07	renegade-solidity-contracts/src	Smart Contracts Are Not Following Some Solidity Coding Standards	**Informational**

ID	COMPONENT	NAME	RISK
#08	valid_malleable_match_settle_atomic.rs	Unnecessary No-op Constraints	**Informational**

#00 - Underconstrained Comparison Can Be Exploited to Drain Funds

Severity: High **Location:** zk_gadgets/comparators.rs

Description. We found a soundness issue in the low-level `greater_than_zero()` gadget, which leads to issues with many gadgets that directly or indirectly depend on it: `greater_than_eq()`, `less_than()`, `constrain_less_than()`, `div_rem()` and `floor()`. The unsoundness of `floor()` can be used to escape a critical check in the malleable-match circuit: that a user's balance is sufficient to cover the trade being performed. We describe below how this might be exploited in an attack that drains the darkpool's entire balance of any token.

This is the code of `greater_than_eq_zero()`, plus two of its subcomponents:

```
/// Evaluate the condition x >= 0; returns 1 if true, otherwise 0
pub fn greater_than_eq_zero(
    x: Variable,
    cs: &mut PlonkCircuit,
) -> Result<BoolVar, CircuitError> {
    // Decompose and reconstruct the value in the given bitlength, if we can do so
    // then the value is greater than zero
    let reconstructed = ToBitsGadget::<D>::decompose_and_reconstruct(x, cs)?;
    EqGadget::eq(&reconstructed, &x, cs)
}
```

```
/// Decompose and reconstruct a value to and from its bitwise representation
/// with a fixed bitlength
///
/// This is useful as a range check for a power of two, wherein the value
/// may only be represented in 2^D bits
pub fn decompose_and_reconstruct(
    a: Variable,
    cs: &mut PlonkCircuit,
) -> Result<Variable, CircuitError> {
    let bits = Self::to_bits_unconstrained(a, cs)?;
    Self::bit_reconstruct(&bits, cs)
}

/// [ZKSECURITY] ...

/// Converts a value to its bitwise representation without constraining the
/// value to be correct
fn to_bits_unconstrained(
    /// [ZKSECURITY] ...
```

In summary, the gadget witnesses some value `reconstructed` which is known to be in the target bit range $[0, 2^D)$. As the code comment correctly points out, if the input `x` equals `reconstructed`, then we can conclude $x \in [0, 2^D)$. So x must be greater or equal zero, using the common definition that *negative* numbers are those greater than half the field size.

The issue is that the converse is not true: Just because `x` does **not** equal `reconstructed`, we can't conclude that `x` is negative. A malicious prover could simply tweak `to_bits_unconstrained()` to witness some arbitrary bits unrelated to the input. This causes `greater_than_eq_zero()` to always return `false`. In other words, we can't rely on this gadget when it returns `false`.

The issue directly affects downstream gadgets:

- `greater_than_eq(a, b)` just calls `greater_than_eq_zero()` on the difference `a - b`, so it has the same issue
- `less_than()` logically negates the output of `greater_than_eq()`, so we can't rely on it when it returns `true`
- `constrain_less_than()` enforces the output of `less_than()` to be `true`, which is meaningless, and can be achieved by a malicious prover regardless of the input
- `div_rem(a, b)` performs Euclidian division to produce a quotient q and remainder r such that $a = qb + r$. However, it uses `constrain_less_than()` to check $r < b$. By escaping this check, a prover can make the quotient smaller than it should be by adding multiples of b to the remainder.

Finally, `floor(x)` is affected as follows: It takes as input a `FixedPointVar`, a fractional number that is represented as the field element $x2^M$, where M is the maximum length of the fractional part. It rounds that number down by doing an integer division by 2^M, so it returns $\lfloor \frac{x2^M}{2^M} \rfloor = \lfloor x \rfloor$. The division is done using `div_rem()`. By tweaking the quotient to be smaller, a malicious prover can change the `floor()` output to be any integer less than the input.

Impact. Of the gadgets identified as unsound, only `floor()` is directly used in protocol circuits. Furthermore, `floor()` is only used in the new malleable-match circuit. Circuits that were deployed prior to this audit are unaffected by this issue.

The impact to malleable-match proofs is severe and directly exploitable, as the following snippet shows:

```rust
// Compute the maximum amounts sent and received
let max_base = match_res.max_base_amount;
let max_quote_fp = match_res.price.mul_integer(max_base, cs)?;
let max_quote = FixedPointGadget::floor(max_quote_fp, cs)?;

// Select the appropriate amount based on the internal party's side
// order.side = 1 implies that the internal party sells the base asset
let max_send_recv = CondSelectVectorGadget::select(
    &[max_base, max_quote], &[max_quote, max_base], order.side, cs)?;
let max_send = max_send_recv[0];
let max_recv = max_send_recv[1];

Self::validate_match_capitalized(max_send, send_bal, cs)?;
```

Note that `floor()` is used to calculate `max_quote`. If the internal party takes the "buy" side of the trade and sells the quote asset, `max_quote` is assigned to `max_send`. The last line in the snippet checks that there is enough balance in the wallet, by constraining `send_bal.amount >= max_send`.

Since an attacker can make `max_send` smaller than its actual value, by tweaking `floor()`, they can make the `send_bal.amount >= max_send` check suceed, regardless of the actual match result. Effectively, the attacker can trade funds that they don't actually have in their wallet.

The underfunded trade is not prevented by any checks in the smart contract, since by design, user balances are not revealed onchain. Instead, the balance will be updated indirectly via an update to its public share, and will underflow and become a large number close to the field size when unblinded and combined with the private share.

The resulting wallet, which now has a huge quote balance, might no longer be usable for withdrawal, due to range checks in the update-wallet circuit. In particular, the base tokens received in the trade might not be accessible to the attacker. However, we believe an underfunded trade can still be exploited: note that we effectively print quote tokens to the external counter-party, and those are withdrawn immediately in the atomic swap. An attacker could take the role of the external party themselves. By setting an artificially low price for the base token, they can move an arbitrary amount of quote tokens out of the darkpool, in exchange for burning a tiny amount of base tokens.

The only quote token supported by the official relayer API is USDC. However, since running relayers is permissionless and the limitation to USDC is not enforced onchain, it appears to us that an attacker is free to choose which token balance to drain in a single swap.

Recommendation. To fix `greater_than_eq_zero()`, we can take inspiration from the classic `LessThan` gadget in circomlib. The following pseudo-code implements the gadget in a sound way:

```
/// input assumption: -2^D ≤ x < 2^D (necessary for completeness)
fn greater_than_eq_zero(D, x) {
  // constrained D+1 bits decomposition of 2^D + x
  let bits = to_bits(D + 1, (1 << D) + x);
  // x ≥ 0 ↔ 2^D + x ≥ 2^D ↔ Dth bit is set
  bits[D]
}
```

#01 - Admin Functions Are Inaccessible If the Darkpool Is Paused

Severity: Medium **Location:** Darkpool.sol

Description. The `Darkpool` contract can be paused, meaning that any user-accessible, state mutating methods (i.e., those that are not scoped to just the contract owner) are tagged with the `whenNotPaused` modifier and will, therefore, revert if the contract is paused. This is meant as an emergency measure, to give the contract owner time to upgrade any faulty or compromised implementations in the case of an attack or other unintended contract functionality. However, the current implementation also applies the `whenNotPaused` modifier to all administrative functions (with the exception of `pause` and `unpause`), preventing the contract owner from executing any state changes if the contract is paused.

Recommendation. Remove the `whenNotPaused` modifier from all administrative functions in *Darkpool.sol*.

#02 - Fee-on-Transfer Tokens Cause Underfunded Deposits

Severity: Medium **Location:** ExternalTransfers.sol

Description. The deposit flow in `executeSimpleDeposit()` assumes that invoking `transferFrom(owner, contract, AMOUNT)` always transfers exactly AMOUNT tokens. However, some non-standard “fee-on-transfer” tokens deduct a percentage on each transfer.

As a result, when depositing AMOUNT, the contract may only receive AMOUNT - FEE, but the darkpool wallet is still credited with the full AMOUNT. This mismatch leaves user balances in darkpool wallets insufficiently backed by actual funds in the contract.

The same issue applies to `executeDeposit()`, which indirectly invokes `safeTransferFrom(owner, contract, AMOUNT)` via Permit2’s `permitWitnessTransferFrom()`, without accounting for fee-on-transfer.

The following public methods in `Darkpool.sol` process external deposits and are susceptible:

- `updateWallet()`
- `processAtomicMatchSettle()`
- `processAtomicMatchSettleWithReceiver()`
- `processAtomicMatchSettleWithCommitments()`
- `processAtomicMatchSettleWithCommitmentsReceiver()`
- `processMalleableAtomicMatchSettle()`

Impact. At the time of writing, fee-on-transfer tokens seem not very prevalent and unlikely to be traded on Renegade. When traded frequently, the accounting mismatch introduced by these tokens on deposit could accumulate, and could leave users unable to withdraw their token balance. Effectively, some users could end up paying the transfer fees for all other market participants.

Worth mentioning, an extreme case of exploiting this issue was the attack against Balancer in 2020, where the attacker manipulated AMM prices by systematically exploiting the fee-on-transfer accounting mismatch. This exploit seems unlikely on Renegade: as an order-book exchange, it should be less vulnerable to price manipulation.

Recommendation. An easy mitigation would be to disallow fee-on-transfer deposits:

- Snapshot token balances on the `Darkpool` contract, before and after each deposit.
- Compute `received = after - before`, and if `received != transfer.amount`, revert.

#03 - Unchecked ERC20 Transfer Return Values Can Cause Silent Token Transfer Failures

Severity: Medium **Location:** ExternalTransfers.sol

Description. The functions `exexecuteWithdrawal`, `executeSimpleWithdrawal`, and `executeSimpleDeposit` do not verify the success of ERC20 token transfers. Instead, these functions call `transfer` and `transferFrom`, respectively, without checking their boolean return values as required by EIP20:

"Callers MUST handle `false` *from* `returns (bool success)`*. Callers MUST NOT assume that* `false` *is never returned!"*

Therefore, failed token transfers may "fail silently" with the transaction succeeding despite no funds being moved. This affects tokens that return `false` on failure rather than reverting. While many tokens actually do revert on failure, this cannot be assumed for all tokens.

Recommendation. Use a SafeERC20 library such as the one by OpenZeppelin. Replace all `transfer` and `transferFrom` calls in *ExternalTransfers.sol* with their corresponding safe equivalents, `safeTransfer` and `safeTransferFrom`.

#04 - Direct Usage of `ecrecover` Permits Malleable Signatures

Severity: Low **Location:** WalletOperations.sol

Description. The `verifyRootKeySignature` function in *WalletOperations.sol* uses the `ecrecover` precompile to recover the `signer` address from the signature and subsequently checks `signer` against the `rootKeyAddress`:

```
// Recover signer address using ecrecover
address signer = ecrecover(digest, v, r, s);
require(signer != address(0), "Invalid signature");

// Convert oldRootKey to address and compare
address rootKeyAddress = addressFromRootKey(rootKey);
return signer == rootKeyAddress;
```

Using `ecrecover` in that way allows for malleable signatures. More concretely, given a `newSharesCommitSig`, one can first split it into `v`, `r`, `s`, and then flip `s` and `v` via

```
bytes32 s2 =
bytes32(uint256(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141)
- uint256(s));

v2 = v == 27 ? 28 : 27;
```

This modification yields a new signature `v2`, `r`, `s2` that fulfills `ecrecover(digest, v, r, s) == ecrecover(digest, v2, r, s2)`. In other words, it gives a different signature that yields the same address.

Recommendation. Taking into account Renegade's protocol design, we currently do not see a realistic exploitation path of this issue. Still, we recommend erring on the side of caution and using an ECDSA library that prevents signatures from being malleable instead of using the `ecrecover` directly.

#05 - Lack of Two-Step Process for Ownership Transfer

Severity: Informational **Location:** Darkpool.sol

Description. The `Darkpool` contract inherits from OpenZeppelin's `Ownable` contract, which implements a single-step ownership transfer mechanism. If ownership is accidentally transferred to an incorrect address, this would result in a permanent loss of administrative control.

Recommendation. The Ownable2Step pattern provides a more secure ownership transfer mechanism by requiring the new owner to accept the transfer in a separate transaction, reducing the risk of accidental transfers. Also, if ownership is never supposed to be renounced, consider overriding `renounceOwnership` to always revert to avoid the risk of accidentally renouncing admin control.

#06 - WETH Interface Differs Between Chains

Severity: Informational **Location:** IWETH9.sol

Description. Renegade currently uses the following interface to interact with the WETH token contract:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "oz-contracts/token/ERC20/IERC20.sol";

/// @title Interface for WETH9
interface IWETH9 is IERC20 {
    /// @notice Deposit ETH and mint WETH
    function deposit() external payable;

    /// @notice Withdraw (burn) WETH and receive ETH
    function withdrawTo(address, uint256) external;
}
```

Notably, the protocol currently relies on `withdrawTo` for withdrawing native coins from the WETH contract. For WETH implementations like the one on Arbitrum, this is perfectly fine. However, `withdrawTo` might not be present in the WETH implementation on other popular EVM chains (notably Ethereum mainnet). Thus, it is possible to run into compatibility issues when expanding the protocol to different networks.

Recommendation. Consider implementing a chain-specific interface pattern to handle different WETH implementations across networks and add comments to explicitly document which chains are supported.

#07 - Smart Contracts Are Not Following Some Solidity Coding Standards

Severity: Informational **Location:** renegade-solidity-contracts/src

While the smart contracts accurately follow the Solidity style guide, there are a few possible adjustments that would improve code quality even further.

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled. To improve reliability and avoid unintended issues caused by differences between compiler versions, consider using a fixed pragma directive, e.g., instead of `pragma solidity ^0.8.0`, use `pragma solidity 0.8.28`. The only exception to this rule are Solidity libraries or "library-like" contracts, such as OpenZeppelin's and Solady's contracts. In these special cases, the pragma should not be fixed because it is usually unclear which compiler version the end user of the library will be using.

There are some instances where functions use the `public` modifier although `external` would suffice. E.g., `getMerkleRoot` in *Darkpool.sol*. For the sake of clarity, consider only using `public` when it is really necessary.

A call to `setTransferExecutor` in *Darkpool.sol* results in an important state change but does not emit a corresponding event. Consider adding an event, indicating that the `TransferExecutor` was changed.

The NatSpec does sometimes not accurately reflect the parameters of the function at hand. As an example, consider `executeWithdrawal` in *ExternalTransfers.sol*:

```
/// @notice Executes a withdrawal of shares from the darkpool
/// -> missing @param here
/// @param transfer The transfer to execute
/// -> missing @param here
function executeWithdrawal(
    PublicRootKey calldata oldPkRoot,
    ExternalTransfer calldata transfer,
    TransferAuthorization calldata authorization
)
    internal
{
    ...
}
```

Lastly, some parts of the code would benefit from more explicit error handling. For example, consider this excerpt from *TransferExecutor.sol*:

```
// Tx will revert if the buy amount is less than the total fees
uint256 totalFees = feeTake.total();
uint256 traderTake = buyAmount - totalFees;
```

Instead of relying on Solidity’s built-in underflow protection, one could add a more descriptive error message.

#08 - Unnecessary No-op Constraints

Severity: Informational **Location:** valid_malleable_match_settle_atomic.rs

Description. The malleable-match circuit and others add “no-op” constraints of the form `x * 0 === 0` on some of their public inputs. The motivation is to make sure these inputs appear in *at least one constraint*, so that they become bound to the proof, i.e. its impossible for an attacker to modify the proof to change these inputs.

We found that `constrain_malleable_fields()` in the malleable-match circuit adds no-op constraints on some variables that are already constrained elsewhere: bit `internal_fee_rates` and `external_fee_rates` already appear in range constraints added by `type_validate_fee_rates()`.

More fundamentally, no-op constraints on public inputs are in general not necessary to ensure non-malleability in PLONK. Every public input is already encoded in a dedicated row in the constraint system, as can be seen in the following snippet from the `jellyfish` library:

```
fn set_variable_public(&mut self, var: Variable) -> Result<(), CircuitError> {
    self.check_finalize_flag(false)?;
    self.pub_input_gate_ids.push(self.num_gates());

    // Create an io gate that forces <!--CODE_BLOCK_825-->.
    let wire_vars = &[0, 0, 0, 0, var];
    self.insert_gate(wire_vars, Box::new(IoGate))?;
    Ok(())
}
```

The PLONK verifier forces these witness values to equal the public inputs it receives, by including a check on the “public input polynomial”.

Furthermore, public inputs are part of the Fiat-Shamir transcript that is used to generate verifier challenges. In other words, the prover already commits to public inputs in two ways, by default: as part of the witness, and as a seperate input to FS. Modifying public inputs in either or both places would cause verification to fail.

Recommendation. No-op constraints on public inputs can safely be removed.