

Crown Smart Contracts

Security Review

Cantina Managed review by:

Devtooligan, Security Researcher
Phil, Associate Security Researcher

June 27, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 Admin reward claiming function exceeds intended scope . . . 4
  - 3.1.2 `wBRLY.sol` is vulnerable to inflation attacks . . . 4
  - 3.1.3 Yield can be "*stolen*" by front running rewards distribution . . . 5
- 3.2 Low Risk . . . 6
  - 3.2.1 Zero address blocking can disable minting functionality . . . 6
  - 3.2.2 Blocklist implementation only checks sender address, not recipient . . . 7
- 3.3 Informational . . . 7
  - 3.3.1 Unnecessary use of super keyword in inherited function calls . . . 7
  - 3.3.2 Incorrect storage gap usage . . . 7
  - 3.3.3 Burn function rounds shares in user's favor . . . 8
  - 3.3.4 Documentation for operation order in transfer . . . 8
  - 3.3.5 `permit()` is incompatible with smart contracts . . . 9
  - 3.3.6 `permit()` functions works differently across contracts . . . 10
  - 3.3.7 Unnecessary complexity for setting `YieldStripping::_underlyingDecimals` . . . 10
  - 3.3.8 Off-chain security review recommendation . . . 10
  - 3.3.9 Develop comprehensive invariant testing suite using stateful fuzzing . . . 11

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Crown aims to become Brazil's leading stable coin provider, architecting its smart contracts such that the majority of risk is transferred to off-chain components.

From Jun 20th to Jun 25th the Cantina team conducted a review of crown-brlc-eth on commit hash 24916469. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	2	1
Low Risk	2	2	0
Gas Optimizations	0	0	0
Informational	9	7	2
Total	**14**	**11**	**3**

3 Findings

3.1 Medium Risk

3.1.1 Admin reward claiming function exceeds intended scope

Severity: Medium Risk

Context: YieldStripping.sol#L108-L115

Summary: The `claimRewards()` function grants administrators excessive privileges beyond its intended purpose. While designed to allow claiming of excess tokens above the total supply, the current implementation permits administrators to drain the entire contract balance, representing a medium severity vulnerability.

Description: The `claimRewards()` function is intended to provide administrators with the ability to claim surplus tokens that exceed the protocol's total supply. However, the current implementation lacks any validation on the `rewards` parameter, allowing unrestricted token transfers:

```
function claimRewards(uint256 rewards, address receiver) public virtual override {
    _claimRewards(_msgSender(), receiver, rewards);
}

function _claimRewards(address caller, address receiver, uint256 rewards) internal virtual {
    SafeERC20Upgradeable.safeTransfer(_asset, receiver, rewards);
    emit RewardsClaimed(caller, receiver, rewards);
}
```

The function accepts any `rewards` value and transfers that amount directly to the specified receiver without checking whether the amount exceeds the intended surplus tokens. This design allows administrators to withdraw the entire contract balance, including tokens that should remain locked within the protocol.

Impact Explanation: The impact is high as administrators can drain the entire contract balance, potentially causing complete loss of user funds and protocol insolvency. This extends far beyond the intended functionality of claiming only excess tokens above the total supply.

Likelihood Explanation: The likelihood is low since the function requires administrative privileges to execute. Exploitation would require either a compromised administrator account or a malicious administrator, both of which are less probable scenarios in well-governed protocols.

Recommendation: Consider implementing a validation check to restrict withdrawals to only the excess tokens above the total supply:

```
  function _claimRewards(address caller, address receiver, uint256 rewards) internal virtual {
+     require(rewards <= (totalAssets() - totalSupply()), "Exceeds available surplus");
      SafeERC20Upgradeable.safeTransfer(_asset, receiver, rewards);
      emit RewardsClaimed(caller, receiver, rewards);
  }
```

Crown: Fixed in PR 9.

Cantina Managed: Fix verified.

3.1.2 `wBRLY.sol` is vulnerable to inflation attacks

Severity: Medium Risk

Context: wBRLY.sol#L34

Summary: By design, ERC4626 vaults are vulnerable to inflation attacks, which consists in inflating the token value per share, to benefit from slippage losses caused to further depositors. `wBRLY.sol` is an enhanced ERC4626, and hasn't mitigated this risk.

Finding Description: ERC4626 vaults hold `tokens`, and mint `shares` that represent the share of the total amount of tokens each shareholder owns. When such a vault is empty, a malicious user can mint themselves 1 wei of shares, and make an enormous donation to the vault, such that instead of `1 share == 1 token`, they can make `1 share == 1e18 token` (or even more).

Given `previewDeposit()` (which calculates the amount of shares to be minted upon `deposit()`) rounds down, if the next user deposits less than `1e18` tokens, the amount of shares they get will be rounded

down to 0, meaning their tokens will be sent to the vault, but no new shares will be minted → unfairly increasing the value per share of the inflation attacker. This is partially mitigated by Open Zeppelin's ERC4626, but the mitigation is insufficient as the attack can still be cheaply performed.

Impact Explanation: High: this issue has 2 main impacts:

- DoS deposits below the inflated token/share ratio.
- Loss of funds for future depositors.

Likelihood Explanation: Low: this can only happen when the vault is 100% empty, i.e. probably only right after deployment.

Recommendation: A mostly harmless mitigation to this is to implement decimals offset, which makes this attack unfeasible, at the cost of causing a negligible loss to shareholders. (You can find a detailed discussion of this topic on OpenZeppelin's issue 3706).

Override `ERC4626Upgradeable::_decimalsOffset()` to a reasonable value, such as 6:

```
+ function _decimalsOffset() internal view virtual override returns (uint8) {
+     return 6;
+ }
```

Crown: Fixed in PR 13.

Cantina Managed: Fix verified.

3.1.3 Yield can be "*stolen*" by front running rewards distribution

Severity: Medium Risk

Context: BRLY.sol#L536-L550

Summary: `BRLY::addRewardMultiplier()` is expected to be called once a day, distributing yield immediately to token holders. This allows users to front run these calls buying as much BRLY as they can, and sell it right afterwards, receiving a full day's worth of yield despite holding the tokens for less than a block (or for just a few blocks).

Finding Description: When `BRLY::addRewardMultiplier()` is called, each user's `balanceOf()` is increased immediately, accruing the yield generated over the past day. This happens regardless of how long the buyer held the token.

Impact Explanation:

- `User A` holds the token for 23:59:59 and sells it to `user B` right before `BRLY::addRewardMultiplier()` is called.
- `User B` buys the token from `user A` and sells it back on the same block, after the `addRewardMultiplier()` call.

In this scenario, `user A` would get 0 yield even though they held the token for the entire day, while `user B` will get a full day's worth of yield despite holding the token for less than a block.

Likelihood Explanation: This usually will only be feasible if BRLY is available on an AMM or similar contract.

Note: front running on the same block is not strictly necessary for this issue. E.g. if the function is called once a day, the user can buy it close to the time the function is usually called (as long as it is predictable).

Recommendation: Consider implementing a time-based distribution mechanism, where `rewardMultiplier` increases gradually over a fixed window (e.g., 24 hours) after each call. This ensures rewards more closely match the duration of token ownership.

Alternatively, if deployed on a chain without a public mempool (e.g. Base), a simpler solution is available: calling `addRewardMultiplier()` at randomized times each day. This mitigates front-running, as it makes the reward distributions moment unpredictable.

Crown: Acknowledged.

- For secondary buyers, the potential impact could be a temporary price destabilization. This is not a significant issue, since the daily reward rates are in the range of 1-10bps. An attacker will most likely not be able to make a profit due to slippage.
- For primary buyers, mint and burns fee would make this attack not viable.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Zero address blocking can disable minting functionality

Severity: Low Risk

Context: BRLY.sol#L420

Description: The `_blockAccount()` function does not prevent blocking the zero address, which can inadvertently disable the minting functionality. When the zero address is blocked, all subsequent mint operations will fail because the `_mint()` function calls `_beforeTokenTransfer(address(0), to, amount)`, which checks if the sender (address(0) in this case) is blocked. The current implementation only validates that the account is not already blocked:

```
function _blockAccount(address account) private {
    if (isBlocked(account)) {
        revert BRLYInvalidBlockedAccount(account);
    }

    _blocklist[account] = true;
    emit AccountBlocked(account);
}
```

However, the minting process involves a transfer from the zero address:

```
function _mint(address to, uint256 amount) private {
    // ...
    _beforeTokenTransfer(address(0), to, amount);
    // ...
}
```

The `_beforeTokenTransfer()` function checks if the sender is blocked:

```
function _beforeTokenTransfer(address from, address /* to */, uint256 /* amount */) private view {
    if (isBlocked(from)) {
        revert BRLYBlockedSender(from);
    }
    // ...
}
```

If address(0) is blocked, this check will cause all mint operations to revert.

Recommendation: Add a validation in the `_blockAccount()` function to prevent blocking the zero address:

```
  function _blockAccount(address account) private {
+     if (account == address(0)) {
+         revert BRLYInvalidBlockedAccount(account);
+     }
      if (isBlocked(account)) {
          revert BRLYInvalidBlockedAccount(account);
      }

      _blocklist[account] = true;
      emit AccountBlocked(account);
  }
```

Crown: Fixed in PR 10.

Cantina Managed: Fix verified.

3.2.2 Blocklist implementation only checks sender address, not recipient

Severity: Low Risk

Context: BRLY.sol#L231-L233

Summary: The current blocklist implementation only validates the sender ("from") address in transfer operations, while documentation suggests both sender and recipient should be blocked. This creates a gap between documented behavior and actual implementation that may not meet regulatory compliance requirements for stablecoins and similar tokens.

Description: The contract's current transfer restrictions only apply to the sender address, allowing transfers to blocked addresses. However, the documentation states "It is the caller's responsibility to ensure that blocked accounts are not provided as `to`," which implies that transfers to blocked accounts should be prevented at the contract level rather than relying on caller validation. The current implementation creates potential regulatory compliance gaps, as many jurisdictions require restrictions on both sending and receiving for blocked addresses.

Consideration should also be given to enforcing the block list on allowance spenders as well. A prime use case for this would be in the event of a phishing attack where a bad actor received approvals from multiple victims.

Recommendation: Evaluate the requirements of your jurisdictional regulatory authorities and consider implementing comprehensive blocklist checks that validate both sender and recipient addresses in all transfer operations, as well as spenders of allowances such as the `msg.sender` in a `transferFrom`.

Crown: Fixed in PR 12.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 Unnecessary use of super keyword in inherited function calls

Severity: Informational

Context: BRLV.sol#L102, BRLV.sol#L111, BRLY.sol#L479, BRLY.sol#L488, wBRLY.sol#L100, wBRLY.sol#L109

Description: The contract uses the `super` keyword when calling inherited functions like `_pause()`, even though it's not required in single inheritance scenarios. In the `pause()` function and other similar locations throughout the codebase, `super._pause()` is called when a direct call to `_pause()` would achieve the same result.

```
function pause() external onlyRole(PAUSE_ROLE) {
    super._pause();
}
```

While this code functions correctly, the `super` keyword is typically used in multiple inheritance scenarios to explicitly specify which parent contract's method should be called. In single inheritance cases, using `super` adds unnecessary complexity without providing additional functionality.

Recommendation: Simplify the function calls by removing the unnecessary `super` keyword:

```
  function pause() external onlyRole(PAUSE_ROLE) {
-     super._pause();
+     _pause();
  }
```

Crown: Fixed in PR 8.

Cantina Managed: Fix verified.

3.3.2 Incorrect storage gap usage

Severity: Informational

Context: BRLV.sol#L209, BRLY.sol#L786, wBRLY.sol#L203

Description: The contract implements storage gaps (`__gap`) in the top level, of a fully derived contract rather than a base contract intended for inheritance. Storage gaps are designed to reserve storage slots in base contracts to allow for future upgrades without affecting the storage layout of derived contracts. However, when used in the final derived contract that won't be inherited from, these gaps serve no purpose and unnecessarily consume storage slots.

According to OpenZeppelin's upgradeable contracts documentation OpenZeppelin's upgradeable contracts documentation, storage gaps should only be included in base contracts that are intended to be inherited by other contracts. The final derived contract in an inheritance chain does not need storage gaps since there are no child contracts that could be affected by future storage layout changes.

Recommendation: Remove the storage gap from the top-level contract if it is not intended to be inherited by other contracts:

Crown: Fixed in PR 7.

Cantina Managed: Fix verified.

3.3.3 Burn function rounds shares in user's favor

Severity: Informational

Context: BRLY.sol#L284

Description: The `_burn()` function uses `convertToShares()` to convert the token amount to shares for burning, which rounds down due to integer division. This means users may burn slightly fewer shares than expected for a given token amount, effectively rounding in the user's favor rather than the protocol's favor.

```
function _burn(address account, uint256 amount) private {
    // ...
    uint256 shares = convertToShares(amount); // Rounds down
    // ...
}

function convertToShares(uint256 amount) public view returns (uint256) {
    return (amount * _BASE) / rewardMultiplier; // Integer division rounds down
}
```

Best practice for rebasing tokens is to round calculations in favor of the protocol's safety. The `burn()` function is permissioned and the impact is immaterial, however, maintaining consistent rounding direction helps prevent potential issues with future development.

Recommendation: Consider creating a separate conversion function for burn operations that rounds up, or modify the burn logic to round up when calculating shares:

```
function convertToSharesRoundUp(uint256 amount) internal view returns (uint256) {
    return (amount * _BASE + rewardMultiplier - 1) / rewardMultiplier;
}

function _burn(address account, uint256 amount) private {
    // ...
    uint256 shares = convertToSharesRoundUp(amount); // Round up for burns
    // ...
}
```

Crown: Fixed in PR 11.

Cantina Managed: Fix verified.

3.3.4 Documentation for operation order in transfer

Severity: Informational

Context: BRLY.sol#L388-L391

Description: The `_transfer()` function in the BRLY contract performs share accounting operations in a specific order that prevents share loss during self-transfers. While the current implementation is safe, reversing the order of these operations could introduce a vulnerability where users lose shares when

transferring tokens to themselves. The current safe implementation decrements shares from the sender before incrementing shares for the receiver:

```
unchecked {
    _shares[from] = fromShares - shares;
    _shares[to] += shares;
}
```

If this order were reversed, `shares[from]` would be set to a previously cached value and self-transfers would result in share loss due to the interaction between cached values and storage updates.

Recommendation: Consider adding an inline comment to document this critical ordering requirement for future development:

```
  unchecked {
+     // The order of decrementing _shares[from] and incrementing _shares[to] must be maintained
+     // to prevent share loss during self-transfers (when from == to)
      _shares[from] = fromShares - shares;
      _shares[to] += shares;
  }
```

Crown: Fixed in PR 14.

Cantina Managed: Fix verified.

3.3.5 `permit()` is incompatible with smart contracts

Severity: Informational

Context: BRLV.sol#L168, BRLY.sol#L772, wBRLY.sol#L162

Summary: `permit()`, available on `BRLV.sol`, `BRLY.sol`, and `wBRLY.sol`, allows users to supply approval through a signature. However, the signature verification method is only compatible with EOAs, and not with smart contracts.

Finding Description: `permit()` verifies signatures via `ECDSAUpgradeable.recover()`:

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    if (block.timestamp > deadline) {
        revert ERC2612ExpiredDeadline(deadline, block.timestamp);
    }

    bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner, spender, value, _useNonce(owner),
    ↪  deadline));
    bytes32 hash = _hashTypedDataV4(structHash);
    address signer = ECDSAUpgradeable.recover(hash, v, r, s); // <<<

    if (signer != owner) {
        revert ERC2612InvalidSignature(owner, spender);
    }

    _approve(owner, spender, value);
}
```

This method checks if a hash has been signed by a specific `private key/ public key/ address`. However, smart contracts cannot sign messages, as they do not posess a private key. EIP-1271 was created to address this, by checking if a specific `private key/ public key/ address` is authorized to sign transactions on behalf of a smart contract.

Recommendation: Consider implementing EIP-1271 signature verification, e.g. with Open Zeppelin's `SignatureChecker.sol`

Crown: Fixed in PR 16.

Cantina Managed: Fix verified.

3.3.6 `permit()` functions works differently across contracts

Severity: Informational

Context: wBRLY.sol#L147

Summary: `BRLY.sol`, `BRLV.sol`, and `wBRLY.sol` all have a `permit()` function. In principle, the three implementations work the same way. However, there is a slight difference that has not been documented:

- In `BRLY.sol` and `BRLV.sol`, `permit()` grants the spender the right to spend up to the approved amount in *tokens*.
- In `wBRLY.sol`, `permit()` grants the spender the right to spend up to approved amount in *shares*, which means an approve of `x` will grant permission to move >`x` tokens.

This is how these functions are expected to work, and is aligned with how the `permit()` function is usually implemented for each of these types of contracts (i.e. tokens-based for ERC20s vs shares-based for ERC4626 vaults). However, having the same function with different outputs might confuse protocol users when giving appropriate approvals.

Recommendation: Add a documentation note to `wBRLY::permit()`:

```
+  * @notice Permission is granted on a shares basis, meaning a permission of `x` value will allow the spender
↪   to spend `>x`, depending on the current share value.
   * @dev See {IERC20PermitUpgradeable-permit}.
   */
  function permit(
```

Or change the `wBRLY::permit()` function name:

```
   * @dev See {IERC20PermitUpgradeable-permit}.
   */
+ function permitShares(
- function permit(
```

Crown: Fixed in PR 17.

Cantina Managed: Fix verified.

3.3.7 Unnecessary complexity for setting `YieldStripping::_underlyingDecimals`

Severity: Informational

Context: YieldStripping.sol#L37-L41

Summary: `YieldStripping.sol` takes BRLY as the underlying asset, which has decimals hardcoded as 18. However, when defining `YieldStripping::_underlyingDecimals`, the contract uses an unnecessarily complex mechanism.

Recommendation: Considering `BRLY.sol` exposes a public `decimals()` function, the logic can be replaced by querying this function directly, which is cleaner than hardcoding it as 18 again:

```
  function __YieldStripping_init_unchained(IERC20Upgradeable asset_) internal onlyInitializing {
-     (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(asset_);
-     _underlyingDecimals = success ? assetDecimals : 18;
+     _underlyingDecimals = asset_.decimals();
      _asset = asset_;
  }
```

Crown: Fixed in PR 18.

Cantina Managed: Fix verified.

3.3.8 Off-chain security review recommendation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol architecture heavily relies on off-chain components to manage critical functionality, including reward multiplier calculations, whitelist management, and yield distribution mechanisms. While this design reduces on-chain complexity and gas costs, it shifts significant security responsibility to off-chain infrastructure and applications. The current off-chain architecture includes:

- TypeScript web3-server handling RPC and Fireblocks interactions.
- Clojure application managing customer accounts and portfolios.
- AWS infrastructure using ECS Fargate and RDS Postgres.
- Auth0 for authentication and authorization.
- Pulumi for infrastructure as code and secrets management.

These systems control protocol-critical functions such as:

- Reward multiplier updates that directly impact token valuations.
- Whitelist management controlling vault access permissions.
- Yield calculations and distributions from the yield stripping mechanism.

This architecture creates a different security risk profile than fully on-chain protocols, introducing potential attack vectors including application-level vulnerabilities, infrastructure misconfigurations, third-party service dependencies, operational security gaps, and traditional web application security issues.

Recommendation: Consider conducting comprehensive security reviews of the off-chain infrastructure and applications, similar to smart contract audits. This should include:

- Application security testing of the TypeScript and Clojure codebases for logic flaws, injection vulnerabilities, and authentication bypasses.
- Infrastructure security assessment covering AWS configurations, network security, database security, and access controls.
- Third-party integration review of Auth0, Fireblocks, and other external service dependencies.
- Operational security evaluation of secrets management, monitoring, incident response procedures, and administrative access controls.
- Web application security testing including domain security, DNS configuration, and client-side vulnerabilities.

Crown: Acknowledged.

Cantina Managed: Acknowledged.

3.3.9 Develop comprehensive invariant testing suite using stateful fuzzing

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol would benefit significantly from implementing a comprehensive invariant testing suite that uses stateful fuzzing to validate system properties across various execution paths. This testing approach is particularly valuable for complex financial protocols involving rebasing tokens, yield distribution, and wrapper contracts. Invariant testing, also known as property-based testing, involves defining mathematical properties that should always hold true regardless of the sequence of operations performed on the system. Stateful fuzzing extends this concept by maintaining state between function calls, allowing the fuzzer to explore complex interaction patterns that might not be apparent in isolated unit tests.

In stateful fuzzing, the testing framework:

- Maintains persistent contract state across multiple function calls.
- Randomly selects functions to call with randomized parameters.
- Executes sequences of operations that simulate real-world usage patterns.
- Continuously validates that defined invariants remain true after each state transition.

For this protocol, invariant testing would be especially effective at catching subtle issues that could arise from the interaction between rebasing mechanics, wrapper functionality, and yield distribution. For example, operations that might appear safe in isolation could break fundamental properties when combined in unexpected sequences - such as issues that could emerge from reordering seemingly independent operations in critical functions.

Recommendation: Consider implementing an invariant testing suite that validates key protocol properties such as:

- Token conservation: Total supply relationships between BRLY, wBRLY, and BRLV remain mathematically consistent.
- Share-to-token conversion accuracy: Conversion functions maintain precision and reversibility.
- Vault accounting integrity: Total assets is always greater or equal to the sum of user balances.
- Access control consistency: Privileged operations maintain their restrictions across state changes.

Tools like Foundry's invariant testing framework or Echidna can be configured to run thousands of randomized operation sequences while continuously checking these properties. This approach would complement existing unit tests by exploring edge cases and interaction patterns that are difficult to anticipate manually.

The investment in invariant testing is particularly valuable for protocols such as stablecoin issuers, as it can identify critical vulnerabilities that traditional testing approaches might miss, including those that could emerge from seemingly minor code changes or refactoring efforts.

Crown: Acknowledged.

Cantina Managed: Acknowledged.