Zellic

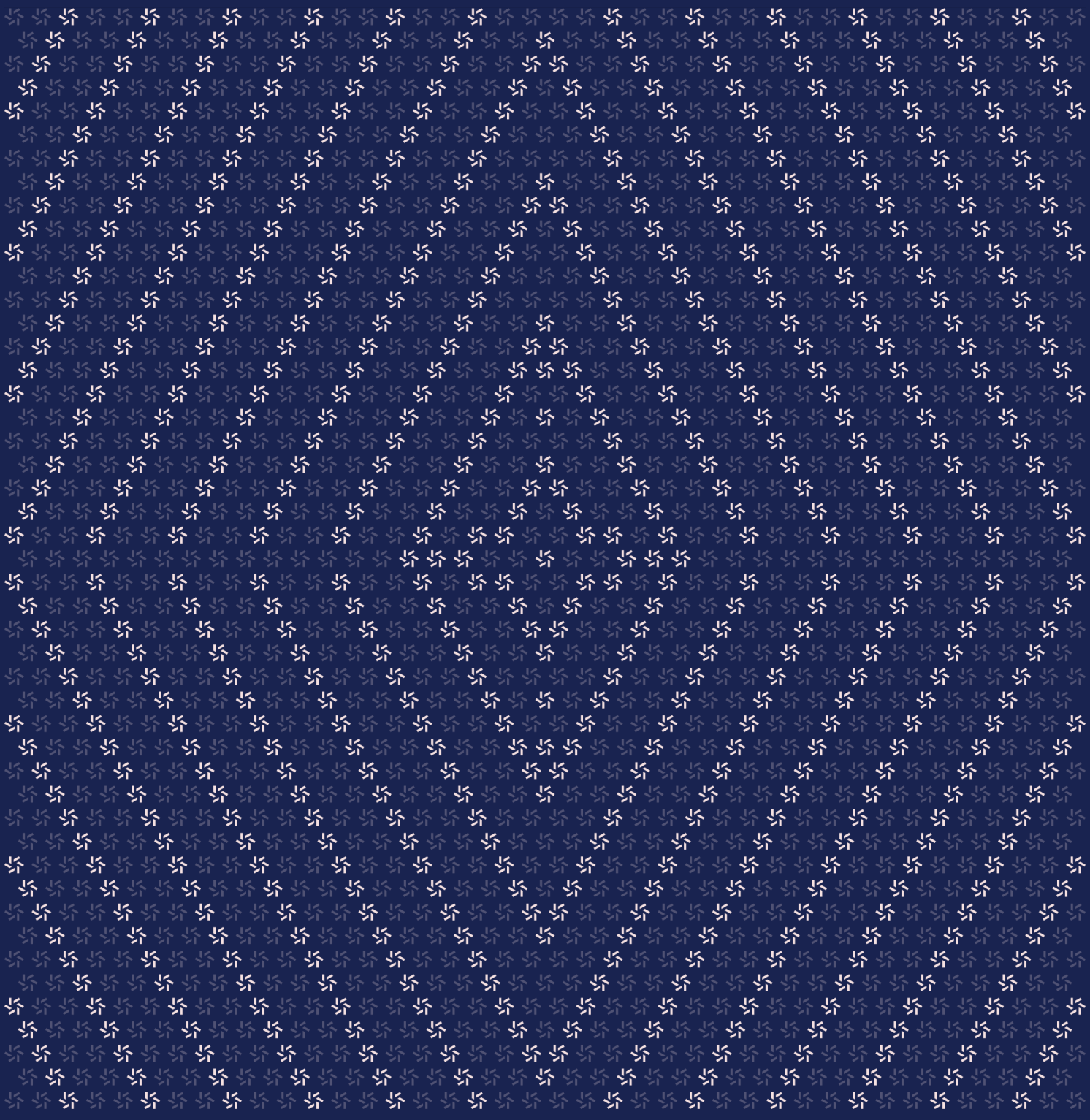
Prepared for
Reza Datoo
Familia Labs Ltd.

Prepared by
Jinseo Kim
Mohit Sharma
Zellic

March 18, 2024

Palmy Finance

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Familia Labs Ltd. from February 14th to March 1st, 2024. During this engagement, Zellic reviewed Palmy Finance's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the business logic implemented correctly?
- Are there any modifications from the Aave project that have unnoticed side effects?
- Is it possible for funds to be locked or drained by flawed business logic or by a malicious interaction?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The unchanged business logic and implementation from the Aave project
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

One should note that we solely focused on the modifications that were made to the original Aave codebase at the request of the client. The possible outcomes that are not the result of these modifications were outside the scope of this engagement.

During this assessment, a lack of comprehensive test suites prevented us from fully assessing the scoped codebase. We discuss this in section 4.1. ↗.

1.4. Results

During our assessment on the scoped Palmy Finance contracts, we discovered 11 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Familia Labs Ltd.'s benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	4
Low	2
Informational	2

2. Introduction

2.1. About Palmy Finance

Palmy Finance is a protocol for users to quickly and easily deposit and borrow assets on Oasys Hub. Depositors can provide liquidity to earn interest as a stable passive income, while borrowers can leverage their assets without selling them out.

A lending protocol is one of the core Lego blocks of DeFi composability. It is a public, decentralized marketplace of assets that can be accessed not only by a user but also by bots and other DeFi protocols. Palmy Finance provides a comprehensive set of lending features, with a focus on the assets handled on Oasys Hub.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Palmy Finance Contracts

Repositories	https://github.com/palmy-finance/palmy-makai ↗ https://github.com/palmy-finance/palmy-stake ↗ https://github.com/palmy-finance/palmy-token ↗ https://github.com/palmy-finance/palmy-ve ↗ https://github.com/palmy-finance/palmy-protocol ↗ https://github.com/palmy-finance/palmy-incentives-controller ↗
Versions	palmy-makai: `c26ad1b3f38ceaff41ebc34b5dd4a146f36e8750` palmy-stake: `d6d9013287355845d80fa946ed844ba6c6d0f256` palmy-token: `681985e99ef21ed6da9ccda8b41b6e3880f8d28e` palmy-ve: `0666d57140c242a0dcb705eab20b87cbe4330147` palmy-protocol: `a1a18fbe6e84befbfc9e6aa78d2596a4fe40974e` palmy-incentives-controller: `9a0ca4beb120b3300e1918bb57287e7de958932d`
Programs	• Leverager • stake/* • stake-v1/contracts/token/base/GovernancePowerDelegationERC20 • token/base/GovernancePowerDelegationERC20 • token/vault/PalmyRewardsVault • token/vesting/Token • token/vesting/TokenVesting • token/PlmyToken • token/PlmyTokenV2 • FeeDistributor • Voter • VotingEscrow • VotingEscrowV2 • VotingEscrowV2Rev2 • VotingEscrowV2Rev3 • libraries/Math • deployments/* • misc/* • protocol/* • flashloan/base/FlashLoanReceiverBase • dependencies/PalmyOwnable • incentives/* • lib/* • utils/*
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jinseo Kim
Engineer
jinseo@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 14, 2024	Start of primary review period
February 16, 2024	Kick-off call
March 1, 2024	End of primary review period

3. Detailed Findings

3.1. Missing checks for stale prices from oracle

Target	ChainsightOracle, PriceAggregatorAdapterChainsightImpl, PalmyFallbackOracle		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

An oracle may fail to update and provide the most recent information due to its nature. Therefore, a protocol must verify whether the oracle is supplying outdated information.

Palmy Finance utilizes a price oracle to obtain a price of an asset. The ChainsightOracle contract has the `updateState` function, which the oracle provider calls in order to supply the price information:

```
function updateState(bytes calldata _data) external override {
  data[msg.sender] = _data;
  emit StateUpdated(msg.sender, _data);
}
```

Because this function does not store the timestamp of the last update of the state, it is impossible to check if the price information is stale or not.

If the oracle does not receive price information, it fall backs to the fallback oracle. The authorized party can submit the price to the fallback oracle using the `submitPrices` function of the PalmyFallbackOracle contract:

```
function submitPrices(address[] calldata assets, uint128[] calldata prices)
    external onlySybil {
  require(assets.length == prices.length, 'INCONSISTENT_PARAMS_LENGTH');
  for (uint256 i = 0; i < assets.length; i++) {
    _prices[assets[i]] = Price(uint64(block.number), uint64(block.timestamp),
    prices[i]);
  }

  emit PricesSubmitted(msg.sender, assets, prices);
}
```

This function also does not store the timestamp of the last price, and it is impossible to check the staleness of the price information.

Impact

When the oracle does not operate properly or the authorized party does not update the price on the fallback oracle in a timely manner, an attacker can take an opportunity of arbitrage by exploiting the difference between the stale and true price. This can lead to the loss of funds in the protocol.

Recommendations

Consider storing the timestamp of the price information and checking it when the business logic uses the price information.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and fixes were implemented in the following commits:

- `fa270e5a` ↗
- `b0dc2721` ↗

3.2. The fallback oracle returns zero price when no price information is provided

Target	PalmyFallbackOracle, PalmyOracle		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

If the oracle does not have price information for the requested asset, the price information in the fallback oracle is used:

```
function getAssetPrice(address asset) public view override returns (uint256) {
  if (asset == BASE_CURRENCY) {
    return BASE_CURRENCY_UNIT;
  } else {
    int256 price = _adapter.currentPrice(asset);
    if (price > 0) {
      return uint256(price);
    } else {
      return _fallbackOracle.getAssetPrice(asset);
    }
  }
}
```

Following is the source code of the `getAssetPrice` of PalmyFallbackOracle, the fallback oracle contract:

```
function getAssetPrice(address asset) external view override returns (uint256)
    {
  return uint256(_prices[asset].price);
}
```

This function returns zero if the fallback oracle has no price information of the asset. As a result, the asset is priced at zero.

Impact

Because the asset is priced at zero, the asset can be borrowed without enough collateral. This leads to the loss of funds for the particular asset.

Recommendations

Consider checking if the fallback oracle has the price information of the asset when the protocol fetches it.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit 06d174bd ↗.

3.3. Reverting of the `close` function due to the incorrect calculation in `getHealthFactor`

Target	Leverager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Oasys Hub, the blockchain where Palmy Finance is deployed, only allows authorized users to deploy contracts. This imposes a limit to a user on calling multiple functions once unless one of the deployed contracts supports the specific series of calls.

The Leverager contract provides the utility functions that allow users to open or close the leveraged position on behalf of themselves. Opening a leveraged position works by repeatedly depositing an asset and borrowing the asset using the deposited asset as collateral. The other way around, closing a leveraged position, works by repeatedly repaying the debt and withdrawing the corresponding collateral.

The `close` function, which closes a leveraged position, involves calculating the withdrawable amount of the asset at each iteration. During this calculation, the `close` function calls the `getHealthFactor` function, which calculates the health factor after the user withdraws the given amount of the given asset. The source code of the `getHealthFactor` function is as follows:

```
function getHealthFactor(
    address account,
    address asset,
    uint256 withdrawAmount
) public view returns (uint256 healthFactor) {
    // ...
    uint256 reserveUnitPrice = priceOracleGetter.getAssetPrice(asset);
    // ...
    uint256 amountETH;
    amountETH = (withdrawAmount * reserveUnitPrice);
    totalCollateralAfter = (totalCollateral > amountETH)
        ? totalCollateral - amountETH
        : 0;
    // ...
}
```

The `amountETH` variable is supposed to represent the total price of the asset to be withdrawn and is calculated by multiplying the `withdrawAmount` and the `reserveUnitPrice`. However, it is incorrect,

because the the unit amount for the `reserveUnitPrice` is 10^{decimals} instead of 1.

If the decimal of the asset is not zero, the total collateral after the withdrawal will be underestimated due to this incorrect calculation. In most cases, this makes the function revert since this number is clamped to zero and used as a divisor.

Impact

A user will not be able to close a leveraged position through this contract. It is important to note that there are no other ways to close a leveraged position in a single transaction because a user is not allowed to deploy a contract that can perform this in a single transaction.

Users can manually close the position by interacting repeatedly with the lending pool contract. However, this process requires more time and exposes the user to the risk of being liquidated if they don't close the position promptly.

Recommendations

Consider fixing the calculation in the `getHealthFactor` function.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit d97128aa ↗.

3.4. The sources of decimal information are inconsistent

Target	Leverager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The information about the number of decimals is required to use the price obtained from the oracle. Several contracts, such as the lending pool and Leverager contract, rely on this information to function correctly.

The Leverager contract refers to the return value of the `decimals()` function of the asset to obtain the number of decimals. However, this number is not necessarily the same as the number of decimals stored in the lending pool, because it uses the number of decimals provided when the asset was initialized on the lending pool.

If these two numbers are different, the `close` function in the Leverager contract will behave in unexpected ways, including reverting, closing a position incompletely, and so on.

Impact

The prices of lent and deposited assets are calculated using the number of decimals. If this information is incorrect, the `close()` function can revert or close a position incompletely.

Although a user can manually close the position of the relevant asset by interacting with the lending pool contract repeatedly, this process requires more time and exposes the user to the risk of being liquidated before they close the position.

Recommendations

Consider fetching the number of decimals information from the lending pool.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit 2ad2372c ↗.

3.5. Suspending a token does not erase the last element of `tokenLastBalance`

Target	Voter		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Voter contract collects fees from the lending pools and distributes fees according to the share of each user. The share is decided by the voting weight of the lock that the user voted to the lending pools through.

In order to track the collected fees, the Voter contract has the `tokenLastBalance` array variable, which stores the amount of tokens recorded at the last checkpoint logic. When the checkpoint logic runs, it calculates how many tokens are added compared to the last checkpoint logic, adds it to the collected fees of the term, and updates the `tokenLastBalance` variable.

The minter, the privileged account in the Voter contract, can add the token on the contract, suspend the distribution of the token, and resume the distribution of the token. When the distribution of the token is suspended, its value in `tokenLastBalance` is erased by shifting all elements after the element of the token to be suspended:

```
function suspendToken(address _token) external onlyMinter {
    // ...
    suspendedTokenLastBalance[_token] = tokenLastBalance[arrIdx]; // save
    current tokenLastBalance to suspendedTokenLastBalance
    for (uint256 i = arrIdx; i < tokens.length - 1; i++) {
        address iToken = tokens[i + 1];
        tokens[i] = iToken;
        tokenIndex[iToken] = tokenIndex[iToken] - 1;
        uint256 nextTLastBalance = tokenLastBalance[i + 1];
        tokenLastBalance[i] = nextTLastBalance;
    }

    tokens.pop();
    tokenIndex[_token] = 0;
    pools[_token] = address(0);
    isSuspended[_token] = true;
}
```

However, this leaves the last element of `tokenLastBalance` undeleted. For instance, if the `tokenLastBalance` array is $[1, 2, 3, 0, 0, \dots]$ and the minter suspends the first token, the `tokenLastBal-`

ance array changes to $[2, 3, 3, 0, 0, \dots]$ instead of $[2, 3, 0, 0, 0, \dots]$.

Impact

This bug has two outcomes. First, it will prevent the minter from suspending another token, because the `suspendToken` checks if the next of the last element of `tokenLastBalance` is not zero:

```
function suspendToken(address _token) external onlyMinter {
    // ...
    uint256 vacantTokenLastBalance = tokenLastBalance[tokens.length];
    require(
        vacantTokenLastBalance == 0,
        "unexpected error: tokenLastBalance without token is greater than 0"
    );
    // ...
}
```

A new token should be added to the Voter contract to suspend another token.

Second, the collected fees will not be distributed as expected. More specifically, if the balance of the token added after suspension is lower than the last element value of the `tokenLastBalance`, the entire token-distribution logic will revert and not work. If not, the token-distribution logic will not revert, but the part of the collected token that was added after suspension will be locked as much as the last element value of the `tokenLastBalance`.

Recommendations

Consider erasing the last element of the `tokenLastBalance` array in the token-suspension logic.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit 1cb86f64 ↗.

3.6. The locked amount is truncated to `int128` in the `_depositFor` function

Target	VotingEscrow		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `_depositFor` function handles creating a new lock, extending the period of the lock and depositing to the lock.

```
function _depositFor(
    uint256 _lockerId,
    uint256 _value,
    uint256 unlockTime,
    LockedBalance memory lockedBalance,
    DepositType _depositType
) internal {
    // ...
    if (_value != 0) {
        _locked.amount += int128(int256(_value));
        supply = supplyBefore + _value;
    }
    // ...
    address from = msg.sender;
    if (_value != 0) {
        require(
            IERC20(token).transferFrom(from, address(this), _value),
            "fail to .transferFrom when ._depositFor"
        );
    }
    // ...
}
```

If the `_value` is nonzero, the locked amount of the lock increases and the token is transferred from the `msg.sender`. However, the `_value` is truncated to the `int128` type when the locked amount increases.

Impact

If a user tries to deposit the amount more than $2^{1}27 - 1$, this function will lock up the tokens from the user or revert.

Recommendations

Consider confirming that the given amount does not overflow in the `_depositFor` function.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit f91f7386 ↗.

3.7. The `withdrawEmergency` function does not clear the value of `ownerToId`

Target	VotingEscrowV2Rev2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The VotingEscrow contract provides the locking mechanism where users can lock their assets to obtain the voting weight. VotingEscrowV2Rev2, the upgraded contract of the VotingEscrow, allows the agency, the privileged account at the VotingEscrow contract, to process the withdrawal of the lock before the lock expires.

The `withdrawEmergency` function removes the lock and clears the information of the owner of the lock. However, it does not clear `ownerToId`, which contains the lock of the owner:

```
function _withdrawEmergency(
    uint256 _targetLockerId,
    uint256 _currentLockerId,
    address _for,
    bool _isToMsgSender
) internal onlyAgency {
    // ...
    // from VotingEscrow._removeLockerIdFrom (from
    VotingEscrow._removeLockerId)
    idToOwner[_targetLockerId] = address(0);
    // ...
}
```

Meanwhile, in the VotingEscrowV2 contract, which is inherited by VotingEscrowV2Rev2, creating a new lock requires the `ownerToId` of the address to be zero:

```
function _addLockerIdTo(address _to, uint256 _lockerId)
    internal
    virtual
    override
{
    // ...
    require(ownerToId[_to] == 0, "_to already has locker id");
    // ...
}
```

Impact

This prevents the user whose assets were withdrawn by the emergency withdrawal mechanism from creating a new lock, assuming the upgrade to the VotingEscrowV2.

Recommendations

Consider setting the `ownerToId` of the address to zero in the emergency withdrawal mechanism.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and a fix was implemented in commit ed20df78 ↗.

3.8. Checkpoint function confuses scaled balance and actual balance of LToken

Target	Voter		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Voter contract invokes the `scaledBalanceOf` function to determine its token balance, add the increased balance directly to the distributable amount of the token, divide it by each user's share, and `transfer` to them.

The token in the Voter contract is the LToken. LToken is minted when you deposit the asset to the lending pool. Conversely, you can burn the LToken to get back the asset from the lending pool.

The lending pool manages the liquidity index of the reserve, which is the cumulated scale of the sum of capital and interest compared to the initial capital. For example, if one of the reserves in the lending pool generated the total interest 25% since the initialization of the reserve, the liquidity index of the reserve is 1.25.

People who deposited to the lending pool passively earn the interest due to the passive increase of their wallet balance. This can happen because the `balanceOf` function of LToken returns the multiplication of scaled balance and the liquidity index of the reserve on the lending pool.

For instance, let us assume Alice deposited 1 OAS when the liquidity index was 1.25, and Alice received 1 lOAS. Internally, when the lending pool mints the lOAS, it divides the amount to be minted by the liquidity index and adds it to the scaled balance. In this case, the scaled balance of Alice will be 0.8.

Now let us assume again that time has passed and the liquidity index became 1.5. When Alice checks her wallet balance, it would be the 1.2 lOAS, which is larger than Alice originally deposited. Internally, the scaled balance of Alice is still 0.8, but Alice's wallet balance increased because the liquidity index increased.

Alice now transfers the 0.6 lOAS to Bob. After that, Alice and Bob will have 0.6 lOAS, respectively. Internally, the amount to be transferred, 0.6, is divided by the liquidity index, 1.5. This is 0.4, and this is subtracted and added to the scaled balances of Alice and Bob. Alice and Bob will have 0.4 of the scaled balance, respectively.

The Voter contract utilizes the `scaledBalanceOf` function to obtain its balance, but this is incorrect since this does not represent the actual balance.

Impact

The part of the cumulative interest of the token will be locked in the contract forever. For example, if the Voter contract receives 1.2 IOAS when the liquidity index is 1.5, it only distributes 0.8 IOAS and locks up the 0.4 IOAS.

Recommendations

Consider multiplying the liquidity index to the token amount to be distributed to a user or using the `balanceOf` function for tracking its balance.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and fixes were implemented in the following commits:

- `ba5b9e00` ↗
- `c4de6f8f` ↗
- `7d4b67da` ↗
- `9516bf4f` ↗
- `109bf5b9` ↗
- `63656b54` ↗

3.9. Collected fees cannot be claimed after withdrawing the locked amount

Target	FeeDistributor, Voter		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The collected fees are distributed in the FeeDistributor and Voter contracts. They use `ownerToId` of the `VotingEscrow` function to process claims for the collected fees:

```
// FeeDistributor:
function claim() external returns (uint256) {
    _checkpointToken();
    address _for = msg.sender;
    uint256 _lockerId = IVotingEscrow(votingEscrow).ownerToId(_for);
    require(_lockerId != 0, "No lock associated with address");
    // ...
}

// Voter:
function claim() external returns (uint256[] memory) {
    address _owner = msg.sender;
    uint256 _lockerId = Ve(_ve).ownerToId(_owner);
    require(_lockerId != 0, "No lock associated with address");
    // ...
}
```

However, when the locked asset is withdrawn from the locker, the `ownerToId` variable of the address is cleared:

```
// VotingEscrow:
function withdraw() external nonreentrant {
    // ...
    _removeLockerId(_lockerId);
    // ...
}

function _removeLockerId(uint256 _lockerId) internal {
    // ...
    _removeLockerIdFrom(msg.sender, _lockerId);
```

```
}

function _removeLockerIdFrom(address _from, uint256 _lockerId) internal {
    // ...
    idToOwner[_lockerId] = address(0);
    ownerToId[_from] = 0;
}
```

Impact

After a user withdraws their asset from the locker, they cannot claim the collected fees for the locker.

Recommendations

Consider refactoring the contracts not to depend on `ownerToId` after the first vote, such as saving the list of lockers of a user in the FeeDistributor and Voter contracts.

Remediation

This issue has been acknowledged by Familia Labs Ltd.. Familia Labs Ltd. stated they plan to document this behavior and encourage users to claim the collected fees before withdrawing assets from the locker in their UI.

3.10. Return value from `transferFrom` should be checked

Target	Leverager, IncentivesController		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Leverager and IncentivesController contracts use the `transferFrom` function but do not validate the return value. The `transferFrom` function is not required to revert when the transfer fails (due to the insufficient balance or approval) under the ERC-20 token standard, and some ERC-20 tokens return false rather than revert in fact.

Impact

This can lead to unexpected results on the features of these contracts (e.g., transferring the incomplete reward to the user in case the rewards vault has insufficient balance or reverting). The lack of checks in these contracts do not cause the loss of funds.

Recommendations

Consider checking the return value from `transferFrom`.

Remediation

This issue has been acknowledged by Familia Labs Ltd., and fixes were implemented in the following commits:

- 93f64e5e ↗
- 48ccb028 ↗

3.11. Centralization risk

Target	Voter, VotingEscrow		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Voter and VotingEscrow contracts allow the privileged accounts, minter and agency respectively, to disrupt the business logic if used with malice.

A malicious minter can add a malicious token to the Voter contract, which can revert the entire checkpoint and/or distribution process. This prevents users from receiving their share of collected fees.

A malicious agency can execute the emergency withdrawal mechanism at the VotingEscrow contract. This mechanism also allows an agency to withdraw the locked asset to themselves, not to the user who deposited the asset.

Impact

The Voter and VotingEscrow contracts can be disrupted in the case where the privileged accounts are exploited with malice (e.g., the malicious actor compromises those privileged accounts). Specifically, a minter in the Voter contract can halt the distribution process, and an agency in the VotingEscrow contract can drain the locked asset.

Recommendations

Consider reducing the privileged features. For instance, the privilege of an agency in the VotingEscrow contract can be reduced by modifying the emergency withdrawal function to only withdraw the asset to the owner of the lock. Also, consider implementing the multi-signature mechanism on the privileged account and adopting the time-lock mechanism in order to minimize the risk of private-key compromise.

Remediation

This issue has been acknowledged by Familia Labs Ltd.. Familia Labs Ltd. stated they plan to introduce a multi-signature wallet contract with a time-lock for privileged accounts.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Test suite

During the audit of the scoped contracts, we observed opportunities for enhancing the test suite to better align with our standards for testing in typical projects.

Implementing comprehensive test suites is crucial to identify potential bugs and ensure that the system behaves as expected. In efforts to do this, we recommend the project to strive for 100% code coverage. For instance, Finding 3.3. ↗ could have been identified in the development workflow if the tests had covered the `close` function.

We could see that many of the tests were utilizing mocking contracts. Because mocking contracts often behave differently compared to the actual contracts, our general recommendations are minimizing the usage of mocking contracts in tests and replacing them with the actual contracts the target contract is supposed to operate with.

Besides testing the entire codebase, it is also important to ensure that all functions actually behave as expected. Finding 3.8. ↗ could have been identified in the development workflow if the tests had considered the case, assuming that time passed since the deployment of the contract.

4.2. Oracle risks

The oracle mechanism is essential to Palmy Finance for evaluating the value of assets, and the correctness of the provided price information is critical for the safety of the protocol. Unlike Aave, which utilizes Chainlink oracle, Palmy Finance utilizes the oracle provided by Chainsight. It is worth mentioning the security implications of this change.

Chainlink oracle aggregates answers from multiple independent oracle operators. Because of this aggregating mechanism, an attacker has to compromise a number of oracle operators in order to manipulate the aggregated answer from the oracle.

On the other hand, Chainsight oracle receives an answer from the given address and provides the answer as is. In this case, an attacker has to compromise the single address of the oracle operator in order to manipulate the aggregated answer from the oracle.

It is recommended for users to consider the security practice of the oracle provider for assessing the risk of the protocol.

4.3. Unused code

The codebase contains multiple pieces of unused code.

For example, the VotingEscrow contract retains the information of whether a user voted or abstained, and the Voter contract calls the VotingEscrow contract to update the information. However, this does not serve any functionality in the current codebase.

Additionally, the Voter contract utilizes the `pools` mapping variable, but its presence does not impact the functionality of the code since this variable simply maps the token address to itself.

We recommend removing any unused code to maintain a clean and efficient codebase.

4.4. Typos

We have found minor typos that do not affect code functionality:

- In the contracts in the `palmy-finance-palmy-ve` repository, multiple variables whose name includes `week` should use `term` instead, because a term is two weeks.
- In the Voter contract, `_calcurateBasisTermTsFromCurrentTs` should be `_calculateBasisTermTsFromCurrentTs`.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: FeeDistributor.sol

Function: `checkpointToken()`

Accumulates the fee minted from the last checkpoint time to the current timestamp.

Branches and code coverage (including function calls)

Intended branches

- Total distributed amount equals balance difference.
  - ☐ Test Coverage

Function: `claim()`

Claims and directly deposits the fees accumulated between the last request and the present.

Branches and code coverage (including function calls)

Intended branches

- Claimed fees is locked in voting escrow.
  - ☐ Test Coverage

Negative behavior

- Reverts if no lock is associated with sender.
  - ☑ Test Coverage

5.2. Module: Leverager.sol

Function: `close(address asset)`

Loops the repaying and withdrawing.

Inputs

- `asset`
  - **Control**: Completely controlled by caller.
  - **Constraints**: Must have a valid reserve on the lending pool.
  - **Impact**: The address of the target token.

Branches and code coverage (including function calls)

Intended branches

- Full repayment succeeds if user has sufficient balance.
  - ☐ Test Coverage
- Partial repayment success.
  - ☐ Test Coverage
- Asset balance of contract is zero after success, regardless of loop count.
  - ☐ Test Coverage

Negative behavior

- Reverts if asset does not have a valid lending pool.
  - ☐ Negative Test

Function: `loop(address asset, uint256 amount, uint256 interestRateMode, uint256 borrowRatio, uint256 loopCount)`

Loops the depositing and borrowing.

Inputs

- `asset`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A
  - **Impact**: The address of the target token.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must have a valid lending pool.
  - **Impact**: The total deposit amount.
- `interestRateMode`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The interest-rate mode at which the user wants to borrow: 1 for stable, 2 for variable.

- `borrowRatio`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The percentage of the usage of the borrowed amount.
- `loopCount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: `loopCount >= 2 && loopCount <= 40`.
  - **Impact**: The loop count for how many times to deposit

Branches and code coverage (including function calls)

Intended branches

- Succeeds if user has sufficient balance.
  - ☑ Test Coverage
- Asset balance of the contract is zero after success.
  - ☐ Test Coverage

Negative behavior

- Reverts if user does not have sufficient balance.
  - ☑ Negative test

5.3. Module: Voter.sol

Function: `addToken(address _token)`

Adds a token (LToken) when the number of lending pools increases.

Inputs

- `_token`
  - **Control**: Completely controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The token address corresponding to the pool added.

Branches and code coverage (including function calls)

Intended branches

- Token is added if valid and checkable through `tokenList()` if it is a valid LToken.
  - ☑ Test Coverage

Negative behavior

- Revert if not called by minter.
  - ☑ Negative test
- Revert if token is already set.
  - ☑ Negative test

Function: `claim()`

Transfers the assigned fees to the owner of locker ID.

Branches and code coverage (including function calls)

Intended branches

- Successful claim of assigned fees.
  - ☑ Test Coverage
- Claimed token amounts transferred to the caller.
  - ☑ Test Coverage
- `tokenLastBalance` is updated for each claimed token.
  - ☑ Test Coverage

Negative behavior

- Reverts if no lock is associated with the caller's address.
  - ☑ Negative test
- Reverts if the token transfer fails.
  - ☑ Negative test

Function: `reset()`

Resets voting from the next tally.

Branches and code coverage (including function calls)

Intended branches

- Clear the user's weights for all whitelisted tokens.
  - ☑ Test Coverage

Negative behavior

- Reverts if no lock is associated with the caller's address.
  - ☑ Negative test

Function: `suspendToken(address _token)`

Suspends registration for a token (LToken) added.

Inputs

- `_token`
  - **Control**: Completely controlled by caller.
  - **Constraints**: Token must be present in `tokens`.
  - **Impact**: Token to be suspended.

Branches and code coverage (including function calls)

Intended branches

- Token removed successfully.
  - ☑ Test Coverage

Negative behavior

- Revert if not called by minter.
  - ☑ Negative test
- Revert if not whitelisted.
  - ☑ Negative test
- Revert if already suspended.
  - ☑ Negative test
- Revert if zero address.
  - ☑ Negative test

Function: `voteUntil(uint256[] _weights, uint256 _voteEndTimestamp)`

Votes the locked weight of the locker ID according to the user vote weights until `_VoteEndTimestamp`.

Inputs

- `_weights`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: `tokens.length == _weights.length`.
  - **Impact**: The vote weights.
- `_voteEndTimestamp`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The vote weights of locker ID.

Branches and code coverage (including function calls)

Intended branches

- Vote is successful with valid weights.
  - ☑ Test Coverage

Negative behavior

- Reverts if `tokens.length != _weights.length`.
  - ☐ Negative test
- Reverts if `_voteEndTimestamp` exceeds `block.timestamp + maxVoteDuration`.
  - ☑ Negative test

Function: `vote(uint256[] _weights)`

Votes the locked weight of the locker ID according to the user vote weights.

Inputs

- `_weights`
  - **Control**: Completely controlled by caller.
  - **Constraints**: `tokens.length == _weights.length`
  - **Impact**: The vote weights of locker ID.

Branches and code coverage (including function calls)

Intended branches

- Vote is successful with valid weights.
  - ☑ Test Coverage
- Vote duration is set to `maxVoteDuration` if vote end exceeds the threshold.
  - ☑ Test Coverage

Negative behavior

- Reverts if `tokens.length != _weights.length`.
  - ☑ Negative test
- Reverts if no lock is associated.
  - ☑ Negative test

Function: `_checkpointToken()`

Accumulates the fee minted for all tokens from last checkpoint time to the current timestamp.

Branches and code coverage (including function calls)

Intended branches

- Accumulates fee for all tokens if `toDistribute[i] != 0`.
  - ☑ Test Coverage
- Sum of fee distributed across weeks for a token equals the total amount to be distributed.
  - ☐ Test Coverage

5.4. Module: VotingEscrow.sol

Function: `createLockFor(uint256 _value, uint256 _lockDuration, address _to)`

Deposits `_value` tokens for `_to` and locks for `_lockDuration`.

Inputs

- `_value`
  - **Control**: Controlled by the caller (agency).
  - **Constraints**: Must be greater than zero.
  - **Impact**: Specifies the amount of tokens to deposit and lock.
- `_lockDuration`
  - **Control**: Controlled by the caller (agency).
  - **Constraints**: Must be greater than the current timestamp and less than or equal to the maximum lock duration (two years).
  - **Impact**: Specifies the duration for which the tokens will be locked.
- `_to`
  - **Control**: Controlled by the caller (agency).
  - **Constraints**: Must be a valid address.
  - **Impact**: Specifies the address to deposit and lock tokens for.

Branches and code coverage (including function calls)

Intended branches

- Successful creation of a new lock for `_to` with the specified `_value` and `_lockDuration`.
  - ☑ Test Coverage

Negative behavior

- Revert if the caller is not an agency.
  - ☑ Negative test
- Revert if `_value` is zero.

☑ Negative test

- Revert if `_lockDuration` is not greater than the current timestamp.
  ☑ Negative test
- Revert if `_lockDuration` exceeds the maximum lock duration (two years).
  ☑ Negative test
- Revert if trying to create lock twice.
  ☑ Negative test

Function: `depositFor(address _for, uint256 _value)`

Deposits `_value` tokens for `_for` and adds to the lock.

Inputs

- `_for`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be a valid address with an existing nonzero lock.
  - **Impact**: Specifies the address to deposit tokens for.
- `_value`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Specifies the amount of tokens to deposit and add to the lock.

Branches and code coverage (including function calls)

Intended branches

- Successful deposit and addition to an existing lock.
  ☑ Test Coverage

Negative behavior

- Revert if no lock is associated with `_for` address.
  ☑ Negative test
- Revert if `_value` is zero.
  ☑ Negative test
- Revert if no existing lock is found for `_lockerId`.
  ☑ Negative test
- Revert if the existing lock has expired.
  ☑ Negative test

Function: `increaseAmount(uint256 _value)`

Deposits `_value` additional tokens for `_lockerId` without modifying the unlock time.

Inputs

- `_value`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of tokens to add to the lock.

Branches and code coverage (including function calls)

Intended branches

- Successful deposit and addition to an existing lock for `msg.sender`.
  - ☑ Test Coverage

Negative behavior

- Revert if no lock is associated with `msg.sender`.
  - ☑ Negative test
- Revert if `_value` is zero.
  - ☑ Negative test
- Revert if no existing lock is found for `_lockerId`.
  - ☑ Negative test
- Revert if the existing lock has expired.
  - ☑ Negative test

Function: `increaseUnlockTime(uint256 _lockDuration)`

Extends the unlock time for `_lockerId`.

Inputs

- `_lockDuration`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must result in a new unlock time that is greater than the current unlock time and less than or equal to the maximum lock duration (two years) from the current timestamp.
  - **Impact**: Specifies the additional duration to extend the lock by.

Branches and code coverage (including function calls)

Intended branches

- Lock duration increased successfully.
  - ☑ Test Coverage

Negative behavior

- Revert if no lock is associated with `msg.sender`.
  - ☑ Negative test
- Revert if the existing lock has expired.
  - ☑ Negative test
- Revert if the locked amount is zero.
  - ☑ Negative test
- Revert if the new unlock time is not greater than the current unlock time.
  - ☑ Negative test
- Revert if the new unlock time exceeds the maximum lock duration (two years) from the current timestamp.
  - ☑ Negative test

Function: `withdraw()`

Withdraws all tokens for sender.

Branches and code coverage (including function calls)

Intended branches

- Locked balance is set to zero for `msg.sender`.
  - ☐ Test Coverage
- Withdrawn tokens are transferred to `msg.sender`.
  - ☑ Test Coverage
- Uses newly checkpointed state.
  - ☐ Test Coverage

Negative behavior

- Revert if no lock is associated with `msg.sender`.
  - ☑ Negative test
- Revert if the lock has not expired.
  - ☑ Negative test
- Revert if the token transfer fails.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Oasys Hub Mainnet.

During our assessment on the scoped Palmy Finance contracts, we discovered 11 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, two were of low impact, and the remaining findings were informational in nature. Familia Labs Ltd. acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.