Vesper Metronome Synth

(+ Vesper Escrowed Met)

Smart Contract Security Assessment

February 28, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of several smart contract modules of two Vesper protocols. The Metronome Synth and the Escrowed Met protocols.

The scope of the audit was dual and this audit report:

- Covers the contracts of the at-the-time private repository autonomoussoftware/metronome-synth of the Vesper Metronome Synth protocol, at commit hash `c2aef1234ac840c03be023f60d3708185cac6572`.

  Dedaub has audited previous versions of the protocol twice in the past. A lot of changes were made since then and this was a full reaudit. The reports of the previous audits can be found here and here.

- Covers the contracts of the at-the-time private repository autonomoussoftware/escrowed-met of the Vesper Escrowed Met protocol, at commit hash `5bc9d011fce12bf1ed39d69f29a33f87e771a3ff`.

  This protocol consists of a set of smart contracts for enabling users to lock their MET tokens for boosting their voting power and earning rewards which are proportional to the locking period. The protocol allows for early unlock of the tokens but with a penalty depending on the elapsed and the remaining time of the locking period.

  The connection between the Escrowed Met protocol and the Metronome Synth is that the users that have locked their MET tokens in Escrowed Met contracts are eligible for getting a fee discount, up to 100%, when they swap their Synthetic tokens for another Synthetic token.

Two auditors worked on the codebase for 17 days on the following contracts:

Metronome Synth

```
 contracts/
 ├─ DebtToken.sol
 ├─ DepositToken.sol
 ├─ FeeProvider.sol
 ├─ NativeTokenGateway.sol
 ├─ Pool.sol
 ├─ PoolRegistry.sol
 ├─ RewardsDistributor.sol
 ├─ SyntheticToken.sol
 ├─ Treasury.sol
 │
 ├─ access/
 │   ├─ Governable.sol
 │   └─ Manageable.sol
 │
 ├─ interfaces/*
 │
 ├─ lib/
 │   ├─ MappedEnumerableSet.sol
 │   └─ WadRayMath.sol
 │
 ├─ storage/
 │   ├─ DebtTokenStorage.sol
 │   ├─ DepositTokenStorage.sol
 │   ├─ FeeProviderStorage.sol
 │   ├─ PoolRegistryStorage.sol
 │   ├─ PoolStorage.sol
 │   ├─ RewardsDistributorStorage.sol
 │   ├─ SyntheticTokenStorage.sol
 │   └─ TreasuryStorage.sol
 │
 ├─ upgraders/
 │   ├─ DebtTokenUpgrader.sol
 │   ├─ DepositTokenUpgrader.sol
 │   ├─ FeeProviderUpgrader.sol
 │   ├─ PoolRegistryUpgrader.sol
```

```
│   ├── PoolUpgrader.sol
│   ├── RewardsDistributorUpgrader.sol
│   ├── SyntheticTokenUpgrader.sol
│   ├── TreasuryUpgrader.sol
│   └── UpgraderBase.sol
│
└── utils/
    ├── Pauseable.sol
    └── TokenHolder.sol
```

Escrowed Met

```
contracts/
├── ESMET.sol
├── ESMET721.sol
├── GovernanceToken.sol
├── Rewards.sol
│
├── access/
│   └── Governable.sol
│
├── interface/
│   ├── IESMET.sol
│   ├── IESMET721.sol
│   ├── IGovernable.sol
│   ├── IRewards.sol
│   │
│   └── external/
│       └── IMulticall.sol
│
├── storage/
│   ├── ESMET721Storage.sol
│   ├── ESMETStorage.sol
│   └── RewardsStorage.sol
│
└── upgraders/
    ├── ESMET721Upgrader.sol
    ├── ESMETUpgrader.sol
    ├── RewardsUpgrader.sol
    └── UpgraderBase.sol
```

SETTING & CAVEATS

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples:

	• User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Unbounded loops can be exploited to prevent liquidation	**RESOLVED**

The AddressSet `debtTokensOfAccount` is used by multiple functions to iterate over all debt tokens with non-zero balance for a specific account (and avoid iterating over all tokens of the protocol). For instance:

Pool::debtOf()

```
function debtOf(
    address account_
) public view override returns (uint256 _debtInUsd) {
    IMasterOracle _masterOracle = masterOracle();
    uint256 _length = debtTokensOfAccount.length(account_);
    for (uint256 i; i < _length; ++i) {
        IDebtToken _debtToken =
            IDebtToken(debtTokensOfAccount.at(account_, i));
        _debtInUsd += _masterOracle.quoteTokenToUsd(
                            address(_debtToken.syntheticToken()),
                            _debtToken.balanceOf(account_)
                        );
    }
}
```

Apart from the gas savings, using debtTokensOfAccount considerably reduces the probability of an out-of-gas denial of service: even if hundreds of synthetic tokens are registered in the protocol, each account will only own a handful of them, hence these loops under usual operation will be bounded.

On the other hand, under adversarial use such loops can be exploited. Imagine a malicious user with a large underwater position risking being liquidated. Assuming that a large number of synthetic tokens are available in the system, this user can completely prevent being liquidated as follows:

- He repeatedly issues new synthetic tokens, with the minimum allowed amount for each one, hence increasing the size of debtTokensOfAccount.
- After multiple repetitions, DebtToken::issue (which calls debtPositionOf) will exceed the block gas limit and fail. At this point no more tokens can be issued for this account.

- As a consequence, `Pool::liquidate` will also necessarily fail, since its gas usage is even higher than that of `DebtToken::issue`. So the user is protected from liquidation.
- In the future, the malicious user can repay the "dummy" tokens, reducing the size of `debtTokensOfAccount` hence removing the DoS. Note that `DebtToken::repay` uses no loops so it is not subject to the DoS.

Note that, if `debtFloorInUsd` is small (in the deployed contract it is actually `0`), then the cost of issuing a large number of tokens can be sufficiently small to make the attack profitable.

To prevent such attacks we recommend adding a maximum allowed size of `debtTokensOfAccount` (which can be large enough not to affect the regular use of the protocol).

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Lack of separation of deposited funds in the `Treasury`	**WON'T FIX**

While the protocol has splitted its logic into separate token contracts and instances for keeping distinct accountancy of each one, there is a single `Treasury` contract which receives the deposits of all underlying assets of all `Deposit` tokens.

Of course, the `Treasury` only allows access from the `DepositTokens`, which are considered as trusted. However, separating the funds as much as possible is a good security practice that could become relevant under some adversarial scenarios.

Consider, for instance, a hypothetical scenario in which a single `DepositTokenA` is compromised, while all other deposit tokens are secure. This could happen, for instance, due to the upgradeable nature of the contracts; if a vulnerability is found in the future and some unused token is not updated to the patched version, the “old” vulnerable contracts could be compromised. Even if such a scenario is unlikely, one should still have the expectation that the compromised `DepositTokenA` should only be able to steal ***its own tokens***, *not* ***all `Treasury`'s assets***.

This is not true, however, due to the lack of separation. The Treasury holds all assets, and blindly trusts the `underlying()` asset reported by the `DepositToken` itself. Hence a compromised token could call `pull` to recover any token from the Treasury.

`Treasury::pull()`

```
function pull(
    address to_,
    uint256 amount_
) external override nonReentrant onlyIfDepositToken {
    if (to_ == address(0)) revert RecipientIsNull();
    if (amount_ == 0) revert AmountIsZero();

    // Dedaub: The underlying asset is fetched from the DepositToken
    //         contract which if compromised can return any other asset
    //         of the Treasury
    IDepositToken(msg.sender).underlying().safeTransfer(to_, amount_);
}
```

Below we quote some possible solutions to address this issue:

- A separate `Treasury` for each `DepositToken` could be used to ensure that if any of them gets compromised it won't be possible for the attacker to drain the funds of the other `DepositToken`'s underlying assets
- A simpler approach could be to fetch the corresponding underlying asset from the `Pool` instead of the token itself for ensuring that even if a `DepositToken` gets compromised the attacker wouldn't be able to access any other underlying asset's funds.

L2	Possible fee discount manipulation when swapping Synthetic Tokens	**RESOLVED**

The `Pool` contract allows users to swap their Synthetic assets with other Synthetic assets, an operation which is subject to a protocol fee. By locking MET tokens, the user can get a discount on the fees. Since locking MET tokens has value for the protocol, it makes sense to provide such a discount. However, a user could still "lock" some MET, perform the swap, and immediately unlock them, getting a discount without effectively locking anything. An early unlock has a penalty, but **it is possible for the discount to still outweigh the penalty**, giving incentive to the user to perform such an action. A discount without really locking any MET has no value for the protocol so it should not be allowed.

The protocol defines a `defaultSwapFee` (=0.25%) and several fee tiers which determine whether a user is eligible of getting a fee discount (up to 100%) or not, based on the user's balance of `esMET` tokens. The tiers are handled by the `FeeProvider` governor and they define a minimum amount of `esMET` tokens that a user should hold in order to be eligible for getting the tier's discount. A user, in order to have balance of `esMET` tokens, he has to lock his `MET` tokens in the `Escrowed Met` contracts for a specific time period. Based on the lock period a user selects, he can boost his `esMET` balance up to 5x times from the initially locked amount. For example, if someone locks 1 MET token (=1 esMET) for the maximum locking period, then his balance will be boosted and will be equal to 5 esMET tokens.

The Escrowed Met contracts allow a user to unlock his tokens before the locking period ends with a penalty which goes up to 50% of his locked amount. It could be the case that the penalty of early unlocking the MET tokens does not exceed the profit one can get by granting a 100% fee discount upon swapping specific Synthetic tokens. Hence, one could buy MET tokens from the market, lock them in Escrowed Met contracts, grant the maximum discount rate, perform the swap and then unlock them losing half of them.

Furthermore, since the maximum boost one can get is able to make his balance 5x times bigger, the losses can be significantly reduced assuming that one may need only the 1/5 of the balance the maximum discount tier requires.

For example, if the 100% discount tier requires one to have a balance of 1000 esMET tokens, the user only needs to lock 200 MET tokens for the maximum locking period. This will make his esMET balance to be boosted to 1000 esMET tokens. Then, after performing the swap with a 100% fee discount he can unlock his tokens losing 50% of them which equals to 100 MET tokens.

For a user to get profit over this manipulation, the following condition has to be met:

```
0.25 * Y > 50 * X
Y > (50 / 0.25) * X
Y > 200 * X
_______________________________________________
X: The MET value
Y: The Synthetic Tokens value after swapping
```

As a result, if the value of the Synthetic Tokens he will get after performing the swap is `200x` times the value of the `MET` tokens he used, he can profit by performing the manipulation described above.

To prevent this kind of abuse, we recommend to select the discount tiers in a way that makes such an action non-profitable (and/or possibly increasing the penalty of early exit when the account has obtained a fee discount).

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Governor can sweep all funds	**RESOLVED**

Although the governor is clearly a trusted account, one should still try to limit its capabilities as much as possible and not allow unrestricted actions without a good reason. The `Treasury::sweep` function (inherited from `TokenHolder`) trivially allows the governor to extract all funds from the treasury. Since users have little risk of sending funds to the `Treasury`, and a large accidental transfer could anyway be recovered by upgrading the contract, the benefits of `Treasury::sweep` likely don't justify the risk. A `sweep` function makes perfect sense for contracts not holding funds, but not so much for a treasury.

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	No check of the Synthetic token status upon liquidating an unhealthy position in Pool contract	**RESOLVED**

In the `Pool` contract, the `liquidate()` function can be used by any user in order to liquidate an unhealthy position by repaying a user's dept and getting share of his collateral as penalty.

However, the function doesn't make use of the `onlyIfSyntheticTokenExists` modifier for ensuring the correctness of the given address.

However, if the Synthetic token is not registered in the system the execution will fail when the function will try to call `accrueInterest()` over the `address(0)`, but adding the modifier would make the code clearer and would be more consistent as well.

`Pool::liquidate()`

```
function liquidate(
    ISyntheticToken syntheticToken_,
    address account_,
    uint256 amountToRepay_,
    IDepositToken depositToken_
) external override whenNotShutdown nonReentrant
  onlyIfDepositTokenExists(depositToken_)
  returns (uint256 _totalSeized, uint256 _toLiquidator, uint256 _fee)
{
    if (amountToRepay_ == 0) revert AmountIsZero();
    if (msg.sender == account_) revert CanNotLiquidateOwnPosition();

    // Dedaub: Here the _debtToken will be equal to address(0) if the
    //         provided syntheticToken doesn't exist. Thus, the call to
    //         accrueInterest() later will fail
    IDebtToken _debtToken = debtTokenOf[syntheticToken_];
    _debtToken.accrueInterest();
    ...
}
```

A2	No check of the Synthetic token status upon repaying in DeptToken contract	**RESOLVED**

In the DeptToken contract, the repay() and repayAll() functions are used to repay back part or the entire amount, respectively, of the dept a user has. However, while other core functions use the onlyIfSyntheticTokenExists and the onlyIfSyntheticTokenIsActive modifiers to check the state of the Synthetic token before minting (i.e. issue() and flashIssue()), these two mentioned functions do not. This behaviour, however, might be desired by the protocol, but we mention it here in case they were left without these checks accidentally.

A3	Local copies of OpenZeppelin contracts are outdated	**WON'T FIX**

The protocol makes use of several contracts coming from OpenZeppelin modules. However, these contracts are copied locally in the Vesper protocol codebase. As a result, some of them might be outdated or unpatched.

For example, the local ERC20.sol contract hasn't been updated and doesn't contain the fix of the transferFrom() function that OpenZeppelin has introduced to not allow decrementing caller's allowance upon transferring tokens on behalf of someone who has granted the caller an infinite allowance.

ERC20::transferFrom() *(local copy)*

```
function transferFrom(
    address sender,
    address recipient,
    uint256 amount
) public virtual override returns (bool) {
    _transfer(sender, recipient, amount);

    uint256 currentAllowance = _allowances[sender][_msgSender()];

    // Dedaub: There is no check whether the sender has granted an
    //         infinite allowance to the caller, which can result in
    //         decrementing the caller's allowance
    require(currentAllowance >= amount,
        "ERC20: transfer amount exceeds allowance");
    unchecked {
        _approve(sender, _msgSender(), currentAllowance - amount);
    }

    return true;
}
```

ERC20::transferFrom() *(@OpenZeppelin)*

```
/**
 * @dev See {IERC20-transferFrom}.
 * ...
 * NOTE: Does not update the allowance if the current allowance
 * is the maximum `uint256`.
 * ...
 */
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

ERC20::_spendAllowance() (*@OpenZeppelin*)

```
/**
 * @dev Updates `owner` s allowance for `spender` based on spent `amount`.
 *
 * Does not update the allowance amount in case of infinite allowance.
 * Revert if not enough allowance is available.
 *
 * Might emit an {Approval} event.
 */
function _spendAllowance(
    address owner,
    address spender,
    uint256 amount
) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        require(currentAllowance >= amount,
            "ERC20: insufficient allowance");
        unchecked {
            _approve(owner, spender, currentAllowance - amount);
        }
    }
}
```

We highly recommend to keep those files updated to the latest versions of the original modules in order to ensure security and correctness.

A4	Possible gas saving when transferring tokens	**RESOLVED**

Following the previous issue, the `DepositToken` and `SyntheticToken` contracts define their own `transferFrom()` functions. However, the allowance check could be moved before the call to the `_trasnfer()` function for saving gas in case the allowance is not sufficient and cause the function to revert.

`DespotiToken::transferFrom()`
`SyntheticToken::transferFrom()`

```
function transferFrom(
    address sender_,
    address recipient_,
    uint256 amount_
) external override (...) returns (bool) {
    // Dedaub: This call to _transfer() can be moved after the allowance
    //         check for saving gas
    _transfer(sender_, recipient_, amount_);

    uint256 _currentAllowance = allowance[sender_][msg.sender];
    if (_currentAllowance != type(uint256).max) {
        if (_currentAllowance < amount_)
            revert AmountExceedsAllowance();
        unchecked {
            _approve(sender_, msg.sender, _currentAllowance - amount_);
        }
    }
    return true;
}
```

This approach has also been adopted by the OpenZeppelin ERC20 latest contract which first spends the allowance by calling the spendAllowance() and performing the actual token trasnfer afterwards.

ERC20::transferFrom() (*@OpenZeppelin*)

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    address spender = _msgSender();
    // Dedaub: Here the allowance is spent before actually transferring
    //         the tokens
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

A5	Double calls to the same modifier	RESOLVED

In the DepositToken contract, the _mint() function is declared as private and is only called by the deposit() function.

DepositToken::deposit()

```
function deposit(uint256 amount_, address onBehalfOf_)
    external
    override
    whenNotPaused
    nonReentrant
    onlyIfDepositTokenIsActive
    onlyIfDepositTokenExists
    returns (uint256 _deposited, uint256 _fee)
```

```
{
    if (amount_ == 0) revert AmountIsZero();
    if (onBehalfOf_ == address(0)) revert BeneficiaryIsNull();

    ...

    (_deposited, _fee) = quoteDepositOut(amount_);
    if (_fee > 0) {
        _mint(_pool.feeCollector(), _fee);
    }

    _mint(onBehalfOf_, _deposited);

    emit CollateralDeposited(
        msg.sender, onBehalfOf_, amount_, _deposited, _fee);
}
```

DepositToken::_mint()

```
function _mint(address account_, uint256 amount_)
    private
    onlyIfDepositTokenIsActive
    updateRewardsBeforeMintOrBurn(account_)
{ ... }
```

However, `_mint()` uses the `onlyIfDepositTokenIsActive` modifier even though it has already been used by its caller which is the `deposit()` function. Thus, it can be removed for saving some extra gas.

A6	Code inconsistencies with declared modifiers	**RESOLVED**

Some contracts of the codebase contain some redundant code that can be simplified by calling the declared modifiers. More specifically:

- In the DepositToken contract the onlyIfUnlocked modifier exists, but the withdraw() function checks whether the given amount_ has been unlocked for the caller or not. For simplicity and consistency, this check could be replaced by the use of the corresponding modifier.

  DepositToken::withdraw()

```
function withdraw(uint256 amount_, address to_)
    external
    override
    whenNotShutdown
    nonReentrant
    onlyIfDepositTokenExists
    returns (uint256 _withdrawn, uint256 _fee)
{
    if (to_ == address(0)) revert RecipientIsNull();

    // Dedaub: This check can be replaced by using the
    //         onlyIfUnlocked modifier
    if (amount_ == 0 || amount_ > unlockedBalanceOf(msg.sender))
        revert AmountIsInvalid();
    ...
}
```

- Similarly, in the Pool contract the functions addToDepositTokensOfAccount and removeFromDepositTokensOfAccount can use the onlyIfDepositTokenExists modifier instead of having code checking the same thing.

  Pool::addToDepositTokensOfAccount()

```
function addToDepositTokensOfAccount(address account_) external {
    // Dedaub: This check can be replaced by using the
    //         onlyIfDepositTokenExists modifier
    if (!depositTokens.contains(msg.sender))
        revert SenderIsNotDepositToken();
    if (!depositTokensOfAccount.add(account_, msg.sender))
        revert DepositTokenAlreadyExists();
}
```

Pool::addToDepositTokensOfAccount()

```
function addToDepositTokensOfAccount(address account_) external {
    // Dedaub: This check can be replaced by using the
    //         onlyIfDepositTokenExists modifier
    if (!depositTokens.contains(msg.sender))
        revert SenderIsNotDepositToken();
    if (!depositTokensOfAccount.add(account_, msg.sender))
        revert DepositTokenAlreadyExists();
}
```

A7	Compiler bugs	INFO

The code is compiled with Solidity 0.8.9. Version 0.8.9, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.