Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Tomo V2

Veridise Inc.

November 17, 2023

► **Prepared For:**

FansTech

► **Prepared By:**

Timothy Hoffman
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Nov. 17, 2023 V1

Contents

1 Executive Summary

From Nov. 8, 2023 to Nov. 15, 2023, FansTech engaged Veridise to review the security of a smart contract implementation of their Tomo V2 protocol. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week from commits `fbef746` - `61e9b22`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Project summary. The Tomo V2 protocol involves a system that allows users to create virtual asset stores called *subjects*, where each subject has a fungible virtual asset called a *key*. Each subject supports three operations: buy, transfer, and sell. A buy operation allows a user to pay native currency to create a given amount of key, a transfer allows a user to give their keys to another user, and a sell operation allows a user to destroy their key to obtain native currency. Buy operations additionally require the user to supply a cryptographic signature (signed by a backend service developed by Tomo) that authenticates the user to the protocol. The price of a key is determined by a *curve module* chosen by the subject owner, a Tomo V2 smart contract that calculates the price as a function of the current "supply" of keys available. The protocol is configured by a *governance* address that is allowed to perform privileged actions such as pausing the protocol and imposing percentage fees to be given to the protocol and/or the subject owners.

Code assessment. The Tomo V2 developers provided the source code of the Tomo V2 contracts for review. The source code appears to be original code written by the developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. No additional documentation was provided to the Veridise auditors for the audit. Although the auditors attempted to understand the intended behavior of the code from the source code, they were unable to accurately assess the intended behavior in several places in the code.

The source code contained a test suite, which the Veridise auditors noted provides partial test coverage of the behaviors of the protocol.

During the audit, the Tomo V2 developers made several functional changes to the code. This is because the code had not been finalized at the time of the audit start date.

Summary of issues detected. The audit uncovered 14 issues, 1 of which is assessed to be of high or critical severity by the Veridise auditors. Specifically, subject owners can steal funds from the protocol by selling their free "preserved" keys for a profit (V-TOMO2-VUL-001). The Veridise auditors also identified 9 low-severity issues, including a lack of validation on fee percentages (V-TOMO2-VUL-003, V-TOMO2-VUL-004) and a replay attack risk (V-TOMO2-VUL-008), as well as 3 warnings and 1 informational-severity issue. The Tomo V2 developers resolved all of the reported issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Tomo V2.

Due to the lack of documentation, the auditors had a hard time understanding the intended functionality of the protocol (e.g., see V-TOMO2-VUL-005). To avoid bugs such as V-TOMO2-VUL-001 and V-TOMO2-VUL-009, we recommend documenting any assumptions, caveats, and failure cases thoroughly and increasing test coverage to cover these cases.

Also, we strongly recommend that the developers reduce code duplication and refactor common functionality into reusable functions. This can help avoid bugs such as V-TOMO2-VUL-013.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Tomo V2	`fbef746 - 61e9b22`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Nov. 8 - Nov. 15, 2023	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	1	1
Medium-Severity Issues	0	0
Low-Severity Issues	9	9
Warning-Severity Issues	3	3
Informational-Severity Issues	1	1
TOTAL	14	14

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Data Validation	3
Maintainability	3
Theft	1
Authorization	1
Denial of Service	1
Replay Attack	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Tomo V2's smart contracts. In our audit, we sought to answer questions such as:

- Is the signature scheme used by `buyKey` and `buyStandardKey` susceptible to replay attacks?
- Do the curve modules correctly implement price calculations?
- Are all configuration values validated correctly?
- Are the special cases for the standard curve module handled correctly?
- Are the special cases for the const curve module handled correctly?
- Are the correct access control policies applied to buy, transfer, and sell key operations?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the `contracts` folder of the source code provided by the Tomo V2 developers, which contains the smart contract implementation of the Tomo V2.

Methodology. The Veridise auditors first inspected the provided test suite. Then they conducted a manual code review of the source code, assisted by both static analyzers and automated testing. During the audit, the Veridise auditors met with the Tomo V2 developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TOMO2-VUL-001	Subject owner can drain protocol funds by selli...	High	Fixed
V-TOMO2-VUL-002	Centralization risks	Low	Acknowledged
V-TOMO2-VUL-003	Total fee percentages can exceed 100%	Low	Fixed
V-TOMO2-VUL-004	Customized fee percentages not validated	Low	Fixed
V-TOMO2-VUL-005	Potential curve initialization DoS in buyStanda...	Low	Acknowledged
V-TOMO2-VUL-006	Behavior of initializeSubject is inconsistent w...	Low	Fixed
V-TOMO2-VUL-007	initializeSubject methods do not add preserved ...	Low	Acknowledged
V-TOMO2-VUL-008	Replay attack risk in buyKey, buyStandardKey()	Low	Fixed
V-TOMO2-VUL-009	Missing zero address checks	Low	Fixed
V-TOMO2-VUL-010	Inconsistent buy/sell price query for standard ...	Low	Fixed
V-TOMO2-VUL-011	Use of magic constant instead of BPS_MAX	Warning	Fixed
V-TOMO2-VUL-012	Inconsistent checks of referralRatio	Warning	Fixed
V-TOMO2-VUL-013	Potentially unimplemented functionality	Warning	Fixed
V-TOMO2-VUL-014	processTransfer() can be pure/view	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-TOMO2-VUL-001: Subject owner can drain protocol funds by selling preserved keys

Severity	High	**Commit**	61e9b22
Type	Theft	**Status**	Fixed
File(s)	`StandardCurveModule.sol`		
Location(s)	_getPrice()		
Confirmed Fix At	9e17ade		

Subject owners can steal money from the protocol by selling preserved keys.

When a subject is initialized with the standard curve, via `TomoV2.buyStandardKey()`, a fixed number of "preserved" keys is added to the supply for that subject. The subject owner can later sell the preserved keys via `TomoV2.sellKey()`. In this process, the `StandardCurveModule._getPrice(supply, amount)` function is called to compute the total change in value of keys as the total supply changes from `supply` to `supply + amount`.

For the standard curve, a documentation comment on `StandardCurveModule` indicates that the standard curve has the following marginal price function (as a function of supply):

> The key price is according to bonding curve,
>
> key price = if x <= _preserved then 0 else x^2 / 43370

$$p(s) = \begin{cases} 0 & \text{if } s \leq preserved \\ s^2/43370 & \text{if } s > preserved \end{cases}$$

That is, preserved keys should have a value of zero. However, the implementation of `StandardCurveModule._getPrice()` does not properly force the value of the preserved keys to be 0, which allows the subject owner to sell the preserved keys and obtain a net profit.

```
function _getPrice(
    uint256 supply,
    uint256 amount
) public view returns (uint256) {
    uint256 start = supply;
    if (supply < _preserved) start = _preserved;
    uint256 sum1 = start * (start + 1) * (2 * start + 1);
    uint256 sum2 = (start + amount) *
        (start + 1 + amount) *
        (2 * (start + amount) + 1);
    return ((sum2 - sum1) * 1 ether) / (43370 * 6);
}
```

Snippet 4.1: Definition of `StandardCurveModule._getPrice()`

Attack Scenario One simple attack scenario occurs as follows:

1. An attacker obtains a cryptographic signature from Tomo's backend that allows them to invoke `buyStandardKey()`, with the subject set to an account that they control, which we will call Account.
2. Account invokes `buyStandardKey()` with Account as the subject and an amount of `1` (the smallest possible amount to purchase). This will initialize a standard curve module for Account, granting Account three preserved keys for free.
3. Account invokes `sellKey()` to sell all four keys, each of which has a nonzero value.
4. The attacker has a net profit from selling four keys, while the protocol has a net loss (from the sale of three keys that were not paid for).

Impact Because the subject owner is able to obtain a net profit from selling keys while the protocol suffers a net loss, attackers may attempt to drain funds from the protocol by repeatedly initializing standard curves for distinct subject addresses and then selling keys using the attack scenario described above. Ultimately, the protocol may become insolvent as a result of such theft.

- The likelihood of the attack occurring will increase as the value of the native currency increases and the gas fees decrease. However, attackers that purely wish to harm the protocol without caring for their personal profits may still perform the attack even if they will lose money.
- The attack requires an attacker to obtain legitimate signatures for `buyStandardKey()`, whose signature scheme requires the `subject` to be specified. The attack can be mitigated with appropriate rate limiting or access controls by the Tomo backend server.

Recommendation Some ways to mitigate this issue include:

- Change the specification so that preserved keys have value, but require callers of `buyStandardKey()` to also pay the price of the preserved keys if the standard curve module is automatically initialized. Note that while this will still allow subject owners to sell their preserved keys for a profit (at the expense of all users owning the subject's keys), it will prevent the protocol from becoming insolvent.
- Fix the implementation so that the total price of the preserved keys is 0, e.g. clamp the interval on which the price change is computed so that the lower bound is at least 3.
- Impose strict rate limiting and access control policies on the Tomo backend server when creating signatures for `buyStandardKey()`.

Developer Response The developers changed the implementation of `_getPrice()` to the following:

```
function _getPrice(
    uint256 supply,
    uint256 amount,
    bool isSell
) private view returns (uint256) {
    uint256 start = supply;
    uint256 end = start + amount;
```



```
8
9       if (supply <= _preserved) {
10          if (isSell) {
11              start = _preserved;
12          } else {
13              start = _preserved;
14              end = start + amount;
15          }
16      }
17      if (start >= end) return 0;
18
19      uint256 sum1 = start * (start + 1) * (2 * start + 1);
20      uint256 sum2 = end * (end + 1) * (2 * end + 1);
21      return ((sum2 - sum1) * 1 ether) / (43370 * 6);
22  }
```

The auditors noted that while this version will ensure that preserved keys will be valued at 0 when sold, buying keys while the supply is below preserved will have a non-zero price. Thus, users that buy preserved keys and then sell them will have a net loss. The developers stated that is in line with the intended behavior of selling preserved keys.

Second, the developers noted that there is also a caveat that when the supply is below `_preserved`, price change will be calculated starting from `_preserved` rather than the current supply, regardless of the amount of keys purchased. They stated that this does not have a noticeable impact on the protocol.

4.1.2 V-TOMO2-VUL-002: Centralization risks

Severity	Low	**Commit**	61e9b22
Type	Authorization	**Status**	Acknowledged
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

Similar to many projects, the TomoV2 contract has an owner/governance that is given special privileges. As an example, the governance can change the curve module whitelist, the protocol fee recipients, the fee percentages, and change the Tomo signer. As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract, as a single externally-owned account is a single point of failure.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the protocol. For example, a malicious governance could, in one step, set the protocol fee to a very high value and cause users to either pay a large sum to sell or abandon at a loss.

Recommendation Utilize a decentralized governance or multi-sig contract as the owner of the contract and the fee recipient, and consider adding additional restrictions to functions accessible by governance such as limiting the maximum fee percentages.

Developer Response The developers acknowledged the issue and indicated that they "will transfer to a governance smart contract" in the future.

4.1.3 V-TOMO2-VUL-003: Total fee percentages can exceed 100%

Severity	Low	**Commit**	fbef746
Type	Data Validation	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	setCurveFeePercent()		
Confirmed Fix At	373f5aa		

The `setCurveFeePercent()` is used to set the default protocol fee (`protocolFeePercent`) and subject fee (`subjectFeePercent`) percentages for a given curve module. The two values are validated to each be less than `BPS_MAX` (100%). However, in `sellKey()`, it appears that it is assumed that `protocolFeePercent + subjectFeePercent <= BPS_MAX`.

```
function setCurveFeePercent(
    address curveModuleAddress,
    uint256 newProtocolFeePercent,
    uint256 newSubjectFeePercent
) external override onlyGov {
    if (!_curveModuleWhitelisted[curveModuleAddress])
        revert Errors.CurveModuleNotWhitelisted();
    if (newProtocolFeePercent > BPS_MAX || newSubjectFeePercent > BPS_MAX)
        revert Errors.FeePercentTooHigh();
    ICurveModule(curveModuleAddress).setFeePercent(
        newProtocolFeePercent,
        newSubjectFeePercent
    );
}
```

Snippet 4.2: Definition of `setCurveFeePercent`

Impact If the governance sets the total fee percentage to be larger than 100%, then `sellKey` will revert due to subtraction overflow.

```
uint256 protocolFee = (price * protocolFeePercent) / BPS_MAX;
payable(_protocolFeeAddress).transfer(protocolFee);

uint256 subjectFee = (price * subjectFeePercent) / BPS_MAX;
payable(vars.keySubject).transfer(subjectFee);

uint256 sellValue = price - protocolFee - subjectFee;
```

Snippet 4.3: Relevant lines in `sellKey()` where subtraction overflow may cause a revert.

Recommendation In `setCurveFeePercent()`, change the condition from

```
newProtocolFeePercent > BPS_MAX || newSubjectFeePercent > BPS_MAX
```

to

```
newProtocolFeePercent + newSubjectFeePercent > BPS_MAX
```

Developer Response Developers added checks in `setCurveFeePercent()` to revert if the sum of the fees is not exactly 100% for the `ConstCurveModule` or if the sum is greater than 10% for any other curve module.

4.1.4 V-TOMO2-VUL-004: Customized fee percentages not validated

Severity	Low	**Commit**	fbef746
Type	Data Validation	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	setCustomizedFeePercent()		
Confirmed Fix At	7032257		

Similar to V-TOMO2-VUL-003, the `setCustomizedFeePercent()` method also allows the sum of the protocol and subject fees to exceed 100%. However, `setCustomizedFeePercent()` does not perform *any* validation on the fee percentage values.

```
function setCustomizedFeePercent(
    address curveModuleAddress,
    address subjectAddress,
    uint256 newProtocolFeePercent,
    uint256 newSubjectFeePercent
) external override onlyGov {
    if (!_curveModuleWhitelisted[curveModuleAddress])
        revert Errors.CurveModuleNotWhitelisted();
    ICurveModule(curveModuleAddress).setCustomizedFeePercent(
        subjectAddress,
        newProtocolFeePercent,
        newSubjectFeePercent
    );
}
```

Snippet 4.4: Definition of `setCustomizedFeePercent()`

Impact Same impact as V-TOMO2-VUL-003.

Recommendation Add a check that causes the transaction to revert when `newProtocolFeePercent + newSubjectFeePercent > BPS_MAX`.

Developer Response Developers added checks in `setCustomizedFeePercent()` to revert if the sum of the fees is not exactly 100% for the `ConstCurveModule` or if the sum is greater than 10% for any other curve module.

4.1.5 V-TOMO2-VUL-005: Potential curve initialization DoS in buyStandardKey

Severity	Low	**Commit**	fbef746
Type	Denial of Service	**Status**	Acknowledged
File(s)	`TomoV2.sol`		
Location(s)	buyStandardKey()		
Confirmed Fix At	N/A		

The `buyStandardKey()` method is similar to the `buyKey()` method that is used to purchase a given amount of keys on a given subject, but with the difference that `buyStandardKey()` will also automatically initialize the subject's curve module to the `StandardCurveModule` if no curve module already exist for the subject.

```
if (_keySubjectInfo[vars.keySubject].curveModule == address(0)) {
    uint256 preserved = ICurveModule(STANDARD_CURVE_ADDRESS)
        .initializeCurveModule(
            vars.keySubject,
            abi.encode() //useless just suit for interface
        );
    _keySubjectInfo[vars.keySubject]
        .curveModule = STANDARD_CURVE_ADDRESS;
    _keySubjectInfo[vars.keySubject].supply += preserved;
    _keySubjectInfo[vars.keySubject].balanceOf[
        vars.keySubject
    ] += preserved;
}
```

Snippet 4.5: Relevant code in `buyStandardKey()` that performs the automatic initialization.

This behavior may cause a denial-of-service problem for the subject: a user may be able to call `buyStandardKey()` to force the module to be initialized as a standard curve module, even if the subject actually intends to initialize the curve module as a different curve module type.

Impact Users may be able to initialize a subject's curve module without the subject's authorization. Specifically, a subject may not be able to initialize their curve module to their own desired type. Note that `buyStandardKey()` requires the sender to provide a valid signature signed by Tomo's backend over the key subject and the sender, so the impact of this denial-of-service problem depends on the behavior of Tomo's backend.

Recommendation Clarify the assumptions about how `buyStandardKey()` is called and add appropriate mitigations (such as requiring the subject's authorization) for this denial-of-service problem.

Developer Response The developers noted that the current behavior is intended:

> Yeah. This is our design, At the beginning, there is only a standard curve, other curve will not open. Anyone can buy other people's key without authorization using standard curves. Later, when we open other curves, the standard curve will be disabled, and anyone need initialize their subject, than others can but.

4.1.6 V-TOMO2-VUL-006: Behavior of initializeSubject is inconsistent with documentation

Severity	Low	**Commit**	fbef746
Type	Logic Error	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	initializeSubject()		
Confirmed Fix At	9c4c499		

The documentation comment on `initializeSubject()` states that:

> initial subject when user want to jump into tomo, user can modify initialized subject many times until supply > 0, if someone buy your key. can not modify anymore

However, the auditors were unable to find any logic in `initializeSubject()` corresponding to this behavior. Based on the current implementation of `initializeSubject()`:

- `initializeSubject()` may only be called once by the sender, as the method requires `_keySubjectInfo[msg.sender].curveModule` to be initially set to the zero address and will set the value to a nonzero address.
- There is no code in `initializeSubject()` that writes or reads any supply value.

Impact The implementation of `initializeSubject()` may not be consistent with the behavior intended by the developers.

Recommendation Clarify the intended behavior of `initializeSubject()` and update the documentation comment and/or `initializeSubject()`.

Developer Response The developers noted that the documentation comment is incorrect, and that the current behavior is intended.

4.1.7 V-TOMO2-VUL-007: initializeSubject methods do not add preserved keys to supply

Severity	Low	Commit	fbef746
Type	Logic Error	Status	Acknowledged
File(s)	TomoV2.sol		
Location(s)	initializeSubject(), initializeSubjectByGov()		
Confirmed Fix At	N/A		

The `ICurveModule.initializeCurveModule()` method may return an unsigned integer representing the number of keys that should be initially granted to some user, such as the owner of the subject that the curve module is being initialized on. In `buyStandardKey()`, when a standard curve module is automatically initialized for the subject, these "preserved" keys are granted to the subject.

```
if (_keySubjectInfo[vars.keySubject].curveModule == address(0)) {
    uint256 preserved = ICurveModule(STANDARD_CURVE_ADDRESS)
        .initializeCurveModule(
            vars.keySubject,
            abi.encode() //useless just suit for interface
        );
    _keySubjectInfo[vars.keySubject]
        .curveModule = STANDARD_CURVE_ADDRESS;
    _keySubjectInfo[vars.keySubject].supply += preserved;
    _keySubjectInfo[vars.keySubject].balanceOf[
        vars.keySubject
    ] += preserved;
}
```

Snippet 4.6: Relevant lines in `buyStandardKey()`.

However, there is no code in `initializeSubject()` and `initializeSubjectByGov()` that will grant the "preserved" keys to the subject.

```
function initializeSubject(
    DataTypes.InitialSubjectData calldata vars
) external override whenInitializeSubjectEnabled {
    if (!_curveModuleWhitelisted[vars.curveModule])
        revert Errors.CurveModuleNotWhitelisted();
    if (_keySubjectInfo[msg.sender].curveModule != address(0))
        revert Errors.SubjectAlreadyInitialized();

    _keySubjectInfo[msg.sender].curveModule = vars.curveModule;
    ICurveModule(vars.curveModule).initializeCurveModule(
        msg.sender,
        vars.curveModuleInitData
    );
}
```

Snippet 4.7: Implementation of `initializeSubject()`. The `initializeSubjectByGov()` method is similar.

Impact Such "preserved" keys will not be recorded in the balance of the subject when curve module is initialized by `initializeSubject()` and `initializeSubjectByGov()`, nor will supply be updated in the `TomoV2` contract.

Currently, the only curve module that returns a non-zero "preserved" keys value is the `StandardCurveModule`. Since the `StandardCurveModule` will still record the key amount in its own storage, all prices calculated by the `StandardCurveModule` will be higher than they should be for the actual amount of keys in circulation. For example, if a subject manually calls `initializeSubject()`, then they will start out with a balance of zero and would have to pay a non-zero amount of currency in order to acquire some keys.

Recommendation Insert code into `initializeSubject()` and `initializeSubjectByGov()` to add the preserved keys to the supply and the balance of the subject. To prevent future inconsistencies, the common initialization code could be moved into a helper function that can be called by `buyStandardKey()`, `initializeSubject()`, and `initializeSubjectByGov()`.

Developer Response The developers note that `initializeSubject` and `initializeSubjectByGov()` are not meant to be used with the `StandardCurveModule`:

> `StandardCurveModule` address is not in whitelist curve module. it's built-in in variable `STANDARD_CURVE_ADDRESS`, so `initializeSubject` and `initializeSubjectByGov` can not initial standard curve if we open Const/Linear/Quadratic curve(at same time we will disable buyStandardKey, so all subject need be initialized first)

4.1.8 V-TOMO2-VUL-008: Replay attack risk in buyKey, buyStandardKey()

Severity	Low	**Commit**	fbef746
Type	Replay Attack	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	buyKey(), buyStandardKey()		
Confirmed Fix At	1f91792		

The `buyKey()` and `buyStandardKey()` can be called by a user in order to purchase a given `amount` of keys from a specific `subject`. These two methods require the user to also provide a valid ECDSA signature over a tuple `(BUY_TYPEHASH, subject, sender, amount)`, where:

- `BUY_TYPEHASH` is a keccak256 hash identifying this as a buy key operation.
- `subject` is the subject corresponding to the keys being purchased.
- `sender` is the address making the key purchase, which `buyKey/buyStandardKey` enforce to be equal to the `msg.sender`.
- `amount` is the amount of keys to purchase.
- The signer is an externally-owned account controlled by Tomo.

This signature scheme does not include sufficient data to guard against replay attacks.

```
unchecked {
    _validateRecoveredAddress(
        _calculateDigest(
            keccak256(
                abi.encode(
                    BUY_TYPEHASH,
                    vars.keySubject,
                    msg.sender,
                    vars.amount
                )
            )
        ),
        _tomoSignAddress,
        vars.sig
    );
}
```

Snippet 4.8: Relevant code in `buyKey()`. A similar code snippet is in `buyStandardKey()`

Impact Once a user obtains a signature corresponding to a call to `buyKey/buyStandardKey()` is obtained, the user will be able to call `buyKey/buyStandardKey()` with the same `subject` and `amount` as many times as they'd like in the future.

Recommendation Include mitigations against replay attacks, such as an expiry time or nonce.

Developer Response The developers stated that this is intended behavior. The signing mechanism is only used to authenticate users so-as to prevent bot accounts from spamming key purchases.

To avoid a signature obtained for `buyKey()` being reused for `buyStandardKey()` (and vice versa), the developers changed the type hash for `buyStandardKey()`.

4.1.9 V-TOMO2-VUL-009: Missing zero address checks

Severity	Low	**Commit**	61e9b22
Type	Data Validation	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	See description		
Confirmed Fix At	582f334		

There are several functions that accept an `address` parameter but do not validate that the address is nonzero. In most cases, an `address` with value `0` indicates invalid or uninitialized data, so such parameters should be validated.

- The `TomoV2._setProtocolFeeAddress()` method allows `_protocolFeeAddress` to be assigned the value `0`. This address is used as the destination of a transfer within the `sellKey` and `_buyKey` functions.

```
function _setProtocolFeeAddress(address newProtocolFeeAddress) internal {
    address preProtocolFeeAddress = _protocolFeeAddress;
    _protocolFeeAddress = newProtocolFeeAddress;
        ...
```

Snippet 4.9: Relevant lines in `_setProtocolFeeAddress()`

- The `TomoV2._setTomoSignAddress()` method allows `_tomoSignAddress` to be assigned the zero address. This address is used within the `buyKey()` and `buyStandardKey()` functions when validating the provided signature `_validateRecoveredAddress(_, _tomoSignAddress )` and will always result in a `SignatureInvalid` revert within that function if the zero address is provided.

```
function _setTomoSignAddress(address newTomoSignAddress) internal {
    address preTomoSignAddress = _tomoSignAddress;
    _tomoSignAddress = newTomoSignAddress;
    ...
```

Snippet 4.10: Relevant lines in `_setTomoSignAddress()`.

- The `TomoV2.setCustomizedFeePercent` function allows `subjectAddress == 0` to be passed to `ICurveModule.setCustomizedFeePercent()`, but there doesn't seem to be a legitimate reason to allow that value. This likely indicates a bug.
- The `TomoV2._setGovernance()` method allows `_governance` to be assigned the value `0`.

Impact

- When `_setProtocolFeeAddress()` sets `_protocolFeeAddress` to `0`, later transfers within the `sellKey` and `_buyKey` functions actually burn the protocol fees, which is likely undesirable.
- When `_setTomoSignAddress()` sets `_tomoSignAddress` to `0`, all calls to `_validateRecoveredAddress` will revert, which means all calls to `buyKey` and `buyStandardKey` will revert, regardless of the subject. Only the governance address can make such a change (e.g., as a result of a mistake), but it will cause a denial-of-service issue for all users attempting to buy keys through those two methods.

```
function setCustomizedFeePercent(
        address curveModuleAddress,
        address subjectAddress,
        uint256 newProtocolFeePercent,
        uint256 newSubjectFeePercent
    ) external override onlyGov {
        if (!_curveModuleWhitelisted[curveModuleAddress])
            revert Errors.CurveModuleNotWhitelisted();
        ICurveModule(curveModuleAddress).setCustomizedFeePercent(
            subjectAddress,
            newProtocolFeePercent,
            newSubjectFeePercent
        );
    }
```

Snippet 4.11: Relevant lines in `setCustomizedFeePercent()`.

```
function _setGovernance(address newGovernance) internal {
    address prevGovernance = _governance;
    _governance = newGovernance;
        ...
```

Snippet 4.12: Relevant lines in `_setProtocolFeeAddress()`

- If the governance accidentally calls `setCustomizedFeePercent()` with a zero `subjectAddress`, the call will succeed even if it likely that the governance is making a mistake.
- If the governance calls `setGovernance()` with the zero address, that change is permanent. All functions marked with the `onlyGov` modifiers (including this one) will always revert. It is not clear whether a lack of a zero address check here is intended behavior (e.g., developers relinquishing control over the protocol to increase decentralization) or whether this is a bug (e.g., governance is controlled by a DAO).

Recommendation For each location indicated above, insert code to revert if the provided address parameter is the zero address.

Developer Response The developers added zero address checks in all of the above locations.

4.1.10 V-TOMO2-VUL-010: Inconsistent buy/sell price query for standard curves

Severity	Low	**Commit**	61e9b22
Type	Logic Error	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	getSellPrice() and getBuyPrice()		
Confirmed Fix At	cfece2e		

The `TomoV2` contract provides functions to query the current buy and sell prices for subject keys. In most cases, these functions call the corresponding curve module functions for the given subject key. Both functions have a special case for when the curve module has not been initialized for the given subject. The `getSellPrice()` function returns `0` when the curve module has not been initialized, but the `getBuyPrice()` function queries the `STANDARD_CURVE_ADDRESS` in this case.

```
function getBuyPrice(
    address subject,
    uint256 amount
) external view returns (uint256) {
    address curve = _keySubjectInfo[subject].curveModule == address(0)
        ? STANDARD_CURVE_ADDRESS
        : _keySubjectInfo[subject].curveModule;

    return ICurveModule(curve).getBuyPrice(subject, amount);
}

function getSellPrice(
    address subject,
    uint256 amount
) external view returns (uint256) {
    if (_keySubjectInfo[subject].curveModule == address(0)) return 0;
    return
        ICurveModule(_keySubjectInfo[subject].curveModule).getSellPrice(
            subject,
            amount
        );
}
```

Snippet 4.13: Relevant functions from TomoV2.sol

Furthermore, `getBuyPrice()` will return the wrong price for uninitialized standard curve modules. When a standard curve module is initialized in `buyStandardKey()`, the `StandardCurveModule` will set the supply to the number of preserved keys. However, `getBuyPrice()` will not perform any automatic initialization, so the supply will remain as zero when it is queried.

Impact

- The inconsistent behavior between `getBuyPrice()` and `getSellPrice()` can be confusing to users.
- On uninitialized curve modules, `getBuyPrice()` will return a lower price than would be required in `buyStandardKey()`.

```
function getBuyPrice(
    address subject,
    uint256 amount
) external view override returns (uint256) {
    return
        _getPrice(
            _dataStandardCurveBySubjectAddress[subject].supply,
            amount
        );
}
```

Snippet 4.14: Definition of `StandardCurveModule.getBuyPrice()`. The supply will be zero for an uninitialized standard curve module.

```
function initializeCurveModule(
    address subjectAddress,
    bytes calldata
) external override onlyTomoV2 returns (uint256) {
    _dataStandardCurveBySubjectAddress[subjectAddress].supply = _preserved;
    return _preserved;
}
```

Snippet 4.15: Definition of `StandardCurveModule.initializeCurveModule()`

Recommendation Any of these options would make the interface clear and consistent:

- Both functions default to `STANDARD_CURVE_ADDRESS` when the curve module is not initialized, and logic is added to correctly handle the preserved keys.
- Both functions return `0` when the curve module is not initialized. Instead, add separate functions to query the standard curve module buy/sell price.

Developer Response The developers changed `getSellPrice()` to be similar to `getBuyPrice()`. The `getBuyPrice()` problem has been fixed with the changes made by the developer for V-TOMO2-VUL-001.

4.1.11 V-TOMO2-VUL-011: Use of magic constant instead of BPS_MAX

Severity	Warning	**Commit**	61e9b22
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	5ac17c8		

The `TomoV2` file defines a constant `BPS_MAX` with value `10000` used to represent `100%`. However, there are multiple locations in the code that use a hardcoded `10000` literal instead of `BPS_MAX`.

```
1 if (newReferralRatio > 10000) revert Errors.ReferralRatioTooHigh();
```

Snippet 4.16: Location in `TomoV2.setReferralRatio()`

```
1 if (referralRatio >= 10000) revert Errors.ReferralRatioTooHigh();
```

Snippet 4.17: Location in `ConstCurveModule.initializeCurveModule()`. The same issue occurs in `LinearCurveModule` and `QuadraticCurveModule`.

Impact If the developers update `BPS_MAX` in the future (e.g., to increase the number of decimals), they may forget to update all of the locations that use the `100%` value.

Recommendation Replace all occurrences of `10000` with `BPS_MAX`, and then move the definition of `BPS_MAX` to another file so that it can be included in all places that use `BPS_MAX`.

Developer Response The developers fixed the hardcoded `BPS_MAX` in `TomoV2`. However, they did not fix the curve modules for the following reason:

> All curve modules will not change or upgrade.

4.1.12 V-TOMO2-VUL-012: Inconsistent checks of referralRatio

Severity	Warning	**Commit**	61e9b22
Type	Maintainability	**Status**	Fixed
File(s)	`TomoV2.sol`		
Location(s)	setReferralRatio()		
Confirmed Fix At	cfece2e		

The `TomoV2.setReferralRatio()` method can be used to change a subject's curve module referral reward percentage (called the "referral ratio"), which is required to be less than or equal to 100% (`10000` in the code). All curve module implementations (except for `StandardCurveModule`) may also allow the initial referral ratio to be set during initialization, but with the additional requirement that the ratio is strictly less than 100%. These two behaviors are inconsistent.

```
function setReferralRatio(uint256 newReferralRatio) external override {
    if (_keySubjectInfo[msg.sender].curveModule == address(0))
        revert Errors.SubjectNotInitialized();
    if (newReferralRatio > 10000) revert Errors.ReferralRatioTooHigh();
    ICurveModule(_keySubjectInfo[msg.sender].curveModule).setReferralRatio(
        msg.sender,
        newReferralRatio
    );
}
```

Snippet 4.18: Definition of `TomoV2.setReferralRatio()`

```
function initializeCurveModule(
    address subjectAddress,
    bytes calldata data
) external override onlyTomoV2 returns (uint256) {
    (uint256 price, uint256 timePeriod, uint256 referralRatio) = abi.decode(
        data,
        (uint256, uint256, uint256)
    );
    if (referralRatio >= 10000) revert Errors.ReferralRatioTooHigh();
    ...
```

Snippet 4.19: Relevant code from `ConstCurveModule.initializeCurveModule()`. `LinearCurveModule` and `QuadraticCurveModule` have similar code snippets.

Impact This issue is indicative of a bug. Using `TomoV2.setReferralRatio` allows a higher referral ratio to be set (e.g., 100% is allowed) than is allowed when initializing the curve modules (e.g., 100% is not allowed). However, it is unlikely that a subject will set a referral ratio of 100%, as all subject fees from a buy key operation will be sent to the referral address (if provided).

Recommendation Clarify the intended behavior of a referral ratio of exactly 100% and make the check consistent in all locations.

Developer Response The developers stated that the intended behavior is to allow a referral ratio of 100%:

> Referrer get benefit from the owner's subject fee. so we assume subject owner can give all subject fee to referrer.

4.1.13 V-TOMO2-VUL-013: Potentially unimplemented functionality

Severity	Warning	**Commit**	61e9b22
Type	Logic Error	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	5ac17c8		

Several files contain code that performs unnecessary storage writes, either because the location written to is never read from or the write is guaranteed to make no change to existing data.

- The `ConstCurveData` struct defined in `ConstCurveModule.sol` has a field named `timePeriod` that is written to storage in `ConstCurveModule.initializeCurveModule()`, but there is no function that reads this field.

```
struct ConstCurveData {
    uint256 price;
    uint256 timePeriod;
    uint256 supply;
    uint256 referralRatio;
}
```

Snippet 4.20: Definition of `ConstCurveData`

- The `_customizedFeePercent` mapping defined in the `StandardCurveModule` contract is written to by `setCustomizedFeePercent()`, but there is no location that reads this data. Note that the other curve module contracts will store customized fee percent values to be used in `processBuy()`.

```
contract StandardCurveModule is ModuleBase, ICurveModule {
    mapping(address => StandardCurveData)
        internal _dataStandardCurveBySubjectAddress;
    mapping(address => CustomizedFeePercent) internal _customizedFeePercent;
```

Snippet 4.21: Relevant lines in `StandardCurveModule`

- Within the `sellKey` function of the `TomoV2` contract, there is a `delete` statement for a memory location wrapped inside a condition that checks if that location has value `0`. However, the main effect of `delete` is to set the memory location to its default value, in this case `0` which means the code has no effect.

```
if (_keySubjectInfo[vars.keySubject].balanceOf[msg.sender] == 0)
    delete _keySubjectInfo[vars.keySubject].balanceOf[msg.sender];
```

Snippet 4.22: Relevant lines in `TomoV2.sellKey()`

Impact The presence of dead code may indicate a mistake in the implementation or unimplemented functionality that is important to the protocol. Another side effect is that unused writes waste gas.

Recommendation Ensure that unused writes and/or dead code do not exist because of a typo, bug, or missing implementation. Otherwise, remove the unused writes and/or dead code.

Developer Response The developers added code to `StandardCurveModule` to handle customized fee percentages, added a method to `ConstCurveModule` to retrieve the `timePeriod` field, and removed the dead code in `TomoV2.sellKey()`.

4.1.14 V-TOMO2-VUL-014: processTransfer() can be pure/view

Severity	Info	Commit	fbef746
Type	Maintainability	Status	Fixed
File(s)	ICurveModule.sol		
Location(s)	processTransfer()		
Confirmed Fix At	5ac17c8		

The `processTransfer()` function is declared in the `ICurveModule` interface and defined within each contract that implements this interface. Every implementation uses the `pure` keyword on the function because they do not read or modify the contract state.

```
1 function processTransfer() external returns (bool);
```

Snippet 4.23: Declaration of `processTransfer()` in `ICurveModule`

Recommendation The declaration in `ICurveModule` can be marked with either the `pure` or `view` keyword.

Developer Response The developers added the `pure` keyword to the declaration of `processTransfer()` in `ICurveModule`.