DEDAUB

DEDAUB.COM

GYSR

Smart Contract Security Assessment

April 04, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the GYSR protocol. Some issues were found, including Medium and Low severity issues and one Critical issue consisting of a reentrancy attack. All of them were properly addressed and resolved.

The GYSR protocol is also accompanied by a very thorough and comprehensive test suite with thousands of test cases covering more than 95% of the entire protocol including functions, branches, statements and lines with multiple scenarios for each case to ensure the correctness and enhance the security of the protocol.

BACKGROUND

GYSR allows developers to easily add on-chain incentive methods to their product. For instance, it can be used for yield farming, distributing a new token, distributing protocol fees, etc. Dedaub audit v3 of the protocol as a standalone full audit. Version 3 of the protocol adds new mechanisms (such as bond sale, continuous auctions, fixed-price payments, etc) and offers more combinations of different staking and reward modules.

SETTING & CAVEATS

This audit report covers the contracts of the at-the-time private repository gysr-io/core-dev of the GYSR protocol at commit `74631752adfa948b0afe814e284032754d82c96e` and the changes introduced in the Pull Request #38 that were delivered in the middle of the audit as part of improvements in the calculations of `ERC20FixedRewardModule` and rework of the bookkeeping mechanisms of that module.

As part of the audit, the auditors also reviewed the fixes for the issues included in the report. The fixes were delivered as part of a secondary, also private, repo (gysr-io/core-private) which is aligned with the one used for the audit. The initial commit of the second repo that aligns both repos is 6e78ee4a152726e34ff9a8da08e3c703415e7d31. The auditors reviewed the fixes up to commit 614dfffb97cb136190594849c50b29766341f3cd.

A few months after the initial audit, Dedaub was also asked to perform a secondary delta audit on the recent changes introduced in PR #47 (commit 33aba064b48b9d7f44ef925af51fad09e2dd4b2f) of the at-the-time private repository gysr-io/core-dev. The changes only affected the ERC20MultiRewardModule.sol contract, which was also the scope of this delta audit, and consisted of adding some extra functionality to allow deregistration of a reward token without having to do a transfer first. The rationale behind this change is that if any of the reward tokens are faulty and cannot be transferred, all users who have registered them will not be able to unstake, as the whole process will revert when trying to transfer the faulty tokens, causing a lockup to this function. However, allowing these tokens to be deregistered and their rewards renounced without performing a transfer would allow users to unstake. As a result, the auditors found that the changes have been properly implemented enabling the desired functionality without introducing any security threats.

Two auditors worked on the codebase for 14 days on the following contracts:

```
contracts/
├─ AssignmentStakingModule.sol
├─ AssignmentStakingModuleFactory.sol
├─ Configuration.sol
├─ ERC20BaseRewardModule.sol
├─ ERC20BondStakingModule.sol
├─ ERC20BondStakingModuleFactory.sol
├─ ERC20CompetitiveRewardModule.sol
├─ ERC20CompetitiveRewardModuleFactory.sol
├─ ERC20FixedRewardModule.sol
├─ ERC20FixedRewardModuleFactory.sol
├─ ERC20FriendlyRewardModule.sol
├─ ERC20FriendlyRewardModuleFactory.sol
├─ ERC20LinearRewardModule.sol
├─ ERC20LinearRewardModuleFactory.sol
├─ ERC20MultiRewardModule.sol
├─ ERC20MultiRewardModuleFactory.sol
├─ ERC20StakingModule.sol
├─ ERC20StakingModuleFactory.sol
├─ ERC721StakingModule.sol
├─ ERC721StakingModuleFactory.sol
├─ GeyserToken.sol
├─ GysrUtils.sol
├─ MathUtils.sol
├─ OwnerController.sol
├─ Pool.sol
├─ PoolFactory.sol
├─ TokenUtils.sol
│
├─ info/
│   ├─ AssignmentStakingModuleInfo.sol
│   ├─ ERC20BondStakingModuleInfo.sol
│   ├─ ERC20CompetitiveRewardModuleInfo.sol
│   ├─ ERC20FriendlyRewardModuleInfo.sol
│   ├─ ERC20LinearRewardModuleInfo.sol
│   ├─ ERC20StakingModuleInfo.sol
│   ├─ ERC721StakingModuleInfo.sol
│   └─ PoolInfo.sol
│
└─ interfaces/*
```

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 behaviour. 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:

ID	Description	STATUS
C1	Reward shares can be drained by the `controller` devalued via a reentrancy attack	**RESOLVED**

This vulnerability arises from two separate issues in different parts of the code:

1. The `TokenUtils::receiveAmount/receiveWithFee` functions compute the amount of received tokens as the difference in balance before and after the transfer.

   `TokenUtils::receiveAmount()`

```
function receiveAmount(
    IERC20 token,
    uint256 shares,
    address sender,
    uint256 amount
) internal returns (uint256) {
    //  transfer
    uint256 total = token.balanceOf(address(this));
    token.safeTransferFrom(sender, address(this), amount);
    uint256 actual = token.balanceOf(address(this)) - total;

    // mint shares at current rate
    uint256 minted = (total > 0)
        ? (shares * actual) / total
        : actual * INITIAL_SHARES_PER_TOKEN;
    require(minted > 0);
    return minted;
}
```

The goal is to support different types of tokens (e.g. tokens with transfer fees). This approach, however, introduces a possible attack vector: the code could miscalculate the amount of tokens transferred if some other action is executed in between the two balance readings. Note that `token.safeTransferFrom()` is an external call outside our control. As such, we cannot exclude the possibility that it returns execution to the adversary (e.g. via a transfer hook).

2. The `fund()` function, of all reward modules, has no reentrancy guards (likely due to the fact that funding sounds “harmless“; we send tokens to the contract without getting anything back).

The possible attack:

We assume a malicious controller that creates a pool with `ERC20FixedRewardModule` (for simplicity). His goal is to receive the benefits of staking but without giving any rewards back. The reward token used in the pool is a legitimate trusted token. We only assume that it has some `ERC777-type` transfer hook (or any mechanism to notify the sender when a `transferFrom` happens).

1. The adversary funds the reward module and waits until several users have staked tokens (giving them rights to reward tokens).

2. He then initiates a number of k nested calls to `ERC20FixedRewardModule::fund` as follows:

ERC20FixedRewardModule::fund()

```
function fund(uint256 amount) external {
    require(amount > 0, "xrm4");
    (address receiver, uint256 feeRate) = _config.getAddressUint96(
        keccak256("gysr.core.fixed.fund.fee"));
    uint256 minted = _token.receiveWithFee(
        rewards,
        msg.sender,
        amount,
        receiver,
        feeRate
    );
    rewards += minted;

    emit RewardsFunded(address(_token), amount,
                       minted, block.timestamp);
}
```

a. He calls `fund()` with an infinitesimal amount (say `1 wei`). `fund` calls `receiveWithFee` which registers the initial `total = balanceOf(this)` and calls `token.safeTransferFrom`.

`TokenUtils::receiveWithFee()`

```
function receiveWithFee(...) internal returns (uint256) {
    uint256 total = token.balanceOf(address(this));
    uint256 fee;

    if (feeReceiver != address(0) &&
        feeRate > 0 && feeRate < 1e18) {
        fee = (amount * feeRate) / 1e18;
        token.safeTransferFrom(sender, feeReceiver, fee);
    }

    token.safeTransferFrom(sender, address(this), amount - fee);
    uint256 actual = token.balanceOf(address(this)) - total;

    uint256 minted = (total > 0)
        ? (shares * actual) / total
        : actual * INITIAL_SHARES_PER_TOKEN;
    require(minted > 0);
    return minted;
}
```

b. The latter passes control to the adversary (via a send hook), which makes a nested call to `fund`, again with `amount = 1 wei`. Which again leads to a new `token.safeTransferFrom`.

c. The process continues until the `k-th` call, which is now made with a larger `amount = N`. The adversary stops making nested calls so the previous calls finish their execution starting from the most nested one.

d. The last (`k-th`) call computes `actual` as the difference between the two balances which will be equal to `N` tokens. This causes rewards to be incremented by the corresponding amount of shares (= `(rewards * N) / total`).

e. Now execution returns to the `(k-1)-th` call, for which the actual transferred amount was just `1 wei`. However, the difference of balances includes the nested `k-th` call, so `actual` will be found to be `N` (not `1 wei`), causing rewards to be incremented again by the same amount of shares.

f. The same happens with all outer calls, causing rewards to be incremented by k times more shares than they should!

3. The previous step essentially devalued each reward share, since we printed k times more shares than we should have. Note that the `controller` can withdraw all funds except those corresponding to the shares in debt. But these now are worth less, so the adversary can withdraw more reward tokens than he should. By picking k to be as large as the stack allows, and a large value of `N` (possibly using a flash loan), the `controller` can drain almost all reward tokens from the pool, leaving users with no rewards.

Note that the other reward modules are also likely vulnerable since they all call `receiveWithFee` and have no reentrancy guard.

To prevent this vulnerability reentrancy guards should be added to all fund methods. Moreover, `TokenUtils::receiveAmount` could check that the actual transferred amount is no larger than the expected one. This check would still support tokens with transfer fees, but would catch attacks like the one reported here.

Resolution:
This vulnerability was fixed by addressing both issues that enabled it. Specifically:

- A check was added in `TokenUtils::receiveAmount` to ensure that the transferred amount is no larger than the expected one
- Reentrancy guards were added to the fund function

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	The use of the factory contracts is only enforced off-chain	**WON'T FIX**

The proper way to deploy a pool and its modules is via the factory contracts. These contracts ensure that the pool is initialized with proper values that prevent a potentially malicious controller from stealing the investor's funds. However, the use of factory contracts is only checked off-chain. `PoolFactory` keeps a list of contracts it created, and this list presumably is used by the GYSR UI to allow users to interact only with officially created contracts.

On the other hand, anyone could still create their own `Pool` contracts and manually initialize in any way. Such contracts would have identical source code as the legitimate ones, and it would be hard to recognize them. They would also be clearly unsafe: by using malicious staking and reward modules, or even a fake GYSR token, an adversary could easily steal all the funds deposited by investors. Although the off-chain checks would ensure that no user actually interacts with such contracts, such checks are inherently less reliable than on-chain ones.

It would be preferable to ensure that contracts with bytecode identical to the official ones can never be improperly initialized, for instance by allowing their constructor to be called by a factory contract.

Resolution:
This issue largely concerns off-chain aspects and cannot be fully addressed on-chain. As a consequence, it will be addressed by adding clear documentation explaining how to verify the validity of a deployed contract.

M2	Unstaking in `ERC20FixedRewardModule` is inconsistent under different use cases	**RESOLVED**

The `ERC20FixedRewardModule` was updated as part of the PR #38 mentioned in the ABSTRACT section.

The fundamental functions for the users are `stake`, `unstake` and `claim`. When a user stakes, the `pos.debt` field holds their potential rewards if they stake for the entire predefined `period`. However, a user can always claim their rewards for the amount already vested.

Here are two scenarios of the same logic that are treated differently:

- **`Case #1`**: The first case assumes that the users will not stake more than once. This happens when this reward module is combined with the `ERC20BondStaking` module since users can't stake twice with a bond. However, if they unstake early, for recovering the remaining principal, their rewards earning ratio should also be reduced. In order for the reward module to achieve this, it treats the user shares as if they were vesting all together. So, when user unstakes early only a percentage of all user shares have vested resulting in losing portion of the earning power as indented.

- **`Case #2`**: The second case is when users can stake more than once. This can happen when this module is combined with other staking modules like `ERC20StakingModule` for example. Then, when a user stakes again, the function calculates the rewards earned up to that point, updates their records and rolls over the remaining (unvested) amount with the newly added one to start vesting from that point forward. This approach treats the user shares as if they were vesting linearly and not all together which means that the user won't lose his earning power.

A detailed example illustrating the inconsistency between the 2 cases is provided in the `APPENDIX` of this report.

Resolution:
This issue was addressed by modifying the staking logic to remove the inconsistency.

LOW SEVERITY:

ID	Description	STATUS
L1	Approximation errors in `ERC20BondStakingModule`	**RESOLVED**

`ERC20BondStakingModule` needs to perform vesting and debt decay on multiple amounts which however have different vesting/decay periods. To perform this operation in `O(1)` an approximation method is used, where vesting/decay happens for the whole amount simultaneously, and the period is essentially restarted in every update. This method necessarily introduces an approximation error. If multiple updates happen the resulting values could be substantially lower than the actual ones.
What is particularly problematic is that such delays can be produced by events that do not add new value to the system. For instance, vesting a large amount could be substantially delayed by staking (maliciously or coincidentally) small amounts. With just 5 updates the amount vested at the end of the period will be only 67% of the total.

Note that there is also an "opposite extreme" strategy: instead of restarting the period on every update, we could choose to never restart until the current amount is fully vested. Of course, this method also introduces an error. If the newly deposited amounts are large, delaying them might introduce a larger error than restarting the period.

So we propose to follow a hybrid approach, alternating between the two extremes: keep a pending amount whose vesting has not started yet, and will start no later than at the end of the current period, but possibly earlier if it's preferable. When a new amount arrives, we will compute how much error will be introduced by starting a new vesting period, and how much error will be introduced if we delay the new amount, and we'll choose the approach of the smallest error.
This report is accompanied by a Jupyter notebook with a discussion of this method, a prototype implementation and some simulations.

The proposed method has the following properties:

- It needs `O(1)` time and is only marginally more complicated than the simple method.
- It is guaranteed to vest at least as much as the simple method, and never more than the maximum amount.
- In order to introduce vesting delays one needs to **add new funds** to the system, larger than the ones currently being vested.

Resolution:
This issue was addressed by an improved logic that resets the time period only on stake operations, improving the accuracy while simplifying the code.

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	`_beforeTokenTransfer()` is not correctly overridden in `ERC20BondStakingModule`	**RESOLVED**

The `ERC20BondStakingModule` contract overrides the `ERC721::_beforeTokenTransfer()` hook. However, the overridden hook hasn't the same signature as the original one causing the compilation to fail. The missing part is the 4th argument which should have been another `uint256`.

ERC721::_beforeTokenTransfer()

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256, /* firstTokenId */
    uint256 batchSize
) internal virtual {
    if (batchSize > 1) {
        if (from != address(0)) {
            _balances[from] -= batchSize;
        }
        if (to != address(0)) {
            _balances[to] += batchSize;
        }
    }
}
```

ERC20BondStakingModule::_beforeTokenTransfer()

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256 tokenId
) internal override {
    if (from != address(0)) _remove(from, tokenId);
    if (to != address(0)) _append(to, tokenId);
}
```

A2	Failing tests	RESOLVED

There are some cases in the test scripts that fail due to the grammar changes that OZ introduced at commit fbf235661e01e27275302302b86271a8ec136fea. They updated the revert messages of the `approve()`, `transferFrom()` and `safeTransferFrom()` functions from:

- “ERC721: caller is not token owner **nor** approved”

to:

- “ERC721: caller is not token owner **or** approved”

However, the tests haven’t been updated to reflect the new changes, so they fail. The affected tests are the following:

- **aquarium.js**
  - LoC:113 - *“when token transfer has not been approved”*
- **erc20bondstakingmodule.js**
  - LoC: 1680 - *“when user transfers a bond position they do not own”*
  - LoC: 1689 - *“when user safe transfers a bond position they do not own”*
  - LoC: 1699 - *“when user transfers a bond position that they already transferred”*

A3	Minor gas optimization	**RESOLVED**

Since the protocol tries to minimize the gas consumption to the minimum possible, we suggest here a minor optimization in `ERC20FixedRewardModule`.

The `pos.updated` value could be updated inside the if statement above instead of having to check again whether the period has ended or not.

ERC20FixedRewardModule::claim()

```
function claim(
    bytes32 account, address, address receiver, uint256, bytes calldata
) external override onlyOwner returns (uint256, uint256) {
    ...
    if (block.timestamp > end) {
        e = d;
    } else {
        uint256 last = pos.updated;
        e = (d * (block.timestamp - last)) / (end - last);
    }
    ...
    // Dedaub: This update could be transferred to the above if statement
    //         for avoiding rechecking whether the period has ended
    pos.updated = uint128(block.timestamp < end ? block.timestamp : end);
    ...
}
```

A4	Inconsistent comment in OwnerController	RESOLVED

The OwnerController contract provides functionality for the rest of the protocol contracts to manage their owners and their controllers.

However, while the comments of the transferOwnership() function state that the owner can renounce ownership by transferring to address(0), this is not possible with the current code as it reverts when the newOwner address is 0.

OwnerController::transferOwnership()

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * This can include renouncing ownership by transferring to the zero
 * address. Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual override {
    requireOwner();
    require(newOwner != address(0), "oc3");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

A5	Gas optimization in duplicate check	**RESOLVED**

In several functions there are nested loops to check that no duplicate arguments are given. For instance:

`ERC20MultiRewardModule::update()`

```
function update(
    bytes32 account,
    address,
    bytes calldata data
) external override {
    ...
    for (uint256 i; i < count; ++i) {
        // verify no duplicates
        for (uint256 j; j < i; ++j) {
            pos = 228 + 32 * j;
            address prev;
            assembly {
                prev := calldataload(pos)
            }
            require(addr != prev, "mrm20");
        }
    }
    ...
}
```

These checks can be performed more efficiently by requiring that arguments are always passed sorted to the function (at the cost of making the caller code a bit more complex).

A6	Compiler bugs	**INFO**

The code is compiled with Solidity `0.8.18`. Version `0.8.18`, at the time of writing, has no known bugs.

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.

APPENDIX

A detailed example for M2

ERC20FixedRewardModule::stake()

```
function stake(
    bytes32 account,
    address,
    uint256 shares,
    bytes calldata
) external override onlyOwner returns (uint256, uint256) {
    uint256 reward = (shares * rate) / 1e18;
    require(reward <= rewards - debt, "xrm3");

    Position storage pos = positions[account];
    uint256 d = pos.debt;
    if (d > 0) {
        ...
    }
    pos.debt = d + reward;
    pos.timestamp = uint128(block.timestamp);
    pos.updated = uint128(block.timestamp);

    debt += reward;
    return (0, 0);
}
```

ERC20FixedRewardModule::unstake()

```
function unstake(
    bytes32 account,
    address,
    address receiver,
    uint256 shares,
    bytes calldata
) external override onlyOwner returns (uint256, uint256) {
    Position storage pos = positions[account];
    require(pos.timestamp < block.timestamp);

    // unstake debt shares
    uint256 burned = (shares * rate) / 1e18;
    {
        uint256 vested = pos.vested; // burn vested shares first
        if (vested > burned) {
            pos.vested = vested - burned;
            burned = 0;
        } else if (vested > 0) {
            burned -= vested;
            pos.vested = 0;
        }
    }
    uint256 unvested;

    // get all pending rewards
    uint256 d = pos.debt;
    uint256 end = pos.timestamp + period;
    uint256 r = pos.earned;
    uint256 e;
    if (block.timestamp > end) {
        e = d;
    } else {
        uint256 last = pos.updated;
        e = (d * (block.timestamp - last)) / (end - last);
        // lost unvested reward shares
        unvested = (burned * (end - block.timestamp)) / period;
        if (d - e - unvested < 1e6) unvested = d - e; // dust
    }

    // update user position
    pos.debt = d - e - unvested;
    ...

    // reduce global debt
    if (unvested > 0) debt -= unvested;
    ...
}
```

ERC20FixedRewardModule::claim()

```
function claim(
    bytes32 account,
    address,
    address receiver,
    uint256,
    bytes calldata
) external override onlyOwner returns (uint256, uint256) {
    Position storage pos = positions[account];
    uint256 d = pos.debt;
    uint256 end = pos.timestamp + period;
    uint256 r = pos.earned;
    uint256 e;
    if (block.timestamp > end) {
        e = d;
    } else {
        uint256 last = pos.updated;
        e = (d * (block.timestamp - last)) / (end - last);
    }

    // update user position
    pos.debt = d - e;
    pos.earned = 0;
    pos.updated = uint128(block.timestamp < end ? block.timestamp : end);
    ...
}
```

In the following example we illustrate the inconsistency between the 2 cases of M2:

Staking Only Once

1. A user stakes for the first time an amount that corresponds to a number of X rewards.

Snapshot #1.1 - ***After 1st stake()***

period	P
block.timestamp	0
pos.debt	X
pos.vested	0
pos.timestamp	0
pos.updated	0

2. He then calls `claim()` when only half of the period has elapsed. This means that 50% of his initial stake is vested and he withdraws 50% of X.

Snapshot #1.2 - ***After claim()***

period	P
block.timestamp	P/2
pos.debt	X/2
pos.vested	0
pos.timestamp	0
pos.updated	P/2

3. Then, another 25% of the period passes and the user unstakes part of his stake. He calls `unstake` with an amount corresponding to the 75% of his initial stake.

    ○ What happens is that since the `pos.vested` hasn't been updated in the meantime when the `claim` happened, the `burned` will not get deducted by the amount of the already vested shares. This means that `pos.debt` will end up subtracting more than the actual amount which corresponds to the already vested shares resulting in losing earning power for the remaining shares which complies with the behavior associated with `ERC20BondStakingModule`.

Snapshot #1.3 - ***After unstake()***

period	P
block.timestamp	3P/4
(unstake) e	X/4
burned	**3X/4**
unvested	**3X/16**
pos.debt	**X/16**
pos.vested	**0**
pos.timestamp	0
pos.updated	3P/4

==============================

Staking Multiple Times

1. A user stakes for the first time an amount that corresponds to an amount of X rewards.

   Snapshot #2.1 - *After 1st stake()*

period	P
block.timestamp	0
pos.debt	X
pos.vested	0
pos.timestamp	0
pos.updated	0

2. He then calls `claim()` when only half of the period has elapsed. This means that 50% of his initial stake is vested and he withdraws 50% of X.

   Snapshot #2.2 - *After claim()*

period	P
block.timestamp	P/2
pos.debt	X/2
pos.vested	0
pos.timestamp	0
pos.updated	P/2

3. Then, another 25% of the period passes and the user wants to unstake part of his initial stake.

- This time, before calling `unstake` he stakes again with an infinitesimal amount (say `1 wei`). This results in updating `pos.vested` value to keep the already vested amount of shares like if they were vesting linearly and not all together.

  Snapshot #2.3 - ***After 2nd stake()***

period	P
block.timestamp	3P/4
(stake) e	3X/4
pos.debt	**~ X/4**
pos.vested	**3X/4**
pos.timestamp	3P/4
pos.updated	3P/4

- He then calls `unstake` with an amount corresponding to the 75% of his initial stake. Now, `pos.vested` has been updated and the `burned` will result in getting a `0` value. This means that `pos.debt` will end up subtracting less amount than the previous case which corresponds only the bahavior of having the shares vest linearly meaning that the user will not lose his earning power this time which may comply with bahaviors associated with other Staking Modules that allow multiple staking.

  Snapshot #2.4 - ***After unstake()***

period	P
block.timestamp	3P/4
(unstake) e	X/4
burned	**0**

unvested	**0**
pos.debt	**X/4**
pos.vested	**0**
pos.timestamp	3P/4
pos.updated	3P/4