Timeless Security Review

Auditors

Christoph Michel, Lead Security Researcher

JayJonah, Security Researcher

Calvin Boehr, Apprentice

Sleepy, Apprentice

Report prepared by: Pablo Misirov & Calvin Boehr

June 18, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Timeless is a yield tokenization protocol that offers Perpetual Yield Tokens (PYTs), which give their holders a perpetual right to claim the yield generated by the underlying principal. Timeless also offers Negative Yield Tokens (NYTs), a protocol-native way to short yield rates.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Timeless according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Timeless engaged with Spearbit to review Timeless Finance. In this period of time a total of 24 issues were found.

Summary

Project Name	Timeless
Repository	timeless-fi
Commit timeless Commit xPYT Commit swapper	018b1f47de6c95d... 4380721b47f946b... dcb20191edca0c2...
Type of Project	Perpetual Yield Tokens, DeFi
Audit Timeline	April 28 - May 12
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	2
Medium Risk	3
Low Risk	8
Gas Optimizations	0
Informational	10
Total Issues	24

5 Findings

5.1 Critical Risk

5.1.1 Mint `PerpetualYieldTokens` for free by self-transfer

Severity: *Critical Risk*

Context: PerpetualYieldToken.sol#L53

Description: The `PYT.transfer` and `transferFrom` functions operate on cached balance values. When transferring tokens to oneself the decreased balance is overwritten by an increased balance which makes it possible to mint PYT tokens for free.

Consider the following exploit scenario:

- Attacker `A` self-transfers by calling `token.transfer(A, token.balanceOf(A))`.
- `balanceOf[msg.sender]` is first set to zero but then overwritten by `balanceOf[to] = toBalance + amount`, doubling `A`'s balance.

Recommendation: Fix the issue in `transfer` and `transferFrom` by operating on the latest storage balances instead of cached values.

Timeless: Would checking for self-transfers and doing an early return be the best way to solve it?

Spearbit: It would be best regarding gas efficiency nevertheless it should still trigger a `gate.beforePerpetualYieldTokenTransfer` call once to accrue the yield because the user would expect any transfer to accrue yields for `from` and `to`, and maybe someone is reliant on this. Additionally, it should also trigger the `Transfer` event for ERC20 compliance.

Timeless: Not sure about triggering `gate.beforePerpetualYieldTokenTransfer` during self transfers, seems like a niche use case.

Spearbit: Then its behavior is inconsistent. Because self-transfers are a niche use case anyway, might as well do the additional call to make it consistent. It would not increase gas cost per execution for non-self-transfer calls as you need the `if` branch + return anyway.

Timeless: Implemented in PR #4.

Spearbit: The bug still exists in `transferFrom` diff PR #4.

You're checking `msg.sender != to` but it should be `from != to` in this case - you always want to check the balance owners. The test should be with a `spender` different from `from` and all three parties are `tester`.

Timeless: Nice catch, fixed in this commit.

Spearbit: Acknowledged.

5.2 High Risk

5.2.1 xPYT auto-compound does not take pounder reward into account

Severity: *High Risk*

Context: xPYT.sol#L179

Description: Conceptually, the `xPYT.pound` function performs the following steps:

1. Claims `yieldAmount` yield for itself, deposits the yield back to receive more PYT/NYT (`Gate.claimYieldEnter`).
2. Buys xPYT with the NYT.
3. Performs a `ERC4626.redeem(xPYT)` with the bought amount, burning `xPYT` and receiving `pytAmountRedeemed` PYT.
4. Performs a `ERC4626.deposit(pytAmountRedeemed + yieldAmount = pytCompounded)`.
5. Pays out a reward in PYT to the caller.

The `assetBalance` is correctly updated for the first four steps but does not decrease by the pounder reward which is transferred out in the last step.

The impact is that the contract has a smaller `assets` (PYT) balance than what is tracked in `assetBalance`.

1. Future depositors will have to make up for it as `sweep` computes the difference between these two values.
2. The xPYT exchange ratio is wrongly updated and withdrawers can redeem xPYT for more assets than they should until the last withdrawer is left holding valueless xPYT.

Consider the following example and assume 100% fees for simplicity i.e. `pounderReward = pytCompounded`.

- Vault total: 1k assets, 1k shares total supply.
- `pound` with 100% fee:
  - claims `Y` PYT/NYT.
  - swaps `Y` NYT to `X` xPYT.
  - redeems `X` xPYT for `X` PYT by burning `X` xPYT (`supply -= X`, exchange ratio is 1-to-1 in example).
  - `assetBalance` is increased by claimed `Y` PYT
  - pounder receives a pounder reward of `X + Y` PYT but does not decrease `assetBalance` by pounder reward `X+Y`.
- Vault totals should be `1k-X` assets, `1k-X` shares, keeping the same share price.
- Nevertheless, vault totals actually are `1k+Y` assets, `1k-X` shares. Although pounder receives 100% of pounding rewards the xPYT price (`assets / shares`) increased.

Recommendation: The `assetBalance` should also decrease by the `pounderReward`.

```
- unchecked {
-   assetBalance += yieldAmount;
- }
+ // using unchecked should still be fine? as pounderReward <= yieldAmount + pytAmountRedeemed. and
↪  pytAmountRedeemed must have already been in the contract because of the implicit `redeem`, i.e.,
↪  assetBalance >= pytAmountRedeemed
+ assetBalance = assetBalance + yieldAmount - pounderReward;
```

Consider adding a test that verifies correct `assetBalance` updates.

Timeless: Implemented in PR #2.

Spearbit: Acknowledged.

5.2.2 Wrong yield accumulation in `claimYieldAndEnter`

Severity: *High Risk*

Context: Gate.sol#L590

Description: The `claimYieldAndEnter` function does not accrue yield to the `Gate` contract itself (`this`) in case `xPYT` was specified. The idea is to accrue yield for the mint recipient first before increasing/reducing their balance to not interfere with the yield rewards computation. However, in case `xPYT` is used, tokens are minted to the `Gate` before its yield is accrued.

Currently, the transfer from `this` to `xPYT` through the `xPYT.deposit` call accrues yield for `this` *after* the tokens have been minted to it (`userPYTBalance * (updatedYieldPerToken - actualUserYieldPerToken) / PRECISION`) and its balance increased. This leads to it receiving a larger yield amount than it should have.

Recommendation: Accrue yield to the address receiving the minted tokens.

```
// accrue yield to recipient
// no need to do it if the recipient is msg.sender, since
// we already accrued yield in _claimYield
- if (pytRecipient != msg.sender) {
+ if (address(xPYT) != address(0) || pytRecipient != msg.sender)
    _accrueYield(
        vault,
        pyt,
-       pytRecipient,
+       address(xPYT) == address(0) ? pytRecipient : address(this),
        updatedPricePerVaultShare
    );
}

// mint NYTs and PYTs
yieldTokenTotalSupply[vault] += yieldAmount;
nyt.gateMint(nytRecipient, yieldAmount);
if (address(xPYT) == address(0)) {
    // mint raw PYT to recipient
    pyt.gateMint(pytRecipient, yieldAmount);
} else {
    // mint PYT and wrap in xPYT
    pyt.gateMint(address(this), yieldAmount);
    if (pyt.allowance(address(this), address(xPYT)) < yieldAmount) {
        // set PYT approval
        pyt.approve(address(xPYT), type(uint256).max);
    }
    xPYT.deposit(yieldAmount, pytRecipient);
}
```

Timeless: Yes, if we use `sweep` below we can accrue yield in the same way as in `_enter`. Fix implemented in PR #5.

Spearbit: Acknowledged.

5.3 Medium Risk

Severity: *Medium Risk*

5.3.1 Swapper left-over token balances can be stolen

Context: Swapper.sol#L133, UniswapV3Swapper.sol#L187

Description: The `Swapper` contract may never have any left-over token balances after performing a swap because token balances can be stolen by anyone in several ways:

- By using Swapper.doZeroExSwap with `useSwapperBalance` and `tokenOut = tokenToSteal`
- Arbitrary token approvals to arbitrary spenders can be set on behalf of the `Swapper` contract using UniswapV3Swapper.swapUnderlyingToXpyt.

Recommendation: All transactions must atomically move all tokens in and out of the contract when performing swaps to not leave any left-over token balances or be susceptible to front-running attacks.

Timeless: Acknowledged, this is the intended way to use Swapper, it should not hold any tokens before and after a transaction.

5.3.2 `TickMath` might revert in solidity version `0.8`

Severity: *Medium Risk*

Context: TickMath.sol#L2

Description: UniswapV3's `TickMath` library was changed to allow compilations for solidity version 0.8. However, adjustments to account for the implicit overflow behavior that the contract relies upon were not performed. The UniswapV3xPYT.sol is compiled with version `0.8` and indirectly uses this library through the `OracleLibrary`. In the worst case, it could be that the library always reverts (instead of overflowing as in previous versions), leading to a broken xPYT contract.

> The same `pragma solidity >=0.5.0;` instead of `pragma solidity >=0.5.0 <0.8.0;` adjustments have been made for the OracleLibrary and PoolAddress contracts. However, their code does not rely on implicit overflow behavior.

Recommendation: Follow the implementation of the official TickMath 0.8 branch which uses `unchecked` blocks for every function. Consider using the official Uniswap files with two different versions of this file, one for solidity versions `<0.8` and one for `0.8` from the 0.8 branch.

Timeless: Implemented in PR #3.

Spearbit: Acknowledged.

5.3.3 Rounding issues when exiting a vault through shares

Severity: *Medium Risk*

Context: Gate.sol#L383

Description: When exiting a vault through `Gate.exitToVaultShares` the user specifies a `vaultSharesAmount`. The amount of PYT&NYT to burn is determined by a `burnAmount = _vaultSharesAmountToUnderlying-Amount(vaultSharesAmount)` call. All implementations of this function in derived `YearnGate` and `ERC4626` contract's round down the `burnAmount`. This means one needs to burn fewer amounts than the value of the received vault shares.

This attack can be profitable and lead to all vault shares being stolen If the gas costs of this attack are low. This can be the case with vault & underlying tokens with a low number of decimals, highly valuable shares, or cheap gas costs.

Consider the following scenario:

- Imagine the following vault assets: `totalAssets` = 1.9M, `supply` = 1M. Therefore, 1 share is theoretically worth 1.9 underlying.
- Call `enterWithUnderlying(underlyingAmount = 1900)` to mint 1900 PYT/NYT (and the gate receives `1900 * supply / totalAssets = 1000` vault shares).
- Call `exitToVaultShares(vaultSharesAmount = 1)`, then `burnAmount = shares.mulDivDown(totalAssets(), supply) = 1 * totalAssets / supply = 1`. This burns 1 "underlying" (actually PYT/NYT but they are 1-to-1), but receive 1 vault share (worth 1.9 underlying). Repeat this for up to the minted 1900 PYT/NYT.
- Can redeem the `1900` vault shares for `3610` underlying directly at the vault, making a profit of `3610 - 1900 = 1710` underlying.

Recommendation: The `_vaultSharesAmountToUnderlyingAmount` function should be replaced by a `_vaultSharesAmountToUnderlyingAmountUp` function which rounds up to avoid users profiting from receiving more value in vault shares than they burn in underlying.

Timeless: Implemented in PR #5.

Spearbit: Acknowledged.

5.4 Low Risk

5.4.1 Possible outstanding allowances from Gate

Severity: *Low Risk*

Context: Gate.sol#L216

Description: The `vault` parameter of `Gate.enterWithUnderlying` can be chosen by an attacker in such a way that `underlying = vault.asset()` is another vault token of the Gate itself. The subsequent `_depositIntoVault(underlying, underlyingAmount, vault)` call will approve `underlyingAmount` of `underlying` tokens to the provided vault and could in theory allow stealing from other vault shares.

This is currently only exploitable in very rare cases because the caller also has to transfer the `underlyingAmount` to the gate contract first. For example, when transferring `underlyingAmount = type(uint256).max` is possible due to flashloans/flashmints and the vault shares implement approvals in a way that do not decrease anymore if the allowance is `type(uint256).max`, as is the case with ERC4626 vaults.

Recommendation: As a best practice, consider resetting the approvals to zero after the `vault.deposit` call (as it is assumed to consume the allowance) to make sure that after the transaction ran, there are never any outstanding approvals on arbitrary token contracts from the gate to arbitrary spenders. This mitigates other unknown attack vectors.

Timeless: Implemented in PR #8.

Spearbit: Acknowledged.

5.4.2 `Factory.sol` owner can change fees unexpectedly

Severity: *Low Risk*

Context: Factory.sol#L141

Description: The `Factory.sol` owner may be able to front run yield calculations in a gate implementation and change user fees unexpectedly.

Recommendation: Put a time lock in place for any fee changes made by the factory owner.

Timeless: Acknowledged, we're fine with this as the Factory is planned to be owned by a Governor contract that already has built in timelock mechanics.

5.4.3 Low `uniswapV3TwapSecondsAgo` may result in AMM manipulation in `pound()`

Severity: *Low Risk*

Context: UniswapV3xPYT.sol#L98

Description: The lower the value of `uniswapV3TwapSecondsAgo` is set with at construction creation time the easier it becomes for an attacker to manipulate the results of the `pound()` function. It becomes easier for attackers to manipulate automated market maker price feeds with a lower time horizon, requiring less capital to manipulate prices, although users may simply not use an xPYT contract that sets `uniswapV3TwapSecondsAgo` too low.

Recommendation: Add a lower bound for `uniswapV3TwapSecondsAgo` in the constructor.

Timeless: We're fine with this, since xPYT creation is permissionless users will just choose xPYTs with security parameters they're comfortable with.

5.4.4 `UniswapV3Swapper` uses wrong allowance check

Severity: *Low Risk*

Context: UniswapV3Swapper.sol#L282, UniswapV3Swapper.sol#L373

Description: Before the `UniswapV3Swapper` can exit a gate, it needs to set an XPYT allowance to the gate. The following check determines if an approval needs to be set:

```
if (
    args.xPYT.allowance(address(this), address(args.gate)) <
    tokenAmountOut
) {
    args.xPYT.safeApprove(address(args.gate), type(uint256).max);
}
args.gate.exitToUnderlying(
    args.recipient,
    args.vault,
    args.xPYT,
    tokenAmountOut
);
```

The `tokenAmountOut` is in an underlying token amount but is compared against an `xPYT` shares amount. A legitimate `gate.exitToUnderlying` call will call `xPYT.withdraw(tokenAmountOut, address(gate), address(swapper))` checks `allowance[swapper][gate] >= previewWithdraw(tokenAmountOut)`.

Recommendation: In practice the actual value does not matter as there is either no approval set, or an infinite approval. We still recommend replacing the incorrect code with a comparison against the xPYT-converted `tokenAmountOut` for correctness:

```
- if (args.xPYT.allowance(address(this), address(args.gate)) < tokenAmountOut)
+ if (args.xPYT.allowance(address(this), address(args.gate)) <
↪   args.xPYT.previewWithdraw(tokenAmountOut))
```

Timeless: Implemented in PR #2.

Spearbit: Acknowledged.

5.4.5 Missing check that `tokenIn` and `tokenOut` are different

Severity: *Low Risk*

Context: Swapper.sol#L133

Description: The `doZeroExSwap()` function takes in two ERC20 addresses which are `tokenIn` and `tokenOut`. The problem is that the `doZeroExSwap()` function does not check if the two token addresses are different from one another. Adding this check can reduce possible attack vectors.

Recommendation: Consider implementing the following check.

```
require(tokenIn != tokenOut, "Duplicate tokens");
```

Timeless: Implemented in PR #1.

Spearbit: Acknowledged.

5.4.6 `Gate.sol` gives unlimitted ERC20 approval on `pyt` for arbitrary address

Severity: *Low Risk*

Context: Gate.sol#L675

```
if (address(xPYT) == address(0)) {
    // mint raw PYT to recipient
    pyt.gateMint(pytRecipient, yieldAmount);
} else {
    // mint PYT and wrap in xPYT
    pyt.gateMint(address(this), yieldAmount);
    if (pyt.allowance(address(this), address(xPYT)) < yieldAmount) {
        // set PYT approval
        pyt.approve(address(xPYT), type(uint256).max);
    }
    xPYT.deposit(yieldAmount, pytRecipient);
}
```

Description: A malicious contract may be passed into the `claimYieldAndEnter()` function as `xPYT` and given full control over any `PYT` the contract may ever hold. Even though `PYT` is validated to be a real `PYT` contract and the `Gate.sol` contract isn't expected to have any `PYT` in it, it would be safer to remove any unnecessary approvals.

Recommendation: Avoid setting any approvals at all by using `gateMint` & `sweep` as `_enter` does.

Timeless: Forgot to use sweep for this part. Implemented in PR #5.

Spearbit: Acknowledged.

5.4.7 Constructor function does not check for zero address

Severity: *Low Risk*

Context: UniswapV3Juggler.sol#L81-L84

Description: The constructor function does not check if the addresses passed in are zero addresses. This check can guard against errors during deployment of the contract.

Recommendation: Require checks should be added to ensure that the addresses passed into the constructor function are not zero addresses.

```
constructor(address factory_, IQuoter quoter_) {
    require(factory_ != address(0), "Zero address");
    require(quoter_ != address(0), "Zero address");
    factory = factory_;
    quoter = quoter_;
}
```

Also see:

- xPYTFactory.sol#L20-L23
- UniswapV3xPYT.sol#L82-L83
- UniswapV3Swapper.sol#L70-L74
- Swapper.sol#L76-L78
- Factory.sol#L52
- Gate.sol#L158-L160
- NegativeYieldToken.sol#L15
- PerpetualYieldToken.sol#L15

Timeless: Acknowledged, we're fine with this.

5.4.8 Accruing yield to `msg.sender` is not required when minting to xPYT contract

Severity: *Low Risk*

Context: Gate.sol#L1009

Description: The `_exit` function always accrues yield to the `msg.sender` before burning new tokens. The idea is to accrue yield for the recipient first before increasing/reducing their balance to not interfere with the yield rewards computation. However, in case `xPYT` is used, tokens are burned on the `Gate` and not `msg.sender`.

Recommendation: For correctness & potential gas efficiency reasons only accrue the yield of the account whose tokens are being burned. That's `_accrueYield(msg.sender)` in case of `address(xPYT) == address(0)`, and `this` otherwise.

```
// accrue yield
- _accrueYield(vault, pyt, msg.sender, updatedPricePerVaultShare);
+ _accrueYield(vault, pyt, address(xPYT) == address(0) ? msg.sender : address(this),
↪  updatedPricePerVaultShare);

// burn NYTs and PYTs
unchecked {
    // Cannot underflow because a user's balance
    // will never be larger than the total supply.
    yieldTokenTotalSupply[vault] -= underlyingAmount;
}
nyt.gateBurn(msg.sender, underlyingAmount);
if (address(xPYT) == address(0)) {
    // burn raw PYT from sender
    pyt.gateBurn(msg.sender, underlyingAmount);
} else {
    /// -----------------------------------------------------------------------
    /// Effects
    /// -----------------------------------------------------------------------

    // convert xPYT to PYT then burn
    xPYT.withdraw(underlyingAmount, address(this), msg.sender);
    pyt.gateBurn(address(this), underlyingAmount);
}
```

Timeless: `_accrueYield` needs to be called before `yieldTokenTotalSupply[vault] -= underlyingAmount;`, so can wrap it in a separate `(address(xPYT) == address(0))` branch.

This call accrues yield to `msg.sender`, not `this`, but I guess when the sender's using xPYT we don't need to accrue yield to the sender since the sender's PYT balance hasn't changed

Fix Implemented in PR #7.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Unlocked solidity pragmas

Severity: *Informational*

Context: Present in most files.

Description: Most of the implementation code uses a solidity pragma of `^0.8.4`. It is particularly the library contracts that use different functions. Unlocked solidity pragmas can result in unexpected behaviors or errors with different compiler versions.

Recommendation: Increase compiler version for the affected contracts and lock it. This has the added benefit of more free safety checks and optimizations by done the compiler. Note: Verify that changing the compiler does not break anything.

Timeless: Implemented in PR #10.

Spearbit: Acknowledged.

5.5.2 No safeCast in UniswapV3Swapper's `_swap`.

Severity: *Informational*

Context: UniswapV3Swapper.sol#L475

Description: It should be noted that solidity version `^0.8.0` doesn't revert on overflow when type-casting. For example, if you tried casting the value `129` from `uint8` to `int8`, it would overflow to `-127` instead. This is because signed integers have a lower positive integer range compared to unsigned integers i.e -128 to 127 for `int8` versus 0 to 255 for `uint8`.

Recommendation: It is highly unlikely that this could become a problem in the mentioned context, however, we still recommend using SafeCastLib for this.

Timeless: Implemented in PR #3.

Spearbit: Acknowledged.

5.5.3 One step critical address change

Severity: *Informational*

Context: Ownable.sol#L37-40

Description: Setting the owner in `Ownable` is a one-step transaction. This situation enables the scenario of contract functionality becoming inaccessible or making it so a malicious address that was accidentally set as `owner` could compromise the system.

Recommendation: Consider making the change of `owner` in the contracts a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) claims the elevation of privileges.

Timeless: Implemented in PR #11.

Spearbit: Acknowledged.

5.5.4 Missing zero address checks in `transfer` and `transferFrom` functions.

Severity: *Informational*

Context: ERC20.sol#84-100

Description: The codebase uses solmate's ERC-20 implementation. It should be noted that this library sacrifices user safety for gas optimization. As a result, their ERC-20 implementation doesn't include zero address checks on `transfer` and `transferFrom` functions.

Recommendation: Consider modifying `ERC20.sol` to include the missing checks. Alternatively, make it very clear to users that these controls aren't in place, and transferring their tokens to the zero address will effectively burn them.

Timeless: Acknowledged, we're fine with this.

5.5.5 Should add `indexed` keyword to deployed `xPYT` event

Severity: *Informational*

Context: xPYTFactory.sol#L15

Description: The `DeployXPYT` event only has the ERC20 `asset_` marked as indexed while xPYT `deployed` can also have the indexed key word since you can use up to three per event and it will make it easier for bots to interact off chain with the protocol.

Recommendation:

```
-    event DeployXPYT(ERC20 indexed asset_, xPYT deployed);
+    event DeployXPYT(ERC20 indexed asset_, xPYT indexed deployed);
```

Timeless: Acknowledged, we're fine with this

5.5.6 Missing check that `tokenAmountIn` is larger than zero

Severity: *Informational*

Context: Swapper.sol#L135

Description: In `doZeroExSwap()` there is no check that the `tokenAmountIn` number is larger than zero. Adding this check can add more thorough validation within the function.

Recommendation: Consider implementing the code snippet below.

```
require(tokenAmountIn > 0, "Cannot be zero");
```

Also, see UniswapV3xPYT.sol#L149.

Timeless: Acknowledged, we're fine with this

5.5.7 `ERC20` does not emit `Approval` event in `transferFrom`

Severity: *Informational*

Context: ERC20.sol#L110

Description: The `ERC20` contract does not emit new `Approval` events with the updated allowance in `transferFrom`. This makes it impossible to track approvals solely by looking at `Approval` events.

Recommendation: Consider adding `Approval` events to the `transferFrom` function.

Timeless: Why would you track approvals using events instead of just calling `allowance()`?

Spearbit: Events can be indexed off-chain once and put into a database which is then queried for performance vs always querying an RPC node for the current allowance. As `allowance` is a mapping, you don't know the keys if you want to list all approvals and therefore cannot call `allowance()`. But one could also argue if you really wanted to index approvals, you can also get this data some other way without events using *the graph* etc. It's a difference between solmate and OpenZeppelin's ERC20 implementations but ERC20 does indeed not require it.

Timeless: Acknowledged, we're fine with this.

5.5.8 Use the official UniswapV3 0.8 branch

Severity: *Informational*

Context: FullMath.sol **Description**: The current repositories create local copies of UniswapV3's codebase and manually migrate the contracts to Solidity 0.8.

- For FullMath.sol this also leads to some small gas optimizations in this LOC as it uses `0` instead of `type(uint256).max + 1`.

Recommendation: Consider using the official Uniswap V3 0.8 branch to make it easier to spot differences in the original code.

Timeless: Implemented in PR #9.

Spearbit: Acknowledged. The code has been modified to match the official library. The official library has not been added as a dependency.

5.5.9 No checks that provided xPYT matches PYT of the provided vault

Severity: *Informational*

Context: Gate.sol#L180-L181

Description: The `Gate` contracts has many functions that allow specifying `vault` and a `xPYT` addresses as parameter. The underlying of the `xPYT` address is assumed to be the same as the vault's PYT but this check is not enforced. Users that call the Gate functions with an xPYT contract for the wrong vault could see their deposit/withdrawals lost.

Recommendation: Consider adding a check to the functions in `Gate` as a safety check.

```
require(address(xPYT) == address(0) || xPYT.asset() == getPerpetualYieldTokenForVault(vault), "xPYT is
↪  for a different PYT than vault");
```

Timeless: Acknowledged, won't fix since we expect users to make this check offchain and not making the check saves gas.

5.5.10 Protocol does not work with non-standard ERC20 tokens

Severity: *Informational*

Context: Gate.sol#L216

Description: Some ERC20 tokens make modifications to their ERC20's `transfer` or `balanceOf` functions. One kind include deflationary tokens that charge certain fee for every `transfer` or `transferFrom`. Others are rebasing tokens that increase in balance over time.

Using these tokens in the protocol can lead to issues such as:

- Entering a vault through the Gate will not work as it tries to deposit the pre-fee amount instead of the received post-fee amount.
- The UniswapV3Swapper tries to enter a vault with the pre-fee transfer amount.

Recommendation: Clarify if fee-on-transfer tokens and other non-standard ERC20 tokens should be supported.

Timeless: We don't need to support fee-on-transfer tokens.