SMART CONTRACT AUDIT REPORT

for

Solidly Protocol

Prepared By: Yiqun Chen

PeckShield

January 30, 2022

Document Properties

Client	Solidly Protocol
Title	Smart Contract Audit Report
Target	Solidly
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Liu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 30, 2022	Xuxian Jiang	Final Release
1.0-rc	January 28, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Solidly` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Solidly

`Solidly` is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. The protocol features a unique `AMM`, which is compatible with all the standard features as popularized by `UniswapV2` with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps, as well as a built-in `NFT`-based voting mechanism and associated token emissions. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Solidly

Item	Description
Name	Solidly Protocol
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/andrecronje/solidly.git (4d34f83)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/andrecronje/solidly.git (8d0ef5a)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Solidly` protocol smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	5	■■■■■
Informational	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 5 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic of BaseV1-core::current()/burn()	Coding Practices	Fixed
PVE-002	Informational	Improved ERC20-Compliance of BaseV1 Token Contracts	Coding Practices	Fixed
PVE-003	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Fixed
PVE-004	Low	Gas Optimization in BaseV1-gauge::deposit()/withdraw()	Coding Practices	Fixed
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logics	Confirmed
PVE-006	Low	Fork-Resistant Domain Separator in BaseV1Pair	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic of BaseV1-core::current()/burn()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseV1-core`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Solidly` protocol is compatible with all the standard features as popularized by `UniswapV2` and further extends with a number of unique features. In the following, we analyze the self-contained price oracle feature without any upkeeps.

In particular, we show below the `current()` routine, which is designed to return the current twap price measured from the given `amountIn * tokenIn`. The logic is implemented correctly but can be improved to eliminate one redundant adjustment on the `timeElapsed` variable. In particular, the current yield of `timeElapsed` (line 428) guarantees its non-zero value, which makes the adjustment at line 429 redundant.

```
420     // gives the current twap price measured from amountIn * tokenIn gives amountOut
421     function current(address tokenIn, uint amountIn) external view returns (uint
            amountOut) {
422         Observation memory _observation = lastObservation();
423         (uint price0Cumulative, uint price1Cumulative,) = currentCumulativePrices();
424         if (block.timestamp == _observation.timestamp) {
425             _observation = observations[observations.length-2];
426         }
427
428         uint timeElapsed = block.timestamp - _observation.timestamp;
429         timeElapsed = timeElapsed == 0 ? 1 : timeElapsed;
430         if (token0 == tokenIn) {
431             return computeAmountOut(_observation.price0Cumulative, price0Cumulative,
                    timeElapsed, amountIn);
432         } else {
```

```
            return computeAmountOut(_observation.price1Cumulative, price1Cumulative,
                timeElapsed, amountIn);
        }
    }
```

Listing 3.1: BaseV1-core::current()

A similar situation also occurs in the burn() function from the same contract. The internal variables of amount0/amount1 are guaranteed to be larger than 0, which therefore makes their positive checks unnecessary!

Recommendation Revise current execution logic of above mentioned functions to remove unnecessary validations or checks.

Status This issue has been fixed in the following commits: 723c5cd and e8e130f.

3.2 Improved ERC20-Compliance of BaseV1 Token Contracts

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BaseV1
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the Solidly protocol is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. And the protocol has its own governance token BaseV1. And In the following, we examine the ERC20 compliance of the BaseV1 token contract.

Specifically, the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue. Specifically, the current implementation has defined the decimals state with the uint256 type. The ERC20 specification indicates the type of uint8 for the decimals state. Note that this incompatibility issue does not necessarily affect the functionality of BaseV1 in any negative way.

In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 spec-

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

ification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `BaseV1` implementation to ensure its ERC20-compliance.

Status This issue has been fixed in the following commit: `6cb64e6`.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	—
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	—
	Reverts while transferring to zero address	—
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	—
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.3 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseV1Router01`
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [3]

Description

In the `Solidly` protocol, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `BaseV1Router01::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        bool stable,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        address _pair = IBaseV1Factory(factory).getPair(tokenA, tokenB, stable);
        if (_pair == address(0)) {
            _pair = IBaseV1Factory(factory).createPair(tokenA, tokenB, stable);
        }
```

```
        (uint reserveA, uint reserveB) = getReserves(tokenA, tokenB, stable);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'BaseV1Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'BaseV1Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        bool stable,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external ensure(deadline) returns (uint amountA, uint amountB, uint liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, stable, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = pairFor(tokenA, tokenB, stable);
        _safeTransferFrom(tokenA, msg.sender, pair, amountA);
        _safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IBaseV1Pair(pair).mint(to);
    }
```

Listing 3.2: `BaseV1Router01::addLiquidity()`

It comes to our attention that the `BaseV1Router01` contract has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two sets of arguments: `amountADesired/amountBDesired` and `amountAMin/amountBMin`. The first set `amountADesired/amountBDesired` determines the desired amount for adding liquidity to the pool and the second set `amountAMin/amountBMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for certain trades on `BaseV1Router01` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `_addLiquidity()` function.

Status The issue has been fixed by adding the suggested requirement as shown in the following commit: `372a2cd`.

3.4 Gas Optimization in BaseV1-gauge::deposit()/withdraw()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseV1-gauge`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Solidly` protocol also features a built-in voting mechanism that tokenizes the lock positions. By doing so, the feature allows a single address to own more than one lock, and lock balances are cumulative and each lock contributes to the overall balance. While examining the current deposit logic, we notice the current implementation can be improved for gas efficiency.

To elaborate, we show below the related implementation of the `deposit()` function. As the name indicates, this function allows users to deposit their assets into the `gauge` for rewards. It comes to our attention that the checkpoint support reads the derived balance from the storage `derivedBalances[msg.sender]` (line 399) while the storage content is already available at the local variable `_derivedBalance` (line 393).

```
387     function deposit(uint amount, uint tokenId) public lock {
388         tokenIds[msg.sender] = tokenId;
389         _safeTransferFrom(stake, msg.sender, address(this), amount);
390         totalSupply += amount;
391         balanceOf[msg.sender] += amount;

393         uint _derivedBalance = derivedBalances[msg.sender];
394         derivedSupply -= _derivedBalance;
395         _derivedBalance = derivedBalance(msg.sender);
396         derivedBalances[msg.sender] = _derivedBalance;
397         derivedSupply += _derivedBalance;

399         _writeCheckpoint(msg.sender, derivedBalances[msg.sender]);
400         _writeSupplyCheckpoint();
401     }
```

Listing 3.3: `BaseV1-gauge::deposit()`

A similar optimization is also applicable to other routines, including `getReward()` in `BaseV1-gauge` and `BaseV1-voter` contracts.

Recommendation Avoid unnecessary storage reads in the above routines for gas efficiency.

Status The issue has been fixed in the following commits: `c93dc74` and `5cc90fc`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses‘
    //  allowance to zero by calling ‘approve(_spender, 0)‘ if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
```

```
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfe()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In the following, we show the `update_period()` routine from the `BaseV1Minter` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transfer(to, amount)` (line 121) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
107     function update_period() external returns (uint) {
108         uint _period = active_period;
109         if (block.timestamp >= _period + week) { // only trigger if new week
110             _period = block.timestamp / week * week;
111             active_period = _period;
112             weekly = weekly_emission();
113
114             uint _growth = calculate_growth(weekly);
115             uint _required = _growth + weekly;
116             uint _balanceOf = _token.balanceOf(address(this));
117             if (_balanceOf < _required) {
```

```
                _token.mint(address(this), _required-_balanceOf);
            }

            require(_token.transfer(address(_ve_dist), _growth));
            _ve_dist.checkpoint_token(); // checkpoint token balance that was just
                minted in ve_dist
            _ve_dist.checkpoint_total_supply(); // checkpoint supply

            _token.approve(address(_voter), weekly);
            _voter.notifyRewardAmount(weekly);

            emit Mint(msg.sender, weekly, circulating_supply(), circulating_emission());
        }
        return _period;
    }
```

Listing 3.6: `BaseV1Minter::update_period()`

Note this issue is also applicable to other routines in `BaseV1-minter` and `BaseV1-voter` contracts. For the `safeApprove()` support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been confirmed and the team clarifies that the supported tokens are expected to have the full ERC20-compliance.

3.6 Fork-Resistant Domain Separator in BaseV1Pair

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `BaseV1Pair`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Solidly` protocol token as well as the related pool tokens strictly follows the widely-accepted ERC20 specification (Section 3.2). In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `constructor()` function (lines 236-244).

```
234     constructor() {
```

```
        uint chainId = block.chainid;
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)'),
                keccak256(bytes(name)),
                keccak256(bytes('1')),
                chainId,
                address(this)
            )
        );

        (address _token0, address _token1, bool _stable) = BaseV1Factory(msg.sender).getInitializable();
        (token0, token1, stable) = (_token0, _token1, _stable);
        fees = address(new BaseV1Fees(_token0, _token1));
        ...
    }
```

Listing 3.7: `BaseV1Pair::constructor()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external {
        require(deadline >= block.timestamp, 'BaseV1: EXPIRED');
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'BaseV1: INVALID_SIGNATURE');
        allowance[owner][spender] = value;

        emit Approval(owner, spender, value);
    }
```

Listing 3.8: `BaseV1Pair::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status The issue has been fixed in the following commit: 3e3de10.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Solidly` protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. The protocol features a unique `AMM`, which is compatible with all the standard features as popularized by `UniswapV2` with a number of novel improvements. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.