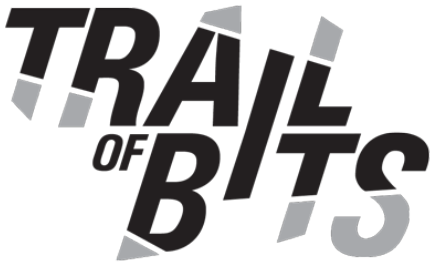

Dharma Labs Smart Wallet

Security Assessment

October 15, 2019

Prepared For:
Nadav Hollander | *Dharma Labs*
nadav@dharma.io

0age | *Dharma Labs*
0age@dharma.io

Prepared By:
Eric Rafaloff | *Trail of Bits*
eric.rafaloff@trailofbits.com

Dominik Czarnota | *Trail of Bits*
dominik.czarnota@trailofbits.com

Executive Summary

From September 30 through October 11, 2019, Dharma Labs hired Trail of Bits to review the security of its smart wallet. Trail of Bits conducted this assessment over the course of four person-weeks, with two engineers working from commit `b1d510d` from the dharma-smart-wallet repository. Specific source code files listed in Coverage were deemed in scope by Dharma Labs and were reviewed.

During the first week, Trail of Bits read relevant documentation and became familiar with the Solidity smart contracts. The Slither static analyzer was run on the codebase and all of its output was triaged. This was followed by manual code review, which identified issues specific to the smart wallet and its security requirements. Manual review continued during the second and last week of the assessment. In addition, Trail of Bits ran Slither's upgradability checker to ensure that contract upgrades can be safely performed (see Appendix C).

A total of 17 issues were found ranging in severity from informational to medium. One medium-severity issue was the result of reuse of prior, modified wallet keys being unsafe due to signature replay attacks under certain conditions. Four low-severity issues were the result of inadequate data validation, including missing checks for zero addresses, and rounding errors that may occur in external Compound contracts during deposits and withdrawals. One low-severity issue was the result of a missing timelock interval limit, while another was due to a potential denial of service condition affecting the `DharmaSmartWalletFactoryV1` and `DharmaKeyRingFactoryV1` contracts. The remaining ten issues were informational in nature, and relate to items such as incorrect error reporting and compiler features that could be dangerous under certain conditions.

Overall, Trail of Bits found most of the smart wallet codebase to follow best practices and contain thorough documentation. It is recommended that Dharma Labs address all issues contained in this report, including the code quality issues highlighted in Appendix B.

Update: On October 18, 2019, Trail of Bits reviewed fixes proposed by Dharma Labs for the issues presented in this report. See a detailed review of the current status of each issue in Appendix D.

Project Dashboard

Application Summary

Name	Dharma Labs Smart Wallet
Version	`b1d510d`
Type	Smart contract
Platforms	Solidity

Engagement Summary

Dates	September 30 - October 11, 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	6	■ ■ ■ ■ ■ ■
Total Informational-Severity Issues	10	■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Total	17	

Category Breakdown

Data Validation	6	■ ■ ■ ■ ■ ■
Access Controls	3	■ ■ ■
Undefined Behavior	3	■ ■ ■
Cryptography	2	■ ■
Error Reporting	1	■
Auditing and Logging	1	■
Denial of Service	1	■
Total	17	

Engagement Goals

The goal of the engagement was to evaluate the security of the Dharma Labs smart wallet and answer questions such as the following:

- Is authorization adequately enforced?
- Are cryptographic signatures securely generated and validated?
- Can users safely change wallet keys?
- Can any arithmetic operations lead to integer underflows or overflows?
- Can any external function calls lead to issues such as denial of service, rounding errors, or unsafe reentrancy?
- Is data validation adequately enforced?
- Does timelocking functionality work as intended?
- Does account recovery functionality work as intended?
- Can the wallet contract be safely upgraded?
- Do sensitive functions emit event logs?

Coverage

Specific files from commit `b1d510d` from the dharma-smart-wallet repository were considered in scope and reviewed, and are listed below in priority from high to low:

1. Smart Wallet Implementation

- `implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol`

2. Smart Wallet Touchpoints

- `registries/DharmaKeyRegistryV1.sol`
- `account-recovery/DharmaAccountRecoveryManager.sol`
- `helpers/Timelocker.sol`
- `factories/smart-wallet/DharmaSmartWalletFactoryV1.sol`

3. Upgradeability

- `upgradeability/smart-wallet/DharmaUpgradeBeacon.sol`
- `upgradeability/DharmaUpgradeBeaconController.sol`
- `upgradeability/DharmaUpgradeBeaconEnvoy.sol`
- `proxies/smart-wallet/UpgradeBeaconProxyV1.sol`

4. Key Ring Implementation

- `implementations/key-ring/DharmaKeyRingImplementationV0.sol`
- `helpers/ECDSAGroup.sol` (not used in V0, used in the future)

5. Upgradeability protections + management

- `implementations/smart-wallet/AdharmaSmartWalletImplementation.sol`
- `implementations/key-ring/AdharmaKeyRingImplementation.sol`
- `upgradeability/DharmaUpgradeBeaconControllerManager.sol`
- `helpers/IndestructibleRegistry.sol`

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Implement a smart wallet nonce that increments after each key change operation within the `DharmaKeyRegistryV1` and `DharmaSmartWalletImplementationV2` contracts.** This will ensure that old wallet signatures cannot be reused. See TOB-DL-01 for additional information.

❑ **Implement a nonce that increments after each signature generation and validation operation within the `DharmaKeyRegistryV1` contract.** As described in TOB-DL-02, this will ensure that old signatures cannot be reused in `DharmaKeyRegistryV1`'s `setGlobalKey` function.

❑ **Ensure that errors are reported correctly.** As described in TOB-DL-03, fix the `_getCTokenDetails` function by setting `functionName` to "`redeem`" when a `_CDAI.redeem.selector` function selector is passed.

❑ **When transferring contract ownership, first ensure in off-chain code that the new owner's address is under control of Dharma Labs.** See TOB-DL-04 for additional information.

❑ **Ensure that adequate data validation is performed.** This includes checking that addresses are not equal to zero (TOB-DL-05, TOB-DL-06, TOB-DL-11, TOB-DL-12) and that certain limits are enforced on values (TOB-DL-12, TOB-DL-14).

❑ **Add logging for all sensitive functions.** As described in TOB-DL-07, ensure that all sensitive functions emit adequate event-based logging.

❑ **Avoid using ABIEncoderV2 and Solidity compiler optimizations.** Both can be dangerous. See TOB-DL-08 and TOB-DL-09 for additional information.

❑ **Use an older version of the Solidity compiler.** Newer versions have been observed to contain bugs and should therefore be avoided, as described in TOB-DL-10.

❑ **Document the risk of making small deposits or withdrawals.** Until extra validation is implemented as described in TOB-DL-13, users should be made aware of the dangers of rounding errors within Compound contracts.

❑ **Ensure timelocking works as expected and cannot be abused.** Extra care should be taken when enforcing timelocking functionality. See TOB-DL-15 and TOB-DL-16 for additional information.

❑ **Prevent potential denial-of-service conditions by generating sender-specific addresses in `DharmaSmartWalletFactoryV1` and `DharmaKeyRingFactoryV1`.** As described in TOB-DL-17, the `_getSaltAndTarget` function should mix in the value of `msg.sender`.

Long Term

❑ **Add additional unit testing for security-related components.** This includes nonces (TOB-DL-01, TOB-DL-02), blacklists (TOB-DL-01), error reporting (TOB-DL-03), data validation (TOB-DL-05, TOB-DL-11, TOB-DL-12, TOB-DL-14, TOB-DL-15), cryptographic signatures (TOB-DL-06), and access controls (TOB-DL-16).

❑ **Use a two-step ownership transfer scheme.** This will help ensure that a new owner address is correct before performing a full ownership transfer, as described in TOB-DL-04.

❑ **Integrate static analysis tools like Slither or Crytic into your CI pipeline.** This will help to detect a wide range of issues, such as the use of unsafe pragmas described in TOB-DL-08.

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity**. Due to concerns over the stability and security of the optimizations produced by the Solidity compiler, optimizations should be carefully reviewed before use. See TOB-DL-09 for additional information.

❑ **Use Compound's exchange rates to determine if a smart wallet deposit or withdrawal is safe**. If such an action is deemed unsafe, do not continue with the action and instead return an error. See TOB-DL-13 for additional information.

Findings Summary

#	Title	Type	Severity
1	Wallet key reuse is unsafe	Cryptography	Medium
2	setGlobalKey is susceptible to signature replay	Cryptography	Informational
3	Compound's redeem call failure emits ExternalError with incorrect function name	Error Reporting	Informational
4	transferOwnership should be split into two separate functions	Access Controls	Informational
5	Missing validation in contract initialization function	Data Validation	Low
6	Missing error check when calling ecrecover	Data Validation	Informational
7	Missing event logging	Auditing and Logging	Informational
8	ABIEncoderV2 is not production-ready	Undefined Behavior	Informational
9	Solidity compiler optimizations can be dangerous	Undefined Behavior	Informational
10	Solidity 0.5.11 not recommended for production use	Undefined Behavior	Informational
11	Missing validation in DharmaUpgradeBeaconControllerManager	Data Validation	Low
12	Missing validation in DharmaSmartWalletImplementationV2	Data Validation	Low
13	Rounding errors in external contracts can result in lost tokens	Data Validation	Low

14	Missing timelock interval limit allows for trapping timelocks until the interval is changed	Data Validation	Low
15	setTimelock functionality is ineffective for modifyTimelockInterval function	Access Controls	Informational
16	Timelock library is missing expiration functionality	Access Controls	Informational
17	Attacker can increase gas cost of _getSaltAndTarget	Denial of Service	Low

1. Wallet key reuse is unsafe

Severity: Medium
Type: Cryptography
Target: Multiple Files

Difficulty: High
Finding ID: TOB-DL-01

Description
Unexpected behavior in the `DharmaKeyRegistryV1` and `DharmaSmartWalletImplementationV2` contracts can lead to smart wallet users mistakenly broadcasting signatures that are invalid in the current block, but can be replayed later by an attacker in a future block.

The `DharmaKeyRegistryV1` contract allows an owner to set a global key via the `setGlobalKey` function (Figure 1.1), which can be used by Dharma Labs to sign smart wallet transactions on behalf of users. Alternatively, address-specific keys can be set via the `setSpecificKey` function (Figure 1.2).

The `DharmaSmartWalletImplementationV2` contract allows an account recovery manager to set a user signing key via the `recover` function (Figure 1.3). The `setUserSigningKey` function increments a nonce that invalidates previous signatures, and is therefore unaffected by this issue.

As implemented, both contracts are capable of changing keys at any time. If a prior, modified key is ever reused, then smart wallet signatures can swap between states of being valid and invalid in different blocks.

```
function setGlobalKey(
   address globalKey,
   bytes calldata signature
 ) external onlyOwner {
   // Ensure that the provided global key is not the null address.
   require(globalKey != address(0), "A global key must be supplied.");

   // Message hash constructed according to EIP-191-0x45 to prevent replays.
   bytes32 messageHash = keccak256(
     abi.encodePacked(
       address(this),
       globalKey,
       "This signature demonstrates that the supplied signing key is valid."
     )
   );

   // Recover the signer of the message hash using the provided signature.
   address signer = messageHash.toEthSignedMessageHash().recover(signature);

   // Ensure that the provided signature resolves to the provided global key.
   require(globalKey == signer, "Invalid signature for supplied global key.");

   // Update the global key to the provided global key.
   _globalKey = globalKey;
 }
```

Figure 1.1: `DharmaKeyRegistryV1`*'s* `setGlobalKey` *function.*

```
function setSpecificKey(
   address account,
   address specificKey
 ) external onlyOwner {
   // Update specific key for provided account to the provided specific key.
   _specificKeys[account] = specificKey;
 }
```

Figure 1.2: `DharmaKeyRegistryV1`*'s* `setSpecificKey` *function.*

```
function recover(address newUserSigningKey) external {
   require(
     msg.sender == _ACCOUNT_RECOVERY_MANAGER,
     "Only the account recovery manager may call this function."
   );

   // Set up the user's new dharma key and emit a corresponding event.
   _setUserSigningKey(newUserSigningKey);
 }
```

Figure 1.3: `DharmaSmartWalletImplementationV2`*'s* `recover` *function.*

Exploit Scenario

A call to a victim's smart wallet is signed by the global key within the same block in which the global key is changed, causing it to fail and preventing the wallet's nonce from incrementing. The victim sees the failed wallet transaction and assumes it is void. However, the global key is later reverted to the old one, either legitimately by an administrator, or maliciously by a privileged attacker who exploits TOB-DL-02. If the victim's wallet has not made any new transactions, the victim's original transaction can now unexpectedly succeed when resubmitted by an attacker.

Recommendation

Short term, implement a smart wallet nonce that increments after each key change operation so that old signatures always remain invalid. Alternatively, implement a blacklist of previously used keys to prevent smart wallet keys from ever being reused. If this is done, inform users that key reuse is prohibited.

Long term, add more unit testing to check that the nonce or blacklist works as intended.

2. setGlobalKey is susceptible to signature replay

Severity: Informational
Type: Cryptography
Target: `DharmaKeyRegistryV1.sol`

Difficulty: High
Finding ID: TOB-DL-02

Description
The `DharmaKeyRegistry` contract attempts to validate that an owner calling `setGlobalKey` has a corresponding private key by requiring a valid signature (Figure 2.1). However, this check does not guarantee that this is the case. While the replay of arbitrary signatures is prevented, an account owner can set a previously used key for which they lack the corresponding private key by replaying a valid signature observed from a previous call to this function.

```
function setGlobalKey(
   address globalKey,
   bytes calldata signature
 ) external onlyOwner {
   // Ensure that the provided global key is not the null address.
   require(globalKey != address(0), "A global key must be supplied.");

   // Message hash constructed according to EIP-191-0x45 to prevent replays.
   bytes32 messageHash = keccak256(
     abi.encodePacked(
       address(this),
       globalKey,
       "This signature demonstrates that the supplied signing key is valid."
     )
   );

   // Recover the signer of the message hash using the provided signature.
   address signer = messageHash.toEthSignedMessageHash().recover(signature);

   // Ensure that the provided signature resolves to the provided global key.
   require(globalKey == signer, "Invalid signature for supplied global key.");

   // Update the global key to the provided global key.
   _globalKey = globalKey;
 }
```

Figure 2.1: The `setGlobalKey` function.

Exploit Scenario
An attacker chains this issue with another, as described in TOB-DL-01's exploit scenario.

Recommendation
Short term, implement a nonce that increments after each signature generation and validation operation.

Long term, add more unit testing to check that the new nonce works as intended.

3. Compound's redeem call failure emits ExternalError with incorrect function name

Severity: Informational
Type: Error Reporting
Target: `DharmaSmartWalletImplementationV2.sol`

Difficulty: Low
Finding ID: TOB-DL-03

Description
The `DharmaSmartWalletImplementationV2` contract checks the results of its interactions with Compound by using the `_checkCompoundInteractionAndLogAnyErrors` function. This function (Figure 3.1) emits an `ExternalError` event specifying which Compound function call failed. The Compound function name is retrieved by forwarding the `functionSelector` argument to the `_getCTokenDetails` function.

The `_checkCompoundInteractionAndLogAnyErrors` function is called three times across the codebase with the following function selectors:

- In `_depositOnCompound` with `_CDAI.mint.selector`
- In `_withdrawFromCompound` with `_CDAI.redeemUnderlying.selector`
- In `_withdrawMaxFromCompound` with `_CDAI.redeem.selector`

However, the `_getCTokenDetails` function (Figure 3.2) only supports the `mint` and `redeemUnderlying` functions. As a result, when the `redeem` Compound call fails in the `_withdrawMaxFromCompound` function, `_checkCompoundInteractionAndLogAnyErrors` inaccurately reports that it was the `redeemUnderlying` function that failed.

```
function _checkCompoundInteractionAndLogAnyErrors(
    AssetType asset, bytes4 functionSelector, bool ok, bytes memory data
  ) internal returns (bool success) {
    // Log an external error if something went wrong with the attempt.
    if (ok) {
      uint256 compoundError = abi.decode(data, (uint256));
      if (compoundError != _COMPOUND_SUCCESS) {
        // Get called contract address, name of contract, and function name.
        (address account, string memory name, string memory functionName) = (
          _getCTokenDetails(asset, functionSelector)
        );

        emit ExternalError(
          account,
          string(
            abi.encodePacked(
              "Compound ", name, " contract returned error code ",
              uint8((compoundError / 10) + 48), uint8((compoundError % 10) + 48),
              " while attempting to call ", functionName, "."
            )
          )
        );
      }
    // (...) - similar calls occurs in the "else" branch
```

Figure 3.1: The _checkCompoundInteractionAndLogAnyErrors function, reformatted to take less space. Text highlighted in red marks the usage of `functionSelector` *and* `functionName`*, which is calculated from* `functionSelector` *in the* `_getCTokenDetails` *function.*

```
function _getCTokenDetails(
    AssetType asset,
    bytes4 functionSelector
  ) internal pure returns (
    address account,
    string memory name,
    string memory functionName
  ) {
    // (...) - sets `account` and `name`

    // Note: since both cTokens have the same interface, just use cDAI's.
    if (functionSelector == _CDAI.mint.selector) {
      functionName = "mint";
    } else {
      functionName = "redeemUnderlying";
    }
  }
```

Figure 3.2: The _getCTokenDetails function that returns `functionName` *based on the value of* `functionSelector`*.*

Exploit Scenario

While not currently an exploitable issue, incorrect error reporting could lead to bugs in the future if other routines were to depend on its accuracy.

Recommendation

Short term, fix the `_getCTokenDetails` function by setting `functionName` to "`redeem`" when a `_CDAI.redeem.selector` function selector is passed.

Long term, add more unit testing to check that error reporting works as intended.

4. transferOwnership should be split into two separate functions

Severity: Informational | Difficulty: High
Type: Access Controls | Finding ID: TOB-DL-04
Target: `DharmaUpgradeBeaconController.sol and Ownable contracts`

Description
The `transferOwnership` function changes ownership of a contract in a single transaction. If an incorrect `newOwner` is provided, ownership may never be recovered. A best practice is to split the ownership transfer into two functions: one for initiating the transfer and one for accepting the transfer.

By splitting the functionality of `transferOwnership` into two functions—`transferOwnership` and `acceptOwnership`—the original owner will retain ownership until the new owner calls `acceptOwnership`. This will help prevent accidental transfer of ownership to an uncontrolled address.

```
function transferOwnership(address newOwner) external onlyOwner {
  require(newOwner != address(0), "Ownable: new owner is the zero address");
  emit OwnershipTransferred(_owner, newOwner);
  _owner = newOwner;
}
```

Figure 4.1: The `transferOwnership` function.

Exploit Scenario

- Alice deploys one of the ownable contracts, then decides to change the owner to another address under her control.
- Subsequently, she enters the new address as `newOwner`, but mistakenly enters the last hex value of the address incorrectly.
- Upon invocation of `transferOwnership` with the incorrect address, Alice loses all ownership of the contract.

The ownable contracts are `DharmaKeyRegistryV1`, `DharmaUpgradeBeaconControllerManager`, and `DharmaAccountRecoveryManager`, while the `DharmaUpgradeBeaconController` contract implements its individual ownable logic.

Recommendation
Short term, ensure in the off-chain code that the `newOwner` address is always under control of Dharma Labs before invoking `transferOwnership`.

Long term, use the described two-step process of `transferOwnership` and `acceptOwnership` to ensure an address is controllable before performing a full ownership transfer.

5. Missing validation in contract initialization function

Severity: Low
Type: Data Validation
Target: `DharmaKeyRegistryV1.sol`

Difficulty: High
Finding ID: TOB-DL-05

Description
The `initialize` function in the `AdharmaSmartWalletImplementation` contract is missing address validation for the key parameter, as highlighted in Figure 5.1.

```
// Keep the initializer function on the contract in case a smart wallet has
// not yet been deployed but the account still contains user funds.
function initialize(address key) external {
  // Ensure that this function is only callable during contract construction.
  assembly { if extcodesize(address) { revert(0, 0) } }

  // Set up the user's key.
  _key = key;
}
```

Figure 5.1: The `initialize` function.

Exploit Scenario
Due to human error or a bug in a deployment script, an address of zero is passed to the initialization function, incorrectly setting the user's key to zero.

Recommendation
Short term, validate that a supplied key is not equal to zero.

Long term, add additional unit testing to check that invalid input is rejected from the `initialize` function.

6. Missing error check when calling ecrecover

Severity: Informational
Type: Data Validation
Target: Multiple Files

Difficulty: Medium
Finding ID: TOB-DL-06

Description
Several calls are made to the `ECDSA` library, which is a wrapper around the built-in `ecrecover` function, without explicitly checking if an error occurred (i.e., an address of 0 is returned).

- `implementations/key-ring/DharmaKeyRingImplementationV0.sol#L141`
- `implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L569`
- `implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L1121`
- `implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L1129`
- `implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L1313`
- `registries/DharmaKeyRegistryV1.sol#L68`

Figure 6.1: List of affected functions.

Exploit Scenario
All identified instances were found to be unexploitable, due to adequate data validation of user and Dharma signing keys elsewhere in the codebase. However, future changes to the codebase risk introducing an exploitable instance of this issue if return values of `ecrecover` are never checked.

Recommendation
Short term, validate that the returned address of calling `ecrecover` is not zero.

Long term, add more unit testing to check that invalid signatures are properly handled throughout the codebase.

7. Missing event logging

Severity: Informational
Type: Auditing and Logging
Target: Multiple Files

Difficulty: Low
Finding ID: TOB-DL-07

Description
Several sensitive functions do not emit events. An absence of event-based logging can make auditing transactions and responding to incidents more challenging.

- contracts/implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L145
- contracts/implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L205
- contracts/implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L315
- contracts/implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L412
- contracts/implementations/smart-wallet/DharmaSmartWalletImplementationV2.sol#L450
- contracts/registries/DharmaKeyRegistryV1.sol#L51
- contracts/registries/DharmaKeyRegistryV1.sol#L85
- contracts/account-recovery/DharmaAccountRecoveryManager.sol#L83
- contracts/account-recovery/DharmaAccountRecoveryManager.sol#L109
- contracts/account-recovery/DharmaAccountRecoveryManager.sol#L140

Figure 7.1: List of affected functions.

Exploit Scenario
An incident occurs, and due to a lack of event-based logging, an investigation takes longer than necessary to complete.

Recommendation
Ensure that all sensitive functions emit adequate event-based logging.

8. ABIEncoderV2 is not production-ready

Severity: Informational
Type: Undefined Behavior
Target: `DharmaSmartWalletImplementationV2.sol`

Difficulty: Undetermined
Finding ID: TOB-DL-08

Description
The `DharmaSmartWalletImplementationV2` contract uses the new Solidity ABI encoder, `ABIEncoderV2`. This encoder is still experimental and is not ready for production use.

Recently, over three percent of all GitHub issues for the Solidity compiler were found to be related to experimental features, with `ABIEncoderV2` constituting the vast majority. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with over a dozen bugs over the past year, and some are so recent they have not yet been included in a Solidity release.

For example, earlier this year a severe bug was found in the encoder and was introduced in Solidity 0.5.5.

Exploit Scenario
Dharma Labs deploys the `DharmaSmartWalletImplementationV2` contract. After deployment, a bug is found in the encoder, which an attacker exploits to permanently lock the contract.

Recommendation
Short term, do not use `ABIEncoderV2` or any other experimental Solidity feature. Refactor the code to alleviate the need to pass or return arrays of strings to and from functions.

Long term, integrate static analysis tools like Slither or Crytic into your CI pipeline to detect unsafe pragmas.

9. Solidity compiler optimizations can be dangerous

Severity: Informational
Type: Undefined Behavior
Target: `truffle-config.js`

Difficulty: Undetermined
Finding ID: TOB-DL-09

Description
Dharma Labs has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them, so it is difficult to determine how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018, and the fix for this bug was not reported in the Solidity CHANGELOG.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug. Optimizations could pose additional risk for limited benefits. Carefully review this tradeoff.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity. Due to concerns over the stability and security of the optimizations produced by the Solidity compiler, optimizations should be carefully reviewed before use.

10. Solidity 0.5.11 not recommended for production use

Severity: Informational
Type: Undefined Behavior
Target: Multiple Files

Difficulty: Undetermined
Finding ID: TOB-DL-10

Description
Solidity 0.5.11 is the latest version of the compiler and is not recommended for production use. This is due to the fact that the Solidity compiler has had a history of low- to medium-severity bugs in recent releases.

For additional information, please see the Solidity Documentation's List of Known Bugs.

Exploit Scenario
A bug recently introduced into the Solidity compiler causes a security vulnerability, which an attacker exploits to lock users' wallet contracts.

Recommendation
If possible, use an old version for production (e.g., 0.5.3) and a more recent version for testing (to benefit from the latest compiler's checks).

11. Missing validation in DharmaUpgradeBeaconControllerManager

Severity: Low
Type: Data Validation
Target: `DharmaUpgradeBeaconControllerManager.sol`

Difficulty: Medium
Finding ID: TOB-DL-11

Description
Within the `DharmaUpgradeBeaconControllerManager` contract, the `armAdharmaContingency` function does not perform adequate validation of its `controller` and `beacon` address parameters (Figure 11.1).

```
function armAdharmaContingency(
   address controller, address beacon, bool armed
 ) external {
   // Determine if 90 days have passed since the last heartbeat.
   (bool expired, ) = heartbeatStatus();
   require(
     isOwner() || expired,
     "Only callable by the owner or after 90 days without a heartbeat."
   );

   // Arm (or disarm) the Adharma Contingency.
   _adharma[controller][beacon].armed = armed;
 }
```

Figure 11.1: The armAdharmaContingency function.

Exploit Scenario
Due to human error or a bug in a script, the `armAdharmaContingency` function is called with zero addresses, so the contract to never becomes “armed” as was expected.

Recommendation
Short term, validate that the `controller` and `beacon` address parameters are not zero.

Long term, add additional unit testing to check that invalid inputs are rejected from the `armAdharmaContingency` function.

12. Missing validation in DharmaSmartWalletImplementationV2

Severity: Low
Type: Data Validation
Target: `DharmaSmartWalletImplementationV2.sol`

Difficulty: Medium
Finding ID: TOB-DL-12

Description
Within the `DharmaSmartWalletImplementationV2` contract, the `withdrawDai` and `withdrawUSDC` functions do not perform adequate validation of their `amount` and `recipient` parameters (Figures 12.1 and 12.2).

```
function withdrawDai(
   uint256 amount,
   address recipient,
   uint256 minimumActionGas,
   bytes calldata userSignature,
   bytes calldata dharmaSignature
 ) external returns (bool ok) {
   // Ensure caller and/or supplied signatures are valid and increment nonce.
   _validateActionAndIncrementNonce(
     ActionType.DAIWithdrawal,
     abi.encode(amount, recipient),
     minimumActionGas,
     userSignature,
     dharmaSignature
   );

   // Set the self-call context so we can call _withdrawDaiAtomic.
   _selfCallContext = this.withdrawDai.selector;

   // Make the atomic self-call - if redeemUnderlying fails on cDAI, it will
   // succeed but nothing will happen except firing an ExternalError event. If
   // the second part of the self-call (the Dai transfer) fails, it will revert
   // and roll back the first part of the call, and we'll fire an ExternalError
   // event after returning from the failed call.
   bytes memory returnData;
   (ok, returnData) = address(this).call(abi.encodeWithSelector(
     this._withdrawDaiAtomic.selector, amount, recipient
   ));

   // If the atomic call failed, emit an event signifying a transfer failure.
   if (!ok) {
     emit ExternalError(address(_DAI), "DAI contract reverted on transfer.");
   } else {
     // Set ok to false if the call succeeded but the withdrawal failed.
     ok = abi.decode(returnData, (bool));
   }
 }
```

Figure 12.1: The `withdrawDai` function.

```
function withdrawUSDC(
   uint256 amount,
   address recipient,
   uint256 minimumActionGas,
```

```
    bytes calldata userSignature,
    bytes calldata dharmaSignature
  ) external returns (bool ok) {
    // Ensure caller and/or supplied signatures are valid and increment nonce.
    _validateActionAndIncrementNonce(
      ActionType.USDCWithdrawal,
      abi.encode(amount, recipient),
      minimumActionGas,
      userSignature,
      dharmaSignature
    );

    // Set the self-call context so we can call _withdrawUSDCAtomic.
    _selfCallContext = this.withdrawUSDC.selector;

    // Make the atomic self-call - if redeemUnderlying fails on cUSDC, it will
    // succeed but nothing will happen except firing an ExternalError event. If
    // the second part of the self-call (USDC transfer) fails, it will revert
    // and roll back the first part of the call, and we'll fire an ExternalError
    // event after returning from the failed call.
    bytes memory returnData;
    (ok, returnData) = address(this).call(abi.encodeWithSelector(
      this._withdrawUSDCAtomic.selector, amount, recipient
    ));
    if (!ok) {
      // Find out why USDC transfer reverted (doesn't give revert reasons).
      _diagnoseAndEmitUSDCSpecificError(_USDC.transfer.selector);
    } else {
      // Ensure that ok == false in the event the withdrawal failed.
      ok = abi.decode(returnData, (bool));
    }
  }
```

Figure 12.2: The withdrawUSDC function.

Exploit Scenario

Due to human error or a bug in a script, either function is incorrectly called with their `amount` and/or `recipient` parameters set to zero.

Recommendation

Short term, perform validation of the `amount` and `recipient` parameters by checking if either value is equal to zero.

Long term, add more unit testing to check that invalid inputs are rejected from both functions.

13. Rounding errors in external contracts can result in lost tokens

Severity: Low
Type: Data Validation
Target: `DharmaSmartWalletImplementationV2.sol`

Difficulty: Low
Finding ID: TOB-DL-13

Description
Within the `DharmaSmartWalletImplementationV2` contract, `_depositOnCompound` transfers tokens to Compound, while `withdrawDai` and `withdrawUSDC` both transfer their respective tokens from Compound back to the owner's smart wallet. Rounding errors within the Compound contracts may cause a smart wallet owner to lose a small number of tokens, and result in some tokens being left behind in their account as "dust." While these issues are external to Dharma Labs, the `DharmaSmartWalletImplementationV2` contract should mitigate them as much as possible.

Exploit Scenario
A user mistakenly initiates a deposit of a small number of tokens that results in zero Compound tokens being minted.

A user mistakenly initiates a withdrawal of a small number of tokens that results in zero Compound tokens being withdrawn.

Recommendation
Short term, document this as a known issue so that users are aware of the risks associated with making small deposits and withdrawals from their smart wallet.

Long term, use Compound's exchange rates to determine if a smart wallet deposit or withdrawal is safe. If such an action is deemed unsafe, do not continue with the action and instead return an error. This can be computed off-chain, and extra parameters such as `minimumDeposit` and `minimumWithdrawal` could be introduced to the associated functions.

14. Missing timelock interval limit allows for trapping timelocks until the interval is changed

Severity: Low
Type: Data Validation
Target: `Timelocker.sol`

Difficulty: Low
Finding ID: TOB-DL-14

Description
The `Timelocker`'s `modifyTimelockInterval` function (Figure 14.1) does not limit the provided `newTimelockInterval` value used in `_setTimelock` function (Figure 14.2) to calculate the timelock date. Setting this value to a big number causes the timelock calculation in the `_setInterval` function to overflow. This makes the `_setInterval` function revert for a given function selector until the timelock interval is changed.

The functions protected by timelocks cannot currently be trapped forever because the `modifyTimelockInterval` function (Figure 14.3) is overridden by the `DharmaAccountRecoveryManager` and `DharmaUpgradeBeaconControllerManager` contracts. These contracts apply a timelock interval limit of eight weeks for the `modifyTimelockInterval` function.

It is important to note that even though the `modifyTimelockInterval` is public, it cannot currently be called by anyone because it is overridden by the `DharmaAccountRecoveryManager` and `DharmaUpgradeBeaconControllerManager` contracts, which both use the `onlyOwner` modifier.

```
function modifyTimelockInterval(
   bytes4 functionSelector,
   uint256 newTimelockInterval
 ) public {
   // Ensure that the timelock has been set and is completed.
   _enforceTimelock(
     this.modifyTimelockInterval.selector, abi.encode(newTimelockInterval)
   );

   // Set new timelock interval and emit a `TimelockIntervalModified` event.
   _setTimelockInterval(functionSelector, newTimelockInterval);
 }
```

Figure 14.1: The `Timelocker`*'s contract* `modifyTimelockInterval` *function.*

```
function _setTimelock(
   bytes4 functionSelector,
   bytes memory arguments,
   uint256 extraTime
 ) internal {
   // Get timelock using current time, inverval for timelock ID, & extra time.
   uint256 timelock = _timelockIntervals[functionSelector].add(now).add(
     extraTime
   );
```

Figure 14.2: The `_setTimelock` function, which calculates the timelock date using the given function's timelock interval.

```
function modifyTimelockInterval(
   bytes4 functionSelector,
   uint256 newTimelockInterval
 ) public onlyOwner {
   // Ensure that a function selector is specified (no 0x00000000 selector).
   require(
     functionSelector != bytes4(0),
     "Function selector cannot be empty."
   );

   // Ensure a timelock interval over eight weeks is not set on this function.
   if (functionSelector == this.modifyTimelockInterval.selector) {
     require(
       newTimelockInterval <= 8 weeks,
       "Timelock interval of modifyTimelockInterval cannot exceed eight weeks."
     );
   }

   // Continue via logic in the inherited `modifyTimelockInterval` function.
   Timelocker.modifyTimelockInterval(functionSelector, newTimelockInterval);
 }
```

Figure 14.3: The `modifyTimelockInterval` *function in the* `DharmaAccountRecoveryManager` and `DharmaUpgradeBeaconControllerManager` *contracts.*

Exploit Scenario

The `DharmaAccountRecoveryManager` contract owner wants to set a timelock interval for its `recover` function. They mistakenly set it to a value that, when added to "`now`," overflows. As a result, calls to the `setTimelock` function that are meant to set the timelock end up reverting and trapping the function. This trap lasts until the timelock interval is set to a proper value for the `revert` function, which can be done after setting and waiting for the timelock to expire for the `modifyTimelockInterval` function.

Recommendation

Short term, set an upper limit for the new timelock interval directly in `Timelocker`'s `modifyTimelockInterval` function. Then, remove the timelock interval limit from the `modifyTimelockInterval` overridden functions, or move it to the `Timelocker`'s implementation.

Long term, add more unit testing to check that invalid inputs are rejected from `modifyTimelockInterval` function.

15. setTimelock functionality is ineffective for modifyTimelockInterval function

Severity: Informational
Type: Access Controls
Target: `Timelocker.sol`

Difficulty: Low
Finding ID: TOB-DL-15

Description
The `_setTimelock` function (Figure 15.1) can be used to set a timelock that allows a given function and arguments to be called once after the set timelock period passes. The timelocks for the `modifyTimelockInterval` function (Figure 15.2) are set and enforced based on both function selector and the new internal timelock value. This allows timelocks for the `modifyTimelockInterval` function to be set with the same function selector but a different internal timelock value, which makes it possible to subvert the intended timelock behavior for the `modifyTimelockInterval` function.

This issue is not applicable to other functions protected by timelocks, as the arguments used are tied to the protected targets. Those functions are:

- In the `DharmaAccountRecoveryManager` contract:
  - `recover`, which enforces a timelock based on a wallet address and `newUserSigningKey`.
  - `disableAccountRecovery`, which enforces a timelock based on a wallet address.
- In the `DharmaUpgradeBeaconControllerManager` contract:
  - `upgrade`, which enforces a timelock based on `controller`, `beacon`, and `implementation` addresses.
  - `transferControllerOwnership`, which enforces a timelock based on `controller` and `newOwner` addresses.

```
function _setTimelock(
   bytes4 functionSelector,
   bytes memory arguments,
   uint256 extraTime
 ) internal {
   // Get timelock using current time, inverval for timelock ID, & extra time.
   uint256 timelock = _timelockIntervals[functionSelector].add(now).add(
     extraTime
   );

   // Get timelock ID using the supplied function arguments.
   bytes32 timelockID = keccak256(abi.encodePacked(arguments));

   // Get the current timelock, if any.
   uint256 currentTimelock = _timelocks[functionSelector][timelockID];

   // Ensure that the timelock duration does not decrease. Note that a new,
   // shorter timelock may still be set up on the same function in the event
   // that it is provided with different arguments.
```

```
    require(
      currentTimelock == 0 || timelock > currentTimelock,
      "Existing timelocks may only be extended."
    );

    // Set time that timelock will be complete using timelock ID and extra time.
    _timelocks[functionSelector][timelockID] = timelock;

    // Emit an event with all of the relevant information.
    emit TimelockInitiated(functionSelector, timelock, arguments);
  }
```

Figure 15.1: `Timelocker`*'s* `_setTimelock` *function.*

```
function modifyTimelockInterval(
    bytes4 functionSelector,
    uint256 newTimelockInterval
  ) public {
    // Ensure that the timelock has been set and is completed.
    _enforceTimelock(
      this.modifyTimelockInterval.selector, abi.encode(newTimelockInterval)
    );

    // Set new timelock interval and emit a `TimelockIntervalModified` event.
    _setTimelockInterval(functionSelector, newTimelockInterval);
  }
```

Figure 15.2: `Timelocker`*'s* `modifyTimelockInterval` *function.*

Exploit Scenario

The owner of `DharmaAccountRecoveryManager` or `DharmaUpgradeBeaconControllerManager` sets multiple timelocks for the `modifyTimelockInterval` function for all protected functions and with many different timelock interval values (e.g., with small differences). As a result, they can change the timelock interval for any protected function at any time, even though it will not be obvious to other users.

Recommendation

Short term, set and enforce the timelocks for `modifyTimelockInterval` function based only on the `functionSelector` value.

Long term, add additional unit testing to check that invalid inputs are rejected from `modifyTimelockInterval` function.

16. Timelock library is missing expiration functionality

Severity: Informational
Type: Access Controls
Target: `Timelocker.sol`

Difficulty: Low
Finding ID: TOB-DL-16

Description
Timelocks protect a given function from being called with given arguments for a given time. However, because there is no expiration time enforced in the `_enforceTimelock` function (Figure 16.1), a timelocked function can be called once at any time in the future, even long after the timelock has expired. This behavior may be unexpected to some users.

```
function _enforceTimelock(
   bytes4 functionSelector,
   bytes memory arguments
 ) internal {
   // Get timelock ID using the supplied function arguments.
   bytes32 timelockID = keccak256(abi.encodePacked(arguments));

   // Get the current timelock, if any.
   uint256 currentTimelock = _timelocks[functionSelector][timelockID];

   // Ensure that the timelock is set and has completed.
   require(func
     currentTimelock != 0 && currentTimelock <= now,
     "Function cannot be called until a timelock has been set and has expired."
   );

   // Clear out the existing timelock so that it cannot be reused.
   delete _timelocks[functionSelector][timelockID];
 }
```

Figure 16.1: The `Timelocker`*'s* `_enforceTimelock` *function.*

Exploit Scenario
An owner of a timelocked contract sets timelocks for all protected functions with all possible arguments (e.g., all wallets' addresses and users' signing keys with an arbitrary short time). This allows sensitive functions to be called at any time in the future, thereby subverting intended timelock behavior.

Recommendation
Short term, add a timelock pass expiration period and enforce it in the `_enforceTimelock` function.

Long term, add more unit testing to check that the time-locked functions cannot be called after the timelock pass expires.

17. Attacker can increase gas cost of _getSaltAndTarget

Severity: Low | Difficulty: Low
Type: Denial of Service | Finding ID: TOB-DL-17
Target: `DharmaSmartWalletFactoryV1.sol` and `DharmaKeyRingFactoryV1.sol`

Description
The `_getSaltAndTarget` function (Figure 17.1) is used by the `DharmaSmartWalletFactoryV1` and `DharmaKeyRingFactoryV1` contracts to calculate an address to be used during contract deployment. If a generated address already has contract code deployed to it, the function loops and continues until an unused address is found.

Since deploy functions (e.g. `newKeyRing`) can be used by anyone, it is possible to make these functions repeatedly deploy contracts and purposely use up generated addresses. This would result in these functions consuming higher amounts of gas when used legitimately, since they have to loop several times before a free address is found.

```
function _getSaltAndTarget(
   bytes memory initCode
 ) private view returns (uint256 nonce, address target) {
   // Get the keccak256 hash of the init code for address derivation.
   bytes32 initCodeHash = keccak256(initCode);
   // Set the initial nonce to be provided when constructing the salt.
   nonce = 0;
   // Declare variable for code size of derived address.
   uint256 codeSize;
   // Loop until an contract deployment address with no code has been found.
   while (true) {
     target = address(            // derive the target deployment address.
       uint160(                   // downcast to match the address type.
         uint256(                 // cast to uint to truncate upper digits.
           keccak256(             // compute CREATE2 hash using 4 inputs.
             abi.encodePacked(    // pack all inputs to the hash together.
               bytes1(0xff),      // pass in the control character.
               address(this),     // pass in the address of this contract.
               nonce,              // pass in the salt from above.
               initCodeHash       // pass in hash of contract creation code.
     ))))));
     // Determine if a contract is already deployed to the target address.
     assembly { codeSize := extcodesize(target) }

     // Exit the loop if no contract is deployed to the target address.
     if (codeSize == 0) {
       break;
     }

     // Otherwise, increment the nonce and derive a new salt.
     nonce++;
   }
 }
```

Figure 17.1: The `_getSaltAndTarget` *function which calculates the nonce and address of the later deployed contract.*

Exploit Scenario
An attacker uses `DharmaKeyRingFactoryV1` or `DharmaSmartWalletFactoryV1` to repeatedly deploy many contracts for a given user key. As a result, a legitimate deployment costs more gas or even reverts due to insufficient gas sent.

Recommendation
Include the original `msg.sender` address for target hash calculation in the `_getSaltAndTarget` function, so the returned target address will be different for different parties that call that function.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

DharmaSmartWalletImplementationV2.sol

- **Reentrancy guard is not enforced in `executeAction`.** This function is missing a call to `_enforceSelfCallFrom`. Although this is not exploitable, it is recommended that contracts use OpenZeppelin's ReentrancyGuard modifier to increase code clarity.
- **Explicitly delete `_selfCallContext` in the functions that set it and make internal transactions.** Currently, `_selfCallContext` is deleted when it is enforced via the `_enforceSelfCallFrom` function. However, if an internal transaction reverts, the `_selfCallContext` would not be deleted. While Trail of Bits has not found a situation where this could be exploited in the current codebase (as it is always set before making internal transactions), this behavior could introduce vulnerabilities into later versions of the Dharma Labs smart wallet.
- **Assign the `bool status` named return values in `_withdrawUSDCAtomic`, `_withdrawDaiAtomic` and `_checkCompoundInteractionAndLogAnyErrors` to `false`.** Currently these functions implicitly return a `false` status when certain code paths are not hit, which can be misinterpreted as the functions being incomplete.
- **Validate and make sure the comments in `withdrawDai` and `withdrawUSDC` functions are consistent.** When a Compound call fails in those functions (Figure B.1), the comment in `withdrawUSDC` suggests that the result should be passed to the `require` solidity function while the comment in `withdrawDai` states that the result is assigned.

```
function withdrawUSDC(/* ... */) external returns (bool ok) {
   // (...)
   else {
     // Ensure that ok == false in the event the withdrawal failed.
     ok = abi.decode(returnData, (bool));
   }
}

function withdrawDai(/* ... */) external returns (bool ok) {
   // (...)
   else {
     // Set ok to false if the call succeeded but the withdrawal failed.
     ok = abi.decode(returnData, (bool));
```

```
    }
  }
```

Figure B.1: The `withdrawDai` *and* `withdrawUSDC` *functions.*

IndestructibleRegistry.sol

- **Do not mix named and unnamed return values in `_isPotentiallyDestructible` function.** Currently, the function explicitly returns true or implicitly returns false by setting `potentiallyDestructible`, which can be misinterpreted as the function being incomplete.

Timelocker.sol

- **Fix the comment for the `oldInterval` field in the `TimelockIntervalModified` event.** The comment is the same as for the `newInterval` field. The text "new minimum timelock interval for the function" should be changed to "old minimum timelock interval for the function."
- **Consider moving the `setTimelock` and `modifyTimelockInterval` functions from `DharmaAccountRecoveryManager` and `DharmaUpgradeBeaconControllerManager` to `Timelocker` or another contract.** Those functions share the same code and should be moved to one place, so that further changes made to one function are not overlooked for the other one.

DharmaUpgradeBeaconEnvoy.sol

- **Mark the `DharmaUpgradeBeaconEnvoy` contract as the one that implements the `DharmaUpgradeBeaconEnvoyInterface` interface.** This allows external tools to use this information to enhance code analysis and code navigation in IDEs.

DharmaKeyRingImplementationV1.sol

- **Use `ActionType.SetUserSigningKey` instead of a hardcoded "1" for the enum value in the `isValidSignature` function.** This makes it easier to read the code and validate its expected behavior.
- **Validate the `requiredKeyType` argument against `KeyType.None` in the `_verifyOrderedSignatures` function.** While the function (Figure B.2) is always called with a hardcoded key type argument, it would benefit Dharma Labs to add this validation to prevent mistakes if the function is reused in the future.

```
enum KeyType {
   None,
   Standard,
   Admin,
   Dual
 }

function _verifyOrderedSignatures(
```

```
    KeyType requiredKeyType, bytes32 hash, bytes memory signatures
  ) internal view {
    uint160[] memory signers = hash.recoverGroup(signatures);

    uint256 threshold = (
      requiredKeyType == KeyType.Standard
        ? uint256(_additionalThresholds.standard)
        : uint256(_additionalThresholds.admin)
    ) + 1;

    // (...)
```

Figure B.2: The `KeyType` *enum and the* `_verifyOrderedSignatures` *function.*

C. Contract Upgradability Check

Trail of Bits ran Slither's `slither-check-upgradeability` command-line tool to assess the safety of upgrading the Dharma Labs smart wallet contracts. No issues were found (Figure C.1). Note that the shadowing warning is incorrect, as described here.

```
$ slither-check-upgradeability . UpgradeBeaconProxyV1 . DharmaSmartWalletImplementationV2
--new-version . --new-contract-name DharmaSmartWalletImplementationV3
INFO:CheckInitialization:Run initialization checks... (see
https://github.com/crytic/slither/wiki/Upgradeability-Checks#initialization-checks)
INFO:CheckInitialization:Initializable contract not found, the contract does not follow a
standard initalization schema.
INFO:CompareFunctions:Run function ids checks... (see
https://github.com/crytic/slither/wiki/Upgradeability-Checks#functions-ids-checks)
INFO:CompareFunctions:Shadowing between proxy and implementation found fallback()
INFO:VariablesOrder:Run variables order checks between the implementation and the proxy...
(see https://github.com/crytic/slither/wiki/Upgradeability-Checks#variables-order-checks)
INFO:VariablesOrder:No variables ordering error found between implementation and the proxy
INFO:VariablesOrder:Run variables order checks between implementations... (see
https://github.com/crytic/slither/wiki/Upgradeability-Checks#variables-order-checks)
INFO:VariablesOrder:No variables ordering error found between implementations
```

Figure C.1: Output of running Slither's upgradeability analysis. Note that the shadowing warning is incorrect, as described here.

D. Fix Log

From October 18, 2019 to October 21, 2019, Trail of Bits reviewed fixes for issues identified in this report, available in the commit 4110d1c of the Dharma smart wallet repository. Dharma Labs addressed or accepted the risk of all discovered issues in their codebase as a result of the assessment. Of those issues, 12 were remediated, one was partially fixed, and four were risk-accepted; we reviewed each of the fixes to help ensure the proposed remediation would be effective.

ID	Title	Severity	Status
01	Wallet key reuse is unsafe	Medium	Fixed
02	setGlobalKey is susceptible to signature replay	Informational	Fixed
03	Compound's redeem call failure emits ExternalError with incorrect function name	Informational	Fixed
04	transferOwnership should be split into two separate functions	Informational	Fixed
05	Missing validation in contract initialization function	Low	Fixed
06	Missing error check when calling ecrecover	Informational	Risk Accepted
07	Missing event logging	Informational	Partially fixed
08	ABIEncoderV2 is not production-ready	Informational	Risk Accepted
09	Solidity compiler optimizations can be dangerous	Informational	Risk Accepted
10	Solidity 0.5.11 not recommended for production use	Informational	Risk Accepted
11	Missing validation in DharmaUpgradeBeaconControllerManager	Low	Fixed
12	Missing validation in DharmaSmartWalletImplementationV2	Low	Fixed
13	Rounding errors in external contracts can result in lost tokens	Low	Fixed

14	Missing timelock interval limit allows for trapping timelocks until the interval is changed	Low	Fixed
15	setTimelock functionality is ineffective for modifyTimelockInterval function	Informational	Fixed
16	Timelock library is missing expiration functionality	Informational	Fixed
17	Attacker can increase gas cost of _getSaltAndTarget	Low	Fixed

Detailed Fix Log

Finding 1: Wallet key reuse is unsafe
Fixed in `DharmaKeyRegistryV2` and `DharmaSmartWalletImplementationV3`. Dharma Labs added checks to ensure global and specific keys are not reused by maintaining `_usedGlobalKeys` and `_usedSpecificKeys` mappings and validating new keys against them. User signing key reuse in `DharmaSmartWalletImplementationV3` is prevented by incrementing the nonce when a new user signing key is set.

Finding 2: setGlobalKey is susceptible to signature replay
Fixed in `DharmaKeyRegistryV2`. Dharma Labs added a check to ensure global keys are not reused by maintaining a `_usedGlobalKeys` mapping and validating new keys against it.

Finding 3: Compound's redeem call failure emits ExternalError with incorrect function name
Fixed in `DharmaSmartWalletImplementationV3`. Dharma Labs added the missing "`redeem`" function selector check to `_getCTokenDetails` function.

Finding 4: transferOwnership should be split into two separate functions
Fixed. Dharma Labs implemented a `TwoStepOwnable` contract that is used by `DharmaAccountRecoveryManager`, `DharmaKeyRegistryV2` and `DharmaUpgradeBeaconControllerManager` contracts. Dharma Labs also added a custom logic (the `_willAcceptOwnership` mapping, `agreeToAcceptOwnership`, and `transferControllerOwnership` functions) to `DharmaUpgradeBeaconControllerManager` to ensure proper ownership transfer of the managed controller contract.

Finding 5: Missing validation in contract initialization function
Fixed. Dharma Labs added a `require` to validate that the passed key is not zero.

Finding 6: Missing error check when calling ecrecover
Risk Accepted. Dharma Labs acknowledged the issue and decided not to introduce the check as it can only be problematic if future changes are introduced to the codebase.

Finding 7: Missing event logging
Partially fixed and residual risk accepted. Dharma Labs added event logging to:

- `DharmaKeyRegistryV2`'s `setGlobalKey` and `setSpecificKey` functions.
- `DharmaAccountRecoveryManager`'s `recover` and `disableAccountRecovery` functions.
- `DharmaSmartWalletImplementationV3`'s `withdrawEther` and `cancel` functions.

Dharma Labs accepted the risk of missing events in `DharmaSmartWalletImplementationV3`'s `withdrawDai` and `withdrawUSDC` functions when the Compound calls succeed as the called Compound functions do event logging.

Finding 8: ABIEncoderV2 is not production-ready
Risk Accepted. Dharma Labs indicated they mitigate the issue by testing their code and allowing users' smart wallet contracts to be upgraded to emergency `AdharmaSmartWalletImplementation` contracts.

Finding 9: Solidity compiler optimizations can be dangerous
Risk Accepted. Dharma Labs indicated that compiler optimizations provide substantial gas savings to users and these savings currently outweigh the risk of optimizer bugs.

Finding 10: Solidity 0.5.11 not recommended for production use
Risk Accepted. Dharma Labs indicated that they will test their codebase against an older Solidity compiler version but use 0.5.11 for now since it fixes some issues with ABIEncoderV2.

Finding 11: Missing validation in DharmaUpgradeBeaconControllerManager
Fixed. Dharma Labs added `require` statements to validate the `beacon` and `controller` address parameters are not zero.

Finding 12: Missing validation in DharmaSmartWalletImplementationV2
Fixed. Dharma Labs added `require` statements to validate the `amount` value and `recipient` address are not zero.

Finding 13: Rounding errors in external contracts can result in lost tokens
Fixed. Dharma Labs changed the balance check in `_depositOnCompound` and added `require` statements for amount checks in `withdrawUSDC` and `withdrawDai` functions to account for DAI and USDC decimals. Note that this issue still persists when a maximum withdrawal occurs. Dharma Labs acknowledged that this is the desired behavior as this case is used when users want to clear out all of the underlying tokens.

Finding 14: Missing timelock interval limit allows for trapping timelocks until the interval is changed
Fixed. Dharma Labs added a limit for the minimum timelock interval value to the `_setTimelock` and `_setTimelockInterval` functions.

Finding 15: setTimelock functionality is ineffective for modifyTimelockInterval function
Fixed. Dharma Labs changed the `Timelocker` to only allow a single timelock for the `modifyTimelockInterval` and the newly added `modifyTimelockExpiration` functions at a given time.

Finding 16: Timelock library is missing expiration functionality
Fixed. Dharma Labs introduced timelock expiration values that are set for each function selector, similar to the timelocks' interval values. The expiration value is also limited in the same way as the interval value.

Finding 17: Attacker can increase gas cost of _getSaltAndTarget
Fixed. Dharma Labs changed the functions in the `DharmaSmartWalletFactoryV1` and `DharmaKeyRingFactoryV1` contracts to take an additional target address parameter and check the deployed (or existing) contract code hash to validate that it has the expected value.

Detailed Issue Discussion

Responses from Dharma Labs for risk-accepted and unfixed issues are included as quotes below.

Finding 6: Missing error check when calling ecrecover

> *While we don't directly check for ecrecover errors, we do check the results against, for example, signing keys. As a result, we believe the issue can't be exploited and could only backfire in case of further changes or code reuse. We are going to keep this situation in mind and also properly test newly added code.*

Finding 7: Missing event logging

> *We believe the additional event logging for the success cases of withdraw from Compound functions is not needed on our side since the logging happens on the Compound side as well.*

Finding 8: ABIEncoderV2 is not production-ready
Finding 9: Solidity compiler optimizations can be dangerous
Finding 10: Solidity 0.5.11 not recommended for production use

> *We may deploy an initial "rollback" version of the V3 implementation with:*
>
> *- No ABIEncoderV2 (and, by extension, no generic atomic batch functionality),*
> *- Solidity 0.5.3, and*
> *- No optimization enabled.*
>
> *If a serious issue is uncovered in any of the above, then we can flip the switch on the upgrade beacon controller manager right away (and functionality of the implementation will be unchanged).*