Tether ERC20 Launch Audit

Philip Daian

https://stableset.com

Overview This report presents a security review for the initial launch of the Tether ERC20 token by Tether Limited / iFinex Inc. on the Ethereum mainnet. The audit is limited in scope to uncovering potential problems and making recommendations to ensure the robustness **within the boundary of the Solidity contract only**.

Table of Contents

Cover Page 1

1 Introduction and Background 2
1.1 Team and Organization 3
1.2 Disclaimers 3
1.3 Highlighting 3
2 Scope 4
2.1 System Overview and Audit Coverage 4
2.2 Versions 5
2.3 Threat Model and Economics 7
3 Common Antipattern Analysis 8
4 Testing and assurance 12
4.1 Recommended unit tests 13
4.2 Recommended integration tests 15
4.3 Regression testing and testing policy 16
5 Code walkthrough 16
5.1 Data invariants - TetherToken.sol 16
5.2 Function-level comments 16
6 General recommendations 18
6.1 Bounty program 18
6.2 Verifiable compilation 18
6.3 Software analysis tools 18
6.4 Documentation 18
6.5 Privacy concerns 18
6.6 Contract failure management 19
6.7 Versioning Scheme 19
7 Conclusion 19
8 Feedback 20

1 Introduction and Background

Tether (`https://tether.to/`) is an upcoming smart contract featuring a series of ERC20 tokens, or "Tethers". Each Tether carries a name and symbol of the underlying asset it represents, and is transferable and tradeable as a standard ERC20 token. The first Tether to launch on the Ethereum blockchain is USDTether, which is intended to allow tokenized US dollars to be exchanged on the Ethereum network (and potentially used in smart contracts). This expands the previous Omni version of the system by allowing for interoperability with protocols such as `https://0xproject.com/`.

Each Tether is a standard ERC20 token with two core additional functions, *issue* and *redeem*. These functions allow the owner of a Tether contract to increase or decrease the total supply of Tether, by creating tokens and crediting their account in the former case or debiting their account and destroying tokens in the latter.

Other supporting functions added to the ERC20 include *deprecate*, which allows the Tether contract to be upgraded to forward all ERC20-compliant calls to a new contract address as a temporary stop-gap measure before client contracts and applications can upgrade their contract addresses to the new version, and *setParams*, which allows Tether to introduce a small/bounded fee for transactions to subsidize potential future development.

The intended use of Tether is as follows: Tethers are issued, collateralized by some funds in the bank account of Tether Limited, and are traded as a standard token. To cash out of Tethers, users can send Tether to a supporting exchange and trade for cryptocurrencies. Alternatively, Tether can initiate USD wires to its member exchanges through a withdrawal process that involves destroying the corresponding tokens. Eventually, many assets other than USD are planned to be supported with similar usage patterns and identical contract code.

As a company, Tether promises to hold 1:1 reserves for all Tether generated in its contracts. A total list of updated balances issued to all Tether can be viewed at `https://wallet.tether.to/transparency`, and can be checked against the on-chain deployments of Tether contracts by any interested party. Furthermore, Tether Limited has contracted new auditors, Friedman LLP, a well-established New York-based accounting firm, to issue reports confirming the available balance of the claimed funds. These reports will be published and communicated on the Tether website.

The Tether token is owned and governed by a multisignature wallet collectively referred to as the "Tether owner". This multisignature wallet contains an address for each partner in the Tether organization, and is the only entity able to perform privileged operations in the Tether contract. These operations are as follows:

- Issue new Tethers (when they are required by the system and corresponding deposits are held).
- Destroy / redeem Tethers leaving the system.
- Upgrade contract code to a new version, replacing all ERC20 API functions with new code and permanently disabling current code.
- Pause the contract, suspending transfer and other user operations.
- Change the owner to a new owner.
- Set a fee within certain hardcoded bounds (with a maximum feerate of 2%).
- Set a maximum fee within certain hardcoded bounds (with a maximum max fee of 50 USDTethers).

This multisignature contract comes with a threshold whereby some number of the Tether partners/administrators must agree before any of the above actions is executed. Overall, the administrators of this multisignature contract are the key base of trust in the Tether token; if they are untrusted or malicious, the contract offers no security guarantees whatsoever. This is a reasonable and rational model for this contract, as there is already centralized trust introduced by the required assumption of full reserve on which the token is based.

The intended uses of the contract are numerous. Several exchanges are partnering with Tether Limited in the launch process, and will allow their users to move funds between exchanges or withdraw Tethers directly, reducing the current inter-bank delays often involved in exchange withdrawal and allowing their users access to tokenized USD. Additionally, blockchain systems can offer prices denominated in and pegged to traditional assets, reducing exposure to volatility for their consumers. The entire system does, however, require trust in Tether and its administrators to remain solvent and honest in their reserve holdings and execution of token governance.

1.1 Team and Organization

Tether is launched by Tether Limited, a subsidiary of iFinex Inc. iFinex Inc. counts as other subsidiaries Hong Kong-based Bitfinex, one of the largest Bitcoin exchanges in the world. The primary developers of the audited version of the Tether smart contract are Will Harborne, the Ethereum Projects Lead at iFinex Inc., and Enrico Rubboli, a Senior Software Engineer at iFinex Inc. Their LinkedIn pages detailing relevant background and experience are located at `https://www.linkedin.com/in/will-harborne-18163742/` and `https://www.linkedin.com/in/rubboli/`

1.2 Disclaimers

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large number of funds.

1.3 Highlighting

To enhance the readability of this report (though we always recommend carefully reading the full document), we use the following highlight colors:

- Client/documentation recommendations: These are recommendations to any client contracts or other software components making use of the Tether API, as well as to the Tether developers to include in documentation to such parties.
- User recommendations: These are actions we recommend to users of or investors in the Tether contract.
- Solved in v1: These are recommendations for the contract that no longer apply to the latest version, as they are solved in the upgrade from v0 to v1.
- Long term: These are recommendations for policy, documentation, or architectural changes that we make to the Tether team as part of a long-term roadmap, that do not result in critical security issues for the audited versions of the contracts.
- To be addressed: These are recommendations to the Tether team for tests or changes before the contract is moved into production that have not been addressed in the latest report version.

2 Scope

The scope of this audit is limited to code-level problems with the Solidity Tether ERC20 contract. The primary focus is on code written by the Tether organization, but we will also cover the sanity of the critical Zeppelin dependency (that implements much of the ERC20 logic), the multisignature wallet used to administer the contract, and the interaction of this wallet with the Tether token.

We will now give a brief overview of the architecture of the Tether contract and precisely scope our efforts.

2.1 System Overview and Audit Coverage

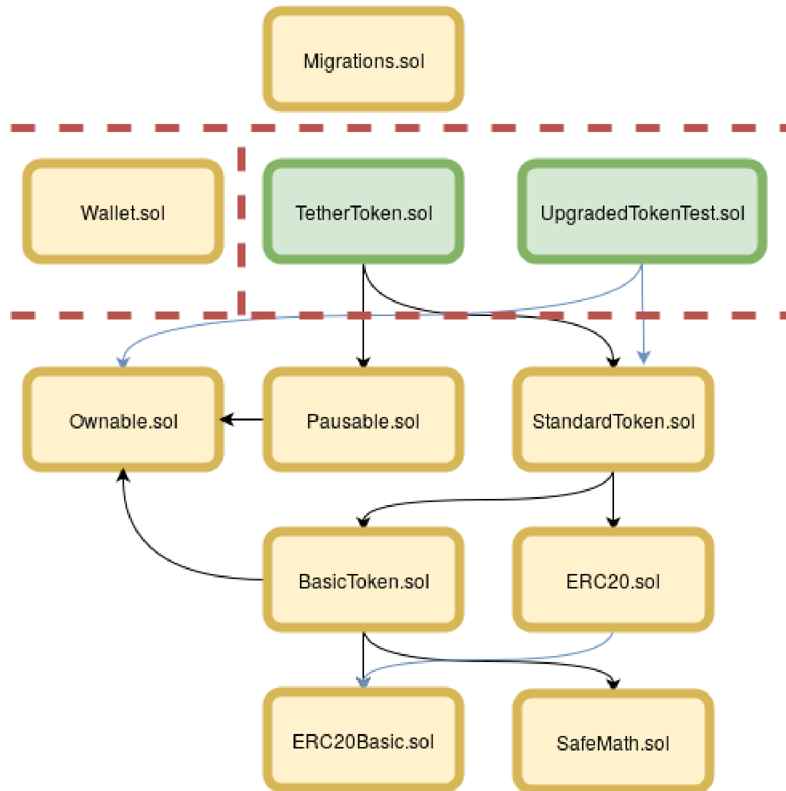

Fig. 1. Tether System Overview

The full on-chain system consists of the components and dependencies shown in Figure 1. The overarching components, developed by different development teams and organizations, are separated by the red dotted lines. The relevant team/system of each is labeled in every case.

Tether depends on Truffle for migrations, testing, and deployment infrastructure, Gnosis for the multisignature wallet that is set as the owner of the Tether contract, and Zeppelin for the underlying implementation of the basic ERC20 functionality, the ownership functionality, and the pause/resume functionality.

Code dependencies between the system components are denoted with arrows (with color differences being purely for visual clarity).

The components denoted in green are the core new code introduced by the Tether project, and will be the focus of the most detailed review in the audits (including invariants

and line-by-line analysis). The components in yellow are off-the-shelf, and will be analyzed only for potential interactions with the Tether contract, and not for full correctness (though special care is taken to test all paths of the core ERC20 logic as part of TetherToken). Components marked in grey are used for testing only, and are not deployed on-chain in a production deployment of the system. They are thus not covered in detail by the audit.

Some changes were made by the Tether team to the core Zeppelin libraries, and these changes are covered fully by the audit despite not being marked in the diagram above.

Also important to the correct execution of the systems are external dependencies:

- The Solidity compiler and language
- The Ethereum implementation (VM, networking client, consensus implementation, etc.)
- The client applications parsing Tether tokens (eg - on exchanges, in contracts, or in user interfaces)

These remaining items and their interactions (with both each other and the Tether contract) pose significant risk to the operation of the contract, though many are not specific to this contract, causing any bugs to likely affect a large number of contracts in the ecosystem.

In the case of client applications, we recommend appropriate care be taken to rigorously test for bad interactions with any of the above components or the Tether contract. We also recommend rigorous input and output validation (eg - of addresses): failure to implement such validation in past client applications has led to stuck tokens being sent to invalid addresses.

2.2 Versions

Each version section listed below represents a version of the contracts we audited, stating with "v0". Comments in the report of the nature "addressed by v1" imply a recommendation has been addressed by a subsequent contract version, and the comment remains for historical purposes only.

All observations in this report have been verified to apply to the latest contract version below. All file hashes are SHA256.

Version 3

```
commit 9b45153fccade2d4a5e82df08d6c6e51bd68132c
Author: plutoegg <willharborne@Wills-MacBook-Pro.local>
Date:   Mon Aug 14 16:12:22 2017 +0100

    update to allow setting unlimited allowance and save gas fee
```

```
Migrations.sol d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376
MultiSigWallet.sol 9c11d634b6460cba73381781bff42e1252c23677ebbf6359c43ecaf919adde17
TetherToken.sol 45e44673d9a3548e4d11f17507588d0fea1a752882137037072c9d8477c995f5
UpgradedTokenTest.sol 56b3590e6a17687d95331704827499aa24f845af18051b9714f85ffe9aa0dfc2
zeppelin/SafeMath.sol e2569cfc3d09006e6c4689c49aa8949528ca09e8bcd23f34405ae4c38584b9ad
zeppelin/lifecycle/Pausable.sol 1c981607effd481b5ad76e72288901791566fe8842ccfeb94c7344c945828266
zeppelin/ownership/Ownable.sol 195a350778f7503c843f25eebc5963edfd27f1b9fe100313cc5086facc7a436b
zeppelin/token/BasicToken.sol 98f0cee9650ec8214570a0f43f3665ca405f7d7012b39922a0a293c43fc92ea5
zeppelin/token/ERC20.sol 3daed22df5910e2a49d5216f2933714b95a03bbe12d16ba95f43cc6cc20499fc
zeppelin/token/ERC20Basic.sol 06c586ce4c9bcf6cba9835336b99dd7a42dad8ea0960ffe7603a2f8b46d10c58
zeppelin/token/StandardToken.sol 50fc5c3858b1fc0aa4fdf8d52e2ea26838188dacb9c85cec7372c2facfb53334
../test/fee_changes.js d5dc9c1cb7dbbbe64af527f92a1e7f3726fa0422c556c395b5bd8752011347a9
../test/test_deploy.js b4c3836e4e68c432b80b132015d1f2564bb8e97a275bc20af13cc9ae21860227
../test/tokens_unit_test.js a7654d2082dd8327b1a0ea2867e37390b98b48ccaab67d3e402c6bc541bf44d0
```

Version 2

```
commit 6688c0062ecb8f058fda7a4d7cd55b0c3c830fb4
Author: plutoegg <willharborne@Wills-MacBook-Pro.local>
Date:   Sat Jul 8 20:42:44 2017 +0100

    added new tests and taking into account decimals

Migrations.sol d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376
MultiSigWallet.sol 9c11d634b6460cba73381781bff42e1252c23677ebbf6359c43ecaf919adde17
TetherToken.sol f3f8e28955b823e3ebbba8a4ffa2fe8d5a45e80bd6bce0db25433a58fea1e36d
UpgradedTokenTest.sol 56b3590e6a17687d95331704827499aa24f845af18051b9714f85ffe9aa0dfc2
zeppelin/SafeMath.sol e2569cfc3d09006e6c4689c49aa8949528ca09e8bcd23f34405ae4c38584b9ad
zeppelin/lifecycle/Pausable.sol 1c981607effd481b5ad76e72288901791566fe8842ccfeb94c7344c945828266
zeppelin/ownership/Ownable.sol 195a350778f7503c843f25eebc5963edfd27f1b9fe100313cc5086facc7a436b
zeppelin/token/BasicToken.sol 98f0cee9650ec8214570a0f43f3665ca405f7d7012b39922a0a293c43fc92ea5
zeppelin/token/ERC20.sol 3daed22df5910e2a49d5216f2933714b95a03bbe12d16ba95f43cc6cc20499fc
zeppelin/token/ERC20Basic.sol 06c586ce4c9bcf6cba9835336b99dd7a42dad8ea0960ffe7603a2f8b46d10c58
zeppelin/token/StandardToken.sol abae06efba99f143d07b03ee4f236e5059ba29f9c4a58751a7ff96256f7152a0
../test/fee_changes.js 1026a9603d0a04c5870f4720567aafc1b6f364f043cabc91f1ec3f2cc93a9670
../test/test_deploy.js b4c3836e4e68c432b80b132015d1f2564bb8e97a275bc20af13cc9ae21860227
../test/tokens_unit_test.js a7654d2082dd8327b1a0ea2867e37390b98b48ccaab67d3e402c6bc541bf44d0
```

Version 1

```
Author: plutoegg <willharborne@Wills-MacBook-Pro.local>
Date:   Tue Jun 13 23:41:40 2017 +0100

    Making deprecated bool and upgraded address public variables

Migrations.sol d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376
MultiSigWallet.sol 9c11d634b6460cba73381781bff42e1252c23677ebbf6359c43ecaf919adde17
TetherToken.sol c10dab8a921e5e4871b592dc9f563a8df7509e628dfc94a3d368796d96aa17bc
UpgradedTokenTest.sol 56b3590e6a17687d95331704827499aa24f845af18051b9714f85ffe9aa0dfc2
zeppelin/SafeMath.sol e2569cfc3d09006e6c4689c49aa8949528ca09e8bcd23f34405ae4c38584b9ad
zeppelin/lifecycle/Pausable.sol 1c981607effd481b5ad76e72288901791566fe8842ccfeb94c7344c945828266
zeppelin/ownership/Ownable.sol 195a350778f7503c843f25eebc5963edfd27f1b9fe100313cc5086facc7a436b
zeppelin/token/BasicToken.sol d30bc2cf4bf49571c8ec44f76b33107809bd78db26dc3a354da121d05aaccf99
zeppelin/token/ERC20.sol 3daed22df5910e2a49d5216f2933714b95a03bbe12d16ba95f43cc6cc20499fc
zeppelin/token/ERC20Basic.sol 06c586ce4c9bcf6cba9835336b99dd7a42dad8ea0960ffe7603a2f8b46d10c58
zeppelin/token/StandardToken.sol 6209f07fe3f8231b4a7a403415acd876b39f1a20b9cac5d239b44478d154382f
../test/test_deploy.js b4c3836e4e68c432b80b132015d1f2564bb8e97a275bc20af13cc9ae21860227
../test/tokens_unit_test.js a7654d2082dd8327b1a0ea2867e37390b98b48ccaab67d3e402c6bc541bf44d0
```

Version 0

```
Checked out of Git June 6, 1918EST

HEAD: commit 43f68d45e5b407cf6729a77891d01784c3674bbd
Author: Enrico Rubboli <rubboli@gmail.com>
Date:   Fri Jun 2 12:31:26 2017 +0200

    allow totalSupply to be upgraded

Migrations.sol d1ba9cc38ae66d7c7b6f2248262205cbe5d76452608f92e0428c2174c7a59376
TetherToken.sol d6074ccd934df6199ec1a39a046d5e9e43ee0fb8615fb05f770f4133f58d8d69
UpgradedTokenTest.sol 56b3590e6a17687d95331704827499aa24f845af18051b9714f85ffe9aa0dfc2
zeppelin/SafeMath.sol e2569cfc3d09006e6c4689c49aa8949528ca09e8bcd23f34405ae4c38584b9ad
```

```
zeppelin/lifecycle/Pausable.sol 1c981607effd481b5ad76e72288901791566fe8842ccfeb94c7344c945828266
zeppelin/ownership/Ownable.sol 195a350778f7503c843f25eebc5963edfd27f1b9fe100313cc5086facc7a436b
zeppelin/token/BasicToken.sol d30bc2cf4bf49571c8ec44f76b33107809bd78db26dc3a354da121d05aaccf99
zeppelin/token/ERC20.sol 3daed22df5910e2a49d5216f2933714b95a03bbe12d16ba95f43cc6cc20499fc
zeppelin/token/ERC20Basic.sol 06c586ce4c9bcf6cba9835336b99dd7a42dad8ea0960ffe7603a2f8b46d10c58
zeppelin/token/StandardToken.sol 6209f07fe3f8231b4a7a403415acd876b39f1a20b9cac5d239b44478d154382f
../test/tokens.js c976fb42a22adb930808ad5a9dfc021ab2d73c3e51a6c489d8147a8e920b5c52
(v0 used a different multisignature wallet, which we omit comments on)
```

Please verify that you are using or validating the appropriate contract version and that the hashes match the above. If subsequent versions of the contract have been released, **all claims in this report must be reevaluated for the potential invalidating impact of these changes.**

A full copy of all the files we audited (encompassing all the versions above) as well as our final report is available at `https://stableset.com/audits/tether_audit_v1.zip`.

2.3 Threat Model and Economics

The security level and required diligence performed in the development of a contract is proportional to its value. The Tether team estimates the initial value of the contract to be in the 1-200M USD range, similar to the amount held by the Omni contract.

Several advantageous economic properties simplify the audit process for this token. Firstly, no Ether is held by the contract. The primary direct incentive of economically rational actors in attacking smart contracts is often to steal the censorship resistant currency for personal profit, an incentive not present here. Secondly, any real-world side effects of the contract with regards to the US dollars held in deposit must be manually approved by human actors, able to notice a suspicious or invalid transaction while having the time to be informed of such due to this naturally imposed delay.

State level adversaries and their capabilities are excluded from this analysis, as these adversaries can disable the system far more easily through its fundamental reliance on the fiat banking model. Despite the above mitigating factors, economically motivated and persistent cybercriminals as well as hobbyist seeking chaos may be incentivized to attack such a contract. If the fiat boundary is not properly checked for malfeasance, such actors could potentially cause monetary loss to holders of Tethered dollars by inducing withdrawals or other actions on incorrect contract state.

Such threats are generally motivated to exploit relatively visible surface-level issues or use known past antipatterns in the system. The development of a 0-day in non-Tether software for the Tether contract would be economically irrational, as other contracts holding immediately liquidateable Ether use such software and are more attractive targets for this skilled development.

Tether is further assisted in its economic security by the strong backing of a large organization, iFinex Inc. IFinex both has the resources to continue operation of Tether in the event of survivable losses and has a history of repaying investors in the event of security failures (`https://www.bitfinex.com/posts/198`).

Potentially greater than the risk of a contract breach itself is the risk of a breach, attack, regulatory obstacle, or other impediment in the functioning of Tether's USD reserves. Such an event occurred briefly during a period of problems with the fiat banking system (see `https://cointelegraph.com/news/tethers-bank-problems-create-unease-as-token-value-slides-below-1`). Economically, attacks of this form are cheap, highly likely to succeed, and potentially more devastating than direct contract failures, though the size of iFinex LLC and its track record in mitigating these issues in the past lends some confidence as to the continued operation of the Tether reserves.

3 Common Antipattern Analysis

In this section, we analyze some common antipatterns that have caused failures or losses in past smart contracts. This list includes the failures in `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/` as well as `http://solidity.readthedocs.io/en/develop/security-considerations.html`, and other literature on smart contract security and the experience of the auditor.

Upgradeability A common problem with (and often feature of) smart contracts is their inability to be upgraded, increasing transition costs when problems with deployed contracts arise or new features are required. This contract includes and tests for (as per our recommendations) an upgrade path to maintain compatibility across upgrades of **all ERC20 methods**. Any client contracts should be aware that these methods are the extent of this contract's upgradeability, though there are no other client-facing methods currently exposed that are nonupgradeable.

One important caveat comes with the upgrade process of this contract. In long chains of upgrades (eg - v0 -> v1 -> v2), a flaw in any version along the chain could potentially violate all guarantees of the contract. We therefore recommend that client contracts do not rely on the upgrade mechanism alone, and supply their own mechanism for handling potential changes to the TetherToken contract address. Relying exclusively on the built-in upgrade mechanism may cause a reduction in the security of the client contract. The upgrade mechanism is intended to give some flexibility and time for client applications to react to changes, and the associated increase in transaction fees may provide the impetus to push client applications to move to interface directly with any upgraded contracts (simultaneously reviewing their code).

Arithmetic issues SafeMath is used for all arithmetic operations in the contract, removing the possibility of overflow. Division is only used (and thus potential integer rounding issues are only present) at one location in the contract, in the fee calculation of Tether transactions. If anything, this calculation will tend to underestimate fees, and may incentivize users to break certain larger transactions into smaller transactions to avoid on-chain fees.

The use of six decimals points in the fee calculation mean that this effect will likely be rather small, with chunking transactions likely to save a user in the range of n * USD.000001 (where n is the number of chunked transactions) in fees, in the extreme case where for example a user enforces zero fees by chunking transactions below the minimum rounding level. A fee savings of .000002 associated with two-transaction chunking is likely not to substantially affect Tether, and the gas costs of a single on-chain transaction will exceed any potential savings. If anything, this policy will favor users with small transactions over large customers (though maximum fee will favor large customers), a not entirely undesireable effect for end-users. This rounding also does not affect the conservation in the token balance calculations, as with each subtraction of value tokens, sendAmount + fee tokens are added, where sendAmount is defined as value - fee.

Denial of Service / Stuck Coins This contract is not payable and thus cannot be affected by stuck coins. Otherwise, one of the only possible denial of services is in forcing throws (each throw in the contract, by inspection, occurs only during validation of inputs). Another possible denial of service relates to gas issues, which we cover next.

The final denial of service could relate to the fees inherent to the Tether service; because for small amounts, the fee will round to 0 (integer division with 1000 after dividing by a max fee of 50), there is no possible denial of service vector with regards to the fee count.

Gas Limit In general, none of the Tether contract or critical multisignature functions have loops or variable runtime control flow. Therefore, the gas use of these functions during testing (with maximum gas substantially below the block-based limit) should determine gas use in the current Ethereum system. During future hard forks, the contract should be re-evaluated for continued feasibility (though the simplicity of its functions make this almost a guarantee).

One exception to the above is the upgrade mechanism of TetherToken, which could potentially consume an arbitrary amount of gas (in a long upgrade chain, this is one call opcode per upgrade and a potentially unbounded amount of gas usage in the target contracts). We recommend, to mitigate against this issue, that one of two actions are taken:

- Implementation of automatic backpropagation for multi-upgrade chains (eg - check latest address in upgraded version, update your local to that address if not equal).
- A clearly stated policy of updating the owners on old Tether contracts to reflect any changes in the structure of the multisignature management contracts, and a policy of re-calling the upgrade functionalities of old contracts on subsequent upgrades, ensuring each historical version of the contract contains the latest address with overhead of at most one call.

The other exception to the above is the iteration through owners (or quantities upper bounded by the number of owners) in several functions of the multisignature wallet. Because these loops are bounded by the maximum number of allowable owners (50), the gas consumption should remain within the limits at all times. The same is true for getTransactionIds, which allows users to submit a range of values to explore in the for loop, allowing for a small enough range to be explored to not violate any potential foreseeable gas limits.

Misc. Issues One major calling pattern to avoid for client contracts is the splitting of function calls to Tether in a single logical procedure across transactions (eg - calling approve and later calling transferFrom). This is because the contract could have been upgraded or paused at any time between the multiple transactions submitted for a single operation by a client.

One change that would make the code substantially cleaner, more modular, and more upgradeable is the moving of the fee calculation functions to the top-level TetherToken file. Currently, the fee calculation is done independently twice in StandardToken.sol and BasicToken.sol. While the calculation has been verified and tested as working, this repetition of code minorly impacts the upgradeability of the token. Furthermore, *modifications directly to the Zeppelin library violate the boundaries drawn in the application diagram in the previous section, potentially making future Zeppelin changes more difficult to integrate and more likely to introduce unintended side effects*. While this change has been verified as not security critical, it is our belief that such a change would improve the readability, testability and modularity of the existing codebase.

An outline of such a change would be as follows:

```
  // Forward ERC20 methods to upgraded contract if this one is deprecated
  function transfer(address _to, uint _value) whenNotPaused {
    if (deprecated) {
      return StandardToken(upgradedAddress).transfer(_to, _value);
    } else {
      return super.transfer(_to, _value);
    }
  }

--- to ----
```

```
// Forward ERC20 methods to upgraded contract if this one is deprecated
function transfer(address _to, uint _value) whenNotPaused {
  if (deprecated) {
    return StandardToken(upgradedAddress).transfer(_to, _value);
  } else {
    uint256 fee = calculateFee(_value);
    balances[owner].add(fee);
    return super.transfer(_to, _value.sub(fee));
  }
}
```

(similarly from transferFrom)

(with the pattern above, the fee calculation can also be independently unit tested as a unit, and the base transferFrom and its interaction with TetherToken transferFrom can similarly be independently tested as units)

For style, consider also adding commented documentation to the setParams style, in the same style as the other functions in TetherToken.sol.

Timestamp Dependence No timestamps, block numbers or otherwise, are used anywhere in the contracts by inspection.

RNG Issues No RNGs are used in this contract by inspection.

Human in the loop Both pause/unpause and upgrade functionality, as well as an owner able to change token supply provides ample human-in-the-loop capability in the event of unforeseen contract failure. In our test recommendations, testing these fail-safes is included.

Unknown failure modes One potential cause of loss is the omission of critical failure modes from a response or testing plan. We consider the following (likely non-exhaustive) list of potential contract failures:

Failure	**Response**
An upgrade to ETH renders TetherToken inoperable	Upgrade contract
Some key loss in multisignature wallet	Remove, readd admins
Multiple party key loss in multisignature wallet	**Failure [external]**
Admins leave multisignature contract without lowering threshold	**Failure [external]**
Tether USD reserves temporarily or permanently captured	**Failure [external]**
Owner fails to move tokens to new owner, old keys lost	Upgrade contract
An unknown security bug leads to balance loss	Upgrade contract
The multisignature wallet fails to call a contract function	Change owner, try with new wallet
The multisignature wallet permanently fails to change owner	Upgrade contract
The multisignature wallet permanently fails to change owner/upgrade	**Failure [tested against]**
TetherTokens are burned or lost permanently	Policy unclear (on how to handle reserves)
Other ERC20-related code failure	Upgrade contract

We recommend specific care be paid by the Tether team to the modes that lead to contract failure, and potential contingencies be established to avoid these scenarios. In general, these failures are addressable through policy, though some are able to be excluded through testing.

Game theoretic bugs This contract does not attempt mechanism design, and thus likely doesn't suffer from direct game theoretic issues. The requirement of trust in the multisignature admins who may also be token speculators (potentially with large positions, short

or long) does present the possibility for perverse incentives, and all users engaging with the contract should be aware that trust in this group and the integrity of the banking model underlying its deposits is absolutely required for any sort of contract guarantees to hold. Furthermore, **all users must be aware that the Tether admins have the ability to impose fees of up to 2% at any time**, and must gauge their interactions with the contract accordingly.

Incorrect / missing modifiers The following custom modifiers are available to the TetherTokens code: `onlyOwner whenNotPaused whenPaused onlyPayloadSize`.

`onlyOwner` should be applied to privileged, admin-level functions callable only by the multisig wallet. This includes pausing the contract, a feature of Pausable, transferring ownership, upgrading the contract, setting new fees, and redeeming/issuing tokens, all of which have the appropriate modifier. The remainder of the methods are all public ERC20 methods or constructors, and are thus intended to be world callable by design.

`whenNotPaused` is used to ensure that no user-driven state changes are possible during a potentially dangerous period in the contract's lifecycle. The methods that modify state in ERC20 (and thus the unprivileged methods in TetherToken) are transfer, transferFrom, and approve. Of these, transfer and transferFrom, the balance altering (and thus most impactful of these) methods are guarded.

`whenPaused` should never be called; there are no methods defined by inspection that should operate only when the contract is paused (except unpause in the Zeppelin library, which is appropriately modified).

`onlyPayloadSize` should be used to ensure the correct address size on any methods which may modify an address's state in the contract. This includes both transfer methods, which are appropriately protected. The only TetherToken function that does not call into these guarded functions is deprecate, which is privileged functionality only callable by admins (whose interest it is in to ensure an appropriate payload size and test the transaction offline). An important caveat is that the forwarders for transfer and transferFrom in TetherToken themselves are not protected by the modifier, placing the burden on any upgraded implementations of these methods to check the payload size property if they want to perform this type of validation; we recommend documenting this in the Tether repo.

Visibility modifiers are not required in the contract, as all methods are either publicly callable or callable by an owner, and there are no internal or utility methods present in the contract.

Stack issues No stack issues are present in the contract. Generally, Solidity will throw on stack overflow, rendering the consequences of a maliciously crafted transaction with large stack moot. The exceptions are with sends, which silently fail and must be explicitly checked for return values. The only send in the Tether system is in the MultiSigWallet, and is appropriately checked.

Scalability The scalability of this contract is at worst that of a standard ERC20, the most popular smart contract token currently deployed. We therefore do not believe scalability issues to be an imminent threat to this contract. The multisignature contract has also, by inspection, only methods with an upper bound on gas price and therefore execution price (see previous section on Gas Limits).

Stuck Ether The TetherToken contract is not payable (with no functions marked payable and no default functions). It is thus unable to receive value, and does not permit the possibility of stuck Ether by design.

The multisignature contract is, on the other hand, payable. We see no reason for legitimate users to send money to this contract, and any stuck Ether is able to be withdrawn and

redistributed appropriately by the multisig admins (who are relied on for honest operation of the TetherToken anyway).

Stuck/Lost Tokens Tokens cannot get stuck within the boundaries of the Tether system. For a token to get stuck or lost, balances[address] must not contain the appropriate amount of tokens for some address. balances[address] is only updated in two methods for any non-owner address: transfer, and transferFrom. In each case, after the balance of a given account is reduced, the balance of an account provided as input to the functionality is increased, conserving the sum total of balances across transfers.

Like in any key-based blockchain system, there is a risk of tokens being lost through their sending to an address for which no private key is known. This risk is not possible nor desirable to mitigate in such systems, and it is up to the user to validate sending addresses. As with all ERC20 tokens, stuck tokens pose a risk if Tether tokens are accidentally sent to contract addresses which are not set up to accept them. We recommend any user of the Tether contract on-chain perform their own out of band testing to ensure that the address they are transferring to is capable of successfully spending TetherTokens.

Because of potential fee calculations, client contracts of Tether tokens cannot assume that their balance will be incremented by the full amount of a successful transfer. As previously mentioned, this could potentially cause client contracts to throw and become stuck, leading to stuck tokens. We recommend clearly communicating this risk to all client contracts.

More generally, deducting a fee within general ERC20, as Tether tokens do, can be viewed as a violation of ERC20 (though in the opinion of the preparers of this report, the standard is properly complied with). The ERC20 specification states, for example, that transfer is intended to “Send _value amount of tokens to address _to". The specification does not state that the balance of the receiver (_to) must be increased by the amount corresponding to the transferred value, though we see it as possible that some contracts may make this assumption (eg - if internally tracking the perceived balance of Tether tokens, or if performing multiple chained transfers). Care must be taken to ensure that if fees are activated, any client contracts either implicitly handle these fees correctly by only performing single transfers and making no assumptions on the receiver’s balance, or explicitly handle the possibility of such fees by querying either the feerate or the balance of the Tether contract. Because of the vague nature of ERC20, this should be standard practice for any ERC20 clients in their handling of assumptions around client tokens. The Tether team has been informed of this potential issue adequately and will handle any potential interactions with more complex client contracts.

Re-entrancy / recursive send : The Tether contract handles no Ether and makes no external calls, and is thus trivially not vulnerable to re-entrancy or recursive send by inspection. Any functions added or modified containing potential external or untrusted calls should be carefully inspected for recursive send before being released to production.

The multisig contract, on the other hand, does hold funds and is payable. There is a single method that makes an external call, as the last action in its control flow other than its log event. Aside from issuing multiple such Execution(transactionId) events, there is no potential for financial loss from the recursive send vulnerability.

4 Testing and assurance

In this section, we analyze the due diligence performed during contract development in writing appropriate tests, and suggest further potential tests that may expand logical coverage.

4.1 Recommended unit tests

Unit tests are a critical part of testing any project. The contracts described above currently have limited unit tests, which are marked below.

Generally, we recommend the following best practices for unit testing smart contracts:

- **Full coverage**: Generally, we recommend full branch, statement, and path coverage (see here for brief background). This ensures that future updates will not break the application, and every component of the code functions properly in isolation, and includes covering all inputs of a given equivalence class.
- **Boundary value analysis**: Generally, we recommend boundary tests for all potential input classes in the application. For example, for the uint256 type, when transfering a balance this means a transfer should be tested that is smaller, equal to, and greater than the balance (with equal to being the boundary between these two cases).

We suggest an expansion of unit tests as follows:

TetherToken.sol

- **Function transfer**: Test transfer from address A (with tokens) to address B (without). Check balance consistency before and after. Test with _value = UINT_MAX and _value = 0, ensure balance consistency. Test failed transfer from address with no tokens. Test transfer of more tokens than in balance, ensure failure and non-modification of balances.
- **Function transferFrom and approve**: Same tests as above. Add all tests both with and without valid approval through approve function. Check that approving a spend of 0 has no effect on allowance. Check that allowance is correctly upgraded after approval. Test that setting a balance to the maximum uint256 value does not cause allowance to be debited on successfull transfer and allows transfers. Check that resetting to a different value then debits allowance on transferFrom appropriately.
- **Function balanceOf**: Test balance of address with or without balance.
- **Function deprecate**: Check functions under contract deprecation. Test double-deprecation and ensure address is updated correctly. Test that deprecate event is in receipt with correct address after all calls.
- **Function totalSupply**: Test that totalSupply is initially 0. Issue tokens several times and ensure totalSupply grows appropriately. Test with upgraded contract.
- **Function issue***: Test that issue with UINT_MAX is not able to overflow total supply. Test that balances and total supply update normally during normal execution. Test that successful execution logs an Issue event in the receipts with correct amount.
- **Function redeem***: Test that redeem cannot be called with more than total supply. Test that redeeming total supply yields 0 tokens. Test that redeem with amount 0 has no effect. Test that redeem with amount INT_MAX throws. Test that totalSupply and balances are appropriately updated in the normal case. Test that on successful execution, Redeem event is logged in receipts with appropriate amount.
- *Suggestion*: Consider disallowing issue/redeem if contract is paused or deprecated.
- *Suggestion*: Consider a getter method for upgrade status in the API, for client contracts to easily check upgraded (or nonupgraded) contract status.
- *Suggestion*: Consider adding a payload size limiter to deprecate to guard against an invalid address gaining ownership.
- **Function setParams***: Test that no fee is applied in default transfers. Test that fee can be added by owner. Test that fee cannot exceed specified bounds. Test that neither parameter can independently exceed specified bounds (eg - excessive maximum with allowable basis and excessive basis with allowable maximum). Test that setting fee correctly affects transfer and transferFrom. Test that event is issued on setParams call with expected data values.

Functions marked with an asterisk contain the key vendor supplied logic of the token, and careful care should be paid to their implementation. All "to be addressed" tests have been manually verified as working and are thus not security critical, though automated testing would improve community assurance of correctness and ease the audit process for future upgrades.

Zeppelin Libraries The Zeppelin libraries already contain tests. Here we briefly reason about their coverage and suggest additional tests on a per-class basis:

- **SafeMath**: Full branch, conditional, and equivalence class coverage is provided. Some value-based tests are missing (eg - testing functions with 0-valued inputs), but the code is simple enough and manual inspection determines that this absence is not a security risk to TetherToken. Furthermore, these cases are implicitly tested by the recommended transfer tests for TetherToken.
- **Pausable**: Full branch, conditional, and equivalence class coverage is provided. Some more complex scenario tests are missing, eg - ensure old owner cannot pause/resume after owner change, test multiple owner changes, etc. Again, the code is simple enough and manual inspection determines that this absence is not a security risk to TetherToken, though elsewhere we did recommend tests that would cover these scenarios in testing the interaction between Pausable and TetherToken.
- **Ownable**: Full branch, conditional, and equivalence class coverage is provided, including checks on address length for transferOwnership to ensure ownership is not stuck (though this does not ensure ownership is not transfered to an account with a known public key, which is impossible to test for). Some coverage is missing, eg - ensure old owner can no longer transfer ownership after their ownership has been deprecated. Again, the code is simple enough and manual inspection determines that this absence is not a security risk to TetherToken, though elsewhere we did recommend tests that would cover these scenarios in testing the interaction between Ownable and TetherToken.
- **ERC20 / ERC20Basic**: This is exclusively an interface class, and by inspection implements the correct set of ERC20 methods. No testing is recommended or implemented for these classes.
- **BasicToken / StandardToken**: These two classes combined implement the full core ERC20 functionality. While coverage is relatively complete, the following items are missing:
  - Testing the code path in approve that does not allow non-zero approval with an existing non-zero allowance. Furthermore, the documentation does not sufficiently warn contract implementers to use the "set to 0, check, then reset" pattern for more than one approval operation.
  - transferFrom should also check for the ability to transfer more than balance when allowance is correctly set.
  - Tests with zero values are generally missing, though manual inspection shows that the class's guarantees will not be violated with such values. Interestingly, the transferFrom and transfer methods impose a payload size restriction representing such a check, for which a test is missing. The allowance method does not feature this modifier, allowing for the creation of transfer approvals that can never be executed. This does not seem to be a substantial security risk in the contract.
  - Tests with zero-length (null) addresses are generally missing, though manual inspection shows that the class's guarantees will not be violated with such addresses.

We of course recommend that the Tether team run the full Zeppelin test suite before each update to the Zeppelin libraries.

The interactions with each Zeppelin class and the core TetherToken contract are enumerated, with testing suggestions for ensuring the relevant inherited functionality is correct:

- **SafeMath**: The TetherToken contract is exposed to SafeMath through BasicToken. The functionality in BasicToken that uses SafeMath is only transfer, which is already tested for in the recommended tests for the transfer function (as well as in the Zeppelin library itself). No further tests are recommended.
- **Ownable / Pausable**: The primary exposure of the TetherToken contract to Ownable is in the onlyOwner modifier used in the contract. This modifier is applied to the deprecate, issue, setParams, and redeem functions. We recommend tests testing that the non-owner of the contract cannot call any of these functions. (complete for issue) The testing of the correct path is already included in the TetherToken recommendations. We also recommend testing transferOwnership, ensuring that after an ownership transfer the old owner is not able to call any of the above functions, and that the old owner cannot transfer ownership after they are no longer owners.
  The TetherToken contract is also exposed to Ownable through Pausable, which only allows owners to pause contracts. We recommend testing the interaction of all these components with a test that has two accounts, A and B. Initially, A is the owner and pause -> resume is tested (ensuring that no methods in TetherToken annotated with whenNotPaused can be used). Then, ownership is transferred to B while paused. A should not be able to pause, and B should be able to resume the contract.
- **ERC20 / ERC20Basic**: This is exclusively an interface class, and by inspection implements the correct set of ERC20 methods. No further testing is recommended for these classes, as TetherToken tests already cover correct implementation of ERC20. It is worth noting that ERC20Basic distributed with Tether has the totalSupply variable renamed to _totalSupply, to avoid shadowing by overridden methods in TetherToken. This change is inconsequential for security and has been applied to all instances of totalSupply, as manually verified.
- **BasicToken / StandardToken**: Basic and StandardToken implement all of the core ERC20 functionality that is overridden by the upgraded methods in TetherToken. The tests we recommended for TetherToken are already exhaustive with regards to the core ERC20 functionality, so we do not recommend any further tests.

Migrations.sol and UpgradedTokenTest.sol Because these contracts have no bearing on the production deployment of the contract, are not executed by any codepaths in TetherToken, and in the case of Migrations represents boilerplate contracts that are widely used in the ecosystem, we do not necessarily recommend testing these contracts.

TetherToken is manually inspected against side effects from Migrations and non-inclusion of UpgradedToken, and these sanity checks should be briefly performed across updates.

4.2 Recommended integration tests

In addition to unit testing, we recommend several end-to-end tests. Ideally, these tests would be executed on a testnet or live version of a blockchain system, and their results verified with multiple consensus clients.

Multisig-Contract interaction We recommend the following tests for interactions between the multisignature contract:

- Attempt a two-owner issue with the approval of both owners
- Attempt a two-owner issue with one owner approving twice (should fail)
- Attempt a two owner issue → add admin → redeem with one old owner and the new owner approving.
- Attempt a two-owner issue with one owner confirming after removal. Attempt the same issue with one owner confirming → that owner being removed → another owner confirming. Both should fail.

- Attempt changing the owner of the TetherToken using a two-owner confirmation, which should succeed. Attempt an issue using both the old and new multisignature contracts. The former should pass, the latter should fail.
- Attempt a contract upgrade and pause using the multisignature wallet, confirm that both have the intended effect.
- Attempt a fee change with the multisignature contract, ensuring that fees are processed, events issued, and owner balance is updated appropriately.

It is also worth manually ensuring that no bypass of the required number of admin confirmations is possible (such a pattern was present in v0, which featured a daily withdraw limit that if set to 0, would allow any admin to bypass the required number of confirmations; this has been resolved by switching from the Parity to Gnosis multisig wallets).

4.3 Regression testing and testing policy

We recommend the following test policies formalized in the Tether Github, and enforced rigorously (both internally and by outside review):

- Code reviews, including one mandatory in-depth review and signoff. Community review periods also recommended if community interest in the contract arises.
- Mandatory full branch, statement, and path coverage on all committed contract code, following the standards of the test recommendations in this report (including boundary value and equivalence class analysis).
- Regression tests for any bugs discovered during development not covered by a test.

Because smart contracts, unlike most software, are designed to be rarely or never upgraded, these standards should suffice.

5 Code walkthrough

In this section, we review the code written by the Tether team in detail.

5.1 Data invariants - TetherToken.sol

- **string public name:** unmodified after initialization in constructor
- **string public symbol:** unmodified after initialization in constructor
- **uint public decimals:** unmodified after initialization in constructor
- **address upgradedAddress:** Only settable by owner in deprecate function, not settable to None once set
- **bool deprecated:** Set on any call of deprecate function by owner, not settable to false once set
- **uint[256] basisPointsRate:** [added to BasicToken] Basis point feerate (/1000) allowed to be charged by Tether; must be below 20.
- **uint[256] maximumFee:** Maximum fee in Tether tokens; must be below 50 for USDTether.

5.2 Function-level comments

- **Constructor**: Correctly initializes _totalSupply variable (ERC20Basic.sol), name, symbol, decimals, and allocates the initial tokens to the contract owner. Deprecated is set to false, and upgradedAddress is left to its default initizalization value (and used nowhere in the code). This covers all global variables by inspection.

- **transfer / transferFrom / balanceOf / approve / allowance**: These methods all follow the same pattern; if (deprecated) [only settable by the deprecate method by inspection], construct a new StandardToken (full ERC20 interface) object from the new address and proxy all calls, forwarding any arguments. Otherwise, the super class's function is called, implemented in StandardToken or BasicToken. Each execution path contains only one line of code, trivial to manually verify for correctness.
- **transfer/transferFrom [Zeppelin modifications]**: Modifications are made and tested to withdraw fees as described elsewhere in this document. Modifications to transferFrom are also made to allow for an unlimited approval, never debiting allowance if the allowance is set to the maximum 256-bit integer. The security risk of this change is low, as the contract has been manually checked against overflows that could accidentally trigger this condition. However, this change is potentially not ERC20-compliant (depending on the reading of the specification), and the implications of this should be communicated to clients, which should not assume that all transferFrom calls will debit allowance in all cases. The purposes for making such an assumption are unclear, making the security risk of making this assumption incorrectly/accideentally low. The potential impact is also low, with clients potentially assuming a lower allowance than stored in the Tether contract (which in a properly designed client would only cause an earlier re-call to approve, wasting gas but not leading to lost funds or security issues).
- **deprecate**: Two data items must be upgraded; deprecated, set to true, enabling proxying of all the ERC20 methods defined above, and correctly setting the address to the upgraded contract. After deprecation, no ERC20 functionality can be called with code specified in this contract, and no receipts are generated, as all code to generate the relevant log events is in the disabled ERC20 functionality overridden by TetherToken.
- **totalSupply**: This function follows a similar upgradeable pattern as the above, but returns the `_totalSupply` variable as specified for tracking total supply in ERC20Basic.
- **issue**: This function is responsible for creating new Tether tokens, a process that requires updates to two data members desribed above, balances and _totalSupply. The balance of the owner and totalSupply are both incrased by the creation amount in this function. Overflow is checked for correctly: if an amount (up to INT_MAX) overflows either variable, the sum will be smaller than the current amount, up to at most one less than the current amount. Though it may have been simpler to use the built-in SafeMath functionality of Zeppelin for this check, they are equivalent by inspection and make no difference to the security of the contract. The relevant event is also emitted with the correct amount post-input validation.
- **redeem**: Identical to above but with subtraction and the redeem event, with correct overflow protection (allowing only values smaller than or equal to the current balance and totalSupply to be subtracted).
- **setParams**: Simple setter method that correctly checks data invariant on fee basis / maximum fee as specified above, and is tested to enforce these invariants. Math is overflow protected using safe multiplication, and multiplies by number of decimals to allow specification of maximum fee in whole units (thus converting to the base amounts specified in the contract).

It is worth noting that in earlier versions of the contract, the maxFee invariant was violated (with the fee ceiling expressed as the maxFee [in base units] multiplied by decimals instead of $10^{decimals}$). The tests were correspondingly incorrectly written, causing the issue to be missed in review and noticed later by BFX developers. While this issue was not security critical and could not lead to funds loss (except for Bitfinex, whose collected max fees would have been lower than anticipated), this highlights the importance of multiple independent parties analyzing and testing code. The new tests were manually inspected for correctness, with a maxFee of 5 appropriately allowing for a maximum fee of $5 * 10^{decimals}$ (an amount that would have yielded a larger fee is tested for appropriate cap). The code

was also checked against an independent pen-and-paper calculation of all the constants included in the tests.

6 General recommendations

In this section, we provide some general recommendations (not specific to implementation details) that we recommend for more assurance in the contract.

6.1 Bounty program

We always recommend a bounty program to repair the perverse incentives of an attacker's ability to profit off breaking and not defending a contract. A sane and publicly announced bug bounty of the funds used in approximately a year of security expenses is always recommended. We further make the special recommendation for the Tether team that a bounty can be offered on https://wallet.tether.to/transparency, with a reward for any vigilant member of the public able to show a discrepancy between https://wallet.tether.to/transparency and any real-world data (either contract or audit).

We recommend that all users of the Tether system continue to monitor and demand the maximum possible level of transparency from the fiat banking infrastructure behind the contract, and invest understanding this as a potential point of failure.

6.2 Verifiable compilation

We recommend that, to assist in community audit, the Tether team includes a README with the contract source on reproducing the exact bytecode deployed on chain (including appropriate version of the Solidity compiler to use). We then recommend that any security-conscious users of or investors in the system audit the on-chain binary, ensuring it faithfully compiles the source.

We also recommend submission of the source to potential sources of third party verification, such as EtherScan's contract source functionality.

6.3 Software analysis tools

Unfortunately, the Solidity software quality tools infrastructure is still immature. The only usable tools we currently recommend for the desired security level of this contract (where formal verification is currently still in the R&D phase in the ecosystem and cannot be cost effectively applied) are `https://github.com/raineorshine/solgraph`, the static analysis feature of Solidity's browser-remix IDE, the Melonport Oyente tool (`https://oyente.melonport.com/`), and the Securify static analysis tool (`http://securify.ch/`).

6.4 Documentation

We recommend the Tether team create developer/user documentation including the client recommendations we provide in this document.

6.5 Privacy concerns

Obvious privacy concerns apply to the Tether token, with the transaction graph remaining fully public and providing no attempts at obfuscation (with privacy implications described in this presentation among many other sources).

We recommend that any entry points to the Tether system warn their users that Tether tokens are potentially traceable by resourced entities able to access either exchange data, KYC data, or other sources of personally identifying information. Similarly, exchanges participating in the Tether system, as well as the multisig operator contract all have traceable balances (though in this case the property is desireable for audit).

6.6 Contract failure management

Despite the best efforts of this author and others towards assuring the security of the underlying contract, failure is still a possibility. We recommend any infrastructure that depends on this contract:

- Check for transaction failures at each call of the contract and handle them gracefully (in this contract, generally represented by throwing an exception, though functions should always be checked for status return value if one exists).
- Include graceful handling of a paused contract, and test these code paths with a mock paused deployment of the Tether contract.
- Include a failsafe in the event of catastrophic contract failure (in the form of mechanism to disable or upgrade contract address).
- Check whether the contract is deprecated before making any calls. A deprecated contract may have been replaced with arbitrary code, and we recommend that clients of the contract reverify any new versions. This will ensure that no required API contracts are violated, and no breaking changes are introduced to the client. Clients requiring lower assurance may choose to skip this re-verification at their own risk.
- For maximum assurance (if using results of contract functions for large irreversible balance transfers or similar), a grace period may be implemented between receiving and actioning on contract results. This grace period would allow any obvious and major problems with the contract to be noticed by humans and handled, giving time to invoke the failsafe mentioned above. Alternatively, a legal or out of band remediation process (or lack thereof) can be formalized between the relevant entities.
- The security of any private keys and accounts they represent is not guaranteed by the Tether contract, and is thus outside the scope of this audit. We recommend Tether make clear that it provides no guarantees in the event of key storage failures.
- Be aware of potential fee calculations in making any client assumptions, as detailed in the “Stuck Tokens" portion of this audit; failure to do so may result in permanent loss of Tether tokens.

We recommend these stipulations be made explicit to clients of the contract, in API documentation and private communication with important contract users. We recommend the Tether team to check that these failsafes have been appropriately implemented in any high value contracts whose failure may affect the operation or reputation of the main contract.

Because the contract does not hold any value (and the only ability to move the tokens it backs require human and organizational intervention by Tether Limited), the possibility for direct funds loss in the event of catastrophic contract failure is low. On the other hand, **contract failures may cause client applications to receive incorrect data**. If these applications move value conditional on this data, there is still the potential for substantial loss of funds.

6.7 Versioning Scheme

We recommend the implementation of a basic versioning scheme, with the first release versioned at 1. Future redeployments of this contract should clearly specify their version (both in source files and as a complete distribution with verifiable compilation).

7 Conclusion

We conclude that if the recommendations herein are followed, the Tether contracts will launch in a well tested, secure state. The contracts appropriately lean on standard, tested libraries to implement their core standard functionality, and properly test the interactions of all their subcomponents with their contract code. The contracts are not vulnerable to any

classically known antipatterns. Their simplicity, lack of storage of decentralized cryptocurrency, and intervention mechanism with humans in the loop make the risk of this contract likely to be low.

We recommend that the team performs all our recommended integration tests post deployment, and we recommend that investors audit the correctness of the deployed bytecode as per our recommendations. The primary risk to this contract is in the multisignature key storage and trust required in the administrators of the contracts to maintain appropriate fiat balances. The former risk is mitigable through a diverse set of stakeholders experienced with holding keys, and the latter is mitigable with thorough and frequent public audit.

Of course, we re-emphasize that this report is not necessarily a guarantee of correctness or contract performance, and always recommend seeking multiple opinions (especially during subsequent upgrades to any Tether system component, or as the system grows in monetary value beyond what was expected in the creation of this report).

8 Feedback

Smart contracts are a nascent space, and no perfect security audit procedure has thus far been perfected for their deployment. We welcome any suggestions or comments on this report, its contents, our methodology, or potential gaps in coverage via e-mail at `feedback at stableset.com`.

Report Versions

```
v1 - update for changes adding fees + fee calculation error [current]
v0 - initial release
```