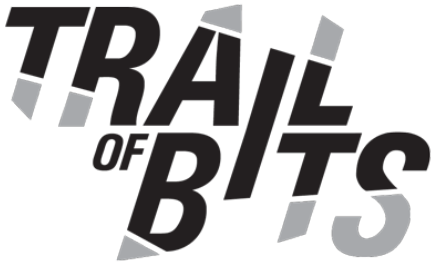

Ampleforth

Ethereum Protocol Security Assessment

November 20th, 2018

Prepared For:
Brandon Iles | *Ampleforth*
brandon@ampleforth.org

Prepared By:
Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

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

Changelog:

November 20th, 2018:	Initial report delivered
December 17th, 2018:	Updates and name change from µFragments to Ampleforth

Executive Summary

Between November 5th and 20th, 2018, Trail of Bits assessed the smart contracts of the Ampleforth[1] Ethereum protocol codebase. Two engineers conducted this assessment over the course of four person-weeks. We evaluated the code from commit ID 888fccaf05786f3f7f49e18ff040f911d44906f4 of the `market-oracle` git repository, and commit ID 07437020b54c535ced2f4b5f1a0cc1a2ee6618e3 of the `uFragments` repository, reflecting the state of the project as of about October 8th, 2018.

The first week consisted of the engineers familiarizing themselves with the codebase, running static analysis tools such as Slither, and manual code inspection. The second and final week concluded our manual analysis of the Solidity code. We extracted a set of security properties by studying the codebase and communicating with the developers, then encoded those properties into tests for Echidna and Manticore. See the appendices for related discussion.

Several of our findings pertain to mishandling of edge cases in market oracle output. These can cause rebasing to fail to self-stabilize the token, either due to rare but plausible natural causes, or due to a malicious or erroneous market source. They can also cause erroneous events to be emitted. One finding concerns the use of a deprecated version of a library. This appears to have been addressed in a subsequent version of the code than what we assessed. The two final findings relate to the potential for catastrophic failure during contract upgrading, as well as the possibility of arbitrage due to rebasing predictability.

In addition to the security findings, we discuss code quality issues not related to any particular vulnerability in Appendix B. A few additional appendices are also provided for guidance on operations and deploying the off-chain portion of the codebase.

The Ampleforth ERC20 token appears to be vulnerable to a well-known race condition vulnerability inherent to the ERC20 specification itself. The token already implements one of our suggested mitigations. We have included Appendix C to provide background on the issue as well as offer additional mitigations.

[1] *µFragments* was rebranded as *Ampleforth* subsequent to our assessment but prior to the finalization of this report. The report has been modified such that all references to the company/product “µFragments” were replaced with “Ampleforth”. However, all references to source code artifacts (*e.g.*, smart contract names) remain as they were in the assessed version of the codebase.

Engagement Goals & Scope

The goal of the engagement was to evaluate the security of the Ampleforth protocol and smart contracts and answer the following questions:

- Can attackers use leverage within the system to undermine the stability of the currency?
- Does the design of the system introduce any risks at the architectural, code dependency, or contract levels?
- Do the contracts perform calculations on Gons and Fragments correctly? Is there a possibility of integer underflow, overflow, or rounding errors?
- Are there any issues with the contract upgrade mechanism?
- What is Trail of Bits's guidance on deploying and operating the off-chain portions of the codebase?

Coverage

We reviewed the `UFragments` ERC20 token and the `MarketOracle` contracts. This included all of Ampleforth's on-chain code and Solidity smart contracts. Off-chain portions of the codebase such as the exchange rate feed were not covered in this assessment.

Contracts were reviewed for common Solidity flaws, such as integer overflows, re-entrancy vulnerabilities, and unprotected functions. Furthermore, contracts were reviewed with special consideration for the complex arithmetic calculations performed in the token as well as the bespoke integer arithmetic library implementation used by the Ampleforth token contract. Special care was taken to ensure that there was no possibility for loss of funds due to arithmetic errors (*e.g.*, overflow, underflow, or rounding) or logic errors.

Project Dashboard

Application Summary

Name	Ampleforth Protocol
Type	ERC20 Token and Protocol
Platform	Solidity

Engagement Summary

Dates	November 5th through 20th, 2018
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High Severity Issues	0	
Total Medium Severity Issues	0	
Total Low Severity Issues	4	■■■■
Total Informational Severity Issues	2	■■
Total Issues of Undetermined Severity	1	■
Total	7	

Category Breakdown

Configuration	1	■
Data Validation	1	■
Patching	2	■■
Undefined Behavior	3	■■■
Total	7	

Recommendations Summary

Short Term

❑ **Gracefully handle rebasing when no market sources are fresh.** For example, leave the exchange rate unchanged and solely apply the damping factor.

❑ **Mitigate the effect of malicious or erroneous market sources.** Consider capping the reported exchange rate to that value in the `volumeWeightedSum` calculation. Also, consider changing the value returned by `MarketSources` to be a `uint128`.

❑ **Upgrade to a newer version of ZeppelinOS as soon as possible.** Zos-lib is deprecated. This appears to have happened concurrently to this assessment. Confirm that all usage of the old Zos-lib has been removed.

❑ **Prevent reentrancy in market sources.** Prevent the minimum rebasing time from being zero. Ensure that rebase times are strictly increasing.

❑ **Document market source removal.** State all assumptions made by `removeSourceAtIndex`, including the requirement that `index` always be strictly less than `_whitelist.length`.

❑ **Document smart contract upgrade procedures.** Record the version of Solidity used for the initial deployment and ensure that that same version of Solidity is used for *all* future deployments. Implement all of the bullet points in the recommendations section of our contract upgrade anti-patterns blog post.

❑ **Include a diversity of market sources.** Ensure that markets like $\delta Y/\delta X$ and Compound that allow for margin trading are included as market sources.

Long Term

❑ **Investigate methods of calculating the aggregate volume that are less sensitive to individual market sources' output.** Consider having `MarketSources` push their updates to the `MarketOracle` at whatever rate they choose rather than relying on the `MarketOracle` to poll. This would help resolve several findings.

❑ **Improve unit test coverage.** Edge cases like that of finding TOB-FRAG-001 could have been revealed in testing.

❑ **Revisit contract upgradability.** Consider switching to a different contract upgrade pattern, such as contract migrations.

❑ **Research the incentives produced by having a predictable rebase mechanism that is susceptible to arbitrage.** Consider implementing more nuanced economic simulations with agents that are capable of exploiting arbitrage.

❑ **Consider additional ERC20 race condition mitigations.** Improve documentation for end-users to educate them about the ERC20 approve race condition.

❑ **Integrate advanced security tools into your secure development lifecycle.** Slither as well as custom Echidna and Manticore scripts have been provided along with this report.

Findings Summary

#	Title	Type	Severity
1	Rebasing will fail if no market sources are fresh	Undefined Behavior	Low
2	Malicious or erroneous MarketSource can break rebasing	Data Validation	Low
3	Zos-lib is deprecated	Patching	Informational
4	Possible reentrancy if the minimum rebase interval is zero	Undefined Behavior	Low
5	Market source removal is dangerous	Undefined Behavior	Informational
6	Contract upgrades can catastrophically fail if the storage layout changes	Patching	Low
7	Rebase predictability may make Ampleforth a target for arbitrage	Configuration	Undetermined

1. Rebasing will fail if no market sources are fresh

Severity: Low
Type: Undefined Behavior
Target: `market-oracle/contracts/MarketOracle.sol` and `uFragments/contracts/UFragmentsPolicy.sol`

Difficulty: Low
Finding ID: TOB-FRAG-001

Description
If no market oracles are fresh, then `getPriceAnd24HourVolume()` on line 57 of `MarketOracle.sol` will revert due to division by zero.

```
    uint256 volumeSum = 0;
    uint256 partialRate = 0;
    uint256 partialVolume = 0;

    bool isSourceFresh = false;

    for (uint256 i = 0; i < _whitelist.length; i++) {

        (isSourceFresh, partialRate, partialVolume) = _whitelist[i].getReport();

        if (!isSourceFresh) {

            emit LogSourceExpired(_whitelist[i]);

            continue;

        }

        volumeWeightedSum = volumeWeightedSum.add(partialRate.mul(partialVolume));

        volumeSum = volumeSum.add(partialVolume);

    }

    // No explicit fixed point normalization is done as dividing by volumeSum normalizes

    // to exchangeRate's format.

    uint256 exchangeRate = volumeWeightedSum.div(volumeSum);
```

***Figure 1.1**: Division by zero in `getPriceAnd24HourVolume()`*

This function is called when rebasing; therefore, rebasing will fail if there is not a fresh oracle. If rebasing fails then the damping factor will not be applied (see `UFragmentsPolicy.sol` line 82).

Exploit Scenario
There is no trading volume over the past 24 hours reported by any market oracle; so there are no fresh oracles. This can happen either naturally or, for example, if the token is paused but rebasing is not paused. This will cause a revert during rebasing, the damping factor will not be applied, and the `UFragments` contract will fail to self-stabilize.

Recommendation
In the short term, add a check during rebasing to gracefully handle this situation. For example, leave the exchange rate unchanged and solely apply the damping factor.

In the long term, improve unit tests to cover edge cases such as this.

2. Malicious or erroneous MarketSource can break rebasing

Severity: Low
Type: Data Validation
Target: `market-oracle/contracts/MarketOracle.sol`

Difficulty: Low
Finding ID: TOB-FRAG-002

Description
If `MarketSource` i ever reports a `partialRate`, r_i, and `partialVolume`, v_i, such that

$$r_i \times v_i \geq 2^{256} - \sum_{j \neq i} r_j \times v_j,$$

then the incremental summation of `volumeWeightedSum` (*q.v.* Figure 1.1) will cause a revert due to integer overflow.

This is called within `rebase`, like issue TOB-FRAG-001.

Similarly, this issue can occur if a `MarketSource` ever reverts when polled by the `MarketOracle`.

This finding is classified as having low severity because once a malicious or erroneous market source is detected, it can be removed from the whitelist. However, this would require continuous monitoring and action on the part of the owner.

Exploit Scenario
A market source returns a very large value for `partialRate` and/or `partialVolume`. This causes a revert in the calculation of `volumeWeightedSum` and thereby prevents rebasing. Self-stabilization through rebasing will not occur until the offending market source is removed from the whitelist.

Recommendation
The maximum exchange rate is hard-coded to roughly 2^{80} in the monetary policy, so in the short term consider capping the reported exchange rate to that value in the `volumeWeightedSum` calculation. Also, consider changing the value returned by `MarketSources` to be a `uint128`.

In the long term, investigate methods of calculating the aggregate volume that are less sensitive to individual market sources' output. For example, consider having `MarketSources` push their updates to the `MarketOracle` at whatever rate they choose rather than relying on the `MarketOracle` to poll.

3. Zos-lib is deprecated

Severity: Informational
Type: Patching
Target: `uFragments`

Difficulty: Low
Finding ID: TOB-FRAG-003

Description
The ZeppelinOS Library (`zos-lib`) was recently deprecated. Users should migrate to the `zos` library.

This finding is classified under informational severity because there are no known `zos-lib` vulnerabilities exercised in the uFragments code, and we have learned that uFragments will soon be migrating away from this deprecated library.

Exploit Scenario
A vulnerability in `zos-lib` is discovered but goes unpatched because it has been deprecated.

Recommendation
Upgrade to a newer version of ZeppelinOS as soon as possible.

4. Possible reentrancy if the minimum rebase interval is zero

Severity: Low
Type: Undefined Behavior
Target: `uFragments`

Difficulty: Medium
Finding ID: TOB-FRAG-004

Description
If the minimum rebase time interval (`_minRebaseTimeIntervalSec`) is set to zero, then a market source can reentrantly call `UfragmentPolicy.rebase`. This will cause the rebase to occur twice, but the second time with an epoch which is *lower* than the first.

The severity of this finding is classified as low because it does not appear to result in any security vulnerabilities *unless* an external observer depends on the monotonicity of epoch events.

Exploit Scenario
`_minRebaseTimeIntervalSec` is set to zero seconds. A rebase is initiated, which executes:

```
(exchangeRate, volume) = _marketOracle.getPriceAnd24HourVolume();
```

leading to a call `getReport` on each source. Alice's source's implementation of `getReport` makes a reentrant call to `UfragmentPolicy.rebase`. As a result,

```
_uFrags.rebase(_epoch, supplyDelta)
```

```
(UfragmentPolicy.rebase(epoch, supplyDelta))
```

is called two times, but the second time with an epoch which is less than the first. This will cause the `LogRebase` events to be emitted with epochs out of sequence.

Recommendation
In the short term, this vulnerability can be addressed by preventing `_minRebaseTimeIntervalSec` from being zero. In addition, ensure that rebase times are strictly increasing by changing the inequality in

```
require(_lastRebaseTimestampSec.add(_minRebaseTimeIntervalSec) <= now);
```

to

```
require(_lastRebaseTimestampSec.add(_minRebaseTimeIntervalSec) <  now);
```

in `UfragmentPolicy.rebase`.

In the long term, consider having `MarketSources` push their updates to the `MarketOracle` at whatever rate they choose rather than relying on the `MarketOracle` to poll, similarly to the recommendation from TOB-FRAG-002.

5. Market source removal is dangerous

Severity: Informational
Type: Undefined Behavior
Target: `market-oracle/contracts/MarketOracle.sol`

Difficulty: Low
Finding ID: TOB-FRAG-005

Description
The `removeSourceAtIndex` private function makes a number of assumptions about the state of the contract. If it is ever called when any of these assumptions are invalid, then there will be serious security implications.

```
/**
  * @param index Index of the MarketSource to be removed from the whitelist.
  */
  function removeSourceAtIndex(uint256 index)
      private
  {
      emit LogSourceRemoved(_whitelist[index]);
      if (index != _whitelist.length-1) {
          _whitelist[index] = _whitelist[_whitelist.length-1];
      }
      _whitelist.length--;
  }
```

***Figure 5.1**: Unchecked index argument and whitelist size.*

The `removeSourceAtIndex` function assumes that `_whitelist` is non-empty. If it is called when `_whitelist` is empty, the `_whitelist` array length will silently underflow.

Similarly, removing a source with an index greater than or equal to `_whitelist.length` will silently remove the last element in the whitelist.

All current usage of this function appears to be safe, which is why this finding has informational severity.

Exploit Scenario
Underflowing the whitelist length allows anyone with write access to `_whitelist` to overwrite *any* storage address within the contract. A future refactor of the code could easily expose this vulnerability.

Recommendation
Document this behavior in the function documentation string, and/or by explicitly adding a check, *e.g.*, with `require(index < _whitelist.length)`.

6. Contract upgrades can catastrophically fail if the storage layout changes

Severity: Low
Type: Patching
Target: All upgradable contracts

Difficulty: Low
Finding ID: TOB-FRAG-006

Description
The contracts in Ampleforth use the ZeppelinOS library for upgradability. Due to the way in which the library implements upgrades, the storage layout of the contracts must not change between deployments. Unfortunately, the Solidity compiler can and does often change its storage layout between versions. Any change in the state variables (new variables, changes of type, *&c.*) will require a thorough assessment before upgrading. Extreme care must be placed in implementing inheritance, as it may also affect the storage layout.

This finding does not represent a current vulnerability in the code. However, a mismanaged upgrade can easily and immediately lead to a broken contract, constituting a high-severity issue. This finding is classified as having low severity because Solidity does not have a good track record of being backward compatible, and it is becoming increasingly hard to install older versions of the compiler.

Exploit Scenario
A newer version of `solc` is used to compile a contract upgrade, causing the storage layout to change. This will cause the contract to silently, catastrophically fail upon upgrade.

Recommendation
In the short term, document this vulnerability in the Ampleforth upgrade procedures. Also record the version of Solidity used for the initial deployment and ensure that that same version of Solidity is used for *all* future deployments. Implement all of the bullet points in the recommendations section of our contract upgrade anti-patterns blog post.

In the long term, consider switching to a different contract upgrade pattern, such as contract migrations.

7. Rebase predictability may make Ampleforth a target for arbitrage

Severity: Undetermined
Type: Configuration
Target: the Ampleforth token

Difficulty: Medium
Finding ID: TOB-FRAG-007

Description
There are increasingly many options for traders to speculate on and profit from swings in ERC20 token values. Exchanges like δY/δX offer instruments for margin trading and short-selling. Since a third-party observer can almost perfectly predict the exchange rate reported by the market oracle, *and* since the rebasing process is deterministically predictable, then anyone can predict the value Ampleforth will have after a rebase. While the Ampleforth whitepaper does make an argument against high-frequency rebasing (*cf.* section 8.3), there is no discussion of the implications of allowing people enough time to exploit arbitrage between rebasings given that the outcome will be deterministic.

Exploit Scenario
Alice queries Ampleforth's market oracles and determines that the next rebase will drastically decrease the value of Ampleforth. Therefore, she borrows Ampleforth tokens from a market like Compound, immediately sells them, and then buys them back at a lower price after the next rebase. Granted, this specific market action will likely cause the value of Ampleforth to stabilize, disincentivizing further arbitrage. However, the macro effects of the incentive and ability for arbitrage do not appear to be well understood.

Recommendation
In the short term, ensure that markets like δY/δX and Compound that allow for margin trading are included as market sources. Requiring the market sources to push their updates rather than having the market oracle poll them—as recommended in TOB-FRAG-002 and TOB-FRAG-004—might also help to prevent the predictability of rebasing, since third parties would not necessarily be able to query the market sources directly.

In the long term, further investigate and model the incentives produced by having a predictable rebase mechanism that is susceptible to arbitrage. For example, consider implementing more nuanced economic simulations with agents that are capable of exploiting arbitrage.

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
Arithmetic	Related to arithmetic calculations
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's 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.

General Recommendations

- **Follow the Solidity naming convention guide.** Following a standard naming convention helps the review of the code. In addition, most style guides that advocate for prepending an underscore before certain variables do so for *private* variables. However, there are several instances in the Ampleforth codebase in which a leading underscore is used for a *public* variable:
  - `MarketSource`: `_name` and `_reportExpirationTimeSec`
  - `MarketOracle`: `_whitelist`
  - `UFragmentsPolicy`: `_uFrags`, `_marketOracle`, `_deviationThreshold`, `_rebaseLag`, `_minRebaseTimeIntervalSec`, `_lastRebaseTimestampSec`, `_epoch`
  - `DetailedERC20`: `_name`, `_symbol`, `_decimals`
  - `UFragments`: `_monetaryPolicy`, `_rebasePaused`, `_tokenPaused`

  We presume this convention is to indicate variables that are only used internally, but have public visibility for some reason (*e.g.*, debugging or transparency). If this convention is to stand, it should be memorialized somewhere in the repository so future developers can maintain consistency.

uFragments/contracts/UFragments.sol

- **Incorrect naming in comments.** The `approve` function's documentation string mentions that two other functions, "`increaseApproval`" and "`decreaseApproval`", should be used instead, while their real names are actually "`increaseAllowance`" and "`decreaseAllowance`".

C. ERC20 approve race condition

The Ampleforth token is exposed to the well-known ERC20 race condition. Code comments and the existence of allowance incrementing and decrementing functions suggest that at least one of the developers is aware of this vulnerability. This appendix reviews the issue, describes the impact, and outlines mitigations.

Issue Description
Ampleforth conforms to the ERC20 token standard, which contains an unavoidable race condition. Ampleforth's compliance with ERC20 inherently introduces this race condition. This race condition is only exploitable by sophisticated attackers, but could result in loss of funds for Ampleforth users.

It is not a smart contract correctness bug, but rather a consequence of the API design and Ethereum's unique execution model. The bug is quite subtle and difficult to understand. Normally, people think of the transaction model as completely separate from the code it executes, but this bug requires a nuanced understanding of their interaction to precisely understand its impact.

Specifically, the ERC20 standard requires two functions, `approve` and `transferFrom`, which allow users to designate other trusted parties to spend funds on their behalf. Calls to any Ethereum function, including these, are visible to third parties prior to confirmation on-chain. In addition to these calls' visibility prior to confirmation, a sophisticated attacker can "front-run" them and insert their own transactions to occur *before* the observed calls.

The `approve` function is defined to take an address and an amount, and set that address's "allowance" to the specified amount. Then, that address can call `transferFrom` and move up to their allowance of tokens as if they were the owners. Here's the issue: `approve` is specified to be idempotent. It sets the approval to a new value regardless of its prior value, it doesn't modify the allowance.

Exploit Scenario
In a scenario where a malicious party is approved for some amount and then the approving party wants to update the amount, the malicious party could end up with significantly more funds than the approving party intended.

Suppose Alice, a non-malicious user, has previously approved Bob, a malicious actor, for 100 Ampleforth tokens. She wishes to increase his approval to 150. Bob observes the `approve(bob, 100)` transaction prior to its confirmation and front-runs it with a `transferFrom(alice, bob, 100)`. Then, as soon as the new approval is in, his allowance is set to 150 and he can call `transferFrom(alice, bob, 150)`.

In this scenario, Alice believes she's setting Bob's allowance to 150, and he can only spend 150 tokens. Due to the race condition, Bob can spend 250. This is effectively a theft of tokens. Bob can then use these stolen tokens at an exchange that accepts Ampleforth. Even if Ampleforth directly modifies balances to refund Alice her tokens, the participating exchange is left with the liability since Bob has already traded the Ampleforth tokens for another cryptocurrency.

Likelihood of Exploitation
As mentioned above, only sophisticated attackers can exploit this bug, and only in very specific circumstances. The attack requires dedicated infrastructure to monitor and quickly react to transactions. Performing it consistently may require collaboration with an Ethereum mining pool. In addition, it is only possible when the attacker has already been approved for some allowance. Even then, the value an attacker can steal is limited (it cannot be more than the initial allowance).

Due to the degree of effort required, the minimal reward, and the unlikely circumstances required (e.g., the vast majority of ERC20 token holders never use `approve` and `transferFrom` in the first place, let alone with untrusted parties), Trail of Bits is unaware of this bug ever having been exploited in the wild. It simply has not proven profitable to exploit. It has been widely known for quite some time now and most large tokens elect to remain standards-compliant rather than mitigate it.

Nonetheless, any issue that could result in loss of funds as well as loss of confidence in Ampleforth is very important, and must be addressed seriously and comprehensively to the extent possible. Just because it has not been exploited in the past does not mean it never will be in the future. As attackers grow more serious and well-resourced, bugs this subtle and difficult to exploit merit thorough consideration.

Available Mitigations
Ampleforth has already implemented our suggested mitigation: adding `increaseApproval` and `decreaseApproval` functions, which are not idempotent and, therefore, do not suffer the above issue. Users who exclusively use these functions will not be vulnerable.

Alternatively, users can ensure that when they update an allowance, they either set it to 0 or verify that it was 0 prior to the update. In the above example, the user would call `approve(0)` then `approve(150)` instead of just the latter.

Notably, both outlined mitigations require users to use the API with some care. The solution is not just modifying code, but creating and publishing documentation. Since this issue is in the standard and Trail of Bits cannot recommend that Ampleforth remove it entirely (by modifying the `approve` function), it is critical that users of this functionality are informed of the risk and understand the best practices for avoiding it.

D. Slither

Trail of Bits has included our Solidity static analyzer, Slither, with this report. Slither works on the Abstract Syntax Tree (AST) generated by the Solidity compiler and detects some of the most common smart contract security issues, including:

- The lack of a constructor
- The presence of unprotected functions
- Uninitialized variables
- Unused variables
- Functions declared as constant that change the state
- Deletion of a structure containing a mapping

Slither is an unsound static analyzer and may report false positives. The lack of proper support for inheritance and some object types (such as arrays) may lead to false positives.

Usage
Launch the analysis on the Soldity file:

```
$ python /path/to/slither.py file.sol
```

Ensure that import dependencies and libraries, such as OpenZepplin, can be found by the solc compiler in the same directory.

E. Echidna property-based testing

Trail of Bits used Echidna, our property-based testing framework, to find logic errors in the Solidity components of Ampleforth.

During the engagement, Trail of Bits produced a custom Echidna testing harness for Ampleforth's ERC20 token. This harness initializes the token and creates an appropriate market oracle. It then executes a random sequence of API calls from different unprivileged actors in an attempt to cause anomalous behavior.

The harness includes tests of ERC20 invariants (*e.g.*, token burn, `balanceOf` correctness, *&c.*), ERC20 edge cases (*e.g.*, transferring tokens to one's self and transferring zero tokens), as well as a test that Ampleforth's gons-per-fragment accounting is always correct.

To add more tests at any point, simply add regular Echidna tests (functions with names beginning `echidna_`, taking no arguments, and returning a boolean) to the contracts ending in `_test`, and the binary will detect and evaluate them as well. Similarly, you can modify or remove any existing tests without having to change the executable.

F. Manticore formal verification

We reviewed the feasibility of formally verifying Ampleforth's Solidity contracts with Manticore, our simple, open-source dynamic EVM analysis tool that takes advantage of symbolic execution.

Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing. For contracts like `SafeMathInt` and `UInt256Lib` that do not require complex initialization or interactions between multiple accounts, Manticore can automatically analyze for common vulnerabilities such as arithmetic underflow and overflow, lost and stolen ether, *etc.* We have analyzed such contracts in Ampleforth and discovered no latent bugs with Manticore's automated detectors.

For contracts like `UFragments` that require more complex initialization, custom Manticore scripts are required to initialize the scenario. All such scripts have been delivered to Ampleforth. They test that the whitelist cannot be corrupted or surreptitiously modified and that the primary accounting invariant of the `UFragments` contract always holds:

```
_gonsPerFragment == TOTAL_GONS.div(_totalSupply)
```

G. Hosting Provider Account Security Controls

Trail of Bits has identified controls that should be considered for Ampleforth's off-chain hosting provider accounts:

1. If the hosting provider allows it, ensure the account has at least two payment methods available for billing. This helps to avoid situations where a card is unable to be billed against, resulting in the account being locked.
2. Ensure that the account is accessible to multiple Ampleforth administrators, such that the departure, theft, or loss of a single user cannot impact the team's ability to respond to security-relevant emails.
3. Ensure the hosting provider account has no single point of failure in regard to account access. At least two people should be able to access the account controlling the hosted instances.
4. Verify the access controls (IAM) for each hosting provider to ensure appropriate hardening.

Trail of Bits recommends that Ampleforth define the processes of performing Disaster Recovery and Incident Response across both their on-chain and off-chain infrastructure. Ampleforth should practice these newly defined Disaster Recovery and Incident Response processes to prevent error in a real-life application. An example of an Incident Response framework can be seen in the openly available Pager Duty documentation.

Additionally, the NIST 800 series includes NIST 800-61, which provides a series of guidelines in order to understand processes and procedures for detecting and responding to security incidents.

It is recommended that Ampleforth define SLAs to help identify areas of concern, and reduce risks in the event Disaster Recovery or Incident Response processes must be performed.

For general key storage in AWS, we recommend using the Key Management Service (KWS) and/or the Systems Manager Parameter Store.

H. SSH Security Checklist

The off-chain portions of the Ampleforth protocol such as the exchange rate feed will likely require hosting. We anticipate interaction with production instances will be performed over SSH. This section provides a simple audit checklist for configuring SSH in the most secure fashion possible.

- If possible, simply use the Mozilla SSH Configuration suggestions, and disable unneeded features.
- Enable key-based authentication by adding "`AuthenticationMethods publickey`" and "`PubkeyAuthentication yes`" to `/etc/ssh/sshd_conf`.
- Disable login by the "`root`" user with "`PermitRootLogin no`".
- Fully disable password authentication with "`PasswordAuthentication no`".
- Use a defined set of `AllowUsers` that can login to the SSH server, and use `DenyUsers` for all other users of the system.
- Set idle timeouts with "`ClientAliveInterval 300`" (300 seconds, or 5 minutes) and "`ClientAliveCountMax 0`".
- Wherever possible, use ED25519 keys for both the server and the client:
    - On the server, this can be enabled with "`HostKey /etc/ssh/ssh_host_ed25519_key`"
    - On the client, an ED25519 key can be generated by specifying "ed25519" as the argument to the "-t" option of "ssh-keygen": "`ssh-keygen -t ed25519`".
- Set the "`LogLevel`" to "`Verbose`" in order to log most user actions within SSH.
- Set a maximum login threshold with "`MaxAuthTries 1`", and audit the server for authentication failures.
- Utilize a system such as Fail2Ban or DenyHosts to reject hosts that attempt and fail to authenticate multiple times.

Other supplementary controls can be added to the SSH server to increase security, such as:

- Use Multi-Factor Authentication (MFA) such as Duo or Yubico.
- Configure short-lived SSH certificates such as with BLESS or ussh-pam.
- Require a second person for all authenticated options, generally called the "two-person rule."

I. Personal Security Guidelines

Online Services

1. Set up 2-factor authentication (2FA) on your G-Suite account. Use the Google Authenticator app or a U2F Security Key. Avoid the use of SMS as a second factor.
2. Run a Security Checkup on your personal Google account.
3. Set up 2FA on your Apple ID, Github, Wordpress.com (the blog), and JustWorks.
4. Turn on Find My iPhone. You'll be able to recover your phone if lost or stolen, or wipe the phone remotely if you can't recover it.

Laptop

1. Change your default browser to Chrome. Install HTTPS Everywhere, Password Alert, and either uBlock Origin or Ghostery.
2. Use a unique Chrome Profile for every identity you have (work, personal, etc). Do not sign into multiple accounts on the same browser instance.
3. Turn on full-disk encryption with FileVault or other full-disk encryption. If on Linux, make sure you encrypt the whole desk and not only your home directory.
4. Install BlockBlock on your Mac. It will prevent new applications from silently installing themselves to run at startup.

Phone

1. Call your cell phone provider and add additional authentication to your account:
    a. Instructions for AT&T, T-Mobile, Verizon
    b. Background from Forbes, the FTC, and Krebs
2. Set an alphanumeric passcode on your iPhone. 4 and 6-digit PINs are trivial to brute force with commonly available forensic software.
3. Android phones are allowed but discouraged. Use only Google-branded devices running the latest major version of Android. All others are prohibited from holding corporate data.

Universal 2nd Factor (U2F) Setup

Obtain the necessary prerequisites:

- Buy one Yubikey 5 Nano or a Yubikey 5C Nano, depending on your laptop configuration. These are permanently insertable and should remain in your laptop at all times.
- Buy one Feitian Multipass. These are accessible over NFC and Bluetooth LE and enable your phone to use U2F. These should go on your keychain, like any other key you own.
- Install the Google Smart Lock on your iPhone. This enables your iPhone to communicate with the Feitian Multipass over NFC, avoiding the hassle of Bluetooth entirely.

Disable the static password on the Yubikey. Yubikeys are more than simple U2F keys. They have "slots" that run different authenticators. You should disable these applications so the Yubikey only performs U2F and nothing else.

1. Download the Yubikey Personalization Tool
2. Click Tools
3. Click "Delete Configuration"
4. Click Slot 1, then click Delete

Now, add the two U2F keys to your Google account:
https://myaccount.google.com/signinoptions/two-step-verification

Note the Feitian Multipass will only connect through your desktop when plugged in over a USB cable. Use the included USB cable to enroll it in your account.

Consider setting up U2F on your personal Github, Facebook, and Google accounts, as well as any other sites that support it: https://www.yubico.com/solutions/#all

Here is what your 2-Step Verification screen should look like when complete.

- Two Security Keys, one wired and one wireless
- *Google Prompt* for applications that do not support U2F (e.g., Apple Mail.app)
- Backups code stored in a safe location offline
- No SMS or TOTP (phishable) authenticators in use

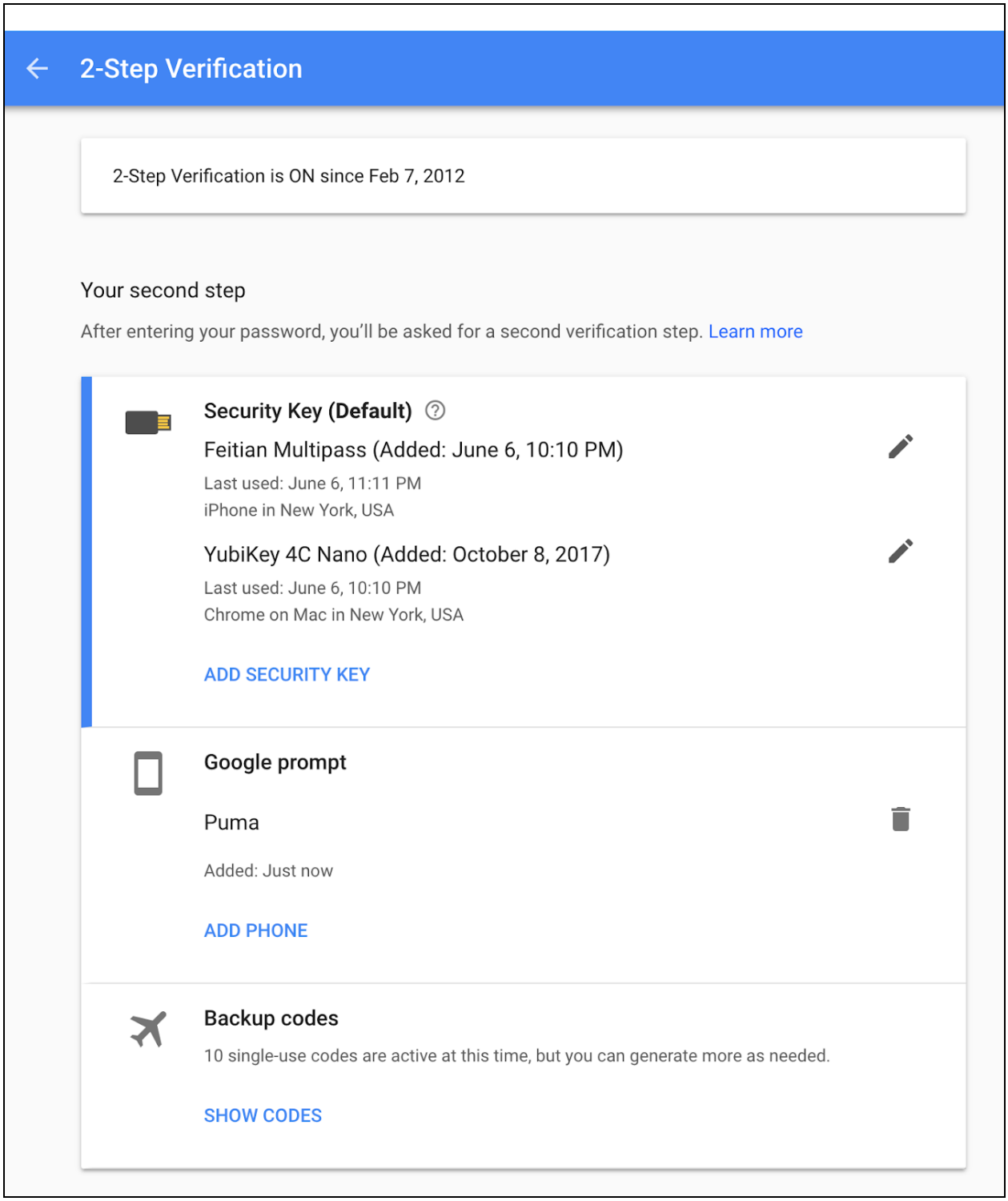

Figure I.1: *Example secure configuration of a Google account*

Yubikey Personalization Tool

Follow these steps to delete the static password on your Yubikey.

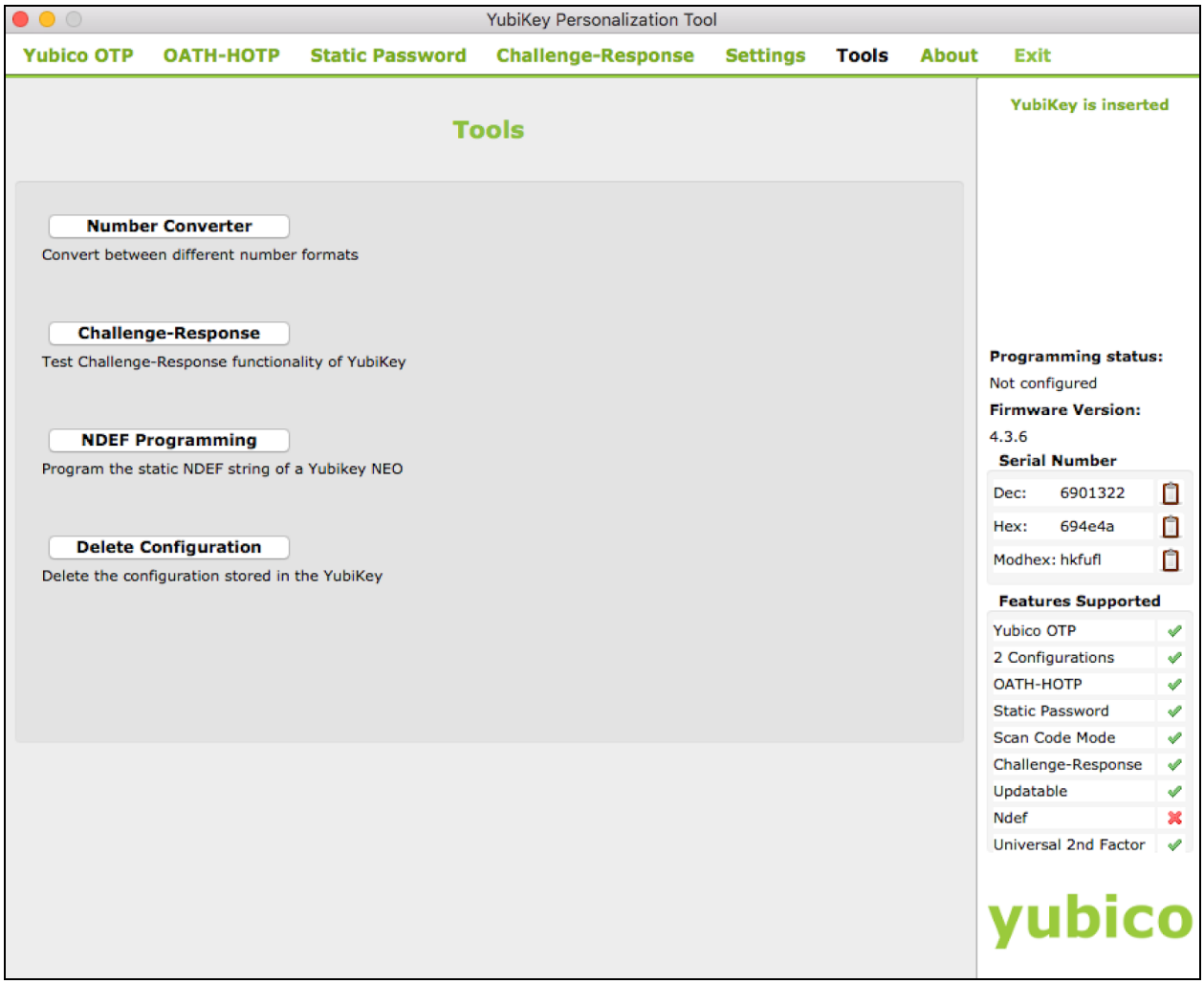

Figure I.2: *Select The "Tools" tab, click "Delete Configuration"*

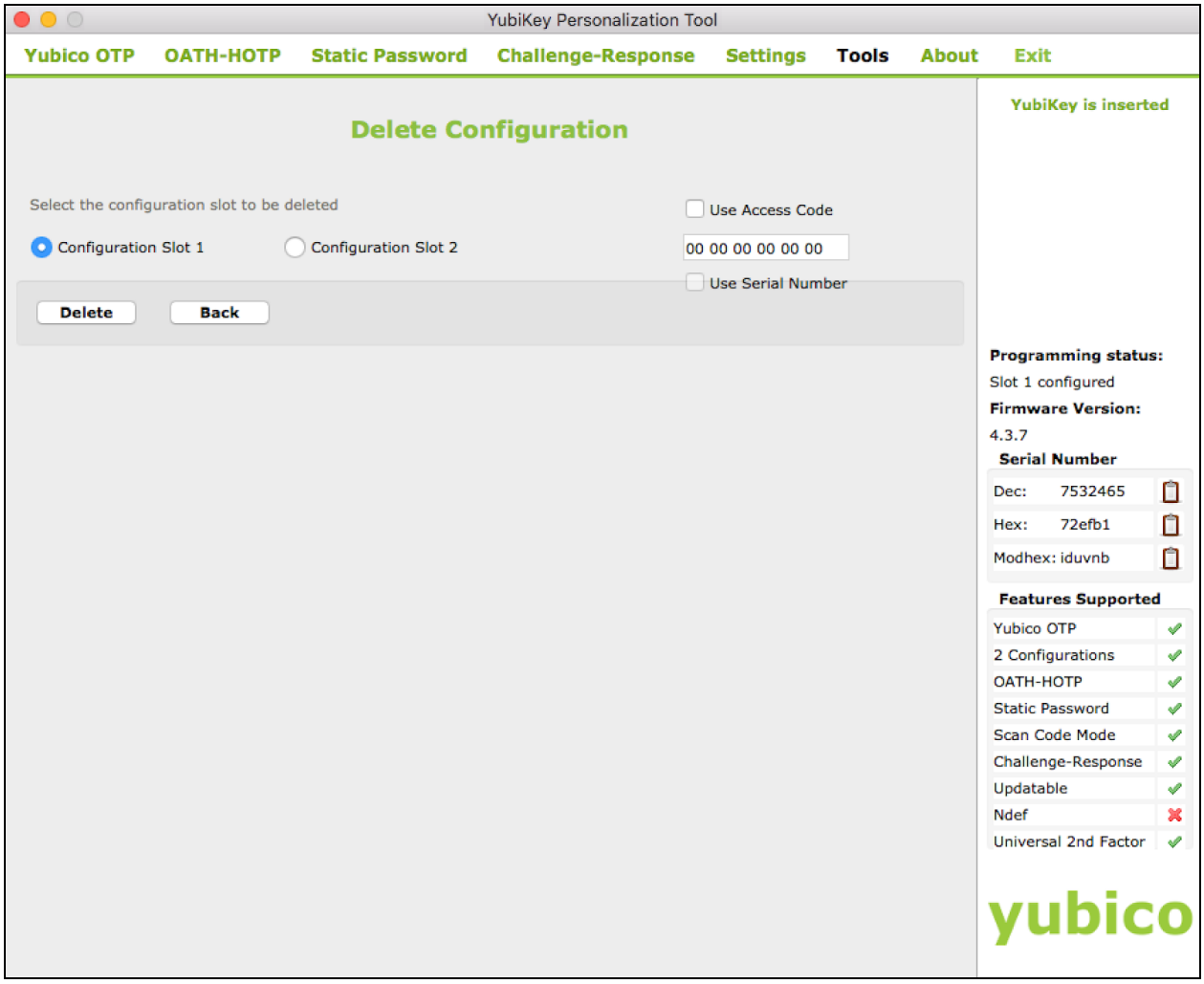

Figure I.3: *Select "Configuration Slot 1" (this contains the static password) and click "Delete"*