AirSwap Exchange Contract Launch Audit

Philip Daian

`https://stableset.com`

Overview This report presents a security review for the initial launch of the AirSwap exchange contract, by swap.tech, 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 2
 1.2 Disclaimers 2
 1.3 Highlighting 3
2 Scope 3
 2.1 System Overview and Audit Coverage 3
 2.2 Versions 4
 2.3 Threat Model and Economics 5
3 Common Antipattern Analysis 5
4 Testing and assurance 9
 4.1 Recommended unit tests 9
 4.2 Recommended integration tests 10
 4.3 Regression testing and testing policy 10
5 Code walkthrough 10
 5.1 Data invariants - Exchange.sol 11
 5.2 Function-level comments 11
6 General recommendations 12
 6.1 Bounty program 12
 6.2 Verifiable compilation 12
 6.3 Software analysis tools 13
 6.4 Phishing concerns 13
 6.5 Privacy concerns 13
 6.6 Contract failure management 13
 6.7 Versioning Scheme 14
7 Conclusion 14
8 Feedback 14

1 Introduction and Background

AirSwap (`https://airswap.io/`) is an upcoming smart contract that powers a decentralized exchange platform based on the Swap protocol architecture (`https://swap.tech/`). The contract is intended to facilitate the settlement of trades of ERC20-compliant tokens. These trades are the net result of users of the AirSwap platform communicating through a peer-to-peer protocol that is described in the Swap whitepaper (available on the swap.tech website). The security guarantees and attack surface are outside the scope of the review provided in this document.

Each trade consists of a token pair (with one of the tokens potentially being Ether, the native currency of the Ethereum blockchain), a maker/taker address, a maker and taker amount, an expiration, and a nonce to allow multiple otherwise identical trades to occur. The broader function of this contract is the execution of an atomic token/token (or token/ETH) swap, executed by a taker successfully posting an order fill on-chain. Functionality for canceling orders is also included, and marks an order as no longer valid on the blockchain, preventing any future fills of the order by the contract.

Two interconnected projects (Swap and Airswap) are involved in the launch of this contract. The primary client of the exchange contract is the AirSwap platform, which furnishes a user interface for interacting with the AirSwap system. This AirSwap platform is distributed by `https://airswap.io/`, in the form of software that users are able to run.

The exchange contract is entirely permissionless and decentralized, with no special ownership roles or higher order permissions possible in the reviewed code.

1.1 Team and Organization

The AirSwap exchange contract is developed primarily by Don Mosites, a graduate of Carnegie Mellon who has been involved in previous startups. Also on the contract development team are Julius Faber, an experienced contract developer at Consensys, and Deepa Sathaye, a specialist in trading systems.

The Swap protocol is conceptualized by a broader team, including Michael Oved, a veteran trader who helped oversee the expansion of the large high-frequency trading firm Virtu to the Asian market. While the scope of the protocol and/or its success on the market is beyond the scope of this audit, we are happy to see the involvement of team members understanding traditional financial markets in protocol design.

The swap.tech contract has also been audited by Solidity security expert Nick Johnson, whose audit was performed entirely independently from and in parallel to the audit conducted in this report. The independent audit of a contract which already has a minimal attack surface lends a high degree of assurance in its being authored correctly.

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 exchange API, as well as to the contract developers to communicate these recommendations to the AirSwap platform team.
- User recommendations: These are actions we recommend to users of the exchange contract.
- Addressed: These are recommendations for the contract that no longer apply to the latest version, as they have been addressed by swap.tech in previously reviewed versions.
- Long term: These are recommendations for policy, documentation, or architectural changes that we make to the Swap 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 Swap 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 exchange contract. We will now give a brief overview of the architecture of the Swap contract and precisely scope our efforts.

2.1 System Overview and Audit Coverage

The full on-chain system consists of a single contract developed by AirSwap, Exchange.sol, which imports an ERC20 developed by Consensys for interface use only (no code dependencies between the contracts are executed). Two test contracts and the Zeppelin Migrations.sol file are also included.

Exchange.sol is the core new code introduced by the Swap project, and will be the focus of the most detailed review (including invariants and line-by-line analysis). ERC20.sol is off-the-shelf, and will be analyzed only for potential interactions with the Swap contract, and not for full correctness. The remainder of the contracts are used for testing and auxiliary functionality only, and are not deployed on-chain in a production deployment of the system. They are thus not covered in detail by the audit.

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 AirSwap platform and associated client applications parsing contract events

These remaining items and their interactions (with both each other and the exchange 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, including the AirSwap platform, we recommend appropriate care be taken to rigorously test for bad interactions with any of the above components and the exchange 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 ERC20 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 2

```
commit 8b01247f9d7f69d36a2c0bda3de18dc8eab7fb3f (HEAD -> master, origin/master, origin/HEAD)
Author: Don Mosites <mosites@gmail.com>
Date:   Wed Oct 4 20:18:16 2017 -0400

    Initial commit of AirSwap exchange and token contracts.

615b88ea728b966197781601199dfeffe85f4ba5eb1703cdee7e47be291626c1  contracts/AirSwapToken.sol
f8ad2693abf2cb74844b9c77a9c52dee0e2344d2c595dea33bca98289ebd4bd2  contracts/Exchange.sol
5e8f867fa69405fe6bf9251f5904c57a8a5631a0f866396ac8f327b793cd34ab  contracts/Migrations.sol
4b5818a69d261c36b679b6f1a41d9224cf1458791d43c1c0c3cb1e3017c90f12  contracts/lib/Ownable.sol
e6878fd303c8769bb3c7b58a973eaba44d04fe803d6a7e8a98d11351f234b4fc  contracts/lib/Pausable.sol
ca8b3587a2ee3d0a64d524a93ca52e1b19bc3303116631e12e8bcdd37a057513  contracts/lib/SafeMath.sol
46212ce4dcd17f50ae900143b3e446fb00e872d61422dceb0ed8b8d20b8174d7  contracts/lib/StandardToken.sol
0d54a6ae757d63860813d88a373a92c03f6930dc2001747c6e81d16696719630  contracts/lib/Token.sol
f559ce45825bc41228e4a024e4ea1d8a98480f4b390361295e13720ef6702896  test/airSwapToken.js
ae66c70e06d1ba61925ab4188b1f7beaa6b9c7915f7519c44af2a4cdaf6ae930  test/exchange.js
f1e078e8a627f179b99ce2fa82774c0a7ca07b5e2fe435c9f75bd60f96145583  test/expiration.js
bd5f78ec6b3c2e7dfdbae8fa78c6e3917156865f36b02aeebe9fcae47a416d53  test/failingTrades.js
be9de344761f7b2c11430224466a4fd95c6cbcafafae2203d337743e64366cbb  test/refund.js
37411144158b72163b021cdee4ecba75cb0c6f2254762beb3a985ed039a8fbff  test/sellEther.js
8ea5a50e71c44c4b69a9a8f05ca6ad9b782443a149b0a04140d0cb9d9e7aa988  test/util.js
```

Version 1

```
commit 8fc5b043b2feaaa238d17eed76e3f73d341ffd25 (HEAD -> master, origin/master, origin/HEAD)
Author: Don Mosites <mosites@gmail.com>
Date:   Tue Oct 3 19:53:25 2017 -0400

    Adding Transfer events to token contract constructor. Closes #60


6369d8a1720f93797ae562a17d5647fbe72d4cab6c75667a8be80c1293dfdab6  contracts/AirSwapToken.sol
53b58a35a2076a947835f66297c7d0e6296e2dd469b1ea84c59d6df1ab73ab3a  contracts/Exchange.sol
5e8f867fa69405fe6bf9251f5904c57a8a5631a0f866396ac8f327b793cd34ab  contracts/Migrations.sol
4b5818a69d261c36b679b6f1a41d9224cf1458791d43c1c0c3cb1e3017c90f12  contracts/lib/Ownable.sol
e6878fd303c8769bb3c7b58a973eaba44d04fe803d6a7e8a98d11351f234b4fc  contracts/lib/Pausable.sol
46212ce4dcd17f50ae900143b3e446fb00e872d61422dceb0ed8b8d20b8174d7  contracts/lib/StandardToken.sol
d5e47782d77d5de3db642eef79a18a9be3ba44618c5d7d475faefbedda13f0ce  contracts/lib/TokenAz.sol
e1303a29cf852384a8fc03c8d33378fafa1169c367b4f10f467db71950c4de35  contracts/lib/TokenBz.sol
0d54a6ae757d63860813d88a373a92c03f6930dc2001747c6e81d16696719630  contracts/lib/Token.sol
f559ce45825bc41228e4a024e4ea1d8a98480f4b390361295e13720ef6702896  test/airSwapToken.js
20e5828fa4a97e8cfd7a75803f83bb91e9d9b36dc61dbd70b9e825d5b5590d0e  test/exchange.js
a12110efc9435bf7e391d55b664e6b89a6180c0704524e431bd71d6d5a8b0ce7  test/expiration.js
a528d8a79056b39ddd6d6336fe75e1eea01e3bd4567ddb8f70a47b4f9be11793  test/failingTrades.js
0eb22cb12efb0a552c90639b6789e73a6295fe28a12ebe9919e7a89444407fa2  test/refund.js
1013fccee4e0a2be808a3ea5f834ff54a24bd4d1c282027254675200 7ce68b2b  test/sellEther.js
8ea5a50e71c44c4b69a9a8f05ca6ad9b782443a149b0a04140d0cb9d9e7aa988  test/util.js
```

Version 0

```
commit 16a1e1ea42a25b88b3dc4d66633151e9388ece7e (HEAD -> master, origin/master, origin/HEAD)
Merge: f5769e2 b480c62
Author: Don Mosites <mosites@gmail.com>
Date:   Tue Aug 15 12:10:30 2017 -0400

    Merge pull request #29 from airswap/adding-event-params

    Adding more params to contract events fixes #26

cac075d4a4efb000c517d6cad2ed32ca2b71486d17802aa16f496a348c9deda4  contracts/Exchange.sol
5e8f867fa69405fe6bf9251f5904c57a8a5631a0f866396ac8f327b793cd34ab  contracts/Migrations.sol
d5e47782d77d5de3db642eef79a18a9be3ba44618c5d7d475faefbedda13f0ce  contracts/lib/TokenAz.sol
e1303a29cf852384a8fc03c8d33378fafa1169c367b4f10f467db71950c4de35  contracts/lib/TokenBz.sol
20e5828fa4a97e8cfd7a75803f83bb91e9d9b36dc61dbd70b9e825d5b5590d0e  test/exchange.js
a12110efc9435bf7e391d55b664e6b89a6180c0704524e431bd71d6d5a8b0ce7  test/expiration.js
a528d8a79056b39ddd6d6336fe75e1eea01e3bd4567ddb8f70a47b4f9be11793  test/failingTrades.js
0eb22cb12efb0a552c90639b6789e73a6295fe28a12ebe9919e7a89444407fa2  test/refund.js
1013fccee4e0a2be808a3ea5f834ff54a24bd4d1c282027254675200 7ce68b2b  test/sellEther.js
8ea5a50e71c44c4b69a9a8f05ca6ad9b782443a149b0a04140d0cb9d9e7aa988  test/util.js
```

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/airswap_audit_v0.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 Swap team estimates the initial value of the contract to be in the 1-200M USD range.

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 large quantities censorship resistant currency for personal profit, an incentive reduced here, as any attacks can only steal orders that are processed on the AirSwap platform before the attack is noticed.

State level adversaries have simpler ways to pursue a shutdown of the AirSwap contract, and would likely not look for vulnerabilities in a contract that is easily patched and redeployed (and which holds no funds). Our primary focus is thus on cybercriminals scouring numerous Ethereum contracts for financially lucrative vulnerabilities.

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-Swap software for the Swap contract would be economically irrational, as other contracts holding immediately liquidate-able Ether use such software and are more attractive targets for this skilled development.

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 / Humans in the Loop 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. Fortunately, this is not an issue for the exchange contract due to AirSwap's ability to replace the Swap contract in their platform with a new one in cases where new features are required. Because the contract is only involved in atomic operations and is primarily aimed at use with the AirSwap platform, replacing this contract would have no long-term consequences.

Arithmetic issues No arithmetic is present in this contract.

Denial of Service / Stuck Coins There is one possible denial of service vector in this contract. Firstly, a token or Ether receiver contract that consumes a large amount of gas or consistently throws (or returns false, once the issues surrounding atomic trades have been mitigated) could potentially cause trades to always fail. Because the AirSwap contract is designed to always throw in such cases, these trades may never be executed. Because these issues are generally isolated to single malformed trades and have no permanent effect on state when the contract is operating as intended, and because issues with failed throws are unavoidable, no further action is required.

Cancels cannot fail unless an order is filled, making the only denial of service vector for the other publicly exposed function frontrunning of cancellations.

We recommend implementing a whitelist-based mechanism for tokens that are known to be non-malicious and to have relatively bounded gas costs to avoid such issues. Such a whitelist can be implemented platform-side (eg through not serving or displaying any orders with invalid addresses), and need not be a restriction added to the contract.

Another possible denial of service vector comes from outside the contract layer, with makers agreeing to orders on the Swap protocol and signing orders for which they do not have the correspondent funds (or in which they move funds immediately before order execution, precluding complete execution). These cases are outside the scope of this audit.

No other denial of service is possible at the contract layer; all operations are completed purely deterministically on input data using bounded-time computations.

Gas Limit / Scalability The only variability in gas costs for the contract is in the external transfer calls detailed above. No loops, recursion, or other potential sources of variance in required gas are present in the contract, and no functions perform extremely gas intensive operations.

The scalability of this contract is constant-time for each transaction (as most ERC20 transferFrom methods are constant time, as is the entire swap.tech code). The low gas usage of this contract make it unlikely that scalability will become an issue for the exchange contract.

Misc. Issues One possible issue is that makers cannot buy tokens for ETH on the platform, a potentially missing feature likely intended to keep the exchange contract's atomic properties. A change to this is not possible without increasing the security requirements of the contract, and this further provides a use value for the AirSwap token. We thus do not consider this a critical issue.

Another potentially important issue is the inclusion of malicious ERC20 tokens or addresses in the Swap contract; such contracts could claim to transfer tokens without updating balances, or selectively DoS transactions by throwing errors any time .

While this was not present in the original analysis of this report, we second the assessment of auditor Nick Johnson in indexing key parameters of the provided events; this change is not security critical, and should serve as a recommendation for an enhancement which would simplify the operation of lite clients.

Timestamp Dependence There is only a single timestamp-based operation involved in this contract: checking the expiry ("(expiration $<$ now)"). We recommend all users be aware that timestamps are manipulable by miners, and checking whether an order is expired involves some mining-introduced margin of error. Miners could, for instance, choose a slightly higher timestamp to cancel a batch of orders, or choose a slightly lower timestamp to process otherwise expired orders. This degree of manipulation is bounded by the Ethereum network, which enforces at the consensus layer that timestamps are accurate to within a margin of error. This is unlikely to cause an issue for small orders which are not large enough to incentivize miner attack, which is likely to be most orders in early iterations of the platform.

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

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 unknown security bug leads to order failure	Upgrade contract
An unknown security bug leads to funds loss	Upgrade contract
Signature verification fails for a valid signature	Resign order, upgrade contract

We do not provide a full enumeration here, as it is clear that all failure modes have similar remediations (upgrading the contract used by the AirSwap platform).

We recommend specific care be paid by the Swap team to the modes that lead to contract failure, and potential contingencies be established to avoid these scenarios. Primarily, we recommend the Swap team provision the AirSwap platform to support a redeployment of the exchange contract if useful or necessary.

Game theoretic bugs A number of game theoretic issues are present in distributed exchange design; because most of these lie at the protocol layer, which is not covered by this audit and are therefore omitted in this report. The primary game theoretic issue introduced by the contract is that takers pay fees for failed orders in the form of gas costs, whereas makers do not. This may incentivze makers to agree to execute the same order with multiple takers, an action that costs them nothing but potentially increases the speed or probability of successful order execution.

Unfortunately, with the taker-posts-to-chain model used by Swap and most other decentralization exchanges, this problem seems unavoidable. In the long term, we recommend higher layer protections against malicious makers, either by whitelisting known honest makers on indexers or blacklisting makers' coins known to be malicious.

Incorrect / missing modifiers There are no custom modifiers available to the exchange contract, making the only modifiers required the default Solidity modifiers: `internal private [external public] constant`. All functions are appropriately marked private save the two intended-to-be-public fill and cancel functions.

It may be worth exploring the new modifiers introduced by Solidity 0.4.17 in the long term (primarily pure), and upgrading the contract to work with this base version.

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 sends in the Swap system use the Solidity-native transfer method, which automatically checks return values and throws on stack overflow.

Stuck Ether One major potential issue with payable smart contracts involves a contract that is sent Ether and never processes this payment, with no functions available to handle any outstanding Ether in the contract. The current exchange contract is vulnerable to several related issues. There is only a single payable function, fill, simplifying our analysis. That being said, several stuck Ether bugs are noted in fill.

The fill function is set up to receive Ether (marked payable), but can return without refunding this Ether to the sender. In this case, Ether will become stuck in the Swap contract. There is no function to currently withdraw Ether from the contract.

Several remediations are possible:

– Applying the suggestions in Github issue 33 (recommended by auditor Nick Johnson as a separate issue) to the entire contract and manually verifying that only the main code path will result in no exceptions being thrown, after the appropriate ERC20 transfer is called and returns true and the appropriate send is sent and returns true.
– Refunding the sender of Ether in the event of a (logged) failure.
– Adding an “escape" function allowing a contract owner to withdraw any stuck balance.

The Swap team should choose and apply a remediation before the launch of the contract. This issue was reported to the team as issue 38 on 17 Sep. This issue is verified as fixed in v1.

An additional issue with accepting Ether in the fill function is that, in the case where no Ether is being traded, it is not enforced that no Ether is sent to the fill function. If Ether is accidentally sent to the fill function in this fashion, this Ether will become stuck in the contract and permanently lost.

We recommend the Swap team add a check to ensure that when no Ether is being traded, msg.value is 0 (eg in the else branch of if (takerToken == address(0x0))). This issue was similarly reported as part of issue 38 above.

Stuck/Lost Tokens Similar to the stuck or lost Ether issue, ERC20 tokens can easily become stuck if sent to an invalid address or address not capable of handling tokens. In the case where tokens are being exchanged with invalid addresses through the exchange contract, this is possible. Unfortunately, checking whether an address is set up to receive ERC20s is not possible, so this is an unavoidable platform limitation and the burden lies on the maker and taker to ensure received tokens can be spent.

Another source of lost tokens is tokens that get sent directly to the exchange contract by mistake. Because this may be a common mistake made by new users, we recommend the addition of a function that takes a token address, and forwards the entire balance of the exchange contract on that token to an owner or hardcoded address. Because this is the only contract functionality in which ownership is relevant, this does not increase centralization and has the potential to save users money by allowing the AirSwap team to return any tokens mistakenly sent to the exchange contract. This is however not a universally accepted practice or standard, and is an optional recommendation.

Re-entrancy / recursive send This contract is by inspection not substantially vulnerable to recursive send vulnerabilities; for Ether transfer, the `transfer` method is used, providing gas below the limit of gas required to trigger re-entrancy when Ether is sent. Some re-entrancy is technically allowed by the external calls to transfer, which use Solidity's native calling functionality and forward all available gas. Such re-entrancy would require a malicious ERC20 to execute, and would present this malicious token with one of two options. The first is to attempt to cancel the currently executing order, which is precluded by the check to fills in the cancel function (currently fills is not checked in cancel, which was reported to the team in issue 39 on Sep 19, and resolved in v1). The second is attempt to execute a new order, which because of the fills re-entrancy guard must be a different order than is currently being executed. This is a calling pattern explicitly intended to be allowed by the Swap design.

Previous versions of this contract could have allowed for richer re-entrant calling patterns on behalf of malicious ERC20s (eg - re-executing an order or canceling an order while its execution was in process). This was communicated to the team on Jul 10 in issue 13 and resolved.

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 unit tests. We make some suggestions for tests we believe are useful in the contract; while there is no requirement for the team to implement these changes, we recommend all these scenarios at least be validated or considered.

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 transferring 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).

This feedback was communicated to the team on Jul 10 in issue 16.
We suggest an expansion of unit tests as follows:

Exchange.sol

– **fill**: Check that valid order trading tokens using two test token contracts executes successfully. Check that balances are updated appropriately and tokens are properly transfered based on amounts passed to the function. Test that if maker has insufficient funds [transferFrom failure], the order fails gracefully and balances are not updated. Do the same test for the taker. Test that if makerToken or takerToken are invalid addresses with no contract code, trade fails gracefully. Test that fill fails and no balances are updated with an invalid v, r, and s (one test for each being invalid). Test that multiple identical orders cannot be filled. Test that fill fails if makerAddress and takerAddress are the same. Test that orders that are already expired fail gracefully and don't update corresponding balances. Test that, in the case where a user tries to take an invalid order (takerAddress != msg.sender), order fails gracefully and no balances are updated. Test that an order cannot be canceled once filled (cancel fails gracefully, no balances updated). Test that Ether is refunded in all failed orders involving Ether. Test that trades for Ether are handled appropriately (Ether transferred to maker, token transferred to taker). Test that sending a message with the wrong value in trades for Ether causes the

fill to fail and no balances to be updated. Test fill with an invalid token that throws on transferFrom, ensure that order fails appropriately (test for atomicity of trades). Test that in all failure cases, any expected logs are present. Test that in successful order, appropriate logs are emitted. Test that orders where 0 tokens are traded (0 maker and taker amount) succeed, do not change balances. Check that if only one of these is zero, balances change appropriately. Check that on all failures, Ether is returned if Ether is sent along with the function.
- **cancel**: Test that after canceling an order successfully, order cannot then be filled. Test that cancel logs a failure event when it fails (or throws depending on changes suggested by Nick Johnson). Test that cancel logs a successful cancellation event on successful cancellation. Test that cancel is non-payable (throws when sent Ether).
- **trade**: Private utility function; both balance updates and atomicity are covered by recommendations for fill above.
- **transfer**: Private utility function; successful balance updates are covered by recommendations for fill above.
- **validate**: Test a number of valid and invalid signatures, generated independently by reference clients. The integration of validation with filling orders is tested by the above recommended fill tests.

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.

The tests we recommend for AirSwap contracts are for testing the interactions between the AirSwap platform and the exchange contract, including placing orders, receiving log events and displaying appropriate information to the user, placing failed orders, attempting to take failed maker orders, and gracefully handling both failures and cancellations on both the maker and taker side.

4.3 Regression testing and testing policy

We recommend the following test policies formalized in the Swap 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 Swap team in detail.

5.1 Data invariants - Exchange.sol

Only one piece of persistent data is present, requiring a very simple set of invariants for this contract:

– fills is a mapping from keccak256(order) = keccak256(makerAddress, makerAmount, makerToken, takerAddress, takerAmount, takerToken, expiration, nonce) to a boolean that is set to false unless an order has either successfully executed or been canceled.

We analyze and ensure that these invariants are preserved in our line by line analysis below.

5.2 Function-level comments

– **fill**: The fill function is responsible for the execution of an order. The conditions on a valid order fill are: maker and taker are using different addresses, maker and taker are trading valid tokens, the transfer between these tokens succeeds (or from ETH to these tokens if the taker is using ETH), the maker has appropriately approved the order fill (by signing a digest of the order, which serves as an ID), the order is not expired, and the order can only be filled once. The base conditions of the order are verified first; the expiry, non-duplication, and signature are manually verified as appropriately checked, and present in our automated test recommendations. (previous versions did not check expiry, though the auditor notified the team in Github issue 15 on 10 Jul, and the issue was addressed by pull request 24). Then, two flows are possible: either the taker is offering Ether, in which case a token transfer is performed and Ether is sent to the maker using an appropriate transfer method (should not fail from out of funds, as msg.value is checked to be the intended send amount first; other failures are covered elsewhere in the audit). Alternatively, two tokens are to be traded and the trade function is forwarded appropriate parameters to execute the trade. The return value of the trade function is never checked and this function can never return false. We recommend modification to the function to either throw on failure, or we recommend asserting its return value as true in fill and modifying the function to have a meaningful return value (in v1, updates were made so that the trade function now throws when either token call throws or returns false; technically this code can be rewritten without returning booleans, since trade and transfer will return true every time, but this is not a security risk). Furthermore, in any case where a failure event is logged, any Ether sent along with the message will be forwarded. It is also checked that no Ether is sent in a call to fill that is intended to exchange between two tokens.
– **cancel**: Cancel simply marks an order as filled and logs its cancellation, allowing the AirSwap platform to show a notification to the user. Nothing else is required for canceling an order due to the stated invariant that an order is only filled once. Unfortunately, the reviewed version of the function violates this invariant, allowing for multiple cancellations or cancellations after an order is successfully filled. This is because cancel does not check the fill status of an order before canceling, and it is not possible for a cancellation to fail. We strongly recommend that the AirSwap team change the cancel function to check fill status, disallowing cancellation of filled orders. This issue was reported to the team in Github issue 39 on 22 Sep. This was addressed in v1, with cancel now reporting a standard failure if the order has already been filled.
– **trade**: The trade function is a private helper function responsible for completing an atomic token-swap (as a helper to fill) in the event when two tokens are being traded. The intended property of the function is an “all or nothing trade", in which either tokens are exchanged between two parties or no state change occurs. For this, the ERC20 transferFrom functionality is called with appropriate parameters, transferring tokens to the counterparty on each trade. This functionality is also extensively covered

in our test recommendations. Unfortunately, one issue in the trade function which must be corrected immediately is its non-atomicity; as per the ERC20 token specification, transfers can return false on failure, in which case the trade function will (in theory; see comments on transfer) return false, but state will not be reverted. This causes a substantial attack whereby a user trades a token they carry no balance in, which can fail returning false, but keeps the tokens successfully transferred in the previous call to transfer. This issue was reported to the team in Github issue 15 on 12 Jul, and was independently discovered during the course of Nick Johnson's contract audit. It was resolved in v1. Now, transfer will throw whenever an ERC20 function either throws or returns false, ensuring that if either of the exchanges fails, the transaction is reverted.

– **transfer**: The transfer function is intended to perform a delegated ERC20 transfer using the transferFrom functionality. The one line of code is trivially correct, and uses the Consensys Standard Token ERC20 interface that has been validated as having an ERC20-compliant method signature. One key issue with this function is that it does not pass along the return value of the client ERC20, missing cases where an ERC20 may return false for a failed transfer. This contradicts the ERC20 specification, which states that clients should never expect an ERC20 to always return true when a boolean return is specified. We recommend changing this code to return the value returned by the ERC20 function. (This was fixed to throw on error and always return true in v1, an acceptable solution)
– **validate**: Validate is a simple signature verification method, whose effects are tested for thoroughly in the provided test recommendations. Issues with parameter padding (such as in the ERC20 short attack) are not consequential to this function, as any valid signature that does not cause the method to throw represents an order that was signed by the maker with the same parameters.

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 recommend the AirSwap team create a bounty program that covers critical, non-DoS funds-loss vulnerabilities in their protocol, platform, and on-chain contract code. We recommend this bounty program provide clear guidance on what constitutes a vulnerability and how much each potential vulnerability would pay an interested user.

6.2 Verifiable compilation

We recommend that, to assist in community audit, the Swap 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/`).

We do recommend addressing the few warnings provided by the Remix IDE. These include the use of explicit "public" visibility modifiers for fill and cancel, and a change form sha3 to keccak256 in the validate function. None of these changes are security critical.

6.4 Phishing concerns

One potential avenue for security compromise and funds loss in the AirSwap architecture is the creation of platform phishing sites, modified software binaries, or DNS/network based hijacks of the AirSwap platform. While such issues at the platform level are outside the scope of this contract audit, we highly recommend that all users check that on-chain calls are being made to the appropriate contract, and that AirSwap prominently publish the address of the correct contract on an SSL-secured page.

6.5 Privacy concerns

Obvious privacy concerns apply to the Swap exchange, with full order history being visible on-chain to any interested observers. This order history includes, for each settled trade, the maker and taker address, amount, fee, and signature information. This means that a clear traceable link exists for any user transacting through the Swap platform between the balances of the tokens they are exchanging. While Swap does not reveal any form of on-chain AML or KYC data, this could potentially create privacy leaks included in the database of blockchain analysis companies (eg Chainalysis).

We recommend that any entry points to the Swap system remind their users that Swap trades occur on-chain, and are thus fully publicly exposed.

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:

- Validate all inputs (addresses, signature values, amounts, balances in the corresponding tokens, ERC20 approvals required to successfully execute) before using the exchange contract.
- Carefully handle throws and/or any possible error return values from both fill and cancel.
- Verify the correctness of signatures by calling validate offline on the inputs that will be passed to the exchange contract, saving gas in the event of a bad signature.
- Ensure they are using the correct exchange contract address.

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 Swap 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, 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 or tokens to be transferred to the wrong address**, so 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 or 0. 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 AirSwap exchange contract will launch in a well tested, secure state. The contract appropriately implements minimal functionality to achieve exchange, is not vulnerable to any classically known antipatterns, and benefits from all key end-user interaction with the contract being mediated by the AirSwap platform, allowing AirSwap to replace the contract with an updated or modified version at its discretion (though being a decentralized exchange, AirSwap cannot enforce that its users follow the recommendations of using a new contract).

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.

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 Swap 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

`v0 - initial release [current]`