AirSwap Token Contract Launch Audit

Philip Daian

https://stableset.com

Overview This report presents a security review for the initial launch of the AirSwap token (AST) 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 Disclaimers . . . 2
- 1.2 Highlighting . . . 2

2 Scope . . . 3
- 2.1 System Overview and Audit Coverage . . . 3
- 2.2 Versions . . . 3
- 2.3 Threat Model and Economics . . . 4

3 Common Antipattern Analysis . . . 4

4 Testing and assurance . . . 7
- 4.1 Recommended unit tests . . . 8
- 4.2 Recommended integration tests . . . 8
- 4.3 Regression testing and testing policy . . . 8

5 Code walkthrough . . . 9
- 5.1 Data invariants - AirSwapToken.sol . . . 9
- 5.2 Function-level comments . . . 9

6 General recommendations . . . 10
- 6.1 Software analysis tools . . . 10
- 6.2 Privacy concerns . . . 10

7 Conclusion . . . 10

8 Feedback . . . 11

1 Introduction and Background

For a general introduction to the AirSwap team and platform, see our associated audit of the exchange contract.

The AirSwap Token (AST), the contract under review in this document, is a mintable ERC20 token designed to power the AirSwap platform by serving as a fee to list trading intent in the AirSwap indexer. To list intent to trade a pair with an indexer, a user must lock up AST, providing a mechanism for denial of service and Sybil resistance on the platform.

To accomplish these goals, the AirSwap token modifies the standard ERC20 token contract with functionality that disables the transfer of locked tokens, allows users to lock a monotonically increasing supply of tokens for a fixed "locking period" (providing the utility of the token), and allows users to query their available balance. Users can also extend existing locks into the future as many times as they wish.

The AirSwap token is also not transferable until after a preset date, to allow for the sale to complete and balances to remain locked for a short time before becoming usable on the platform. This minimizes security risks associated with errors in the sale. The contract also carries an owner, who is allowed to use the transferFrom functionality, allowing other senders to transfer the owners' balance to their own wallet if approved. The owner can also initiate a pause, which freezes token balances and prevents future transfers, also potentially denying service to the AirSwap platform. The Swap team plans to potentially nullify this owner if it is clear that the pause functionality is no longer required. The owner has no power to manipulate balances, mint tokens, or do anything other than the explicit functionality mentioned above.

1.1 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.

1.2 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 token 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 token 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 token 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, AirSwapToken.sol, which imports an ERC20 developed by Consensys for core functionality (StandardToken.sol), and ownership and pause functionality contracts developed by Zeppelin Solutions (Pausable.sol and Ownable.sol). Pausable and StandardToken are directly subclassed by the AirSwapToken class, and Pausable further subclasses Ownable. StandardToken also subclasses token, an ERC20 abstract / pure interface class. Two test skeleton contracts and the Zeppelin Migrations.sol file are also included.

AirSwapToken.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, Pausable.sol, and Ownable.sol are off-the-shelf, and will be analyzed only for potential interactions with the Swap token contract, and not for full correctness. The core ERC20 logic in StandardToken.sol, while also off the shelf and previously audited, will be analyzed for both correctness and compliance to the recently finalized EIP20 standard (`https://github.com/ethereum/EIPs/issues/20`).

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 token 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 token 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

Please see the accompanying exchange audit for version information and information on validating the audited version of the code. We consider the contracts listed in v2 of the corresponding version section, with commit hash `8b01247f9d7f69d36a2c0bda3de18dc8eab7fb3f`.

2.3 Threat Model and Economics

Because the threat model and economics of the token contract are similar to those enumerated in the exchange contract, we defer to the analysis in the attached audit. Similar to the exchange contract, the AirSwap contract holds no Ether, and can potentially be replaced with a new contract which imports system balance snapshots at any time. Similar to the exchange contract, the token contract can therefore cause loss of funds on only a transient basis, if some undiscovered vulnerability in the contract allows for inappropriate or incorrect token transfers that lead to lost funds after trading on an exchange or similar. This bounds the possible damage from a major security problem by the number of trades that could potentially be executed before the vulnerability is discovered and all relevant stakeholders are informed.

This makes the potential damage from token contract vulnerabilities marginally higher than potential damage from exchange contract vulnerabilities, as far more software and businesses will likely interact with the token contract than the exchange contract (all entities interacting with the exchange contract also use the token contract, as do other independent entities like cryptocurrency exchanges). We do not belive this difference is enough to change the threat model of the contract, or to incentivize more sophisticated actors to attack the AirSwap token over the exchange.

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. By design, the AirSwap token is non-upgradeable. The risk from this is low, as the AirSwap token is a bare-bones mintable ERC20, implements a finalized token standard, and is unlikely to require change in the near future. Many such tokens are currently deployed, and any fundamental changes required at the token level would require redeployments of all these contracts.

We therefore deem the AirSwap token's non-upgradeability to not pose a substantial risk. The only caveat is that some hardcoded parameters, such as the monotonically increasing property of the lock and the locking period, cannot easily be upgraded without redeploying the contract. We therefore recommend that AirSwap internally validate and confirm these numbers before their scheduled launch. Unfortunately, it is impossible to have such a trustless utility lock that is also upgradeable, and the AirSwap team has explicitly decided to make this tradeoff in their design.

Arithmetic issues Arithmetic is present in the contract in several locations; the standard token library, where only addition and subtraction are used (no potential for rounding errors), and all operations are overflow checked, and the AirSwapToken contract.

In the latter, again only addition and subtraction are used, precluding any overflow issues. We reason about each of the operations sequentially:

- **Constructor**: the balance of the owner is subtracted from the total supply to yield the deployer balance. In the case that the AirSwap team is malicious, they could underflow this operation by providing a deployer balance larger than the total supply, violating the standard ERC20 invariant that the total supply is the sum of all balances. Because

this is easily checkable and provable by users before tokens are unlocked for transfer, and because the deployer is assumed trusted, this is not a security risk for the platform.
- **lockBalance**: the timestamp `now` is added to the predefined locking period `604800`. Because `now` is bounded by the current timestep, there is no possibility for overflow here.
- **availableBalance**: One subtraction is present here, `balances[_owner] - balanceLocks[_owner].amount`. If balanceLocks is greater than balances, this subtraction could underflow. However, the data-based invariant on balanceLocks is theoretically that a user's balanceLocks entry will always be less than their balances entry. Unfortunately this does not hold and introduces a major vulnerability; see below. This was fixed in v2 of the reviewed contracts.
- **Other functions**: Contain no arithmetic.

The current token contract has a major vulnerability whereby users can arbitrarily bypass the balance lock. The flow of the exploit is as follows:

- The user creates an active lock with any amount of tokens.
- The user creates a new lock, attempting to lock a value greater than their balance. The check for `balanceLocks[msg.sender].unlockDate > now` will evaluate to true. Then, the contract checks `require(_value >= balanceLocks[msg.sender].amount)` when deciding whether to grant the lock to the user. This also evaluates to true, as they are now trying to lock a huge quantity. The balance lock is created and the event is issued.
- Now, the user calls `transfer` with any amount that is held in their balance. `availableBalance` is called, which returns `balances[_owner] - balanceLocks[_owner].amount`, since an active lock exists. This subtraction is unchecked, and therefore underflows. A huge number larger than either the balanceLock or the user's balance is returned.
- The check in transfer for `require(availableBalance(_from) >= _value);` passes, as the user's available balance is massive. The user is allowed to transfer any amount of tokens up to their balance (which is checked by the base ERC20 function), even if these tokens are locked, bypassing the lock functionality entirely.

Remediation

- Move the check for `require(balances[msg.sender] >= _value);` outside the else branch; it should always hold that a user tries to lock at most their balance.
- **Optional**: consider checking the line `return balances[_owner] - balanceLocks[_owner].amount;` in availableBalance for overflow by asserting that `balances[msg.sender] >= balanceLocks[msg.sender]`. This comes at a small gas penalty, but ensures that if the locks or user balances somehow obtain unexpected balances, the balance lock can not be bypassed.

Denial of Service / Stuck Coins No denial of service is possible at the contract layer; all operations are completed purely deterministically on input data using bounded-time computations. Also, the lack of any payable functions means that no stuck Ether is possible with the current code. The only other potential denial of functionality (cases in which a function outside the standard ERC20 code does not return) are:

- Users try to use a paused contract.
- Users trying to lock balances greater than they hold, or trying to lock less than they had previously locked before a lock has expired.
- Users transferring before the token becomes transferable, one week after launch.

All cases are intended functionality.

Gas Limit / Scalability There are no loops or external calls to introduce variability of gas costs in this contract, meaning that all operations nicely retain a fixed gas cost. The scalability of the remainder of the contract is at worst that of an ultra-basic ERC20 with a few cheap additional checks, meaning that the contract will continue to scale well into the future (and has scaling properties equivalent or favorable to the majority of value-holding contracts currently on the Ethereum network).

Misc. Issues No miscellaneous issues were observed during the review of this contract.

Timestamp Dependence There are several timestamp-based operations involved in this contract: checking the time that tokens become transferable, and checking the time that locks become unlockable. The former can only be manipulated before the AirSwap token becomes transferable, as after this time the Ethereum consensus rule that timestamps must be monotonically increasing (as well as the substantial quantity of elapsed time) will ensure that miner rollbacks of the token to a paused state are impossible.

For the checking of lock expiries, we recommend all users be aware that timestamps are manipulable by miners, and checking whether a lock has expired involves some mining-introduced margin of error. Miners could, for instance, choose a slightly higher timestamp to unlock their funds before expected. 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 locks which are not large enough to incentivize miner attack, which is likely to be all locks 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 token transfer failure	Snapshot balances and re-deploy
An unknown security bug leads to a broken lock mechanism	Snapshot balances and re-deploy

We do not provide a full enumeration here, as it is clear that all failure modes have similar remediations (replacing the contract with a new one snapshotting balances, communicating this to all exchanges and relevant partners).

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 maintain open lines of communications with all exchanges and other clients of the token code.

Game theoretic bugs The majority of this contract simply implements a standard ERC20, with the only clear game theoretic mechanism provided by the locking functionality, in which a user sacrifices liquidity of their AST tokens to fuel their use of the platform and their listing of orders on the indexer. This mechanism is intended as both a source of Sybil resistance for the orderbook and a mechanism for reflecting the net utility of the AirSwap platform in the token value (as more tokens locked up in the platform implies higher utility of the platform, and the decreased token liquidity will correspondingly increase token value). A cursory evaluation of this mechanism seems sane, as users who are lacking up large enough quantities of AST to mount a denial of service attack on the network are also risking decreasing the value of their substantial quantity of tokens at unlock time, providing

a base financial cost to mounting a wide-scale order flooding attack on the Swap peer-to-peer protocol and granting some degree of cryptoeconomic security. Obviously, if the cost to place an order is too low in quantity of tokens locked, this can still allow for attacks by rational players. The full analysis of this mechanism is beyond the scope of this work.

It is likely that no security critical game-theoretic design flaws are present in this contract, though we recommend the Swap team carefully evaluate and validate the choice of constants in their mechanism, specifically the required locking period; such validation is beyond the scope of this report.

Incorrect / missing modifiers There are several custom modifiers available to the token contract, including `whenPaused, whenNotPaused, onlyAfter, onlyAfterOrOwner, onlyOwner`. The other modifiers required the default Solidity modifiers: `internal private [external public] constant`. All functions are public in this contract, as intended. Furthermore, transfers are only allowed after the unlock period (and recommended to test conformance to this), as are transferFroms but with an exception for the owner. Neither transfer or transferFrom are allowed while paused, as intended. All other functions should be allowed always, and not restricted either by time or owner. No owner-only functions are present.

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 Ether cannot get stuck in the target contract, as there are no payable functions anywhere in the contract.

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.

Another source of lost tokens is tokens that get sent directly to the token 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.

Re-entrancy / recursive send This contract is by inspection not vulnerable to recursive send vulnerabilities, as it is a mintable ERC20 with no external calls anywhere in the contract.

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:

AirSwapToken.sol

- **Constructor**: Check that appropriate transfer events are emitted with expected amounts. Check that balance of owner and deployer match expected balances. Check that their sum is equal to totalSupply. Check that owner has been set appropriately.
- **lockBalance / availableBalance**: Check that locking several balances produces the appropriate event and updates both availableBalance and the balanceLocks data structure appropriately. Check that after a lock is initialized, attempting to lock a smaller amount fails before the timeout. Check that this succeeds after the timeout. Check that locking 0 tokens succeeds. Check that availableBalance is updated appropriately in all cases. Check that availableBalance is equal to the balance when no lock is present.
- **transfer / transferFrom**: For both transfer and transferFrom, check that transfers before the specified transferability date fail. Check that transfers after succeed, emit the appropriate events, and update balance accordingly. Check that transfer fails when a user locks some or all of their tokens and attempts to transfer more than the unlocked sum. Check that balances are not updated in this case, the transaction throws, and no logs are issued. Check that pausing the contract disables both functions, and unpausing resumes their functionality. Check that transfering ownership allows the new owner to pause, prohibits the old owner from pausing or transfering ownership again.

4.2 Recommended integration tests

In addition to unit testing, we recommend end-to-end testing the integration between the Swap token and the AirSwap platform, ensuring that credits on the AirSwap platform are appropriately issued for locking tokens. Also check the interaction between the transfer of tokens and the platform. Ideally, these tests would be executed on a testnet or live version of a blockchain system, and their results verified with multiple consensus clients.

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 - AirSwapToken.sol

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

- balanceLocks is a mapping from addresses to a balanceLock order that contains an amount and date, representing the locked balances using the mechanism described earlier in this audit. The invariants include that balanceLocks will never decrease in amount before expiring, and that when updated, a balanceLock's locking periods will be updated to the current timestamp plus 604800 (7 days, manually verified).
- (other standard ERC20 datastructures, including balances and totalSupply, maintaining the invariant that sum(balances) = totalSupply)

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

5.2 Function-level comments

AirSwapToken

- **Constructor**: The constructor is a simple function assigning some balance to the owner, the remainder of the supply to the deployer, issuing the appropriate ERC20-compliant transfer events, and transfering ownership to the provided owner.
- **lockBalance**: Two cases exist in the lock balance. A user is either extending a previous lock or creating a new one. In the former case, the contract ensures that the balanceLock increases monotonically before expiry. In either case, the contract checks that the user has the balance available to lock, and issues an event with both the new and old lock amount. The balanceLocks data for the sender is also updated with the appropriate new value and expiry.
- **availableBalance**: Three cases exist when calculating the available balance; either the user has an expired lock, no lock, or an active lock. In the first two, the lock will be less than the current timestamp (with no lock returning the default value of 0 for the lock timeout), and the full balance will be returned. In the second, the lock is subtracted from the available balance, with a check for overflow (which should never be triggered as reasoned in the arithmetic issues section above).
- **transfer** / **transferFrom**: Both forward to the standard ERC20 methods, after checking that the user has the balance available (not locked) and that the token has become transferable. An exception is made for the owner in transferFrom, required to perform the sale before tokens can be transferred publicly.

Token and StandardToken: The contents of Token and StandardToken were reviewed, and shown to be line-by-line equivalent to the two Solidity implementations in `https://github.com/ethereum/viper/pull/366/`, which were written and tested by the auditor. This gives high assurance of correctness in their code, as these contracts were written independently and tested against implementations by several developers in multiple languages.

Ownable and Pausable: The contents of Ownable and Pausable are outside the scope of our line-by-line review, but were similarly reviewed by the auditor in previous contract audits and confirmed to be identical. Both contracts are short and easily validated manually.

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 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 all functions in the token contract. None of these changes are security critical.

6.2 Privacy concerns

Obvious privacy concerns apply to the Swap token, with transfer history being visible on-chain to any interested observers. This transfer history includes, for each settled transfer, the to and from address and amount. 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 AirSwapToken transfers occur on-chain, and are thus fully publicly exposed.

7 Conclusion

We conclude that if the recommendations herein are followed, the AirSwap token contract will launch in a well tested, secure state. The contract appropriately implements minimal functionality to achieve the desired utility, is not vulnerable to any classically known antipatterns, and benefits from a lack of holding of any decentralized irreversible cryptocurrency.

We recommend that the team performs all our recommended 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]`