Havven Australia Ltd

Havven Contract Review

Version: Second Review: 3.0

June, 2018

Contents

Introduction

Sigma Prime was commercially engaged to perform a second time-boxed security review of the smart contracts dictating the blockchain dynamics and business logic of the Havven (HAV) and USD Nomins (nUSD) tokens.

This second review is in relation to the second iteration of development on the Havven token system. The first review can be found on Github [1].

The review focused solely on the security aspects of the Solidity implementation of the platform. The economic structure of the platform and related economic game theoretic attacks on the platform are outside the scope of the review. Readers should note that this iteration of Havven does not implement the full design of the Havven platform, as detailed in the Havven whitepaper [2].

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contracts. Sigma Prime makes no judgements on, or provides any security review regarding, the underlying business model or the individuals involved in the project.

Document Structure

The first section introduces the project and provides an overview of the functionality of the contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an open/closed status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as "informational". Outputs of automated testing that were developed during this assessment are also included for reference (in Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Havven contracts.

Project Overview

The Havven system is comprised of two ERC-20 [3] tokens and is intended to establish a new form of stable coin. The two tokens are:

- "Havven" tokens (`havven` or `havvens`), which serve as collateral for the second, nominally stable coin.
- "Ether-backed USD Nomins" (`nomin` , `nomins` or `nUSD`), which are the designated stable coins.

The Havven Whitepaper [2] (hereafter "the whitepaper") describes a dynamic process by which `havven` holders are incentivised to dynamically adjust a locked-up portion of their `havven` holdings. It is intended that these locked-up `havvens` will serve as collateral and thereby stabilise the value of one `nomin` at or near one USD. `Nomin` transactions incur a fee and the resulting fee-pool is distributed to users who supply `havvens` as collateral, thereby incentivising `havven` holders to participate in the dynamic `nomin` stabilisation process.

The current implementation of the Havven system is not final but instead represents a step towards a complete implementation. Accordingly, a number of features differ, relative to the system described in the whitepaper, including the following:

- The ability to issue `nomins` against held `havvens` is restricted to certain white-listed accounts.
- `Havven` holders receive fees from the fee-pool in proportion to the amount of `nomins` they have issued, relative to the total issuance of `nomins`, irrespective of any designated `havven` collateralisation target.
- The dynamic fee structure detailed in the whitepaper would require `havven` holders to regularly adjust the proportion of their `havvens` locked-up as collateral, to maximise the fees received. This dynamic fee structure is absent.
- The number of `nomins` that can be issued against a users' `havven` holdings is determined by the USD value of `havvens` (which is injected into the blockchain by an oracle) and the `issuanceRatio`.
  - If the USD value of `havvens` decreases, such that the USD value of held `havvens` falls below the nominal value required to collateralise the `nomins` already issued by the user, the user is unable to (a) issue more `nomins`, nor (b) transfer any `havvens` from their account.
  - These conditions remain enacted until the user either adds additional `havvens` to their account or the USD value of `havvens` increases, such that, in either case, the value of held `havvens` is sufficient to reach the designated value of collateral.

Sigma Prime understands that the manifest discrepancies between the current smart contract functionality and the *Havven* system described in the whitepaper have been communicated to the wider community.

Contract Overview

Note: This document is based upon a time-boxed analysis of the underlying smart contracts. Statements are not guaranteed to be accurate and do not exclude the possibility of undiscovered vulnerabilities.

There are four main Solidity files, named `Court.sol`, `HavvenEscrow.sol`, `Havven.sol` and `Nomin.sol`, each possessing its own distinct logic:

- The `Court` contract defines a voting mechanism allowing users of the system to participate votes to confiscate nomins from the accounts of bad-actors. The primary purpose of the `Court` contract is to prevent token wrapping contracts (as discussed in the whitepaper). `Court.sol` is not included in the scope of the present audit.
- The `HavvenEscrow` contract allows the foundation to allocate `havvens` to investors with designated vesting schedules. During a vesting period, investors may still issue `nomins` against the vested `havvens`. Investors are also entitled to a share of the `nomin` fee-pool, in proportion to the total amount of `nomin` tokens they have issued. `HavenEscrow.sol` is not included in the scope of the present audit.
- The `Havven` contract specifies the `havven` tokens along with the mechanisms required to collect fees for passive token holders. This contract determines the initial creation and subsequent distribution of `havven` tokens.
- The `Nomin` contract specifies the `nomin` tokens, including their creation and destruction, and associated fees.

The `Nomin` and `Havven` contracts each have their own `Proxy` and `TokenState` contracts. The system is designed such that users should not interact directly with the main contracts (`Nomin` and `Havven`) but should

instead interact with the relevant `Proxy` contracts. This design is advantageous for a project still in development, as it allows the main contracts to be replaced without requiring users to learn a new contract address. Furthermore, the `TokenState` contracts, which store the token balances for the main contracts, also promote simplified system upgrades. In the event that the main contracts are replaced, the new contracts can be directed to the pre-existing `TokenState` contracts, thereby forgoing the need to perform iterative token balance migrations in the event of an upgrade.

A simple illustration of the contract interactions is as follows:

```
Proxy <- pass call & return data -> Nomin <- read/write state -> TokenState
```

```
Proxy <- pass call & return data -> Havven <- read/write state -> TokenState
```

Only the main contracts (`Havven` and `Nomin`) can interact directly with their associated `TokenState` contracts, however, the main contracts themselves can be accessed either directly or through the `Proxy` contract. The `Proxyable` contract contains modifiers and variable logic that ensures the main contracts function correctly irrespective of whether a user has accessed them directly or via the `Proxy` .

Audit Summary

The intial review was conducted on commit [957664f]. Within this commit the following contracts were reviewed:

```
└── contracts
    ├── DestructibleExternStateToken.sol
    ├── ExternStateFeeToken.sol
    ├── Havven.sol
    ├── Nomin.sol
    ├── Owned.sol
    ├── Proxyable.sol
    ├── Proxy.sol
    ├── SafeDecimalMath.sol
    ├── SelfDestructible.sol
    ├── State.sol
    └── TokenState.sol
```

The issues that were raised in the first audit [1] relating to the centralisation aspects of this system are omitted from this report.

The final review (including modifications and amendments in response to the initial report) was conducted on the same contracts in commit [b5a6cdb].

Per-Contract Vulnerability Summary

Havven Token Contract `Havven.sol`

The Havven token was found to be vulnerable to a potential front-running attack. This vulnerability has little impact on the current implementation, and the authors have acknowledged and addressed this issue.
No further vulnerabilities were identified.

SelfDestructible `SelfDestructible.sol`

All issues discovered have been resolved.
No further vulnerabilities were identified.

DestructibleExternStateToken Contract (`DestructibleExternStateToken.sol` **)**

No potential vulnerabilities have been identified.

ExternStateFeeToken Contract `ExternStateToken.sol`

No potential vulnerabilities have been identified.

Nomin Token Contract `Nomin.sol`

No potential vulnerabilities have been identified.

Owned `Owned.sol`

No potential vulnerabilities have been identified.

Proxyable `Proxyable.sol`

No potential vulnerabilities have been identified.

Proxy `Proxy.sol`

Storage variables can be overwritten if proxied to `Havven` or `Nomin` contracts. This is acknowledged by the authors and expected for Proxy contracts which utilise the `DELEGATECALL` opcode.
No further vulnerabilities have been identified.

SafeDecimalMath `SafeDecimalMath.sol`

No potential vulnerabilities have been identified.

State Contract `State.sol`

No potential vulnerabilities have been identified.

Token Contract `TokenState.sol`

No potential vulnerabilities have been identified.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the *Havven* smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Summary of Findings

ID	Description	Severity	Status
HAV-01	Havven price update is vulnerable to front-running	Low	Resolved
HAV-02	DELEGATECALL can overwrite proxy storage variables	Low	Resolved
HAV-03	Unvalidated constructor input allows a delay which can overflow for early selfdestruct.	Low	Resolved
HAV-04	Noteworthy design observations	Informational	Resolved
HAV-05	Miscellaneous notes, gas saving and general comments	Informational	Resolved

HAV-01	Havven price update is vulnerable to front-running		
Asset	Havven.sol		
Status	Closed: See Author's Response		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Front-running [4] is an attack vector whereby users watch the blockchain for transactions, then submit transactions with higher gas prices to give their transactions a greater order preference.

Havven uses an oracle to set the price of Havven's in the `updatePrice()` function on line [685]. If the price is lowered (relative to the previous price), this will lock up a greater number of Havvens for users who have partially locked some Havvens (as dictated by `lockedHavvens()` on line [626]). These users could (with some likeliness) implement a simple program which watches for price updates by the oracle. If the price decreases, the program submits a transaction with a higher gas price than the oracle which transfers their remaining Havvens to an new address, preventing the locking mechanism.

The impact of this vulnerability is set to low due to HAV-04, but we raise this as something that may need to be addressed in future iterations.

Recommendations

There are a number of techniques that can be used to prevent front running. One simple example is to limit the gas price of transactions which invoke the `transfer()` and `transferFrom()` functions. Using a higher gas price than this limit for updating the price can prevent user front-running. This technique has the obvious drawback that user's transfer transactions could be prevented during periods of high gas usage.

Another technique is a commit-reveal scheme, however, this would involve locking of transfers between the commit and the reveal. A final approach, which may have some relevance, is the concept of *Submarine Transactions* [5], however, the `CREATE2` opcode is required for an efficient implementation.

Author's Response

In addition to the impact of HAV-04 upon this front-running question, the fact that only white listed (foundation) addresses may issue. nomins until a future version of the contract also ameliorates this issue.

We do not consider front running to be a concern for this iteration of the system, but it is something we have given some consideration to. In what follows we discuss some of our thinking, and the consequences for future versions of the system.

In the previous version of the contract, ether-backed nomins, it was important to ensure the contract was not front-runnable, since any user able to get in ahead of price updates could siphon ether out of the collateral pool, given that nomins were always bought and sold from this pool at a price of US$1. The solution to this was a combination the oracle design and a fee which was charged on buying and selling into the nomin pool. Buying or selling entailed a fee of 50 basis points, so a 1% round-trip fee. Meanwhile the price oracle was designed such that observed ether price movements of 1% or greater since the last price update (with a base update frequency of 15 minutes) would force a price update on the contract. Ether price corrections of greater than 1% in fifteen minutes turn out to be quite infrequent. Together these measures make the risk-free profit opportunities from front-running available only in the rare instances of 15 minute corrections substantially greater than 1%. Such an issuance fee is something to be considered for the next version of Havven, although there are efficiency concerns.

In the final system, issuance and burn operations will place orders a market, so any front running opportunities will require these orders to fill within a block. There are profit opportunities available here, in the potential first-mover advantage conferred in more-rapid C_{opt} targeting for fee-collection, and in the seigniorage profit derived from the ability to burn nomins more cheaply or issue them more expensively than after the pending price update. These strategies rely on exchange conditions being favourable, and occurring rapidly enough. The former is advantageous for more-rapid money supply adjustments. The latter potentially extracts some value from orders that are on the book; it may not be a great problem, but it is something we will continue to think about more closely as the next phases of the project come to fruition.

HAV-02	DELEGATECALL can overwrite proxy storage variables		
Asset	Proxy.sol		
Status	Closed: See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `owner` can set the boolean flag `useDELEGATECALL` which switches the proxy from using a `CALL` opcode to use a `DELEGATECALL` opcode. This design feature is added to allow potential future functionality.

The proxy contract, uses three storage slots (each of 32 byte length). They are,

- **Slot[0] -** `owner`
- **Slot[1] -** `nominatedOwner`
- **Slot[2] -** `target` and `useDELEGATECALL` - (These are tightly packed, with the last 20 bytes (Big-endian, right most bytes) representing `target` with the next byte representing `useDELEGATECALL`.

The proxy contract is currently designed to target both the `Havven` and `Nomin` contracts. The first four storage slots on both of these contracts are,

- **Slot[0] -** `owner`
- **Slot[1] -** `nominatedOwner`
- **Slot[2] -** `initiationTime` (from `SelfDestructible`)
- **Slot[3] -** `selfDestructInitiated` and `selfDestructBeneficiary` (tightly packed, with last byte representing the boolean and the next 20 bytes representing the address)
- **Slot[4] -** `proxy`

Thus if the `useDELEGATECALL` flag is set, any call to `Nomin` or `Havven` to modify `owner` or `nominatedOwner` will modify the `Proxy`'s `owner` or `nominatedOwner`. Therefore, the `owner` state of the `Nomin` or `Havven` contract will likely also be able to control the `proxy`, which can `setTarget()` and `setUseDELEGATECALL()`.

Furthermore, the `target` address and the `useDELEGATECALL` flag will both be simultaneously modified if a `DELGATECALL` is made to `Havven` or `Nomin` which sets the `initiationTime` in the `SelfDestructible` contract. As this cannot be set to any arbitrary value, any attack vector to modify `target` or `useDELEGATECALL` is minimal, and localised to the `owner` account.

Recommendations

As the proxy contract only requires three basic (not complex type) storage slots, it's not too difficult to reserve these addresses in proxied contracts, to ensure there is no accidental/malicious overwrites of the `Proxy`'s storage variables. This could be done easily, for example by creating the following basic contract,

```
contract ReserveStorage {
  uint Slot0;
  uint Slot1;
  uint Slot2;
}
```

and then ensuring that all `proxied` contracts inherit this contract first. i.e

```
contract Nomin is ReserveStorage ... { ... }
```

Resolution

This has behaviour has been acknowledged by the authors and is as intended. It is also noted, that this complication is inherent in all such delegatecall stateful proxy contracts.

HAV-03	Unvalidated constructor input allows a delay which can overflow for early selfdestruct.		
Asset	SelfDestructible.sol		
Status	Closed: Resolved in commits [3e1c845] and [5db7270]		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `selfDestructDelay` variable is set on line [52] in the constructor without validation. It is then used in the `require()` on line [52]. If `selfDestructDelay` is set to a value of the order `uint(0) - now`, this will allow the owner to self-destruct any time after initialisation (equivalent to a `selfDestructDelay = 0`). Alternatively, `selfDestructDelay` could be set to `NULL_INITIATION + 1` which would allow the owner to self destruct the contract without having to `initiateSelfDestruct()`.

Recommendations

Although all instantiations of `SelfDestructible` entities set reasonable `selfDestructDelay` values, we recommend validating the constructor input to make this isolated contract over-flow safe and to prevent possible future vulnerabilities (i.e allowing user input in the constructor).

Resolution

The `selfDestructDelay` variable has been set as a constant, preventing arbitrary values being set in the constructor and the overflow vulnerability discussed in this section. A `selfDesctructInitiated` boolean flag has also been introduced which ensures that the `initiateSelfDestruct()` function has been called, before running the `selfdestruct()` function.

HAV-04	Noteworthy design observations
Asset	Havven.sol
Status	Closed: See Author's Response
Rating	Informational

Description

The locking mechanism for `havven` tokens is enforced in the `transfer()` and `transferFrom()` functions. The current implementation of this locking mechanism incentivises users to either lock their entire balance of `havvens` or none at all. If a user were to partially lock their `havvens`, by issuing an amount of `nomins` that was below their maximal allowed issuance, the user's remaining `havvens` could become locked in the future, in response to a decrease in the USD value of `havvens`. To ensure that their additional `havvens` do not become locked, a rational user would create a new account and send their remaining `havvens` to that account. Thus, the user ends up with two accounts, one in which all `havvens` are locked, the other in which no `havvens` are locked. This situation applies more generally - a user is incentivised to either lock all or none of the `havvens` in their individual accounts. The logic of locking extra tokens when the `havven` value decreases (in this current implementation) seems superfluous. Because of this design, the front-running attack in HAV-01, has little impact.

It is also noted that dimensional analysis on the issuance and destruction of `nomins` reveals a disparity between these processes. On lines [600] and [602], the dimensions of the issuance equation for `nomins` have the form `[nomin] = [USD]/[havven] * [havven]`. For this to be valid (i.e., have the correct dimensions), it implies that `[USD]/[nomin] = 1`. This design is such that the creation of `nomins` is tied to the USD value of `havvens`, but the destruction of `nomins` has no such tie. We simply point out this implicit assumption, to verify that the authors' are aware of such caveats.

Recommendations

It is noted that the economic incentive structures outlined in the white paper have not yet been implemented in this iteration. This section aims to point out specific details of the design to ensure they are intended by the authors.

Author's Response

The highlighted considerations are known to the contract authors, and are intended. The assumption of a $1 nomin price, and the lack of incentive to do anything other than to lock up all havvens in an account will be rectified in upgraded versions of the contract. The whitelist of authorised issuers is in place to ensure that only benevolent actors (to wit, the foundation) may issue nomins until a fuller incentive structure has been implemented.

HAV-05	Miscellaneous notes, gas saving and general comments
Asset	All Contracts
Status	Closed: All issues addressed
Rating	Informational

Description

This section details miscellaneous informational aspects found within the contract. Actions need not be taken, this is mainly for authors reference.

- **Gas Saving - Variable Initialisation -** Havven.sol line [435] - Initializing a `uint` to 0 is more expensive than simply initialising the variable, (i.e `uint feesOwed;`)
  ✓ Resolved in commit [ccded41]
- **Gas Saving - Unused memory variable -** Havven.sol line [619] - The `maxIssuanceRights(issuer)` result is stored in memory on line [615]. This can be re-used to save gas and from re-calling the `maxIssuanceRights` function.
  ✓ Resolved in commit [8d4c40c8]
- **Failed calls have return data size of `calldatasize` -** Proxy.sol line [115] - Failed calls will revert, with the contents at memory address `free_ptr` and length `calldatasize`. The `calldata` is located at this address, and is overwritten by `returndatacopy`. The data length of the new data will be `returndatasize`.
  ✓ Resolved in commit [78b0529]
- **Variable Name Typo -** Havven.sol line [209] - Variable name `initalHavPrice`.
  ✓ Resolved in commit [57bb688]
- **Comment Typo -** Havven.sol line [41] - "When issuing or burning for the issued nomin balances and when transferring for the havven balances." (incomplete sentence)
  ✓ Resolved in commit [6400e30]
- **Comment Typo -** Havven.sol line [106] - “All nomins issued require some value of havvens to be locked up for the proportional to the value of issuanceRatio (The collateralisation ratio). "
  ✓ Resolved in commit [6400e30]
- **Comment Typo -** ExternalStateFeeToken.sol line [149] - "* @notice Query an account’s balance from the state" (repeat of comment at line [138]).
  ✓ Resolved in commit [af7228b]
- **Comment Typo -** TokenState.sol line [26] - ".. to to make the changeover as easy as possible, since mappings .."
  ✓ Resolved in commit [af7228b]
- **Comment Typo -** State.sol line [24] - ".. to to make the changeover as easy as possible, since mappings .."
  ✓ Resolved in commit [af7228b]

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `truffle` framework was used to perform these tests and the output is given below.

```
Contract: Havven scenarios
  ✓ should allow vested tokens to be withdrawn (4372ms)
  ✓ should not allow non-whitelisted address to issue Nomins (504ms)
  ✓ should transfer the amount of havven to nomin (689ms)
  ✓ should transfer the defined amount of havven to nomin (684ms)
  ✓ should distribute fees evenly (1691ms)
  ✓ should not allow double fee withdrawal (1332ms)
  ✓ should be able to burn nomins (791ms)
  ✓ should not be able to burn nomins if none issued (388ms)
  ✓ should lock extra tokens with lower Havven price changes (1261ms)

Contract: Ownable
  ✓ should set the owner to 0x0 if _setOwner() is called (50ms)

Contract: Test Rig
  ✓ should build a test rig without throwing (383ms)
  ✓ should allow for the reading of havven price after deployment (contract
    directly) (324ms)
  ✓ should allow for the reading of havven price after deployment (via proxy
    ) (386ms)

Contract: Variable Return Data
  ✓ should allow for variable return data (138ms)
  ✓ should allow a call directly to the proxy using the .at() method (96ms)


15 passing (13s)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

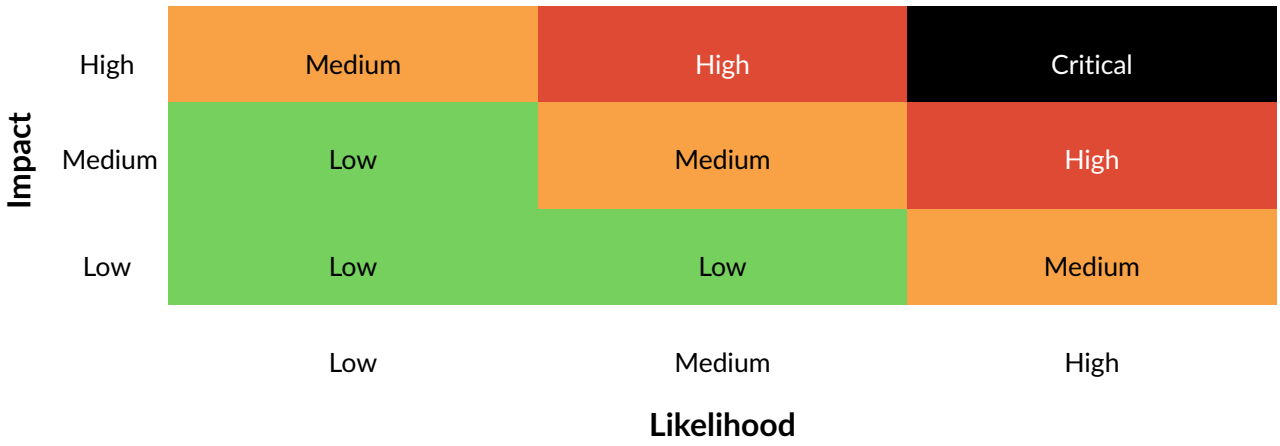

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Havven token contract audit. Github, 2018, Available: https://github.com/sigp/havven-audit.

[2] Havven: A decentralised payment network and stablecoin. Website, February 2018, Available: https://havven.io/uploads/havven_whitepaper.pdf.

[3] ERC-20 Token Standard. Github, Available: https://github.com/ethereum/EIPs/issues/20.

[4] Ethereum Smart Contract Best Practices - Front Running. Website, Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/#transaction-ordering-dependence-tod-front-running.

[5] Ari Juels Lorenz Breidenbach, Phil Daian and Florian Tramèr. To Sink Frontrunners, Send in the Submarines. HackingDistributed.com, August 2017, Available: http://hackingdistributed.com/2017/08/28/submarine-sends/. Date Accessed, May 2018.