PUBLIC

Expert Security Audit

of Smart Contracts

February 20, 2018

Produced by

CHAINSECURITY

for

aeternum

one fund. one coin.

Table Of Content

Foreword

We first and foremost thank AETERNUM for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

- ChainSecurity

Token Name	AETERNUM COIN (AET)
Decimals	8
Smallest Unit (Atom)	10^{-8} AET
Token Amount	$250{,}000{,}000$
Token Type	Pre-minted
Token Price	Defined by `rate` variable (See Page 2)
Maximum tokens sold	Defined by `tokenSalesCap` variable (See Page 2)
Minimum contribution	1 Wei
Maximum contribution	Unlimited
Refund	None

Table 1: Facts about the AET token and the token sale.

System Overview

The AET is supposed to be a crypto currency which ensures that its intrinsic value grows steadily. Based on connection with the Aeternum Fund and the mining rigs, the AET's value is supposed to increase, providing value to token holders.

In the following, we describe the AETERNUM COIN (AET) and its corresponding token sale. Table 1 gives the general overview.

Token Sale Overview

All of the $250{,}000{,}000$ AETERNUM COINS will be created upfront and given to the token owner. The token owner can then allow the crowdsale contract to sell off some of the tokens. The tokens are sold through a separate crowdsale contract. Different phases can be set up in the crowdsale contract as it can be configured through transactions. However, the beginning and end of these phases are not well defined. The currently available tokens for sale are given by `tokenSalesCap`. As the token sale happens on-chain, the token buyer receives its tokens as soon as its purchase has been recorded inside the blockchain.

As discussed, the token owner controls the majority of the tokens and also controls the crowdsale. This is an intentional part of the project. The AETERNUM COIN is supposed to provide goodwill services and support the project with bounty campaigns and marketing strategies, which is enabled by giving the token owner the described control.

Note that all of the mentioned contract variables can be queried using a client, such as geth, or can be viewed online at a website such as etherscan.

Token Price

The `rate` variable defines the conversion rate between ether and token atoms, i.e. if `rate` is $200{,}000{,}000$, token buyers will receive $200{,}000{,}000$ AET atoms = 2 AETs per ETH.

The token price can be freely set by the owner during the whole course of the Token Sale. This can be done using the `adjustRate` function. As a consequence, a race condition might be observed during token purchases. A token buyer might check the current `rate` and send a purchase transaction. However, by the time the transaction is executed on the blockchain, the `rate` might have changed.

Extra Features

Delayed Token Transfers A flag in the token contract allows AETERNUM to prevent token transfers from other users until the owner decides otherwise. Once token transfers have been activated, they cannot be deactivated anymore. Thereby, the token owner can define one common starting time after which token transfers are allowed for all token buyers.

Pausable Sale AETERNUM has the power to pause and unpause the crowdsale using the `onHold` variable. During this time, neither token transfers nor purchases can be made.

Affiliate System The contracts contain an affiliate system that allows affiliates to receive token bonuses as commissions. The bonus is determined based on the purchased amount and `commissionPercentage`. Note that the `commissionPercentage` can be adjusted during the course of the Token Sale, which is why the exact commission is not guaranteed.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on February 8th, 2018, and their latest version on February 27th, 2018:

- AeternumCrowdsale.sol
  - SHA-256: `bf6547793e53b54f4fae346ffc42eb56bfa7e21cd4062700cd0101150a0aba3b`
- AeternumToken.sol
  - SHA-256: `e502783ee9779ad014e2c7fcbe36d22adb6b3672da2c69f3a6190295c4517bda`

Depth of Audit

The scope of the security audit conducted by ChainSecurity was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into four distinct categories, depending on their criticality:

- L - can be considered as less important
- M - needs to be considered to be fixed
- H - should be fixed very soon
- C - needs to be fixed immediately

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. Finally, if during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise we label it as ✓ Addressed.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

Details of the Findings

In this section we detail our findings, including both positive and negative findings.

No restrictions on token owner M

The white paper says: “Upon use of this function or upon first purchase of a given token, this functionality will be permanently locked, after which point that token will only be acquirable through standard means.” There is no such functionality in the smart contracts. Quite contrary, the token owner keeps full control over the tokens.

This design was chosen to retain full flexibility combined with simplicity. However, it comes at the cost that token buyers have to trust the token owner not to misuse his power. If they have this trust, then the presented concern is not an issue for them.

The given design has the advantage that AETERNUM COIN, which doesn’t perceive itself as a market-standard ICO model, can use its focus on community and trust building and perform goodwill services in cases where, for example, the buyer loses his tokens.

Likelihood: Low
Impact: High

Reliance on the trustworthiness of the owner of the contract L

As described in Section 2, the crowdsale gives a lot of power to its owner. While a sale is running, the cap, the token price and commission percentage can be adjusted.

Although these features contrast with standard ICOs in which the power of the owner is limited, they fit the specification in the case of AETERNUM COIN. More precisely, any user (either as a potential token buyer or token holder) is assumed to *completely trust* the owner during the crowdsale as well as after the owner ended it.

As before, these functionalities provide a lot of flexibility and functionality to the token owner without an overly complicated smart contract. However, if the owner account should get compromised or start misbehaving, a number of issues would follow. If the owner is well-behaving and *as long as this assumption of trust by the token buyers in the token owner holds*, these features do not represent a security risk.

Likelihood: Low
Impact: Medium

Crowdsale only checks allowance L ✓ Fixed

When performing a sale the crowdsale contract determines the amount of available tokens for sale through its allowance. An example can be seen here:

```
        require(add(effectiveChange, maxCommission) <= token.allowance(owner, this));
```

While the allowance is a necessary condition, it is not sufficient. The token also needs a sufficient balance. This is never checked. Hence, even if only a few or no tokens would be left, the sale could still indicate that many tokens are available for sale. This issue is also related to the unrestricted token ownership issue seen above.

Likelihood: Low
Impact: Low

Post-audit fix: The AETERNUM team has modified the smart contracts and added the following check in different places:

```
        require(add(effectiveChange, maxCommission) <= token.balanceOf(owner));
```

This check ensures that, not only is the allowance sufficient, but also the balance is sufficient.

Improperly used data types L

The smart contracts often use `uint` data types. These are unsigned integers. However, they are frequently checked against negative values which they cannot represent. Some examples are:

```
        require(sub(tokenSalesCap, sold) >= 0);
```

The return value will always fulfil this property.

This sample combines two apparent operations that have no effect:

```
        uint256 tokens = _wei.mul(rate).div(1000000000000000000);
        require(tokens >= 0);
        return uint256(tokens);
```

This is actually equivalent to the following:

```
        return _wei.mul(rate).div(1000000000000000000);
```

since the require is always valid, and that the cast has no effect.

There are several more occurrences of this behaviour. These mistakes unnecessarily complicate the code and consume gas.

Likelihood: Medium

Impact: Low

Reentrancy Analysis ✓ No Issue

We did not discover any reentrancy issues. This is because no invocations of untrusted code are made.

No Callstack Bugs ✓ No Issue

We did not discover any callstack issues.

Ether Transfers ✓ No Issue

We did not discover unusual or dangerous ether transfers in the code as the AET contract directly forwards received ether to a specified wallet.

Safe Math ✓ Fixed

The AETERNUM contracts use the safe math library to avoid over-/under-flows. However, as smaller data types, such as `uint64` are used, they need to implement custom SafeMath operations for these data types.

As different data types are being used, they have to be converted as seen in this example:

```
        return uint64(commission);
```

Such casts can lead to overflow issues, which are not addressed properly.

Fix: The AETERNUM contracts now use `uint256` everywhere. However, the safe math library is still redefined at the end of the contract for no reason.

Recommendations/Suggestions

- The contracts make frequent use of the `uint64` data type. While it is generally a good idea to reduce the required memory consumption, in Ethereum it is not necessarily beneficial. Some of the computational operations are actually more expensive for smaller data types. This is because the underlying EVM instruction works with 256-bit inputs and conversion becomes necessary.

  Furthermore, it adds additional complexity and costs as a custom SafeMath library has to be used.

- Through the overall design of the crowdsale (e.g. integration into web frontend) it has to be assured that the `writeAffiliateMapping` function is called before the respective token purchase takes place.

- For the crowdsale to work correctly, the token owner has give an allowance. This is currently not contained in the deployment script where it could be added to allow testing and avoid manual mistakes.

- The AET is not pausable, meaning that token transfers can never be restricted. This can be a desirable feature depending on the intended use cases.

- The following check:

  ```
          require(tokens <= sub(tokenSalesCap, sold));
  ```

  should take the commission variable into account.

- The motivation for the following line is not quite clear:

  ```
          require (newRate > 1);
  ```

  Here, the rate has to be bigger than 1. Either a significantly higher value would make sense (as the rate is typically in the order of 10^8) or simply a positivity check would make sense. Also there is no check for an upper bound. However, that might be fine.

- It is unclear why the `withdrawFunds` function is so complicated, given that the crowdsale contract is never supposed to contain any ether. It seems that a simple function which withdraws all funds would be sufficient.

- Some test cases fail, as they seem to contain a mistake. They have the value of “2500000” instead of “250000000”.

- In a number of historical cases, token buyers have (presumably accidentally) send ERC20 tokens to token sale contracts. These sent tokens are then locked within the receiving contract if it contains no functionality to forward/spend them. AETERNUM could add such a functionality in order to avoid this locking.

Conclusion

ChainSecurity has thoroughly audited the Aeternum Coin smart contracts using research-driven, in-house analysis tools as well as manual analysis by experts. The token and the token sale have the required functionality and no security issues from the owner's perspective.

From the investor's perspective, the contracts require an unusually high amount of trust in the token owner, as the token owner has wide-ranging capabilities before, during and after the token sale. These capabilities have been described above and are significantly greater than in comparable token sales. Therefore, the investors need to be willing to trust the token owner with these powers. It can be argued that such trust is reasonable, as investors already trust the token provider and its management team in terms of the business model.

CHAINSECURITY

Disclaimer