CertiK Audit Report for Ampleforth

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Ampleforth (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets. For more information: https://certik.org.

Executive Summary

This report has been prepared for Ampleforth to assess any issues and vulnerabilities that have arisen in the updates of the source code of their uFragments smart contracts. The initial version was on tag v1.0.0 with commit hash 5bec128f9005c0f40c0ce70a4b7069d8c05a8895.

A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Source Code Deltas

The audit was carried out at the v1.0.2 tag of the uFragments repository of the ampleforth project with commit hash fb6eab6ad37b891158dcd280b250baecc5f8fc25. The file that has been updated based on the SHA-256 checksums of the v1.0.0 tag is as follows:

- **UFragmentsPolicy.sol**

  Previous: 81231683bd400eabba5c8a81f98fd71729329833f35b4e6763b37e593ca7f437

  Latest: 7346355b1a35565e07c58dd87bdd2229cbcce503d76a18893213c5aeabd9e463

Additionally, a newly added file was observed and fully audited:

- **Orchestrator.sol**

  Latest: 55339eb64c49d957ca3824d2468787905f25c18025d8a843738e52827a39a573

To properly assess which lines of code would need to be audited, the deltas needed to be factored in as well as whether the deltas merely added code or edited existing functionality that affects unaltered code in other places. In total, 155 lines of code were audited non-inclusive of newlines with the following deltas into account:

- **UFragmentsPolicy.sol**

  Delta: 19 newly added lines within the contract's body (L85-L91, L94-L95, L100, L164-L173)

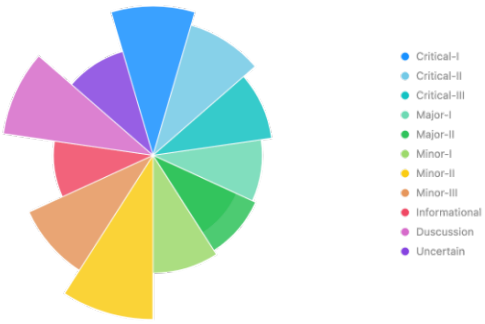
- **Relayer.sol**

  Delta: 134 newly added lines as the whole file was nonexistent in the previous audit

Testing Summary

SECURITY LEVEL

VULNERABILITY OVERVIEW



Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Ampleforth. This audit was conducted to assess issues and vulnerabilities in the updated source code of Ampleforth's uFragment Smart Contracts.

TYPE	Smart Contracts
SOURCE CODE	https://github.com/ampleforth/uFragments/tree/v1.0.2
PLATFORM	EVM
LANGUAGE	Solidity
START DATE	May 04, 2020
REVISION DATE	May 06, 2020 May 30, 2020
METHODS	Dynamic Analysis, Static Analysis, and Manual Review, a comprehensive examination has been performed.

Dynamic & Static Analysis Review Notes

The CertiK team applied state-of-the-art dynamic and static analysis tools on the source code of the smart contracts to pinpoint any vulnerabilities and issues that can be detected using analytical techniques applied on the source code deltas directly.

The tools applied for the deltas of the codebase were unable to identify any potential issues or vulnerabilities, inferring that the codebase is secure against common attack vectors and vulnerabilities.

An additional thing to note is that the team already enforces a strict Solidity linter and specifically solhint which is a capable tool and points to the fact that the team that has undertaken the development of the smart contracts is capable of maintaining a secure codebase and practically applies the best coding practices.

Manual Review Notes

Introduction

The CertiK team was invited by the Ampleforth team to audit the design and implementations of the updated code of Ampleforth's "uFragments" protocol smart contracts.

The goal of this audit was to review the source code deltas of the protocol implementation between v1.0.0 and v1.0.2 for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

Methodology

The work was conducted and analyzed under different perspectives and with different tools such as static analysis tools as well as manual reviews by smart contract experts. In general, we make

observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

Documentation

We used the following sources of truth about how the Ampleforth protocol should work:

1. Ampleforth Whitepaper: https://www.ampleforth.org/paper/
2. Ampleforth Project Documentation: https://www.ampleforth.org/docs/

These were considered as the specification.

Summary

The manual line-by-line review of the source code revealed **no vulnerabilities or issues with the codebase** itself, confirming our initial interactions with Ampleforth that their team enforces **high security standards** in developing their codebase. **The team has demonstrated consistency and reliability with regards to their code ethics and practices**.

High-quality code was located in the repository with **comprehensive unit tests** that cover most of the would-be-faced use-cases as well as **extensive, easily digestible markdown files** that cover the documentation aspect of the project and contains **descriptive build and development instructions** ensuring that all **development carried out on the codebase is fully aligned and of the highest quality possible**.

Recommendations

The codebase of the project has demonstrated **a high degree of security** and as such, **no urgent recommendations are necessary**. Certain findings that **are informational and mostly act as advanced optimization steps** are included below to aid the Ampleforth team in bringing the optimization aspect of

their protocol another step further. However, all of these points are negligible and **do not impact the operational capacity and validity** of the codebase.

All the findings of the report were evaluated by the Ampleforth team in due time and were swiftly dealt with, once again reinforcing the idea that the team is capable of developing secure software in a professional manner and further elevating the standard their codebase possesses.

Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Unnecessary initialization of variable	Ineffectual Code	Informational	UFragmentsPolicy.sol Line 86

[INFORMATIONAL] Description:

The aforementioned line contains an assignment of the zero value casted to an “address” type to the “orchestrator” variable of the contract. This type of assignment is unnecessary as Solidity, akin to many other programming languages, assigns a default value to uninitialized variables which in the case of addresses is the zero address.

Recommendations:

We propose the removal of the assignment. The compiler may already optimize this step removing the assignment altogether so reduced gas costs on deployment may not be observed.

Alleviation:

The assignment was completely removed from the contract per our recommendation.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Transaction Struct Optimization	Code Optimization	Informational	Orchestrator.sol Line 15 - 19

[INFORMATIONAL] Description:

The “Transaction” struct included on the Orchestrator contract can be optimized to occupy one less slot, significantly reducing the gas cost of all types of transactions relating to it such as read and write operations.

Recommendations:

Solidity possesses what is known as a tight-packing mechanism. This means that, whenever possible, two variables that together occupy less than 32-bytes will be packed into a single 32-byte slot and be read and written to via batch operations halving the cost of operations applied on them. This depends on the ordering of the variable declarations and the compiler does not automatically “sort” variable declarations optimally.

We propose that the “Transaction” struct be restructured to have the declaration of an “address” type followed or preceded by the declaration of the “bool” type. In Solidity, an “address” occupies 160-bits whereas a “bool” occupies 8-bits and as such, both of these variables can fit into one 32-byte slot.

Alleviation:

The variables of the struct where re-ordered as per our recommendation to optimize gas costs.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Potentially Misleading Event Variable	Ineffectual Code	Informational	Orchestrator.sol Line 21

[INFORMATIONAL] Description:

The “TransactionFailed” event contains a “uint” variable that represents the index of the transaction that failed. However, this variable may be misleading under certain circumstances as transaction ordering in Ethereum is up to the miners of the blocks. As an example, should a single block contain the following transaction ordering, the “index” variable will be misleading and lead to an out-of-bound access on both the previous and current block of the blockchain:

1. Inclusion of new transaction in transaction queue. This will increase the transaction queue length and add the final transaction to the end.
2. Execution of “rebase” and failed execution of the newly included transaction, leading to the “TransactionFailed” event being emitted with the “index” variable being the last element of the array.
3. The deletion of the last transaction in the transaction queue. This will decrease the transaction queue length and remove the final transaction that was previously executed but failed.

The above sequence will lead to an “index” being emitted that is otherwise inaccessible. The above scenario, although unlikely, can occur should the owner of the contract accidentally include a transaction and attempt to remove it with users invoking “rebase” in the process.

Recommendations:

We propose the removal of the “index” variable from the event as it does not provide any surplus contextual information that would be helpful in debugging the failed transaction. The timestamp of the Event should be used instead as a provider of the transaction’s context of execution.

Alleviation:

After consideration by the Ampleforth team, they decided not to proceed with our recommendation as they stated that the current structure aids in debugging and is of minimal impact to their operation.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Potential Out-of-Gas Exception	Contract Freeze	Informational	Orchestrator.sol Line 51 - 60

[INFORMATIONAL] Description:

The loop included in the aforementioned lines iterates through an array of dynamic size that could potentially lead to the function being impossible to execute if many transactions are included or if the gas cost of the transactions collectively exceeds the block's gas limit.

Recommendations:

As the "transactions" array is fully malleable by the owner of the contract, we simply note this potential out-of-gas exception to be monitored by the Ampleforth team and appropriate remediations be applied on the "transactions" array should it occur. It is of negligible importance as it is an after-effect of what the Orchestrator contract is meant to achieve and is unavoidable unless complex batch-execution mechanisms are introduced that would defeat the purpose.

Alleviation:

This Exhibit is purely informational and the Ampleforth team has been properly informed and will keep track of the gas usage of the function to ensure they are fully capable of dealing with any issue that may arise.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION (OLD)
Unnecessary “delete” Call	Ineffectual Code	Informational	Orchestrator.sol Line 93

[INFORMATIONAL] Description:

Whenever the “length” of an array is reduced, Solidity internally calls the “delete” operator on the last element of the array and as such, it is unnecessary to explicitly call it.

Recommendations:

We propose the removal of the “delete” call on line 93 as it is internally called by the decrement operand applied on the “length” of the array on line 94.

Alleviation:

The Ampleforth team proceeded with removing the unnecessary “delete” call contained in the statements described above.

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Unnecessary Declaration of Variable in Function Name	Ineffectual Code	Informational	Orchestrator.sol Line 55, Line 128 (N125)

[INFORMATIONAL] Description:

The “externalCall” function contains a “transferValueWei” variable in its signature referring to the amount of wei that will be transferred in the external call. This function is labelled as “internal” and is only invoked in a single location, line 55, where the “transferValueWei” argument is hard-coded as zero.

Recommendations:

We propose the removal of the “transferValueWei” variable as it will always be zero since the function is “internal”. This will also reduce the gas cost of executing the transaction as no memory allocation will occur.

Alleviation:

The “externalCall” function signature was adjusted to reflect both the changes of Exhibit 6 and Exhibit 7, optimizing its execution and gas cost as a result.

Exhibit 7

TITLE	TYPE	SEVERITY	LOCATION (OLD & NEW)
Unnecessary Memory Transmission of Bytes Length	Ineffectual Code	Informational	Orchestrator.sol Line 55, Line 128 (N125), Line 138 – 139 (N146 – 152)

[INFORMATIONAL] Description:

The “externalCall” function contains a “dataLength” variable in its signature referring to the length of the “bytes” dynamic array that will be transferred in the external call. This is unnecessary because, as mentioned in comment line 138, the length of the “bytes” is already included in the variable at the first location of the “bytes” pointer.

Recommendations:

We propose the removal of the “dataLength” variable and the read of the “bytes” length directly in assembly by calling “mload” on the “bytes” pointer directly. This will once again reduce the gas cost of invoking the “externalCall” function of the contract.

Alleviation:

The “externalCall” function signature was adjusted to reflect both the changes of Exhibit 6 and Exhibit 7, optimizing its execution and gas cost as a result.

Exhibit 8

TITLE	TYPE	SEVERITY	LOCATION (OLD)
Possible Named Declaration of Return Variable	Code Optimization	Informational	Orchestrator.sol Line 130, Line 132, Line 141, Line 157

[INFORMATIONAL] Description:

The “externalCall” function returns a variable of type “bool” as stated in its function signature. Internally, an in-memory variable is declared called “result” of type “bool” which is assigned the result of the low-level “call” invocation and then explicitly returned as the last statement of the function body.

Recommendations:

We propose the naming of the return variable to “result” and the removal of lines 132 and 157 as named variables are automatically returned and exist in the scope of the function. This will not lead to a reduction of gas cost, however, it will make the code more legible.

Alleviation:

After consideration by the Ampleforth team, the team decided to move forward with the original declare & return pattern as it is consistent with the rest of their codebase.

Exhibit 9

TITLE	TYPE	SEVERITY	LOCATION
Unusual Function Naming Convention	Code Optimization	Informational	Orchestrator.sol Line 112

[INFORMATIONAL] Description:

The function “transactionsLength” is meant to return, as its name implies, the “length” of the “transactions” storage array. It is generally considered ill-practice to declare a dot-accessor in camel-case format as the function name.

Recommendations:

Instead, a more readable name could be utilized like “totalTransactions”. This is a very minor comment and is meant to aid in the readability and maintainability of the codebase as well as the reduction of issues by external users of the contract via its ABI.

Alleviation:

Our comment was assimilated by the Ampleforth team and they proceeded with renaming the function to “transactionsSize”, conforming to the convention of Ampleforth’s Market Oracle repository.

Exhibit 10

TITLE	TYPE	SEVERITY	LOCATION
Potential Optimization of Transaction Creation	Code Optimization	Informational	Orchestrator.sol Line 75

[INFORMATIONAL] Description:

This Exhibit relies on whether Exhibit 2 is applied on the codebase. If Exhibit 2 is applied, this Exhibit can be safely ignored.

In the aforementioned line, whenever a new “Transaction” struct is constructed the hard-coded value of “true” is assigned to its “enabled” member. This will presently cause a 32-byte write operation to be conducted as the memory needs to be initialized to a non-zero value.

Recommendations:

In Solidity, if a value remains uninitialized it will cost less gas-cost to retain rather than an initialized value. As such, it is possible to optimize this step by renaming the variable from “enabled” to “disabled” and setting the default value to “false”. This will cause transactions to cost less gas to create but increase the gas cost of creating them. As disabling transactions is anticipated to be seldomly used in contrast to creating them, applying the aforementioned change would lead to an overall reduction in gas consumption.

Alleviation:

Exhibit 2 was applied in full rendering this Exhibit null.