Ampleforth's Token Geyser v2

Security Assessment

February 12th, 2021

For :
Ampleforth's Token Geyser v2

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	Ampleforth's Token Geyser v2
Description	A time-based emission "geyser" of token rewards proportionate to the amount staked on the platform
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	1. c970676aaecb08e942fe1088a4b1ddcb26655fe6 2. 24a84284f937c9b6c3fc1c32aa7b34d67a6586bb

Audit Summary

Delivery Date	February 12th, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	February 3rd, 2021 - February 12th, 2021

Vulnerability Summary

Total Issues	23
Total Critical	0
Total Major	1
Total Medium	2
Total Minor	6
Total Informational	14

Executive Summary

We were contracted by the Ampleforth team to perform a security review of their Geyser v2 implementation. The v2 implementation of the Token Geyser differs greatly from its predecessor, adopting a token-agnostic nature and enabling any type of token to be utilized as a staking token and reward token. Additionally, the way the stake and unstake mechanisms work do not require token transfers to the contract itself and instead rely on a validated contract vault deployed utilizing the minimal proxy pattern.

As a result of these new features, the Token Geyser was re-written and thus does not rely on any code from its predecessor. Over the course of the audit, we validated that all state transitions occur within sensible bounds and that the new mechanisms introduced for balance keeping operate sanely. To this end, we identified a major flaw in the way the balance sheet of the vault is retained that allows a single vault to be utilized across two seperate geyser contracts with the same balance provided that same underlying token utilized by the geysers is the same.

We pinpointed issues relating to certain novel concepts introduced in v2 that should be remediated as soon as possible by the Ampleforth team. Along with the security-related exhibits, we pointed out certain informational-level exhibits that we believe can greatly optimize the system in terms of gas cost and generated bytecode.

We were able to identify more optimizations relating to over-utilization of certain security principles, such as `SafeMath`, but chose not to include them in the report and instead relay them to the Ampleforth team directly as they can remain in the codebase for readability purposes.

Files In Scope

ID	Contract	Location
EIP	EIP712.sol	contracts/Access/EIP712.sol
ERC	ERC1271.sol	contracts/Access/ERC1271.sol
GEY	Geyser.sol	contracts/Geyser.sol
GRY	GeyserRegistry.sol	contracts/Factory/GeyserRegistry.sol
IFY	IFactory.sol	contracts/Factory/IFactory.sol
IER	IERC20Permit.sol	contracts/Libraries/IERC20Permit.sol
IRY	InstanceRegistry.sol	contracts/Factory/InstanceRegistry.sol
OER	OwnableERC721.sol	contracts/Access/OwnableERC721.sol
POW	Powered.sol	contracts/PowerSwitch/Powered.sol
PSH	PowerSwitch.sol	contracts/PowerSwitch/PowerSwitch.sol
PSF	PowerSwitchFactory.sol	contracts/Factory/PowerSwitchFactory.sol
RV1	RouterV1.sol	contracts/RouterV1.sol
RPL	RewardPool.sol	contracts/RewardPool.sol
RPF	RewardPoolFactory.sol	contracts/Factory/RewardPoolFactory.sol
SPA	Spawner.sol	contracts/Factory/Spawner.sol
UVT	UniversalVault.sol	contracts/UniversalVault.sol
VFY	VaultFactory.sol	contracts/Factory/VaultFactory.sol

File Dependency Graph (BETA)

Findings

ID	Title	Type	Severity	Resolved
EIP-01	Mutability Specifiers Missing	Gas Optimization	Informational	✓
IRY-01	Redundant Code	Dead Code	Informational	✓
VFY-01	Mutability Specifiers Missing	Gas Optimization	Informational	✓
VFY-02	Input Sanitization	Logical Issue	Minor	✓
POW-01	Modifier `require` To Function Call	Gas Optimization	Informational	✓
PSH-01	Input Sanitization	Logical Issue	Minor	✓
UVT-01	Invalid Balance Sheet Evaluation	Logical Issue	Major	✓
UVT-02	Dynamic Evaluation of Loop Length	Gas Optimization	Informational	✓
UVT-03	Potential Misbehaviour of the System	Logical Issue	Minor	✓
UVT-04	Dynamically Computed Static Value	Gas Optimization	Informational	✓
UVT-05	Signature Validation Race Condition	Logical Issue	Medium	✓
UVT-06	Insufficient Prevention of Allowance	Logical Issue	Minor	✓
RV1-01	Function Visibility Optimization	Gas Optimization	Informational	✓
GEY-01	Variable Shadowing	Data Flow	Informational	✓
GEY-02	Bytecode Optimization	Gas Optimization	Informational	✓
GEY-03	Conditional Optimization	Gas Optimization	Informational	✓
GEY-04	Inversion of `if` Clause	Gas Optimization	Informational	✓

ID	Title	Type	Severity	Resolved
GEY-05	Denial-of-Service Attack	Logical Issue	Minor	✓
GEY-06	Potential of Zero Transfer	Logical Issue	Minor	✓
GEY-07	Potentially Misutilized Implementation	Gas Optimization	Informational	✓
GEY-08	Unnecessarily Convoluted Logic	Gas Optimization	Informational	✓
GEY-09	Function Visibility Optimization	Gas Optimization	Informational	✓
GEY-10	Function Comment Inconsistency	Inconsistency	Medium	✓

EIP-01: Mutability Specifiers Missing

Type	Severity	Location
Gas Optimization	Informational	EIP712.sol L26-L31

Description:

The linked variables are assigned to only once, either during their contract-level declaration or during the `constructor`'s execution.

Recommendation:

For the former, we advise that the `constant` keyword is introduced in the variable declaration to greatly optimize the gas cost involved in utilizing the variable. For the latter, we advise that the `immutable` mutability specifier is set at the variable's contract-level declaration to greatly optimize the gas cost of utilizing the variables. Please note that the `immutable` keyword only works in Solidity versions `v0.6.5` and up.

Alleviation:

The team introduced the `immutable` mutability specifiers to the linked declarations thus optimizing the codebase.

IRY-01: Redundant Code

Type	Severity	Location
Dead Code	Informational	InstanceRegistry.sol L51-L54

Description:

The implemented function `_unregister` is meant to remove an instance from the `_instanceSet` and emit a corresponding event, however, it remains unutilized throughout the project.

Recommendation:

We advise that the function is either properly utilized via the derivative contracts such as `GeyserRegistry`, or that the function and associated `event` declaration are removed completely from the codebase to reduce bytecode.

Alleviation:

The linked function was completely omitted from the codebase.

VFY-01: Mutability Specifiers Missing

Type	Severity	Location
Gas Optimization	Informational	VaultFactory.sol L13

Description:

The linked variables are assigned to only once, either during their contract-level declaration or during the `constructor`'s execution.

Recommendation:

For the former, we advise that the `constant` keyword is introduced in the variable declaration to greatly optimize the gas cost involved in utilizing the variable. For the latter, we advise that the `immutable` mutability specifier is set at the variable's contract-level declaration to greatly optimize the gas cost of utilizing the variables. Please note that the `immutable` keyword only works in Solidity versions `v0.6.5` and up.

Alleviation:

The linked variable was properly set to be `immutable` optimizing the gas cost involved in utilizing it.

VFY-02: Input Sanitization

Type	Severity	Location
Logical Issue	Minor	VaultFactory.sol L15

Description:

The `constructor` of the `VaultFactory` contract accepts a single `address` argument that remains immutable beyond its assignment and is used as the underlying implementation of spawned instances. However, no check is imposed in the `constructor` to ensure that it is not accidentally set to the `0x0` address.

Recommendation:

We advise that a `require` check is imposed here to ensure the address is non-zero.

Alleviation:

Input sanitization for the `template` of the `constructor` was properly introduced.

POW-01: Modifier `require` To Function Call

Type	Severity	Location
Gas Optimization	Informational	Powered.sol L28-L46

Description:

In Solidity, `modifier`s work by essentially wrapping the function they are utilized in with the corresponding code of the `modifier`, either appending or prepending statements. As `require` calls with an error message significantly increase the bytecode size and gas cost, it is more optimal to instead have the `modifier` implementations perform a function call, leading to the `require` checks not being duplicated and instead being existent on a single location.

Recommendation:

We advise that the `require` calls are swapped with internal calls performing the same checks to reduce the bytecode size of the contract as well as gas cost.

Alleviation:

The `modifier` implementations were properly refactored to utilize internal function calls thus greatly optimizing the resulting bytecode size of all contracts that inherit them.

PSH-01: Input Sanitization

Type	Severity	Location
Logical Issue	Minor	PowerSwitch.sol L48

Description:

The `constructor` of the `PowerSwitch` contract transfers ownership of itself to the address provided as input to it. However, no check exists that ensures the `owner` is non-zero in either the `PowerSwitch` implementation or parent contracts, such as `Geyser`, that create instances of it.

Recommendation:

We advise that a check is imposed on the specified address that ensures it is non-zero.

Alleviation:

The `owner` variable of the `constructor` is properly validated via `require` checks in the latest version of the codebase.

UVT-01: Invalid Balance Sheet Evaluation

Type	Severity	Location
Logical Issue	Major	UniversalVault.sol L237-L247

Description:

The `checkBalances` function is meant to iterate over all locks existent on the vault and ensure that the locked balances do not exceed the amount of tokens held by the vault.

The issue with the current implementation is that the loop iteration between L239 and L244 checks the balances of the locks sequentially whilst the same token can exist under two different lock IDs within the `_lockSet` set.

For example, if two different `Geyser`s rely on the same vault, the same token will be "locked" under two different lock IDs, for the sake of this example let's consider that value to be `100`.

If I were to maliciously call either `externalCall` or `externalCallMulti`, I would be able to withdraw `100` of the `200` units held by the vault as the two locks created above would each be `100` which would successfully pass the check imposed on L243.

Recommendation:

We advise that the balance sheet evaluation mechanism is refactored to account for duplicate tokens existing within the lock set. Various schemes can be utilized such as the prohibition of a token being registered by another address, the `checkBalances` mechanism to accumulate locked balances and more. The most sensible and gas-optimized solution should be utilized by the Ampleforth team.

Alleviation:

The Ampleforth team responded by stating that this is expected behaviour, as the `Geyser`s are not meant to validate whether the supply of tokens is being staked on other `Geyser`s as long as the desired balance is simply locked. As such, this exhibit is rendered null.

UVT-02: Dynamic Evaluation of Loop Length

Type	Severity	Location
Gas Optimization	Informational	UniversalVault.sol L229, L239

Description:

The linked `for` loops iterate from `0` until a specified length that is the result of a function invocation on the `_lockSet`. As the conditional statement is evaluated on each iteration, it is more optimal to store the length evaluation in-memory prior to the loop to optimize the conditional evaluation.

Recommendation:

We advise that the advice provided in the exhibit's description is assimilated in the codebase.

Alleviation:

The linked loops were adjusted to properly cache the loop length in memory instead of evaluating it on each iteration.

UVT-03: Potential Misbehaviour of the System

Type	Severity	Location
Logical Issue	Minor	UniversalVault.sol L278-L293

Description:

The `externalCallsMulti` function iterates and executes all `calls` provided to it before evaluationg that the balance sheet of the vault is correct and finalizing the function's execution. This allows one to actually withdraw tokens and utilize them prior to returning them in the sequence of external calls performed by the contract which may be an undesired capability of the system.

Recommendation:

We advise that this feature is documented if desired or prohibited by evaluating the balance sheet on each invocation.

Alleviation:

The ability to perform arbitrary calls was completely omitted from the system thus rendering this exhibit void.

UVT-04: Dynamically Computed Static Value

Type	Severity	Location
Gas Optimization	Informational	UniversalVault.sol L314, L370

Description:

The linked lines compute the `keccak256` value of `string` literals.

Recommendation:

We advise that the full evaluations are instead stored as `constant` contract-level variables. Constant variables are hot-swapped during compilation and are as such safe to utilize in proxied contracts.

Alleviation:

The `keccak256` computed values were properly stored in contract-level `constant` variable declarations.

UVT-05: Signature Validation Race Condition

Type	Severity	Location
Logical Issue	Medium	UniversalVault.sol L305-L400

Description:

The linked functions `lock` and `unlock` perform signature validation based on a globally incremented `nonce` on both functions regardless of the party that is providing the signature. This introduces a race condition whereby a validly signed `lock` can be invalidated by submitting another validly signed `lock` with a higher gas fee. This can render the system unusable in a real-world use case with high influx of activity as lock / unlock signature nonces will be overlapping between users.

Recommendation:

We advise that a seperate `mapping` is introduced that keeps track of the `nonce` of each account which is subsequently relayed by the `Geyser` or accessed via `tx.origin`, depending on the types of addresses the `Geyser` should properly support.

Alleviation:

The Ampleforth team decided to retain the current implementation as is and stated that the race condition edge case will instead be handled by the UI layer. We should note that any third party integration of the contracts will also inherit this issue and as such, we adivse that comments regarding it are included in the function declaration for the sake of brevity.

UVT-06: Insufficient Prevention of Allowance

Type	Severity	Location
Logical Issue	Minor	UniversalVault.sol L450-L465

Description:

The `_externalCall` implementation is meant to perform a function call that conducts an arbitrary action except from approving another address of an allowance, presumably to ensure that funds aren't exited from the vault after the `checkBalances` evaluation successfully passes. This check, however, is insufficient as a lot of contracts, including the Ampleforth token itself (AMPL), derive from OpenZeppelin and support the `increaseAllowance` and `decreaseAllowance` functions, thus circumventing the check.

Recommendation:

We advise that these two widely implemented functions are also added to the list of prohibited function calls. We should note, however, that each token implementation differs and the approval mechanism may be circumvented via other means. As such, the introduction of new tokens supported by the `Geyser` as staking tokens should be properly vetted to not allow such an incident to occur.

Alleviation:

The ability to perform arbitrary calls was completely omitted from the system thus rendering this exhibit void.

RV1-01: Function Visibility Optimization

Type	Severity	Location
Gas Optimization	Informational	RouterV1.sol L26-L33, L56-L63, L105-L110, L120-L130

Description:

The linked function is declared as `public`, contains array function arguments and is not invoked in any of the contract's contained within the project's scope.

Recommendation:

We advise that the functions' visibility specifiers are set to `external` and the array-based arguments change their data location from `memory` to `calldata`, optimizing the gas cost of the function.

Alleviation:

All functions were properly optimized by specifying their visibility as `external` and optimizing the argument data locations were applicable to `calldata`.

GEY-01: Variable Shadowing

Type	Severity	Location
Data Flow	Informational	Geyser.sol L260

Description:

The linked variable shadows an existing declaration in a parent contract, `OwnableUpgradeable`, of the `owner` function that retrieves the owner of the contract.

Recommendation:

We advise that the `initialize` variable is renamed to ensure no such shadowing occurs, even though it does not pose an issue in the current implementation.

Alleviation:

The variable was properly renamed to `ownerAddress` preventing the naming collission.

GEY-02: Bytecode Optimization

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L375-L398, L411-L442, L444-L479

Description:

The linked functions differ in the code they execute by a single variable which is passed as a literal in one implementation and as a variable in the other.

Recommendation:

We advise that the literal-using implementation invokes the variable-using implementation to reduce the bytecode of the contract significantly.

Alleviation:

The functions were simplified where possible by introducing inward calls with any additional arguments necessary.

GEY-03: Conditional Optimization

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L391-L397

Description:

The `newStakeUnits` calculated depend on whether time has passed between the `timestamp` and the `lastUpdate` of the `_geyser`, meaning that if those two are equal no change will occur.

Recommendation:

We advise that such an `if` conditional is introduced that returns the `_geyser.totalStakeUnits` immediately.

Alleviation:

The `timestamp` based conditional was introduced to optimize the gas cost of the function in case no change has occured.

GEY-04: Inversion of `if` Clause

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L556-L562, L663-L669

Description:

The linked `if` clauses perform a conditional check whereby within the value literal `0` is assigned to the variable declared before the `if` block. As `uint` variables are initialized at `0` by default in Solidity, it is possible to invert the `if` clause and drop the `else` leg entirely.

Recommendation:

We advise that the optimization described in the exhibit's description is applied to the codebase.

Alleviation:

The `if` clauses were properly inverted optimizing their gas cost.

GEY-05: Denial-of-Service Attack

Type	Severity	Location
Logical Issue	Minor	Geyser.sol L997-L1015

Description:

Although this particular attack vector requires escalated privileges, it is possible to freeze any `unstakeAndClaim` invocations by introducing numerous bonus tokens to the `_bonusTokenSet` thus causing the loop iteration of L998-L1014 to run out-of-gas.

Recommendation:

We advise that a limit is imposed on the number of bonus tokens at the setter function located between L802-L811 to ensure no malicious party is able to act in this way.

Alleviation:

A `require` check was introduced in the code segment that introduces new bonus tokens preventing a prohibitively expensive number of tokens to be introduced to the bonus token system.

GEY-06: Potential of Zero Transfer

Type	Severity	Location
Logical Issue	Minor	Geyser.sol L1010

Description:

Certain tokens throw when a zero-value transfer is attempted, meaning that the `unstakeAndClaim` mechanism may break if all bonus tokens of such a token have been claimed.

Recommendation:

We advise that the linked statement is wrapped in an `if` clause that ensures the `reward` to be paid out is non-zero.

Alleviation:

The Ampleforth's Token Geyser v2 development team has acknowledged this exhibit but decided to not apply its remediation in the current version of the codebase due to time constraints.

GEY-07: Potentially Misutilized Implementation

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L1082-L1092

Description:

The `_truncateStakesArray` implementation is meant to remove a number of `StakeData` members from the input `array` by initializing a new one in-memory, assigning to it in a sequential fashion and returning the new array copy. However, the function is solely utilized in L617 to remove the last stake of the array.

Recommendation:

We advise that the `calculateRewardFromStakes` function is refactored to conduct a truncation at the end of the `while` loop within the `return` statement to greatly optimize the gas cost of the function. This can be achieved by retaining an in-memory `uint256` variable tracking the number of stakes that should be omitted from the array at the end of the functtion.

Alleviation:

The code was adjusted to track the total number of stakes to drop from the array, however, a new issue has been introduced whereby the `_truncateStakesArray` is actually not invoked with the `stakesToDrop` argument, never reducing the size of the array. We strongly suggest that this issue is remediated as soon as possible.

GEY-08: Unnecessarily Convoluted Logic

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L957-L978

Description:

The linked code segment executes the `calculateRewardFromStakes` function which truncates the stakes consumed to produce the reward and returns the truncated array. In the code segment linked, the truncated array is not utilized directly and only the last remaining element is used along with the array's `length`.

Recommendation:

We advise that the `calculateRewardFromStakes` function is refactored to return the last `StakeData` processed and the number of elements that should be removed from the end of the array via `pop`, greatly optimizing the gas cost of this code segment.

Alleviation:

The `calculateRewardFromStakes` function was adjusted to instead return a `struct` with the necessary data to conduct the optimizations linked in the recommendation section of this exhibit, thus greatly optimizing the linked segment.

GEY-09: Function Visibility Optimization

Type	Severity	Location
Gas Optimization	Informational	Geyser.sol L861-L865, L919-L924

Description:

The linked function is declared as `public`, contains array function arguments and is not invoked in any of the contract's contained within the project's scope.

Recommendation:

We advise that the functions' visibility specifiers are set to `external` and the array-based arguments change their data location from `memory` to `calldata`, optimizing the gas cost of the function.

Alleviation:

The visibility and data location optimizations were properly applied to the linked functions.

GEY-10: Function Comment Inconsistency

Type	Severity	Location
Inconsistency	Medium	Geyser.sol L856, L887-L889

Description:

The `stake` function comments denote that the `totalStakeUnits` should be adjusted when a `stake` occurs. However, the implementation of `stake` adjusts the units only before submitting the new stake and does not update the `totalStakeUnits` after, as `unstakeAndClaim` does.

Recommendation:

We advise that this discrepancy is investigated and properly remediated to ensure the accounting mechanisms of the `Geyser` operate correctly.

Alleviation:

The Ampleforth team has responded and stated that "This is expected behavior as modifying `totalStakeUnits` in the `stake` function after the initial update would be a noop given no time has elapsed.

The comment mentioning `increase _geyser.totalStakeUnits` is meant to represent the modification made by the `_updateTotalStakeUnits()` call exclusively.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.