HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: DAOhaus
Date: September 01st, 2022

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for DAOhaus
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; DAO
Platform	EVM
Network	Ethereum, BSC
Language	Solidity
Methods	Manual Review, Automated Review, Architecture Review
Website	https://daohaus.club/
Timeline	03.08.2022 - 01.09.2022
Changelog	15.08.2022 - Initial Review 01.09.2022 - Second Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by DAOhaus (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project is smart contracts in the repository:

Initial review scope

Repository:
https://github.com/HausDAO/Baal/tree/milestone/audit-715
Commit:
e704c3cc87684b1d5f2b7ad0e217e6a9dfe2f19c
Technical Documentation:
Whitepaper (partial functional requirements provided)

Technical description

Functional requirements

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/Baal.sol
SHA3: 8dd88c20f18a1c04cc305e1deb1db772f29c906747185c855006a88565e548b2

File: ./contracts/SharesERC20.sol
SHA3: 8dc857c93d5a5d2d0ec700ca5b2c96c8ae53e40c50a631be71b29ba6053a0387

File: ./contracts/LootERC20.sol
SHA3: 054e2a93e64ae43d0d2f6fcc8864f26581ac67f639f2b2a981662e80d3c58596

File: ./contracts/tools/TributeMinion.sol
SHA3: 7e1d74829af4dc6bdf4f2a30549bf74554560121750651b3496a767e6f1979a9

File: ./contracts/tools/Poster.sol
SHA3: 0af8c54e5f3ea31afa2ff3c8de5d448060a1cd2d89ddd2c3e3d7aea7a7db9b69

File: ./contracts/mock/TestAvatar.sol
SHA3: 433708291fa939afb6ef1501f66b940c83b46bed4c4f4e7423e9f5cdc35ff0de

File: ./contracts/mock/TestERC20.sol
SHA3: abd3e3b21b423596f141f4d02159a13a4ffd2a5468380007eea5599fc033b097

File: ./contracts/mock/MockBaal.sol
SHA3: 59666ca0042468722ee8ed910598485917bf74dd3a942fcf5db6c123a56bf3b6

File: ./contracts/fixtures/GnosisImports.sol
SHA3: 43e50a9258d54c7d5c2f024b5a80310c58920ebb7dbe40de358df1c3e4f5a343

File: ./contracts/interfaces/IBaal.sol
SHA3: 9cbd7ed49c8267414ff18ab9da502cd8faa4ced6a45165b380626082093180f3

Second review scope

Repository:

https://github.com/HausDAO/Baal/commit/15bf835955e20c75c47e2d3d89341b394607d691

Commit:
15bf835955e20c75c47e2d3d89341b394607d691
Technical Documentation:
Whitepaper (partial functional requirements provided)

Technical description

Functional requirements

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/Baal.sol
SHA3: d245813820963bfacabefbc9006a9c169a03e70718b374f68e9ab1224a9cb579

File: ./contracts/SharesERC20.sol
SHA3: 8dc857c93d5a5d2d0ec700ca5b2c96c8ae53e40c50a631be71b29ba6053a0387

File: ./contracts/LootERC20.sol
SHA3: 054e2a93e64ae43d0d2f6fcc8864f26581ac67f639f2b2a981662e80d3c58596

File: ./contracts/tools/TributeMinion.sol
SHA3: 7e1d74829af4dc6bdf4f2a30549bf74554560121750651b3496a767e6f1979a9

File: ./contracts/tools/Poster.sol
SHA3: 0af8c54e5f3ea31afa2ff3c8de5d448060a1cd2d89ddd2c3e3d7aea7a7db9b69

File: ./contracts/mock/TestAvatar.sol
SHA3: 433708291fa939afb6ef1501f66b940c83b46bed4c4f4e7423e9f5cdc35ff0de

File: ./contracts/mock/TestERC20.sol
SHA3: abd3e3b21b423596f141f4d02159a13a4ffd2a5468380007eea5599fc033b097

File: ./contracts/mock/MockBaal.sol
SHA3: 59666ca0042468722ee8ed910598485917bf74dd3a942fcf5db6c123a56bf3b6

File: ./contracts/fixtures/GnosisImports.sol
SHA3: 43e50a9258d54c7d5c2f024b5a80310c58920ebb7dbe40de358df1c3e4f5a343

File: ./contracts/interfaces/IBaal.sol
SHA3: 9cbd7ed49c8267414ff18ab9da502cd8faa4ced6a45165b380626082093180f3

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions.
Medium	Medium-level vulnerabilities are important to fix; however, they cannot lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the methodology.

Documentation quality

The total Documentation Quality score is **6** out of **10**. The Customer provided superficial functional and technical documentation.

Code quality

The total CodeQuality score is **10** out of **10**.

Architecture quality

The architecture quality score is **10** out of **10**.

Security score

As a result of the audit, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.6**.

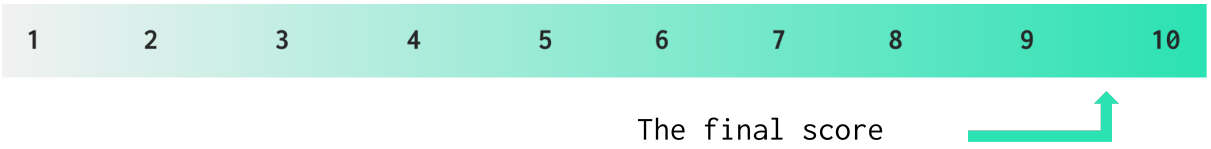

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
5 August 2022	7	2	1	0
01 September	0	0	0	0

Checked Items

We have audited provided smart contracts for commonly known and more specific vulnerabilities. Here are some of the items that are considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Passed
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless it is required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed

Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifier should always be used. All parameters from the signature should be used in signer recovery	Not Relevant
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP standards violation	EIP	EIP standards should not be violated.	Passed
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Passed
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer.	Passed
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style guide violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed

Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

System Overview

Baal is a minimal yet composable DAO template continuing work from the Moloch, Minion and Compound frameworks to make it easier for people to combine and command crypto assets with intuitive membership games. It has the following contracts:

- *Baal* – is a minimal yet composable DAO template continuing work from the Moloch, Minion and Compound frameworks to make it easier for people to combine and command crypto assets with intuitive membership games.
- *Shares* – have direct execution, voting and exit rights around actions taken by the main DAO contract. Shareholders are the collective DAO admins.
- *Loot* - has only exit rights against the DAO treasury, so loot does not have the ability to admin the DAO config. However, because it has exit rights, it is still a powerful unit, and because it is an ERC-20 can be used in many composable ways.
- *TributeMinion* - is a helper contract for making tribute proposals. Provides contract to approve ERC-20 transfers. Provides a simple function/interface to make a single proposal type.
- *Poster* - is a simple function that posts some data to events, these events can then be indexed for access by frontends; sed for all types of content and metadata capture.

Privileged roles

- Shamans - are specific addresses that have more granular control outside the standard governance proposal flow. These addresses should always be contracts that have been explicitly given these rights through the standard proposal flow or during initial DAO setup.
- Governor - can cancel a proposal, set Governance Config (change the length of proposals, if there is a required quorum, etc.).
- Manager - can mint/burn shares/loot.
- Admin - can set Admin configuration and pause/unpause shares/loot.
- DAO - is always a super admin over its config. Can vote to make changes to its configuration at any time.

Risks

- In case of Baal keys leak, an attacker can get access to Baal (admin) functionalities, burn, mint, give shaman roles etc.
- The Baal contract uses the getPriorVotes function that accepts timestamp instead of commonly used block number. The developers should ensure that they use the correct implementation of the token.

Findings

Critical

No high severity issues were found.

High

1. Library code should not be copied

Code from the popular OpenZeppelin library is copied into the codebase.

This leads to an unnecessary increase in the audit scope and introduces accidental change risks to otherwise safe and audited code.

Paths: ./contracts/LootERC20.sol : transferFrom, name, symbol

./contracts/SharesERC20.sol : transferFrom, name, symbol

Recommendation: Remove the copy-pasted code. Remove overriding for transferFrom function.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

Medium

1. Denial of Service vulnerability

External calls can fail accidentally or deliberately, which can cause a DoS condition in the contract. Inside the Baal's totalSupply function, there are two external calls 'lootToken.tokenSupply' and 'sharesToken.tokenSupply'.

This can lead to DoS condition in the contract

Path: ./contracts/Baal.sol : totalSupply()

Recommendation: Isolate each external call into its own transaction that can be initiated by the recipient of the call.

Status: Mitigated (with Customer notice)

2. Assembly usage

CloneFactory implements 'createClone' functionality using assembly. Assembly usage can lead to error in implementation.

Path: ./contracts/Baal.sol : CloneFactory:createClone(address)

Recommendation: Use clone functionality from OpenZeppelin library.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

Low

1. **Floating pragma**

   Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

   The project uses floating pragmas 0.8.0.

   Paths: ./contracts/Baal.sol

   ./contracts/LootERC20.sol

   ./contracts/SharesERC20.sol

   ./contracts/interfaces/IBaal.sol

   ./contracts/mock/MockBaal.sol

   ./contracts/mock/TestAvatar.sol

   ./contracts/mock/TestERC20.sol

   ./contracts/tools/Poster.sol

   ./contracts/tools/TributeMinionr.sol

   Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

   Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

2. **State variable default visibility**

   Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

   Paths: ./contracts/Baal.sol : status, multisendLibrary, gnosisSafeProxyFactory, moduleProxyFactory

   ./contracts/tools/TributeMinion.sol : escrow

   Recommendation: Variables can be specified as being public, internal, or private. Explicitly define visibility for all state variables.

   Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

3. **Variable shadowing**

   Solidity allows for ambiguous naming of state variables when inheritance is used.

   Loot's and Shares state variables '_name' and '_symbol' shadow ERC20 state variables.

   Paths: ./contracts/LootERC20.sol : _name, _symbol

./contracts/SharesERC20.sol : _name, _symbol

Recommendation: Rename related variables/arguments.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

4. **Commented code parts**

Commented parts of code in a contract. They will not cause any security issues, but make code less clear.

In the contracts : Shares (lines 116-120, 125, 137, 233) TributeMinion (lines 5, 126, 133, 138), Baal (lines 381, 1164) are commented parts of code.

This reduces code quality.

Paths: ./contracts/LootERC20.sol

./contracts/SharesERC20.sol

./contracts/Baal.sol

Recommendation: Remove commented parts of code.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

5. **Unused variable**

Unused variables should be removed from the contracts. Unused variables are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease the readability.

The variable ‘nonces’ is never used inside the Baal contract.

Path: ./contracts/Baal.sol

State variable : nonces

Recommendation: Remove unused variables.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

6. **Redundant import**

The use of unnecessary imports will increase the Gas consumption of the code. Thus they should be removed from the code.

The second usage of Enum.sol is unnecessary for the Baal.sol contract.

Path: ./contracts/Baal.sol

Import: "@gnosis.pm/safe-contracts/contracts/common/Enum.sol"

Recommendation: Remove the duplicate import.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

7. **Missing zero address validation**

Address parameters inside the BaalSammoner contract are being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Path: ./contracts/Baal.sol

Constructors: _lootSingleton, _sharesSingleton, _gnosisSingleton

Recommendation: Remove the duplicate import.

Status: Fixed (Revised commit: 15bf835955e20c75c47e2d3d89341b394607d691)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted to and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, Consultant cannot guarantee the explicit security of the audited smart contracts.