Introduction

This document describes the 0.8 version of our procedure for conducting a Process Quality Review, or PQR, on a DeFi protocol's deployed smart contracts and documentation. The focus of this process is on DeFi contracts.

In passing these protocols through these quantitative tests, we create a simple quality score for the smart contract application being reviewed. This score will indicate the overall quality of the development process and the documents that the development team created during the creation process of their protocol. The reader is encouraged to dig into the details of the review in order to see exactly how the score was generated. Every step will be documented and available for critique, as DeFiSafety wants as much feedback on the process as we can receive.

The basic premise of these reviews is that developers following, documenting and maintaining a good software development process should have secure code and maintain a level of security that users can trust. For blockchain projects, a good process should have public and readily auditable documents in addition to other measures - all of which makes the development process clear and transparent.

These reviews are developed and focused towards deployed and operating smart contract applications. Our initial focus will be on DeFi protocols as these bring in many new users investing significant sums, and they must be able to trust the smart contracts they interface with.

PQRs are initially written without the developers' support using exclusively documents that are publicly available such as the website, the software repository of the code (GitHub, etc), and the Contract Address analytics available from the relevant blockchain explorer (Etherscan, BscScan, etc). After the initial Process Quality Review is curated, it is presented to the developers of the reviewed protocol for correction or improvement. The results of these corrections will be clear and publicly documented in a new version of the report. We do this because we want protocols to score highly and it is clear by their cooperation that they do too. This is their opportunity to improve their score, and our opportunity to provide any help we can.

The author has no commercial arrangements with the organizations being audited. These reviews are not funded by the organization being reviewed. Score improvements or initial reviews cannot be bought.

About the Process

The initial author of this process has a long history in the avionics industry. Aerospace software development has always maintained that a rigorous and frequently audited software development process leads to safe and secure software that can be supported for many decades. The avionics process is DO-178C. It is significantly more rigorous than the expectation of our DeFi review process, however the steps used provide an overarching philosophy that has guided the specification of this review process.

Aerospace Requirements

For more detail, this presents the software and system requirements for aerospace code certification in an extremely simplified format. It makes a useful comparison.

1. All System Requirements documented in relevant and proper language.

2. There is strong and clear documented traceability from each system requirement to software requirements or low level requirements.

3. All software requirements (or low-level requirements/comments) are met for each part of the software code.

4. There is documented traceability from the low level requirements to the software code they cover.

5. Every piece of software code is covered by both unit tests and system tests. If a unit test covers the requirements of the system test, the single test is sufficient.

6. There is documented traceability from the software to the tests.

7. There were frequent review meetings held according to a documented process to review every requirement and ensure that the software was in compliance.

8. There are documented software requirement reviews held according to a predetermined and documented process to ensure compliance with each software requirement.

9. There are test reviews held in compliance with a predetermined process for each test.

10. When there is a requirement to change software code that has already met the requirements, the code or test must change. After this, a change impact analysis document will be conducted which reviews the changes and recommends which tests should reoccur again on the requirements, software and tests.

11. During the audit any test can be chosen by the auditor and the auditor has to be able to review traceability from the test through the code to the requirements with reviews for each step in the process.

Disclaimer

Summary of the Process

Very simply, the review looks for the following declarations from the developer's site. With these declarations, it is reasonable to trust the smart contracts.

- Here are my smart contracts on the blockchain
- Here is the documentation that explains what my smart contracts do
- Here are the tests I ran to verify my smart contract
- Here are the audit(s) performed on my code by third party experts
- Here are the admin admin controlss and how they're used
- This is where the protocol gets its data, and these are some proactive steps we've taken to avoid problems.

Sections of the Review Process

The process breaks the scores into the following sections:

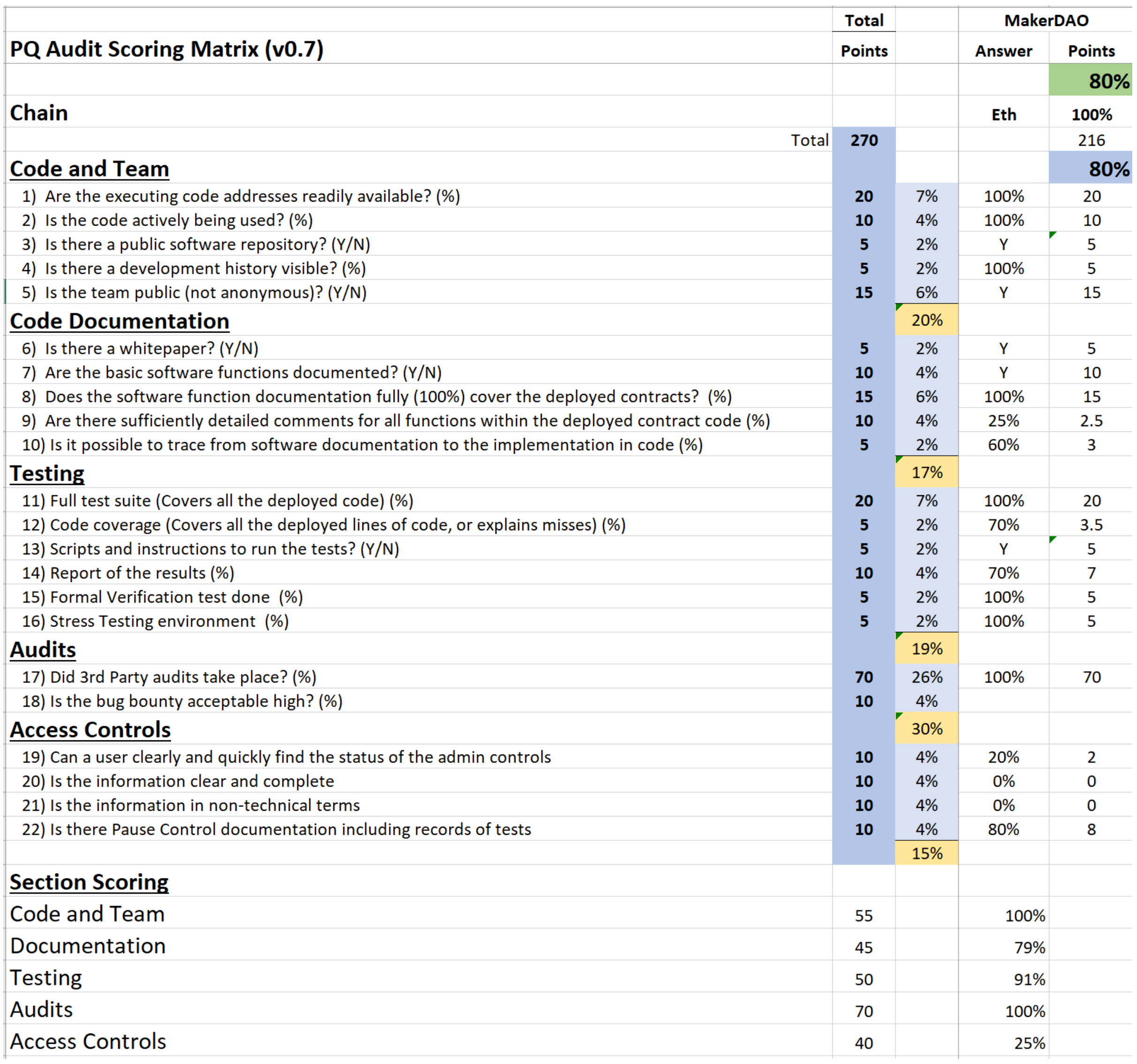
- Code and Team -- Deployed Code Verification
- Documentation -- Software Documentation for the Deployed Code
- Test -- Overall testing measures for the Deployed Code
- Security -- Review of the Software Security Audits and Bug Bounty
- admin controls -- Review of the public information about admin admin controls
- Oracles -- Explanation of the data sources used by the protocol and identification of potential vulnerabilities

Scoring

A review's Final Score is indicated as a percentage. This percentage is calculated as total Achieved Points divided by the total Possible Points. For each question the answer can be either yes (Y), no (N), or a percentage (%). Each of these questions has a "Scoring Weight", as some are more important than others. For example, “Question 1” is more important than “Question 15”, and therefore has a higher weight.

The individual question’s Achieved Points is the Scoring Weight times the answer (yes, no, %). The review’s Total Achieved Points is the sum of every question’s points. For our purposes, a passing score is one that receives 70% or more.

Please see this example of a scoring matrix for reference:



	Total		MakerDAO	
PQ Audit Scoring Matrix (v0.7)	**Points**		**Answer**	**Points**
				80%
Chain			**Eth**	**100%**
Total	**270**			216
Code and Team				**80%**
1) Are the executing code addresses readily available? (%)	**20**	7%	100%	20
2) Is the code actively being used? (%)	**10**	4%	100%	10
3) Is there a public software repository? (Y/N)	**5**	2%	Y	5
4) Is there a development history visible? (%)	**5**	2%	100%	5
5) Is the team public (not anonymous)? (Y/N)	**15**	6%	Y	15
Code Documentation		20%		
6) Is there a whitepaper? (Y/N)	**5**	2%	Y	5
7) Are the basic software functions documented? (Y/N)	**10**	4%	Y	10
8) Does the software function documentation fully (100%) cover the deployed contracts? (%)	**15**	6%	100%	15
9) Are there sufficiently detailed comments for all functions within the deployed contract code (%)	**10**	4%	25%	2.5
10) Is it possible to trace from software documentation to the implementation in code (%)	**5**	2%	60%	3
Testing		17%		
11) Full test suite (Covers all the deployed code) (%)	**20**	7%	100%	20
12) Code coverage (Covers all the deployed lines of code, or explains misses) (%)	**5**	2%	70%	3.5
13) Scripts and instructions to run the tests? (Y/N)	**5**	2%	Y	5
14) Report of the results (%)	**10**	4%	70%	7
15) Formal Verification test done (%)	**5**	2%	100%	5
16) Stress Testing environment (%)	**5**	2%	100%	5
Audits		19%		
17) Did 3rd Party audits take place? (%)	**70**	26%	100%	70
18) Is the bug bounty acceptable high? (%)	**10**	4%		
Access Controls		30%		
19) Can a user clearly and quickly find the status of the admin controls	**10**	4%	20%	2
20) Is the information clear and complete	**10**	4%	0%	0
21) Is the information in non-technical terms	**10**	4%	0%	0
22) Is there Pause Control documentation including records of tests	**10**	4%	80%	8
		15%		
Section Scoring				
Code and Team	55		100%	
Documentation	45		79%	
Testing	50		91%	
Audits	70		100%	
Access Controls	40		25%	

Private Software Repository

Development teams, especially those in DeFi, often prefer a private repository to reduce the ability for an easy fork/copy of their development. We clearly understand the business incentive for this, but we cannot give scores for what we cannot see. Hence, the developers will be penalized for not having a publicly accessible repository.

This is due to the importance of public repositories in quantifying a steady development process as well as being able to see the testing records, and be able to verify the code in general . Although we fully encourage developers to have a private repository alongside a public one, the public repository must be an industry standard for protocols wishing to establish themselves as leaders in the DeFi space.

Audits are also of special importance. With a public repository, anyone can check the differences between the audited and deployed code because all the information is publicly available.

However, if there is no public repository, the audit is of lesser value because the code cannot be cross referenced. Assuming the audit takes place on private code, then 25% is deducted from the audit question score. This is because differences between deployed and audited code are too important to provide points for.

Chains

This section indicates the blockchain used by this protocol. You must pick one of the Chains and fill in the blank. This DeFiSafety Process Quality Review covers all the blockchains that the protocol is deployed onto, so long as the code is the same as deployed to Ethereum mainnet.

So far our Process Quality Reviews have covered protocols deployed to the following (in no particular order) chains:

- Ethereum
- Binance Smart Chain
- Polygon
- Avalanche

- Terra
- Celo
- Arbitrum
- Solana

In addition, our Process Quality Reviews have covered protocols deployed on a multi-chain basis that span across these following chains:

- xDai
- Fantom
- Harmony
- Moonriver
- Optimism
- Loopring
- OKEx
- Palm
- HECO

Smart contracts and team

Any of our Process Quality Reviews start with the code which the protocol deploys, and how frequently it is interacted with by users.

<u>The score for this section is derived from the individual scores of the following questions</u>:

1) Are the smart contracts easy to find? (%)
2) How active is the primary contract? (%)
3)Does the protocol have a public GitHub repository? (Y/N)
4) Is there a development history visible? (%)
5) Is the team public (not anonymous)? (Y/N)

1) Are the smart contract addresses easy to find? (%)

Scoring weight: 20

Summary:

Are the addresses of the deployed smart contract(s) easy to find in the project's public documents?

This is a very important question for users and developers alike, as it has an effect on the audit score for Question 17. It is extremely rare for a protocol to get a 0% on this question. However, when this does happen, it is a very easy fix for the developers of the protocol.

Key Specifications:

- Essentially, all the contract addresses must be publicly visible in a protocol's documentation. Not just the token address - DeFiSafety does not consider tokens in our analyses. Rather, the listed contract addresses should be the ones that enable the key offerings of your protocol. This could include Farms, Pools, Staking, Exchanges, and anything you could think of that is not a token.

- Most importantly, the listed contract addresses of your key products should ideally be your implementation (logic) contracts that execute a majority of your transactions. A good example would be a "*MasterChef.sol"* contract.

Additional Specifications:

- The addresses can be over multiple pages as long as one page has the latest addresses or links to their locations.

- Each contract address must have the software visible on the relevant block explorer. If we cannot verify active addresses using the block explorer, we cannot give marks for them.

All of this is essential in order to declare a protocol as passing because users must be able to quickly identify the contracts that they use (even if they do not understand Solidity). Being capable of verifying what the protocol does and how they do it plays a key part in users being able to place their trust in that protocol.

An image of the web page with the addresses will be screenshotted as part of the review, and this screenshot will be viewable in the "*Contract Address Appendix*" of the PQR.

Scoring

For a project to meet our standards for this question, the addresses must be clearly labelled on the website or on a page on the GitBook or in the readme of the GitHub repository. Ideally, a section in the protocol's documentation should be identified as "*Smart Contracts*", "*Contract Addresses*", "*Deployed Addresses*", etc.

- If the contract addresses are found by having to use the search function, then the protocol will receive 70% for this question.

- If the contract addresses are found, but have required significant research to find (such as in an audit report, through a discord server or even in *mainnet.json*), then the protocol will receive 40% for this question.

- If we find the contract addresses in an obscure location, then the protocol will receive 20% for this question.

- Finally, If no contract addresses are found, then the protocol fails this question and gets a 0% for it.

As mentioned beforehand, this question is related to audits. In the event that the smart contract addresses are not found, the remainder of our review will have to take place without contract code. The audit score for Question 17 will immediately become 0% (even if the code is audited), as there is no ability to prove that the code that is being executed within the smart contracts matches the code that has been audited.

Based on this, we follow Question 1 in every PQR with the guidance provided below.

Guidance:

100% Clearly labelled and on website, documents or repository, quick to find
70% Clearly labelled and on website, documents or repository but takes a bit of looking
40% Addresses in mainnet.json, in discord or sub graph, etc
20% Address found but labelling not clear or easy to find

0% Executing addresses could not be found

<u>How to improve this score:</u>

Make the addresses of the smart contracts utilized by your application available on either your website, Gitbook or GitHub (in the README for instance). Ensure the addresses are up to date, and that they can be verified using a block explorer. Your documentation should ideally comprise a section called “Smart Contracts”, “Smart Contract Addresses”, or “Deployed Addresses” in order to make them easy to find.

2) How active is the primary contract? (%)

Scoring weight: 5

<u>Summary:</u>

This metric looks at the transaction history of the main smart contract of a protocol through the native blockchain explorer (Etherscan, BscScan, PolygonScan, etc). We reserve the right to use whichever chain we perceive to be the most relevant to the protocol. For instance, PoolTogether will naturally receive a higher transaction count on Polygon than on Ethereum, but on the other hand, Yearn’s primary deployment is on the Ethereum blockchain, so it will therefore receive a higher transaction count there.

Key Specifications:

- In order to pass this question with flying colours, be sure to include your executing smart contract addresses in your documentation, as these are the ones that execute your transactions and record them. Such contracts include “*MasterChef.sol*”, “*Farms.sol*”, “*Vaults.sol*”, etc.

- We will not look at token contract addresses for this question.

<u>Scoring:</u>

Naturally, Question 2 will automatically get a 0% if Question 1 gets a 0%. If no smart contract addresses were found, we cannot verify the transaction count.

`

The results are scored as indicated below and the proof is documented in the "*Contract Interactions Appendix*". The reviewer will choose an important contract that is executed regularly and is of critical importance to the protocol.

The screenshot of the smart contract's transaction analysis graph for the "*Contract Activity Appendix*" is generated using the relevant blockchain explorer. If necessary, we will display an addresses' internal transactions.

<u>Guidance:</u>

100% More than 10 transactions a day

70% More than 10 transactions a week

40% More than 10 transactions a month

10% Less than 10 transactions a month

0% No activity or no smart contract addresses found

<u>How to improve your score:</u>

Make sure to list your main executing smart contract addresses in a section of your documentation. This will make it easier for us to evaluate your contract's user interactions, as well as make it easier for you to get 100% for this question.

3) Does the protocol have a public software repository? (Y/N)

Scoring weight: 5

<u>Summary:</u>

For this metric, we look at the availability of the protocol's software repository. A public software repository is essential for best practices, as it enables any user to go through it and read a specific smart contract's code. This practice enables a lot of cross-referencing between the deployed smart contracts and the source code, and subsequently provides increased transparency to the space.

Additionally, Questions 4, 12, and 13 are dependent on this Question's answer, as a "No" here would immediately give 4, 12, and 13 a 0% score.

<u>Key Specifications:</u>

- We define a public GitHub as a repository that contains all or most of the protocol's executing smart contract code. If a protocol only has frontend code for their website, or only code forks that have not been elaborated upon, we do not consider this a public GitHub.

Scoring:

If a protocol's GitHub is visible, even one just made for deployment, then this scores as "Yes". For teams with no public software repository, then "No".

As mentioned beforehand, this question is critical to our PQR process. Without a public software repository, a protocol is unlikely to score highly as this indicates low protocol transparency.

How to improve this score

Ensure your contracts are available for viewing on a public software repository (like GitHub). The link to it can be from your protocol's website or GitBook documentation. Any alternative public repository service is acceptable as well, provided it is identified.

4) Is there a development history visible? (%)

Scoring Weight: 5

Summary

This metric checks a protocol's software repository for a robust development history. We measure this by counting GitHub commits, branches and releases in a software repository. All of these act as a ledger of edits, collaboration, and general workflow.

While not an iron rule, a repository that has been frequently updated over a long period of time is indicative of good developer practices, whereas a quick fork or upload of a deployment is indicative of rushed development or developers wishing to obscure their work. Progression and transparency create bulletproof code, and this question is a good probe of these.

Key Specifications:

- Some protocols prefer to actively develop on a private GitHub as opposed to a publicly available repository. Note that we will only grade the GitHub repositories that are public.

- The specific repository that we will evaluate for this metric is the one that includes the smart contracts of your protocol's main functionality.

Scoring

Naturally, this question cannot be scored in the event that no public GitHub repository could be found. As such, a "No" for question #3 would result in a 0% for this question.

Guidance:

100% Any one of 100+ commits, 10+branches

70% Any one of 70+ commits, 7+branches

50% Any one of 50+ commits, 5+branches

30% Any one of 30+ commits, 3+branches

0% Less than 2 branches or less than 10 commits

How to improve your score:

In order to maximize your score for this question, make sure to actively develop your public GitHub. This doesn't mean that you cannot also have a private GitHub, but your users should at least be able to see a decent amount of development history in it. A good way to rack up the commits is to also add your testing suite to your GitHub, and then continuously test for bugs which would go straight into your CI.

5) Is the team public (not anonymous)? (Y/N)

Scoring Weight: 15

Summary

While there are many leading protocols led by anonymous development teams, developers willing to put their faces on things and break anonymity are usually an indicator of a long term commitment to a project / ecosystem, which in turn boosts its security. It is important to remember that all projects can suffer exploits, but what the protocols do about these issues is key to investor security. Anonymous developers can easily disappear, whereas public ones will be held accountable.

Scoring

To score full marks on this question requires both the protocol to name at least two individuals as working for them, as well as the named individuals publicly confirming this information through LinkedIn, Twitter or a personal website. To get 50%, at least one public name should be attributed to working for the protocol anywhere on the internet. Finally, a protocol with no public members will not receive any marks on this question. The information sources we use for this question are documented in our Team Appendix at the end of our PQRs.

Guidance:

100%: At least two names can be easily found in the protocol's website, documentation or medium. These are then confirmed by the personal websites of the individuals / their linkedin / twitter.

50%: At least one public name can be found to be working on the protocol.

0%: No public team members could be found.

How to improve this score

Create a section of documentation linking to employees, founders or contributors in an easily accessible place such as a website, LinkedIn etc. To score fully, name the contributors to this protocol in your documentation and ensure that this corroborates information that can be found elsewhere on the internet (e.g. LinkedIn/ a personal website). Alternatively, software repository contributors can be public.

Documentation

The documentation section analyses the quality of the protocol's software documentation. For these metrics, we will be looking at the general accessibility of the documentation, as well as verifying if the functionalities of the software are explained or not. This iteration of the Process Audit standard requests only basic documentation, although increasingly detailed documentation will naturally score higher.

The score for this section is derived from the individual scores of the following questions:

6) Is there a whitepaper? (Y/N)

7) Is the protocol software architecture documented? (Y/N)

8) Does the software documentation fully cover the deployed contracts' source code? (Y/N)
9) Is it possible to trace the documented software to its implementation in the protocol's source code? (%)

6) Is there a whitepaper? (Y/N)

Scoring weight: 10

Summary

Is there a white paper or other basic (or complex) description of what the project is doing referenced to by either the protocol's website or GitHub? In addition, Medium articles, GitBooks, or GitHub README.md explaining the application are all admissible.

Scoring

This is a straightforward "Yes" or "No" answer. However, it should be noted that this question's score will have an impact on questions 7,8 and 9. Therefore, since all questions below deal with software function documentation, a "No" here would result as a 0% for questions 7, 8 and 9.

How to improve this score

Ensure that your white paper is available for viewing from your website's front page, GitBooks documentation, or at least the README.md of your GitHub repository. Moreover, protocols should update their whitepapers frequently to meet the capabilities of their present application.

7) Is the protocol software architecture documented? (Y/N)

Scoring weight: 10

Summary

This score requires a section of the documentation that specifically covers the protocol's architecture. Architecture is a loose term that boils down to a section that details "software function used in contract (code) + what it does/how it does it". In addition, this can also be presented as a diagram that may include the following:

- Arrows indicating how the smart contracts interact with each other
- Specific reference to the software functions themselves
- A written explanation on how the smart contracts interact alongside the directional arrows.

Scoring

In order to receive full marks on this question, protocols should include either a description of smart contract architecture (code + how it works) or a diagram to go along with it. Ideally, both would be included in a protocol's documentation.

How to improve this score

Write this document based on the deployed code and how it operates. For guidance, refer to the SecurEth System Description Document. This document can be written after deployment, though as with all documentation earlier is better.

8) Does the software documentation fully cover the deployed contracts' source code? (%)

Scoring weight: 15

Summary

This score requires documentation specifically written about a protocol's smart contract source code. As such, any generalized math formulas or state diagrams without directly referencing the code do not count towards this score.

Key Specifications:

- Something we see quite a bit of is instructions on how to interact with the contract, how to deploy/build on it, etc. We will not consider this as smart contract source code documentation. On the other hand, we do factor API documentation in the overall score for this question. However, API documentation on its own weighs no more than 20%.

Scoring

In order to comply with our standards and meet a 100% score, the documentation (in either the website, GitBooks or the GitHub) must cover *all* of the developed source code. Covering public libraries is not needed. This question requires smart contract source code being identified, explained and located in order to earn full marks. This ensures protocol users know how the protocol works, and is a necessary step to explain what the code does.

Guidance:

100% All contracts and functions documented

80% Only the major functions documented

79-1% Estimate of the level of software documentation

20% Only API function documented

0% No software documentation

How to improve this score

This score can be improved by ensuring protocol documents fully and comprehensively cover everything deployed by the protocol. A good way to do it is to list literally every function for every smart contract that you have deployed for your protocol. Not only that, but you also include a brief description of what the specific contract function does and how it does it. For guidance, refer to the SecurEth System Description Document .

9) Is it possible to trace the documented software to its implementation in the protocol's source code? (%)

Scoring weight: 5

Summary

Traceability are documented links between a smart contract's software architecture, documentation, source code and tests. It can involve code snippets within the documents as a simple identifier (such as a shortened content hash) that connects it to the protocol's source code. It could also be a direct link from code explanations in the documents to where the code is in their Github repository. This is important for users to relate what the protocol documentation explains and its exact location in the GitHub source code, thereby promoting transparency through traceability.

Scoring

In order to meet our standards, every piece of deployed code must have 100% traceability. For reference, check the SecurEth guidelines on traceability.

Guidance:

100% - Clear explicit traceability between code and documentation at a requirement level for all code
60% - Clear association between code and documents via non explicit traceability
40% - Documentation lists all the functions and describes their functions
0% - No connection between documentation and code

How to improve this score

This score can be improved by adding traceability from documents to code such that it is clear where each outlined code function in the documentation is coded in the protocol's GitHub repository. For reference, check the SecurEth guidelines on traceability.

Testing

This section covers the testing process of the protocol's smart contract code previous to its deployment on the mainnet. The testing suite should be located in the same GitHub repository as the contracts themselves. This suite also should completely cover all the code in the deployed contracts and come with a detailed report that indicates a successful test run. This is often referred to as a "code coverage" test. Any additional testing can include a Formal Verification test as well as active test/stress environments (on a chain-specific testnet like Kovan for ethereum, for example).

The score for this section is derived from the individual scores of the following questions:

10) Has the protocol tested their deployed code? (%)
11) How covered is the protocol's code? (%)
12) Does the protocol provide scripts and instructions to run tests? (Y/N)
13) Is there a report of the protocol's test results? (%)
14) Has the protocol undergone formal verification? (Y/N)
15) Were the smart contracts deployed to a testnet? (Y/N)

10) Has the protocol tested their deployed code? (%)

Scoring weight: 20

Summary

Testing is described in the SecurEth guidelines. Unit tests are written on a file by file basis and generally are used for code coverage. System tests are for functionality, and testing usage of the code. This is an integral part of pre-deployment and post-deployment of a smart contract, as continuously testing your code ensures that there are no bugs or potential vulnerabilities in the software. In addition, testing ensures that in the event that a bug is found it can be resolved right away, therefore proving as an effective method of detection and resolution.

Key Specifications:

- Do the tests allow for comprehensive testing of the code? Meaning is every smart contract tested in itself, and is this clearly outlined through individual testing files in the GitHub repository.
- Are there both system and unit tests? Unit testing is a verification that the individual smart contracts work well. On the other hand, and equally as important are system tests that confirm that the contracts interact seamlessly with one another without any issues. A complete testing suite would include both of these.
- It is better to test after deployment than never test at all. Testing is an integral part of evaluating the safety of a DeFi protocol. It is better to have some than none at all, even if the suite only gets developed post-deployment.

Scoring

For our purposes, this score is guided by the Test to Code ratio (TtC), which is calculated by dividing the lines of code in the deployed smart contracts by the lines of code in the protocol’s testing suite. Generally a good TtC ratio is over 100%, which means that approximately every line of deployed code has undergone some form of testing, which makes for better code. An ideal TtC would be 120%, as this proves a rigorous testing process. However, the reviewer's best judgement is the final deciding factor.

Guidance:

100% - TtC > 120% Both unit and system test visible

80% - TtC > 80% Both unit and system test visible

40% - TtC < 80% Some tests visible

0% - No tests found

How to improve this score

This score can be improved by adding tests to fully cover the code. Document what is covered by traceability or test results in the software repository. Ideally, you should have a test file per smart contract deployed on the blockchain. In addition, providing scripts to run your tests is an essential part of testing transparency, and no testing suite should come without them.

11) What is the documented code coverage from the tests? (%)

Scoring weight: 5

Summary

Here we consider if the unit tests fully cover the code. Ideally, a protocol should generate a report of a code coverage run with results posted in their GitHub repository. All projects should aim for 100% code coverage, as this ensures that every line of code was checked for quality. Without a report, the author determines a percentage based on the test suite's percentage, artifacts in the test scripts and qualitative estimation. If there are misses in the coverage report, the documentation should explain why the specific code was not tested. The code that is not covered should be as little as possible and well-justified as to why it is uncovered.

Key Specifications:

- A code coverage test can be performed by the protocol itself, a third-party auditor, or a code coverage service such as Coveralls.

- Ideally, a coverage test should include the coverage of both lines of code and branches on the GitHub.

Scoring

The score for this question will reflect the exact percentage of the code coverage results. However, in the event that a protocol does not have a code coverage report, we will still give a maximum of 50% depending on the score given to the testing suite in the previous question. This is due to the fact that testing is essentially code coverage, but a code coverage report is visual proof of its depth.

Guidance:

100% - Documented full coverage
99-51% - Value of test coverage from documented results
50% - No indication of code coverage but clearly there is a reasonably complete set of tests as identified by the TtC being greater than 120%
30% - Some tests evident (TtC between 20% to 120%) but not complete
0% - No test for coverage seen

How to improve this score

This score can be improved by performing code coverage tests that are as close to 100% coverage as possible. In the event that some lines of code or entire contracts are missed, you should clearly outline why this is the case in your coverage report. Hence, you should also be aiming to perform code coverage tests upon every single deployment. This proves that the code is rigorously tested, and therefore has a degree of reliability attributed to it. Integrate the result of your code coverage result in your GitHub repository readme, like Tetu protocol has done here.

12) Does the protocol provide scripts and instructions to run their tests? (Y/N)

Scoring weight: 2.5

Summary

Making your test scripts available is a key part of adding traceability to your GitHub repository. In essence, the scripts will allow anyone to experience a protocol's testing environment firsthand and allow them to compile your code. This in itself makes a protocol's testing process easier to understand as well as increasingly transparent. Since anyone can run these scripts to ensure that the tests work, it opens up a lot of visibility towards malfunctions or potential vulnerabilities. Equally so, in the absence of scripts, there need to be a certain degree of test instructions in a protocol’s GitHub repository. This is due to test environments often needing pre-installed dependencies in order to be able to interface with the scripts, and instructions go a long way into making this process more accessible.

Key Specifications:

- Although we give points for the presence of testing scripts OR instructions on how to compile them in the proper environment, it is best practice to have both ready and available.

- Scripts can be seen as an implicit way to permit users to run the same tests that a specific protocol has. However, not everyone knows how to run scripts without instructions, and that is why instructions to run the tests and/or the scripts are an important part of it as well.

Scoring

The scoring for this metric is relatively simple. We simply check the reviewed protocol’s GitHub repository for scripts or instructions needed in order to interact with their testing suite. Depending on what we find, this is a simple Yes/No question.

Guidance:

Yes - Scripts and/or instructions to run tests are available in the testing suite
No - Scripts and/or instructions to run tests are not available in the testing suite

How to improve this score

Add the scripts to the repository and ensure they work. Ask an outsider to create the environment and run the tests. Improve the scripts and documents based on their feedback.

13) Is there a detailed report of the protocol’s test results?(%)

Scoring weight: 12.5

Summary

Ideally there is a test report in the repository of the deployed code that was generated by running the full test suite. This is as straightforward as providing the test report created by running the coverage in the protocol's test environment. A complete test report manufactured by the protocol's developer can also be used as a testing guide and visualizer for users that wish to execute those same tests. Therefore, adding reports like these provide a great deal of traceability. An ideal report would include a complete visual breakdown of a performed test run by a protocol, as well as explanations of the general testing methodology used.

Key Specifications:

- The important distinction between a coverage test report and a coverage test output is that a coverage output is almost exclusively software functions and percentages in a table format, while a coverage test report would include additional explanations that provide in-depth information about the methodology used. A report like this also leaves more room for detailing exactly why a line of code was not covered, why there was a miss, errors, etc.

Scoring

If you have a private repository, a coverage test report becomes vital. A good test report will show evidence of a full test suite (Q10) and that you have code coverage (Q11). If it shows the script information, then you will additionally receive marks for Q12. This can be used as the proof needed to properly validate a protocol's testing procedures in the event that their GitHub is private.

For the scoring of this specific metric, we will give full points for test reports that are intricately detailed with the help of methodology explanations. On the other hand, we will give less points for a report that only contains a coverage test output with the functions and percentages.

Guidance:

100% - Detailed test report as described below

50% - Code coverage report visible

0% - No test report evident

How to improve this score

Add a code coverage test report with the results. This should not only be a code coverage output, but rather a combination of your coverage output and a deeper insight on your methodology used. An exemplary test report from Balancer Finance can be found here.

14) Has the protocol undergone formal verification? (Y/N)

Scoring weight: 5

Summary

Formal verification is a process of testing software processes that is well suited to most blockchain software in the DeFi space. More specifically, Formal Verification tests a protocol's software algorithms with formal methods of mathematics in order to prove that they work correctly. This test can be tailored to any specification or property, which is why it is a very useful tool. Nevertheless, its accessibility is limited and few protocols have undergone Formal Verification. For this reason, the weight is lower than other elements in our scoring matrix.

Key Specifications:

- Although certain auditors use certain formal methods, these are not what we consider a full-fledged Formal Verification. To get a good representation of what it actually looks like, established formal verifiers include Certora and Runtime Verification.

Scoring

This is a simple Yes/No question regarding the availability of a Formal Verification test report of the protocol being reviewed. If there is no such report, this metric receives a "No". If a report is provided, this metric receives a "Yes".

Guidance:

Yes - Formal Verification was performed and the report is readily available

No - Formal Verification was not performed and/or the report is not readily available.

How to improve this score

Undergo a Formal Verification for your protocol’s algorithms, and use the services provided by the reputable formal verifiers in the space such as Certora and Runtime Verification. Although this can become expensive, it is an essential part of legitimizing your software’s integrity.

15) Were the smart contracts deployed to a testnet? (Y/N)

Scoring weight: 5

Summary

A stress testing environment is an active test environment (not the active mainnet) that is used for testing limits of a protocol’s smart contracts. This would run on a testnet with simulated inputs such as Rinkeby, Goerli, Kovan, etc. By deploying smart contracts to a testnet environment and keeping it up to date, the developers can test possible limit cases as they develop and proactively mitigate problems - all of which adds another layer of smart contract protection.

Key Specifications:

- A testnet shines in the testing of a smart contract before it is deployed to the mainnet. Testnets being similar to simulated mainnet deployments, it is an essential part of stressing a software to its core in order to point out instabilities. Doing this before a launch would ensure that the smart contracts are stable and should be functional.

Scoring

This question is evaluated through a simple Yes/No answer. In order to get a “Yes”, a protocol must provide their deployed testnet smart contract addresses. A protocol cannot get a “Yes” without these addresses, even if they clearly mention that they use testnet networks in their documentation.

Guidance:

Yes - Protocol has proved their tesnet usage by providing the addresses

No - Protocol has not proved their testnet usage by providing the addresses

How to improve your score

Make sure to utilize testnet networks in your developing and deployment process. It is an environment that lets you fully test the capability and functionality of your smart contracts. Subsequently, it ensures a certain degree of stability. Finally, don't forget to link your testnet smart contract addresses in your documentation, as we need proof in order to score this question.

Security

This section looks at the security of a protocol through 3rd party software audits and bug bounties. Security is the most important section in this review, as it is essential to uphold robust safety measures in the DeFi space. Therefore, the questions in this section weigh the most out of all the others, making them a key point of our focus in intricately analyzing and commenting on them. As such, audits and bug bounties are metrics that we use in order to evaluate how much a protocol values smart contract safety. More technical details are explained in this document.

The score for this section is derived from the individual scores of the following questions:

16) Is the protocol sufficiently audited? (%)
17) Is the bounty value acceptably high? (%)

16) Is the protocol sufficiently audited? (%)

Scoring weight: 70

Summary

Smart contract audits are an indicator of code quality that is adherent to best practices in DeFi. Protocol audits occur when a third-party blockchain software security organization reviews a specific protocol's smart contract code. Specifically, auditors conduct tests to look for overall quality in the software architecture in addition to the subtle blockchain-specific weaknesses that could be utilized by an attacker. Audits are one of the pillars of smart contract security in the DeFi space, and that is why it is the most important metric within DeFiSafety reviews.

Key Specifications:

- A protocol that has undergone an audit is automatically perceived as being legitimized in the DeFi space. This is because most audits go through a protocol's software with such a fine-tooth comb that it is sure to prevent vulnerabilities to some degree.

- An audit typically weighs more if the report has been published before one of a protocol's mainnet deployments. Equally so, having multiple audits performed before deployment is the best-case scenario. This assures users that the protocol's team have a vested interest in making sure that their deployed smart contracts are safe, and that their users know it.

- A public audit report is a necessity. A protocol that claims to be audited without having a public audit report can effectively be considered as unaudited. In the DeFi space, transparency is everything, and having a private audit report is non-transparent.

- An audit report usually generates a good amount of feedback from the auditors for the developers of the protocol to implement. These can range from informational issues that are typically structure related, to critical issues that usually underline an important vulnerability in the code. It is of utmost importance that the developers of the team being audited implement these recommendations for the sake of safe code and best practices.

Scoring

If the smart contract addresses on the mainnet are not found or if the addresses are found but the code is hidden, the audit results will be zero. This is because even if an audit is available, there is no ability to verify that the code deployed is the same code that has been audited. This is especially important for new versions of protocols as new code being shipped for a V2 / V3 etc. is now no longer covered by the previous audit.

If the quality of the report does not reflect a proactive audit done on the code, then the authors reserve the right to reduce the score to as low as 0%. This attempts to cover valueless documents that say “Audit” and “PASS” but are not real audits, as no oversight is provided. Some audits in DeFi simply act as rubber stamps, so it is important to verify their methods.

The authors also reserve the right to reduce audit scores if they do not cover some economic issues. A smart contract audit that covers solidity well but ignores financial risks specific to DeFi has limited value (but not zero). Financial risk in DeFi is significant, making this audit question of massive importance.

With a public repository, anyone can check the differences between the audited and deployed code because all the information is publicly available. Assuming the audit takes place on code that cannot be seen publicly, then 25% is deducted from the usual score. This has the effect of making a 100% become 75%, and so forth. If the auditing firm provably indicates that their audit report is relevant to the deployed code, then full marks are regained.

This is the most important question in the process, as good audit practises almost always indicate good protocol development practises.

Guidance:

100% - Multiple Audits performed before deployment and the audit findings are public and implemented or not required
90% - Single audit performed before deployment and audit findings are public and implemented or not required
70% - Audit(s) performed after deployment and no changes required. The Audit report is public.
65% - Code is forked from an already audited protocol and a changelog is provided explaining why forked code was used and what changes were made. This changelog must justify why the changes made do not affect the audit.
50% - Audit(s) performed after deployment and changes are needed but not implemented.
30% - Audit(s) performed are low-quality and do not indicate proper due diligence.
20% - No audit performed

0% Audit Performed after deployment, existence is public, report is not public OR smart contract address' not found.

Deduct 25% if the audited code is not available for comparison.

How to improve your score:

Your score for this question can improve by having your future deployments rigorously audited before deployment. In order to achieve full marks, you must have had your smart contracts audited twice before being deployed. However, even having just one audit performed before deployment gives you a great score for this metric. Another point you can seek to improve on is implementing the recommendations brought to you by the audit. We read every audit thoroughly, and not implementing important fixes will definitely affect your score negatively. Finally, having one audit published either before or after a deployment is infinitely better than having none at all. Even if you have just one audit, you're on the right track. Just make sure that it is traceable to your own code, and that the audited smart contracts have their addresses publicly available.

17) Is the bounty value acceptably high? (%)

Scoring Weight: 20

Summary

This section checks the value of the bug bounty, if it exists. A Bug Bounty program is another fundamental element of smart contract security. When a bounty offers a substantial reward, the incentives to outsource the search for bugs to the community become increasingly apparent. More efforts in bug searches will naturally lead to a higher chance of finding a critical issue in the protocol's backend. Moreover, a high Bug Bounty reward not only signifies that the development team cares about finding bugs, but it also means that the code has been battle-tested to absolute confidence.

High TVL bug bounty rewards are specifically designed to incentivize black hat hackers to take the white hat route as there is a legal route to significant revenue. Our scoring metric reflects this.

Key Specifications:

- We do consider Code Arena competitions as being a Bug Bounty program. Although the concepts between the two are slightly different, the fundamental goal of finding bugs for a cash reward is the same.

Scoring

First, a score of 0% is given if there is no Bug Bounty program for the protocol being reviewed or if the bug bounty is no longer available. Based on discussions with Immunefi we have given a priority towards active and very rewarding Bug Bounty programs as these improve the general incentives for safety in DeFi.

An active program means a third party (such as Immunefi or CodeArena) is actively driving hackers to the site. An inactive program would be a static mention of a Bug Bounty on the documents. A dead program is one that was offered in the past, but has since expired.

Guidance:

100% - Bounty is 10% TVL or at least $1M AND active program (see below)

90% - Bounty is 5% TVL or at least 500k AND active program

80% - Bounty is 5% TVL or at least 500k

70% - Bounty is 100k or over AND active program

60% - Bounty is 100k or over

50% - Bounty is 50k or over AND active program

40% - Bounty is 50k or over

20% - Bug bounty program bounty is less than 50k

0% - No bug bounty program offered / the bug bounty program is dead

How to improve your score

The whole idea of a Bug Bounty program is to increase the amount of eyes that are continuously checking your source code for bugs. How do you improve that? Through improving the monetary incentives of your program. In this scenario, the more incentives you put in for community members to find errors in your code, the more your code will be secure in the long run. The other side of the coin is that posting a million dollar bounty will naturally make you feel more inclined to be absolutely certain that your code does not have any flaws in it. Therefore, Bug Bounties are an incentive for both the users and the developers.

Admin controls

This section covers the documentation of a protocol's admin controls for their DeFi protocol. The admin controls are the contracts that allow upgrading contracts or coefficients in the protocol. In short, admin control documentation clearly outlines what the admins can do with their contracts, and what they cannot do.

Since these contracts can allow the protocol admins to "change the rules", complete disclosure of capabilities is vital for a user in being able to quickly identify, as this is a crucial factor relating to investment decisions. If a protocol can change on a dime with just one admin's approval, it is extremely problematic for investor safety as all information the investor uses to make their decisions is subject to immediate change. This is further explained in this document.

The score for this section is derived from the individual scores of the following questions:

18) Is the admin control information easy to find? (%)

19) Are relevant contracts clearly labelled as upgradeable/immutable? (%)

20) Is the type of contract ownership clearly indicated? (%)

21) Are the contract change capabilities described? (%)

22) Is the information easy to understand? (%)

23) Is there sufficient Pause Control documentation? (%)

24) Is there sufficient timelock documentation? (%)

25) Is the TimeLock an adequate length? (%)

18) Is the admin control information easy to find? (%)

Scoring weight: 10

Summary

This question answers the ease of which a user can find the admin Control documentation. Admin controls documentation defines any capabilities that the developers have to make fundamental changes to the contract while it is deployed.

This documentation can include anything from governance, smart contract ownership, emergency procedures, upgradeability, and more.

There are a few metrics that we consider while scoring this question. Is it clearly labeled? Is it where the users expect it to be? As extremely important information for investor decision making, this must be explicitly identified in documentation.

Key Specifications:

- Ideally, admin controls would be clearly outlined as the title of a section in a protocol's documentation. However, it is rarely called that. Usually, we find admin controls information under a number of different terms. These include but are not limited to: Governance, DAO, Admins, Ownership, etc.

Scoring

We approach the scoring of this question with the users in mind. As we consider admin controls to be one of the most important pieces of documentation that a protocol can have, it needs to be relatively easy to find for the average person. In order to get full points here, a protocol must ideally have a section of their documentation titled "Admin controls". However, we also give full points if this information is found under a section called "Governance", or "DAO". We start withdrawing points when the admin controls are found in a subsection of another overarching section. Oftentimes, we find bits and pieces of information under sections such as "Tokenomics" or "Staking". The more we find information in a dispersed manner, and through search methods that become increasingly difficult, the more we start withdrawing points.

Guidance:

100% - Clearly labelled and on website, documents or repo, quick to find

70% - Clearly labelled and on website, documents or repository but takes a bit of looking

20% - Admin controls documents in multiple places and not labelled

0% - Admin Control information could not be found

How to improve your score

You can improve this score by putting all of your admin controls information under one overarching section of your documentation. It needs to be as easy to find and as easy to identify as possible. Additional acceptable identifiers of admin controls documentation include: “Governance”, “DAO” (if applicable), or “Admin Access/Powers”.

19) Are relevant contracts clearly labelled as upgradeable or immutable? (%)

Scoring weight: 10

Summary

This question looks at the admin controls information supplied by the protocol on contract upgradeability. Upgradeability can be defined as a change in a smart contract following its mainnet deployment. If documentation specifically states that contracts are either upgradeable or immutable (non-upgradeable), then full points are awarded for this question. Mentions of upgradeability are important, as it clearly identifies an essential aspect of the protocol’s admin powers. More details regarding admin controls can be found here.

If certain contracts are immutable and others are upgradeable, it must be specifically mentioned on a contract-to-contract basis, for every individual contract. Implicitly, this means that upgradeable contracts are not the highest standard of quality according to DefiSafety.

Key Specifications:

- Although upgradeable contracts are not the highest standard of smart contract security, they are still an elemental part of the DeFi space today. This is why we do take upgradeability standards such as EIP-1167 into consideration when it comes to the scoring of this question.

Scoring

When it comes to scoring this question, we look at the number of deployed smart contracts and compare the amount of upgradeability information for each one. In order to get 100%, each smart contract must be immutable. However, protocols can still get a very good score for this question even if some of the contracts are upgradeable, as long as they are labelled as such.

Guidance:

100%: Both the contract documentation and the smart contract code state that the code is not upgradeable or immutable.
80%: All Contracts are clearly labelled as upgradeable (or not)
50%: Code is immutable but not mentioned anywhere in the documentation
0%: Admin control information could not be found

How to improve your score

You can improve your score by clearly labelling the upgradeability capabilities of each of your deployed smart contracts. Doing this is important from a user perspective. However, in order to get full points here you need to make all of your contracts immutable. Examples can be found here.

20) Is the type of contract ownership clearly indicated? (%)

Scoring weight: 10

Summary

Every deployed smart contract needs to have its ownership defined in the code. Ownership refers to which addresses can call various functions within the software code, and we evaluate to what extent they are highlighted in the code's comments or in the protocol's documentation. For example, does the protocol's admin controls documentation highlight the use of OnlyOwner functions, MultiSig addresses, or clearly defined smart contract governance roles? This is all essential information for evaluating the extent of ownership privileges that the admins have over their smart contracts.

In theory, the more restrictive and rigorous a smart contract's governance is, the more you can have faith in its security and the team's good faith. This is due to the fact that if a protocol's admins have removed their own individual powers over a certain smart contract, whether through smart contract self-governance or requiring a MultiSig, the capabilities for malpractice become inherently limited.

Key Specifications:

- An OnlyOwner function declares that there is only one governing owner that can make calls to a certain smart contract. Ideally, to ensure immutability, the team would either declare the smart contract as its own OnlyOwner or make the OnlyOwner an address that the team has renounced all ownership to by burning the keys (and proving that this transaction has gone through).

- A MultiSig function declares that there are multiple owners that can make calls to a certain contract. However, before being able to make the changes, a specific amount of owner signatures are required. This is usually documented in fractions, as a MultiSig will typically require 2/4, 4/6, 6/9 signatures, etc. A MultiSig function makes it impossible for only one team member to make changes to a contract, hereby eliminating a lot of potential malpractice opportunities.

- Defined Roles make various functions of a smart contract governed by a specific address or a whitelist of addresses. These roles can include the whitelist admin, the pauser, minter, and more. This library of defined roles is curated by OpenZeppelin, and the in-depth information can be found here.

Scoring

We score this metric based on the extent of which the information above is detailed in a protocol's documentation. In order to get full points, a protocol must clearly identify the ownership functions of each of their deployed smart contracts, as well as explain how they have structured it. However, if all of this is only explained in the code or the code's comments, we will only award half the points for this metric.

Guidance:

100%	The type of ownership is clearly indicated in their documentation. (OnlyOwner / MultiSig / etc)
50%	The type of ownership is indicated, but only in the code. (OnlyOwner / MultiSig / etc)
0%	Admin Control information could not be found

How to improve your score

You can improve your score through clearly outlining the ownership functions (OnlyOwner, MultiSig, Defined Roles) attributed to your deployed smart contract. Ideally, this would all be clearly formatted within your admin controls section. At the bare minimum, these functions should be established and explained in your protocol's code or code comments. If you do not have any ownership functions within your software's code, do consider adding them in the next deployment of your smart contracts.

21) Is the contract change capability described? (%)

Scoring weight: 10

Summary

The capabilities for change in a specific smart contract refer to what extent the configurations of a contract can be changed post-deployment. These configurations refer to the various constants, variables and functions that are established in a protocol's backend. In an upgradeable structure, changes could be deployed through a new implementation contract that is integrated through a proxy via libraries or interface contracts. In the event that a protocol's structure allows this, they should ideally outline exactly what configurations can be tweaked as well as establish the margins of their smart contracts' upgrades. This is an essential part of a protocol's backend transparency, as users should be made aware of the parameters that the developers can change at a moment's notice.

Key Specifications:

- The information detailed above needs to be identified especially in a DAO-like structure where the users are actively voting on upgrade implementations. However, even an upgradeable DEX, LP, Vault, and more should have their capabilities for change identified in the admin controls section of their documentation.

- Typically upgraded parameters include burning rate, minting rate, staking rewards, supply, and more.

Scoring

We score this metric based on how much a protocol details their contracts' upgradeability capabilities, and also to what extent. Meaning, how many contracts out of the total amount of contract does the protocol touch upon regarding their status of upgradeability. Have the admins renounced ownership over the contract, and it is therefore immutable? Can a team member burn and mint tokens whenever it wants without going through a voting process? Can the parameters of a protocol's smart contract be tweaked by the developers whenever they see fit? These are all important details that the users need to be given access to in order for them to be able to properly assess the safety of the contracts themselves.

Guidance:

100% - The documentation covers the capabilities for change for all smart contracts

50% - The documentation covers the capabilities for change in some, but not all contracts

0% - The documentation does not cover the capabilities for change in any contract

How to improve your score:

You can improve your score for this question by adding all the necessary information regarding the capabilities of your smart contracts' upgradeability in your documentation. An example of this can be found here.

22) Is the admin controls information easy to understand? (%)

Scoring weight: 10

Summary

Upgradable smart contracts allow the admins to make changes that could potentially impact the investments of the protocol's users. Any users interfacing with the protocol need to understand how the contract can change, and what kind of risks these changes involve. As such, it is recommended that this information be provided by the protocol in the admin controls section of their documentation. This is to ensure best protocol-to-user transparency so that users have all the information needed in order to feel safe about their investments.

Key Specifications:

- The best case scenario will always be immutable smart contracts, as these do not permit any changes or upgrades post-deployment. However, upgradeable smart contracts certainly have their uses in the DeFi space, and can therefore still score high for this metric in the event that all the investment safety information is provided in a protocol's documentation.

Scoring

This question grades the quality of the explanations with higher scores given to plain language descriptions relevant to the investments that require no special software background. Of course, as in the question above, the only way to get a perfect score is to have immutable contracts.

Guidance:

100% - All the contracts are immutable

90% - Description relates to investments safety in clear non-software language

30% - Description all in software-specific language

0% - No admin control information could not be found

How to improve this score

Create a document that covers the items described above in plain language that investors can understand. An example is enclosed.

23) Is there sufficient Pause Control documentation? (%)

Scoring weight: 10

Summary

A Pause Control function allows the admins to freeze certain aspects of a protocol. If a hack is underway, this function can allow the developers to properly mitigate the damages that a hack or exploit is causing to one or multiple of their smart contracts. Subsequently, having such a function is an efficient way to add an extra layer of protection to user investments.

Key Specifications:

- We define Pause Control as anything that immediately halts a smart contract. Such an event can be executed using contracts like Pausable.sol, or a Pause Guardian.

- Note that the Pause function we are referring to is inherently different from a TimeLock, which is another function evaluated further in our review.

Scoring

For this question, we look at the admin controls section of a protocol's documentation in search for specific information on a function or smart contract that could halt some of their smart contracts while being deployed. Some common examples include Pausable.sol, Pause Guardian, various emergency pausing measures, and more. In addition, when documenting a protocol's pause control, protocols should include some form of test record that is updated at least once every three months. This should serve as proof that a "fire drill" test was performed, and that the tested Pause Control works as intended. A test like this should be performed frequently so that no unforeseen problems arise while trying to pause the protocol during emergencies.

Guidance:

100% - Pause control(s) are clearly documented and there is records of at least one test within 3 months

80% - Pause control(s) explained clearly but no evidence of regular tests

40% - Pause controls mentioned with no detail on capability or tests
0% - Pause control not documented or explained

How to improve this score

To improve your score, you should either implement or expand upon the information available on your Pause Control function. Additionally, make sure to perform consistent "fire drill" tests on your emergency pause so that we can be assured that it has been battle-tested. You should also create a document that covers the items described above in plain language that investors can understand. An example is enclosed.

24) Is there sufficient Timelock documentation? %

Scoring weight: 10

Summary

A Timelock is a smart contract that requires any calls made to a contract to pass through a time delay, allowing users to exit the contract in advance of the change taking effect if they so desire. This is a function used mostly in upgradeable contract structures, as well as DAOs. It also plays a key role in preventing rug pulls, as investors would be able to see the transaction in the Timelock transaction queue before it goes through and plan accordingly. This is another key player in adding that extra layer of security to a protocol.

Key Specifications:

- Timelocks are inherently different from the Pause Control contracts that we mention in Question #23.

- A Timelock removes a certain degree of administrative power from the protocol admins and subsequently grants more control to the community due to having a component that can proactively protect their funds. Therefore, it can play an important role in user transparency as well as best practices.

- Detailing the duration of the Timelocks used by a protocol is essential, as this can be used as a frame of reference for users who want to closely watch the queued transactions in the Timelock.

Scoring

We will score this metric based on the overall availability of information on the reviewed protocol's Timelock. In order to achieve a high score for this question, the duration of the Timelock should be documented and should be explained in-depth. Additionally, each contract that the Timelock is applicable to should be identified. Ideally, an explanation of the software functions behind the Timelock should also be provided. Lastly, a protocol could find itself not having a use for a Timelock in the event that their smart contracts are immutable, or if they do not have a governance system in place. In this scenario, the protocol would only receive full points for this question if they provide an adequate explanation on why a Timelock is not needed.

Guidance:

100% - Documentation identifies and explains why the protocol does not need a Timelock OR Timelock documentation identifies its duration, applicable contracts and justifies this time period.

60% - A Timelock is identified and its duration is specified

30% - A Timelock is identified

0% - No Timelock information was documented

How to improve this score

Create a document that identifies the timelock, its length and which contracts it is applicable to. This should be included in the protocol documentation. An example of good timelock documentation is Uniswap's. Protocols may seek to use this as a starting point for their own Timelock documentation.

25) Is the timelock of an adequate length? (Y/N)

Scoring weight: 5

Summary

A timelock should have a length of between 48 hours to 1 week. This will allow all users to be informed of the changes made but not long enough to collapse a protocol entirely if malfunctioning code is shipped. In addition, such measures go a long way into preventing developer malpractices as they will no longer be able to instantaneously ship out a rug pull.

Scoring

We will only score this metric based on the availability of the information regarding the duration of the protocol's Timelock. If this information is not directly available in their documentation, either in a GitBooks or a GitHub README.md, we will not score it. This means that no Timelock information described in a third-party audit will be factored into the scoring of this question. Again, if the protocol does not use upgradeable contracts, and has no governance structure, they will get full points for this question only if they clearly explain why they do not need a Timelock.

Guidance:

100% - Timelock is between 48 hours to 1 week OR justification as to why no Timelock is needed / is outside this length.

50% - Timelock is less than 48 hours or greater than 1 week.

0% - No Timelock information was documented OR no timelock length was identified.

How to improve this score

In your Timelock documentation, make sure to include the duration of it and an adequate description of why this specific timeframe was chosen. If you do not use Timelocks due to immutable contracts or not having a governance structure, make sure to clearly outline this in your docs and appropriately justify it.

Oracles

This section goes over the documentation that a protocol may or may not supply about their Oracle usage. Oracles are a fundamental part of DeFi as they are responsible for relaying tons of price data information to thousands of protocols using blockchain technology. Not only are they important for price feeds, but they are also an essential component of transaction verification and security.

Since Oracles play such an important role in the space, it is vital that protocols disclose the complete extent of their Oracle usage. Such information includes the provider, the refresh rate of price feeds, which smart contracts are using Oracles, the Oracle's key software functions, and more. The availability of this documentation is not only important for developers, but also for users who want to get a good grasp on the robustness of the Oracle used by their favourite protocols. In an era where Oracle manipulations are rampant, it becomes increasingly crucial to prioritize them in terms of best practices and transparency.

<u>The score for this section is derived from the individual scores of the following questions</u>:

26) Is the Oracle sufficiently documented? (%)
 N
28) Can flashloan attacks be applied to the protocol, and if so, are those flashloan attack risks mitigated? (Y/N)

26) Is the Oracle sufficiently documented? (%)

Scoring weight: 10

Summary

The specific Oracle provider that a protocol uses for price data should be identified. This will inform users as to the origin and reliability of the data which protocols' smart contracts depend upon. More specifically, the documentation should go over the different components of an Oracle including software function documentation (if the protocol provides its own price data source), Oracle source, the contracts that are using Oracles, and the timeframe between each price feed. This information is very important as Oracles are critical to the function of many protocols, and being more transparent about this data will lead to increasingly robust Oracle practices.

Key Specifications:

- By Oracle source, we essentially mean the provider. This could be anyone from Chainlink to Tellor, MakerDAO, Compound, Band Protocol, Augur, etc. Every provider has their own philosophy, methods, and backend. Protocols identifying their Oracle provider goes a long way into users being able to do their due diligence, and therefore promotes overall transparency.

Scoring

When scoring this metric, we look specifically at how much information is available on a protocol's Oracle. In order to get full points for this question, a protocol must identify its various components. These are: source, contracts, price feed timeframe, and basic software functionality (if they provide their own price data source). In the event that the protocol has no need for price feeds, they do not have a use for an Oracle. In this scenario, the protocol must adequately explain why they have no use for an Oracle in order to get full points for this question.

Guidance:

100% - If it uses one, the Oracle is specified. The contracts dependent on the oracle are identified. Basic software functions are identified (if the protocol provides its own price feed data). Timeframe of price feeds are identified.

100% - The reason as to why the protocol does not use an Oracle is identified and explained.

75% - The Oracle documentation identifies both source and timeframe, but does not provide additional context regarding smart contracts.

50% - Only the Oracle source is identified.

0%: No oracle is named / no oracle information is documented.

How to improve this score

Include a section within protocol documentation explaining the oracle it employs. If no oracle is used, state this. Pancakeswap documentation identifies the oracles that it uses, and justifies why it chose them. Protocols might consider using their documentation as a starting point. On the other hand, a protocol may simply not need a data source and should this be explained full marks will be awarded.

27) Is front running mitigated by this protocol? (Y/N)

Scoring weight: 2.5

Summary

Front running is problematic for DeFi users interacting with protocols. In essence, front running refers to a DeFi user gaining privileged access to orderbook or mempool transaction information that would allow them to plan their own transactions accordingly. This could look like seeing a multitude of buy orders on a specific token pair, and then creating a buy transaction with a higher gas so that it goes to the top of the transaction queue, effectively creating a profit due to the subsequent buy orders. Protocols should implement measures to prevent transactions being front run by other actors.

Key Specifications:

- As front-running is done through accessing Oracle data in the mempool, there are multiple ways to mitigate the potential of attacks like this. A good example of this would be combining a Chainlink Oracle with a TWAP such a Uniswap V3's, as this would require an attacker to front-run both Oracles.

- An additional way to mitigate the possibilities of front-running would be to put a gas limit on transactions.

- Another notable safety measure would be to add a ReentrancyGuard function to your contracts so that no fraudulent calls cannot be made to your executing smart contracts. This would greatly reduce the possibility of an attacker manipulating your Oracles.

Scoring

We score this metric based on the amount of information that a protocol has documented about their mitigation strategy for front-running. This is a simple “Yes” or “No” question.

Guidance:

Yes - The protocol cannot be front run and there is an explanation as to why OR documented front running countermeasures are implemented.

No - The protocol does not mention front running or does not document any countermeasure against it

How to improve this score

Document the frontrunning countermeasures your protocol employs. If no frontrunning countermeasures are employed, state and justify this. For an example of documentation relating to frontrunning countermeasures, see mistX’s documentation. KeeperDAO also has good documentation on this matter.

28) Can flashloan attacks be applied to the protocol, and if so, are those flashloan attack risks mitigated? (Y/N)

Scoring weight: 2.5

Summary

Flash loans are a fundamental part of DeFi, originally brought in by Aave. However, flash loan attacks are significant threats to DeFi protocols, and have crippled a number of them. This is due to flash loans being used as a means to manipulate the liquidity of certain token pairs in a protocol’s vaults, and subsequently affecting the accuracy of the prices reported by their Oracles. Protocols should thus implement measures to mitigate them, as this is very important for their users’ fund protection. A protocol that documents this kind of information is inherently making itself more trustworthy to investors.

Key Specifications:

- Although flash loan exploits cannot currently be completely blocked, there are certainly methods that protocols can use to mitigate their extent. For instance, using multiple Oracle sources can prevent the reporting of inaccurate or very volatile price changes. This can be done by using a Chainlink Oracle that uses multiple data nodes in combination with a TWAP Oracle such as Uniswap's. Other ways of preventing large-scale flash loan exploits include robust tokenomics suchs as minting caps, withdrawal caps, and more.

- Additional methods of protecting your smart contracts from attacks include making your contracts "uncallable" by other contracts. This will prevent other contracts from interacting with your contracts in a malicious fashion.

Scoring

We will score this question based on how much of the flash loan mitigation information is available in the protocol's documentation. Countermeasures should be documented and explained so that users can understand what steps have to be taken to mitigate the risks of a potential flashloan attack. This is a simple "Yes" or "No" question.

Guidance:

Yes: Flashloan attack countermeasures have been implemented and documented.
No: No flashloan attack countermeasure is documented.

How to improve this score:

Identify in your documentation what countermeasure the protocol employs to prevent a flashloan attack. An example of how this might be prevented can be found here and this should be documented in this way [SECUREETH LINK].