Stater-LendingData

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **May 13th-21st, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/13/2021	Gabi Urrutia
0.2	Document Edits	05/16/2021	Gokberk Gulgun
1.0	Final Edits	05/21/2021	Gokberk Gulgun
1.0	Remediation Plan	05/25/2021	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Stater engaged Halborn to conduct a security assessment on their Smart contract beginning on May 13th, 2021 and ending May 19th, 2021. The security assessment was scoped to the smart contracts LendingCore.sol, LendingMethods.sol and LendingTemplate. An audit of the security risk and implications regarding the changes introduced by the development team at Stater prior to its production release shortly following the assessments deadline.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverables set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

1.2 AUDIT SUMMARY

The team at Halborn was provided four weeks for the engagement and assigned two full time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit to achieve the following:

- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified few security risks, and recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit.While manual testing is recommended to uncover flaws in logic, process,and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart Contract manual code read and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions(solgraph)
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Testnet deployment (Truffle, Ganache)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.

3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

IN-SCOPE:

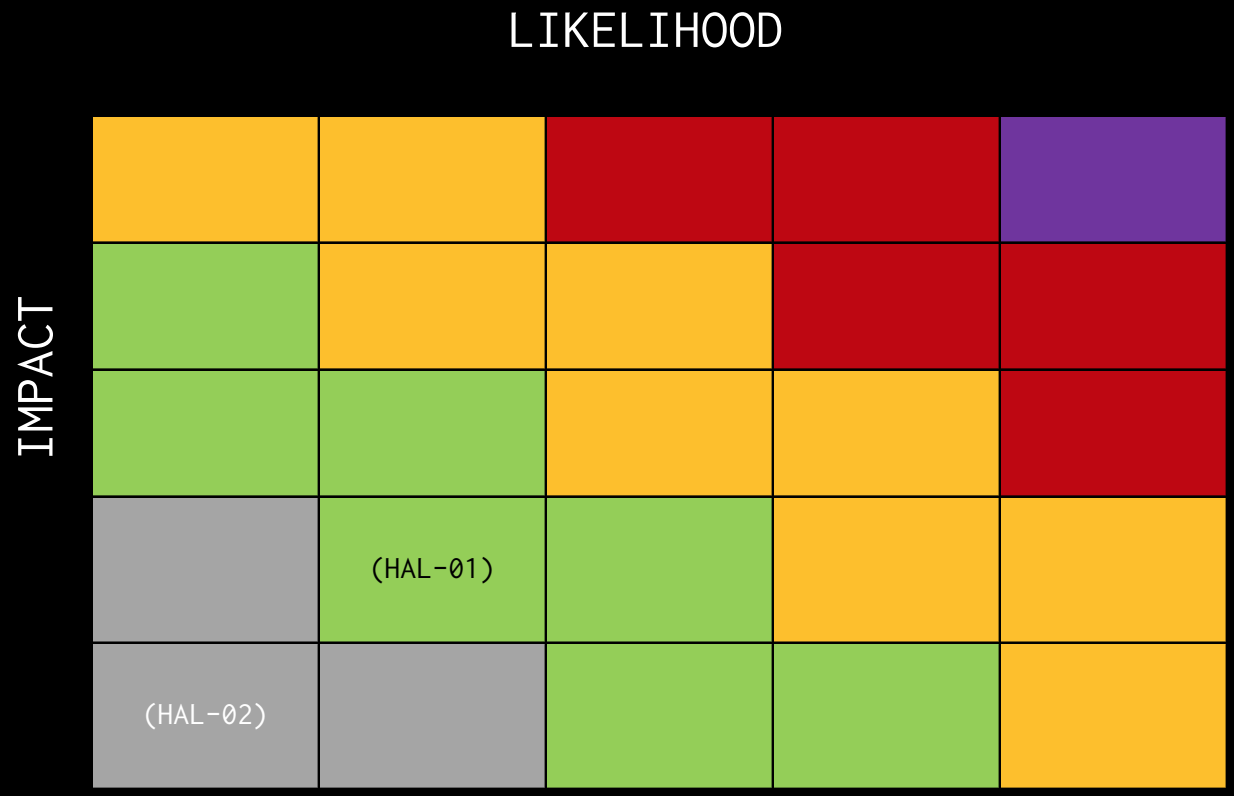
Code related to:
- LendingCore.sol
- LendingMethods.sol
- LendingTemplate.sol

Specific commit of contract:

1b20fc106b26ee6a5c3357f5b2997544d5bb9af4

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	1	1



EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
USAGE OF BLOCK-TIMESTAMP	Low	SOLVED - 05/24/2021
FOR LOOP OVER DYNAMIC ARRAY	Informational	RISK ACCEPTED - 05/14/2021
WORKFLOW TESTING	-	-
STATIC ANALYSIS	-	-
AUTOMATED SECURITY SCAN RESULTS	-	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) USAGE OF BLOCK-TIMESTAMP - LOW

Description:

During a manual review, we noticed the use of block.timestamp. The contract developers should be aware that this does not mean current time. Miners can influence the value of block.timestamp to perform Maximal Extractable Value (MEV) attacks. The use of block.timestamp creates a risk that time manipulation can be performed to manipulate price oracles. Miners can modify the timestamp by up to 900 seconds.

Code Location:

LendingCore.sol Line #133

```
119     * @DIMIIM Determines if a loan has passed the maximum unpaid installments limit or not
120     * @ => TRUE = Loan has exceed the maximum unpaid installments limit, lender can terminate the loan and get the NFTs
121     * @ => FALSE = Loan has not exceed the maximum unpaid installments limit, lender can not terminate the loan
122     */
123     function lackOfPayment(uint256 loanId) public view returns(bool) {
124         return
125             loans[loanId].status == Status.APPROVED
126                 &&
127             loans[loanId].startEnd[0].add(
128                 loans[loanId].nrOfPayments.mul(
129                     loans[loanId].installmentTime.div(
130                         loans[loanId].nrOfInstallments
131                     )
132                 )
133             ) <= block.timestamp.sub(
134                 loans[loanId].defaultingLimit.mul(
135                     loans[loanId].installmentTime.div(
136                         loans[loanId].nrOfInstallments
137                     )
138                 )
139             );
140     }
141
```

LendingMethods.sol

```
258     // Borrower can withdraw loan items if loan is LIQUIDATED
259     // Lender can withdraw loan item is loan is DEFAULTED
260     function terminateLoan(uint256 loanId) external {
261         require(msg.sender == loans[loanId].borrower || msg.sender == loans[loanId].lender);
262         require(loans[loanId].status != Status.WITHDRAWN);
263         require((block.timestamp >= loans[loanId].startEnd[1] || loans[loanId].paidAmount >= loans[loanId].amountDue) || lackOfPayment(loanId), "Not possible to finish this loan yet");
264         require(loans[loanId].status == Status.LIQUIDATED || loans[loanId].status == Status.APPROVED, "Incorrect state of loan");
265 
266         if ( lackOfPayment(loanId) ) {
267             loans[loanId].status = Status.WITHDRAWN;
268             loans[loanId].startEnd[1] = block.timestamp;
269             // We send the items back to lender
270             transferItems(
271                 address(this),
272                 loans[loanId].lender,
273                 loans[loanId].nftAddressArray,
274                 loans[loanId].nftTokenIdArray,
275                 loans[loanId].nftTokenTypeArray
276             );
277         } else {
278             if ( block.timestamp >= loans[loanId].startEnd[1] && loans[loanId].paidAmount < loans[loanId].amountDue ) {
279                 loans[loanId].status = Status.WITHDRAWN;
280                 // We send the items back to lender
281                 transferItems(
282                     address(this),
283                     loans[loanId].lender,
284                     loans[loanId].nftAddressArray,
285                     loans[loanId].nftTokenIdArray,
286                     loans[loanId].nftTokenTypeArray
287                 );
288             } else if ( loans[loanId].paidAmount >= loans[loanId].amountDue ){
289                 loans[loanId].status = Status.WITHDRAWN;
290                 // We send the items back to borrower
291                 transferItems(
292                     address(this),
293                     loans[loanId].borrower,
294                     loans[loanId].nftAddressArray,
295                     loans[loanId].nftTokenIdArray,
296                     loans[loanId].nftTokenTypeArray
```

LendingMethods.sol

```
210     // Borrower pays installment for loan
211     // Multiple installments : OK
212     function payLoan(uint256 loanId,uint256 amount) external payable {
213         require(loans[loanId].borrower == msg.sender, "You're not the borrower of this loan");
214         require(loans[loanId].status == Status.APPROVED, "This loan is no longer in the approval phase, check its status");
215         require(loans[loanId].startEnd[1] >= block.timestamp, "Loan validity expired");
216         require((msg.value > 0 && loans[loanId].currency == address(0) && msg.value == amount) || (loans[loanId].currency != address(0) && msg.value == 0 && amount > 0), "Insert the correc
217 
218         uint256 paidByBorrower = msg.value > 0 ? msg.value : amount;
219         uint256 amountPaidAsInstallmentToLender = paidByBorrower; // >> amount of installment that goes to lender
220         uint256 interestPerInstallement = paidByBorrower.mul(interestRate).div(100); // entire interest for installment
221         uint256 discount = discounts.calculateDiscount(msg.sender);
222         uint256 interestToStaterPerInstallement = interestPerInstallement.mul(interestRateToStater).div(100);
223 
224         if ( discount != 1 ){
225             if ( loans[loanId].currency == address(0) ){
226                 require(msg.sender.send(interestToStaterPerInstallement.div(discount)), "Discount returnation failed");
227                 amountPaidAsInstallmentToLender = amountPaidAsInstallmentToLender.sub(interestToStaterPerInstallement.div(discount));
228             }
229             interestToStaterPerInstallement = interestToStaterPerInstallement.sub(interestToStaterPerInstallement.div(discount));
230         }
231         amountPaidAsInstallmentToLender = amountPaidAsInstallmentToLender.sub(interestToStaterPerInstallement);
232 
233         loans[loanId].paidAmount = loans[loanId].paidAmount.add(paidByBorrower);
234         loans[loanId].nrOfPayments = loans[loanId].nrOfPayments.add(paidByBorrower.div(loans[loanId].installmentAmount));
235 
236         if (loans[loanId].paidAmount >= loans[loanId].amountDue)
237         loans[loanId].status = Status.LIQUIDATED;
238 
239         // We transfer the tokens to borrower here
240         transferTokens(
241             msg.sender,
242             loans[loanId].lender,
243             loans[loanId].currency,
244             amountPaidAsInstallmentToLender,
245             interestToStaterPerInstallement
246         );
247 
248         emit LoanPayment(
249             loanId,
250             msg.value,
```

LendingMethods.sol

```
185     // Borrower cancels a loan
186     function cancelLoan(uint256 loanId) external {
187         require(loans[loanId].lender == address(0), "The loan has a lender , it cannot be cancelled");
188         require(loans[loanId].borrower == msg.sender, "You're not the borrower of this loan");
189         require(loans[loanId].status != Status.CANCELLED, "This loan is already cancelled");
190         require(loans[loanId].status == Status.LISTED, "This loan is no longer cancellable");
191 
192         // We set its validity date as block.timestamp
193         loans[loanId].startEnd[1] = block.timestamp;
194         loans[loanId].status = Status.CANCELLED;
195 
196         // We send the items back to him
197         transferItems(
198         address(this),
199             loans[loanId].borrower,
200             loans[loanId].nftAddressArray,
201             loans[loanId].nftTokenIdArray,
202             loans[loanId].nftTokenTypeArray
203         );
204 
205         emit LoanCancelled(
206             loanId
207         );
208     }
209 
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Use block.number instead of block.timestamp or now to reduce the risk of MEV attacks. Check if the timescale of the project occurs across years, days and months rather than seconds. If possible, it is recommended to use Oracles.

Remediation Plan:

Solved: Stater team assumes that the use of block.timestamp is safe because their timescales are higher than 900 seconds.

3.2 (HAL-02) FOR LOOP OVER DYNAMIC ARRAY - INFORMATIONAL

Description:

When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas,

based on how much computation is needed to complete them. The Ethereum network specifies a block gas limit and the sum
of all transactions included in a block cannot exceed the threshold.

Programming patterns that are harmless in centralized applications can lead to Denial of Service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. Modifying an array of unknown size, that increases in size over time, can lead to such a Denial of Service condition.

A situation in which the block gas limit can be an issue is in sending funds to an array of addresses. Even without any malicious intent, this can easily go wrong. Just by having too large an array of users to pay can max out the gas limit and prevent the transaction from ever succeeding.

Code Location:

- LendingMethods.sol Line #~315

```
309      * @notice Used by the Promissory Note contract to change the ownership of the loan when the Promissory Note NFT is sold
310      * @param from The address of the current owner
311      * @param to The address of the new owner
312      * @param loanIds The ids of the loans that will be transferred to the new owner
313      */
314     function promissoryExchange(address from, address payable to, uint256[] calldata loanIds) external isPromissoryNote {
315         for (uint256 i = 0; i < loanIds.length; ++i) {
316             require(loans[loanIds[i]].lender == from, "Lending Methods: One of the loans doesn't belong to you, rejected.");
317             require(loans[loanIds[i]].status == Status.APPROVED, "Lending Methods: One of the loans isn't in approval state, rejected.");
318             require(promissoryPermissions[loanIds[i]] == from, "Lending Methods: Permission to exchange promissory not allowed, rejected.");
319             loans[loanIds[i]].lender = to;
320             promissoryPermissions[loanIds[i]] = to;
321         }
322     }
```

- LendingMethods.sol Line #~329

```
325      * @notice Used by the Promissory Note contract to approve a list of loans to be used as a Promissory Note NFT
326      * @param loanIds The ids of the loans that will be approved
327      */
328     function setPromissoryPermissions(uint256[] calldata loanIds, address sender, address allowed) external isPromissoryNote {
329         for (uint256 i = 0; i < loanIds.length; ++i){
330             require(loans[loanIds[i]].lender == sender, "Lending Methods: You're not the lender of this loan");
331             if (allowed != address(0))
332                 require(loans[loanIds[i]].status == Status.APPROVED, "Lending Methods: One of the loans isn't in approval state, rejected.");
333             promissoryPermissions[loanIds[i]] = allowed;
334         }
335     }
336 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

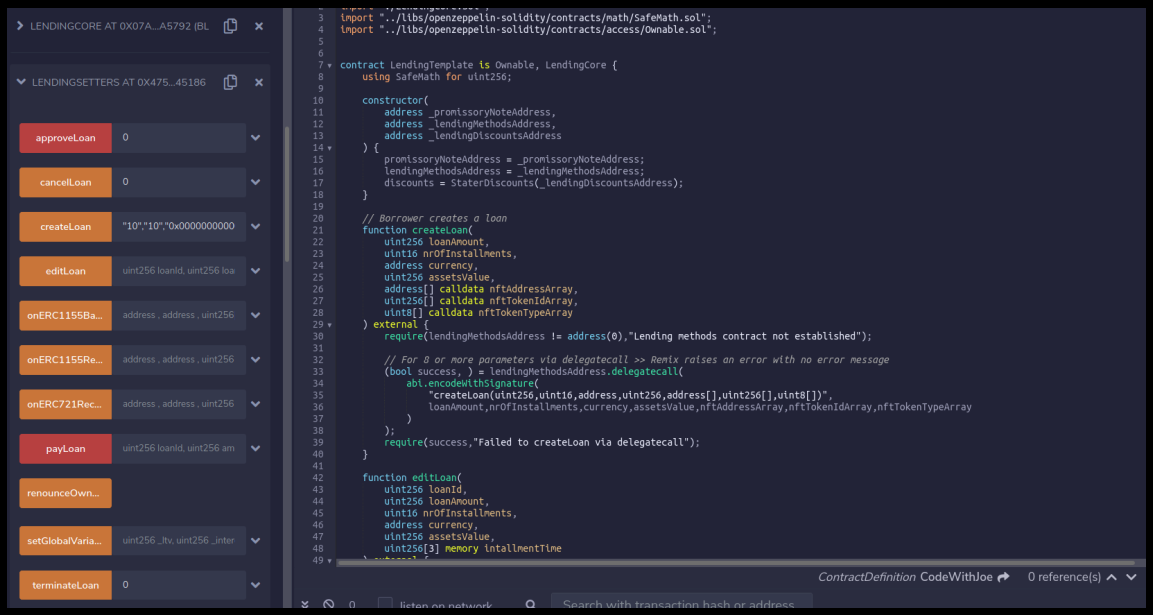
Remediation Plan:

Risk Accepted: Stater team considers appropriate the use of loops.

3.3 WORKFLOW TESTING

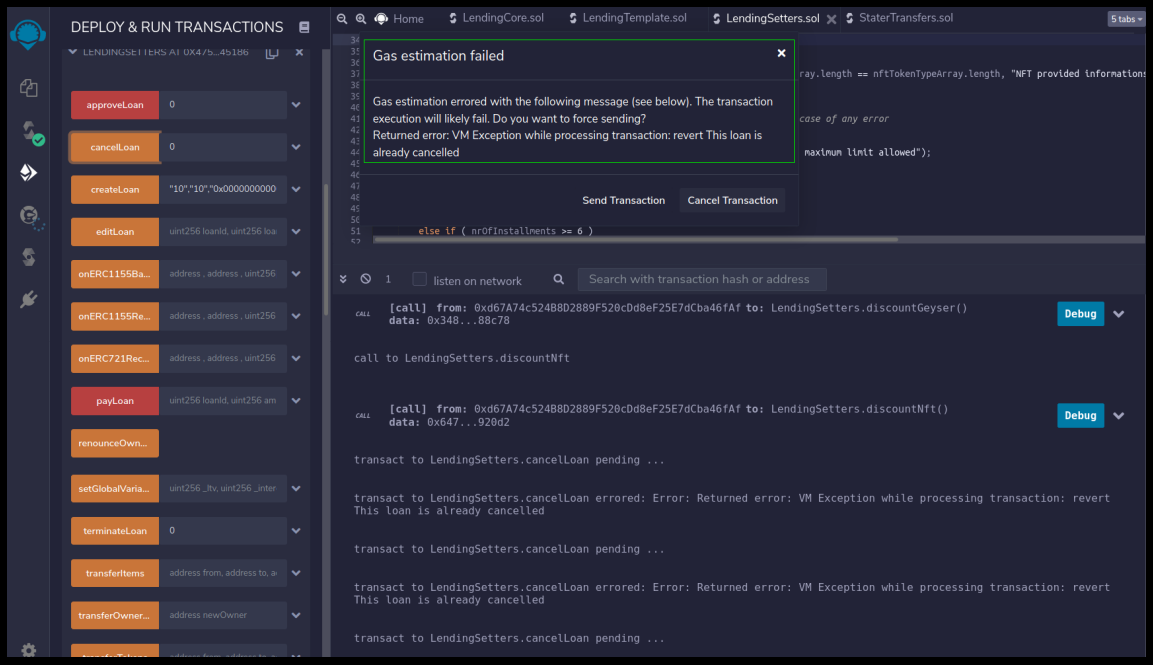
Description:

Custom tests are useful for developers to check if functions and permissions work correctly. Furthermore, they are also useful for security auditors to perform security tests behaving like a malicious user. In the updated lending contracts, The analysis of the workflows was carried out and the tests were performed dynamically. The Lending contracts are deployed on the local network and test accounts used through ganache-cli.

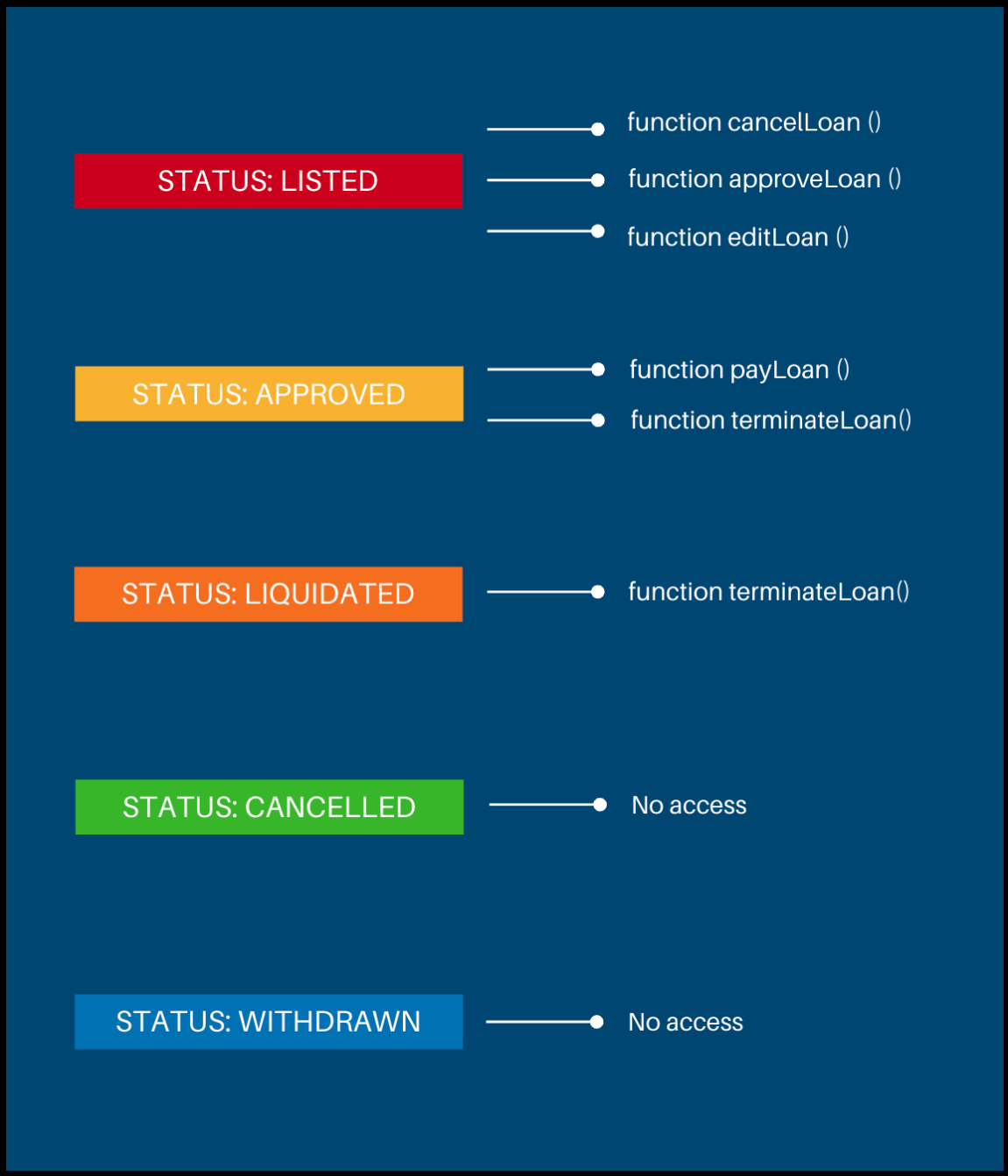


Then, All workflow of functions are reviewed with static analysis. Next, Dynamic analysis step began with createaLoan function.

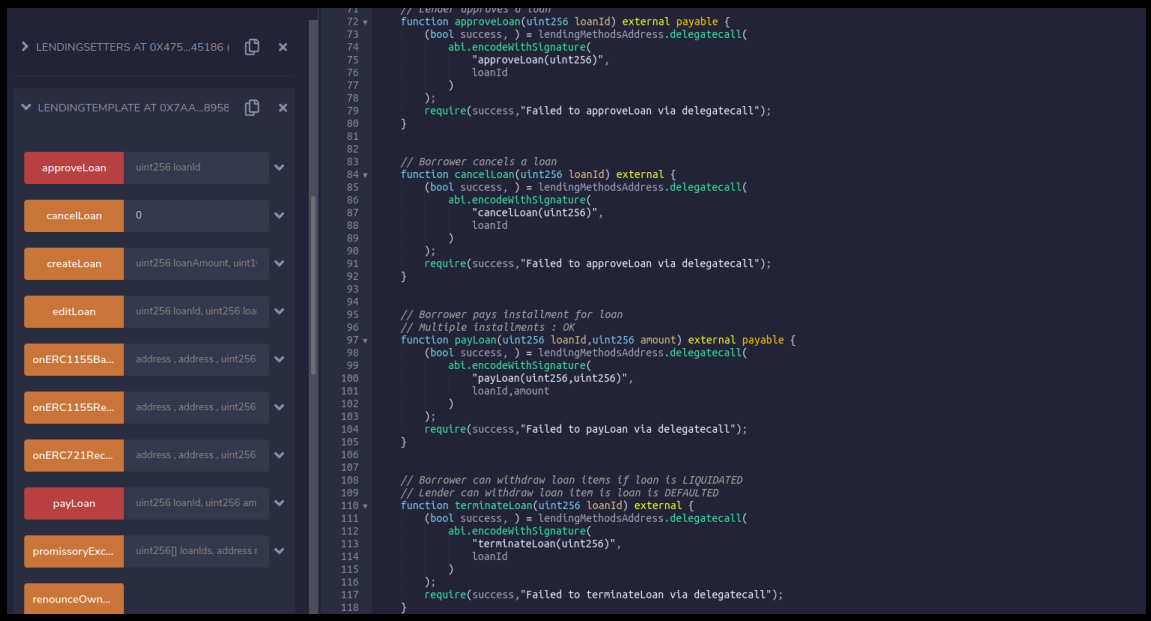
After the creation of the loan, accessible capabilities are tested on the Remix.



As a result of the tests, a graph was generated depending on the behavior.



Finally, Delegate call implementation analyzed on the LendingTemplate.sol.



Results:

As a result of the tests, Depending on the loan and variable states, Exceptional behaviors are handled and implementations have been put. The situations that may occur on the smart contract have been prevented with the correct implementations.

3.4 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

Results:

LendingData.sol

```
INFO:Detectors:
LendingCore.promissoryNoteAddress (contracts/lending/LendingCore.sol#29) is never initialized. It is used in:
LendingCore.loans (contracts/lending/LendingCore.sol#107) is never initialized. It is used in:
        - LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140) performs a multiplication on the result of a division:
        -loans[loanId].status == Status.APPROVED && loans[loanId].startEnd[0].add(loans[loanId].nrOfPayments.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInst
allments))) <= block.timestamp.sub(loans[loanId].defaultingLimit.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) (contracts/lending/LendingCore.s
ol#124-139)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140) uses timestamp for comparisons
        Dangerous comparisons:
        - loans[loanId].status == Status.APPROVED && loans[loanId].startEnd[0].add(loans[loanId].nrOfPayments.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfIns
tallments))) <= block.timestamp.sub(loans[loanId].defaultingLimit.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) (contracts/lending/LendingCore.
sol#124-139)
```

LendingMethods.sol

```
INFO:Detectors:
LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140) performs a multiplication on the result of a division:
        -loans[loanId].status == Status.APPROVED && loans[loanId].startEnd[0].add(loans[loanId].nrOfPayments.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInst
allments))) <= block.timestamp.sub(loans[loanId].defaultingLimit.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) (contracts/lending/LendingCore.s
ol#124-139)
LendingMethods.approveLoan(uint256) (contracts/lending/LendingMethods.sol#143-183) performs a multiplication on the result of a division:
        -loans[loanId].startEnd[1] = block.timestamp.add(loans[loanId].nrOfInstallments.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) (contract
s/lending/LendingMethods.sol#158-164)
LendingMethods.payLoan(uint256,uint256) (contracts/lending/LendingMethods.sol#212-256) performs a multiplication on the result of a division:
        -interestPerInstallement = paidByBorrower.mul(interestRate).div(100) (contracts/lending/LendingMethods.sol#220)
        -interestToStaterPerInstallement = interestPerInstallement.mul(interestRateToStater).div(100) (contracts/lending/LendingMethods.sol#222)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Reentrancy in LendingMethods.createLoan(uint256,uint16,address,uint256,address[],uint256[],uint8[]) (contracts/lending/LendingMethods.sol#32-94):
        External calls:
        - transferItems(msg.sender,address(this),nftAddressArray,nftTokenIdArray,nftTokenTypeArray) (contracts/lending/LendingMethods.sol#76-82)
                - IERC721(nftAddressArray[i]).safeTransferFrom(from,to,nftTokenIdArray[i]) (contracts/plugins/StaterTransfers.sol#55-59)
                - IERC1155(nftAddressArray[i]).safeTransferFrom(from,to,nftTokenIdArray[i],1,0x00) (contracts/plugins/StaterTransfers.sol#61-67)
        State variables written after the call(s):
        - ++ id (contracts/lending/LendingMethods.sol#93)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

LendingTemplate.sol

```
INFO:Detectors:
LendingTemplate.createLoan(uint256,uint16,address,uint256,address[],uint256[],uint8[]) (contracts/lending/LendingTemplate.sol#22-41) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(createLoan(uint256,uint16,address,uint256,address[],uint256[],uint8[]),loanAmount,nrOfInstallments,currency,assetsValu
e,nftAddressArray,nftTokenIdArray,nftTokenTypeArray)) (contracts/lending/LendingTemplate.sol#34-39)
LendingTemplate.editLoan(uint256,uint256,uint16,address,uint256,uint256[3]) (contracts/lending/LendingTemplate.sol#43-60) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(editLoan(uint256,uint256,uint16,address,uint256,uint256[3]),loanId,loanAmount,nrOfInstallments,currency,assetsValue,in
tallmentTime)) (contracts/lending/LendingTemplate.sol#53-58)
LendingTemplate.approveLoan(uint256) (contracts/lending/LendingTemplate.sol#64-74) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(approveLoan(uint256),loanId)) (contracts/lending/LendingTemplate.sol#67-72)
LendingTemplate.cancelLoan(uint256) (contracts/lending/LendingTemplate.sol#78-88) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(cancelLoan(uint256),loanId)) (contracts/lending/LendingTemplate.sol#81-86)
LendingTemplate.payLoan(uint256,uint256) (contracts/lending/LendingTemplate.sol#93-103) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(payLoan(uint256,uint256),loanId,amount)) (contracts/lending/LendingTemplate.sol#96-101)
LendingTemplate.terminateLoan(uint256) (contracts/lending/LendingTemplate.sol#108-118) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(terminateLoan(uint256),loanId)) (contracts/lending/LendingTemplate.sol#111-116)
LendingTemplate.setGlobalVariables(uint256,uint256,uint256,uint32,address,address,address) (contracts/lending/LendingTemplate.sol#121-139) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(setGlobalVariables(uint256,uint256,uint256,uint32,address,address,address),_ltv,_interestRate,_interestRateToStater,_l
enderFee,_promissoryNoteAddress,_lendingMethodsAddress,_lendingDiscountsAddress)) (contracts/lending/LendingTemplate.sol#132-137)
LendingTemplate.promissoryExchange(address,address,uint256[]) (contracts/lending/LendingTemplate.sol#141-152) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(promissoryExchange(address,address,uint256[]),from,to,loanIds)) (contracts/lending/LendingTemplate.sol#145-150)
LendingTemplate.setPromissoryPermissions(uint256[],address,address) (contracts/lending/LendingTemplate.sol#154-165) uses delegatecall to a input-controlled function id
        - (success) = lendingMethodsAddress.delegatecall(abi.encodeWithSignature(setPromissoryPermissions(uint256[],address,address),loanIds,sender,allowed)) (contracts/lending/LendingTemplate.sol#15
8-163)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#controlled-delegatecall
INFO:Detectors:
LendingCore.loans (contracts/lending/LendingCore.sol#107) is never initialized. It is used in:
        - LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140)
        - LendingTemplate.getLoanRemainToPay(uint256) (contracts/lending/LendingTemplate.sol#167-169)
        - LendingTemplate.getLoanApprovalCost(uint256) (contracts/lending/LendingTemplate.sol#172-180)
        - LendingTemplate.getLoanInstallmentCost(uint256,uint256) (contracts/lending/LendingTemplate.sol#182-201)
        - LendingTemplate.getLoanStartEnd(uint256) (contracts/lending/LendingTemplate.sol#203-205)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
LendingCore.lackOfPayment(uint256) (contracts/lending/LendingCore.sol#123-140) performs a multiplication on the result of a division:
        -loans[loanId].status == Status.APPROVED && loans[loanId].startEnd[0].add(loans[loanId].nrOfPayments.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) <= block.timestamp
.sub(loans[loanId].defaultingLimit.mul(loans[loanId].installmentTime.div(loans[loanId].nrOfInstallments))) (contracts/lending/LendingCore.sol#124-139)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

According to the test results, most of the findings found by slither were considered as false positives. Relevant findings were reviewed by the auditors.

3.5 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. In addition, security detections are only in scope.

Results:

LendingCore.sol

Report for lending/LendingCore.sol
https://dashboard.mythx.io/#/console/analyses/49407f16-f7ce-46c0-bb88-7b0acf1fb872

Line	SWC Title	Severity	Short Description
9	(SWC-123) Requirement Violation	Low	Requirement violation.
123	(SWC-000) Unknown	Medium	Function could be marked as external.
143	(SWC-000) Unknown	Medium	Function could be marked as external.

LendingMethods.sol

Report for lending/LendingMethods.sol
https://dashboard.mythx.io/#/console/analyses/95119068-f5aa-4171-af12-8fb256596e90

Line	SWC Title	Severity	Short Description
226	(SWC-134) Message call with hardcoded gas amount	Low	Call with hardcoded gas amount.

LendingTemplate.sol

Report for lending/LendingTemplate.sol
https://dashboard.mythx.io/#/console/analyses/d7ef6811-ad58-48c8-b5e5-a0d09ae8ea75

Line	SWC Title	Severity	Short Description
11	(SWC-100) Function Default Visibility	Low	Function visibility is not set.

THANK YOU FOR CHOOSING

// HALBORN