Unicrypt

Locker v3

SMART CONTRACT AUDIT

28.04.2023

Made in Germany by Chainsulting.de

Table of contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of SDD Tech OÜ. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (18.04.2022)	Layout
0.4 (19.04.2022)	Automated Security Testing Manual Security Testing
0.5 (20.04.2023)	Verify Claims and Test Deployment
0.6 (21.04.2023)	Testing SWC Checks
0.9 (21.04.2023)	Summary and Recommendation
1.0 (25.04.2023)	Final document
1.1 (27.04.2023)	Re-check

2. About the Project and Company

Company address:

SDD Tech OÜ
Mustamäe tee 6b
Tallinn Harjumaa 10616

Website: https://unicrypt.network

Twitter: https://twitter.com/UNCX_token

Telegram: https://t.me/uncx_token

Medium: https://unicrypt.medium.com

2.1 Project Overview

UniCrypt is a decentralized services provider which offers several ways for DeFi projects to build community trust and keep users safe. Famously, UniCrypt created the first-ever liquidity locking smart contracts for Uniswap on Ethereum, known as Proof-of-Liquidity or POL. From there the project continued to develop new features, combining liquidity locking with a decentralized launchpad.

Liquidity Lockers: these are smart contracts that enable teams to publicly lock liquidity on Uniswap or other AMMs for a predetermined period. Essentially, it’s a guarantee to investors that the project developers can’t drain the pool of all the funds. A key innovation is UniCrypt’s lockers will be able to migrate liquidity to Uniswap V3 when the time comes.

FaaS: This is a yield farming-as-a-service protocol that enables the creation of a farm for any token. Launch a farm in a couple clicks using the UI, all automatic with no coding necessary.

Launchpad: Perhaps the most interesting service, a 100% decentralized and automated presale platform that is connected to the liquidity lockers. Once the presale ends a portion of the raised funds (between 30% to 100%) will create the DEX pair on a supported AMM and the liquidity will be locked.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i. Review of the specifications, sources, and instructions provided to Chainsulting to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Chainsulting describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review

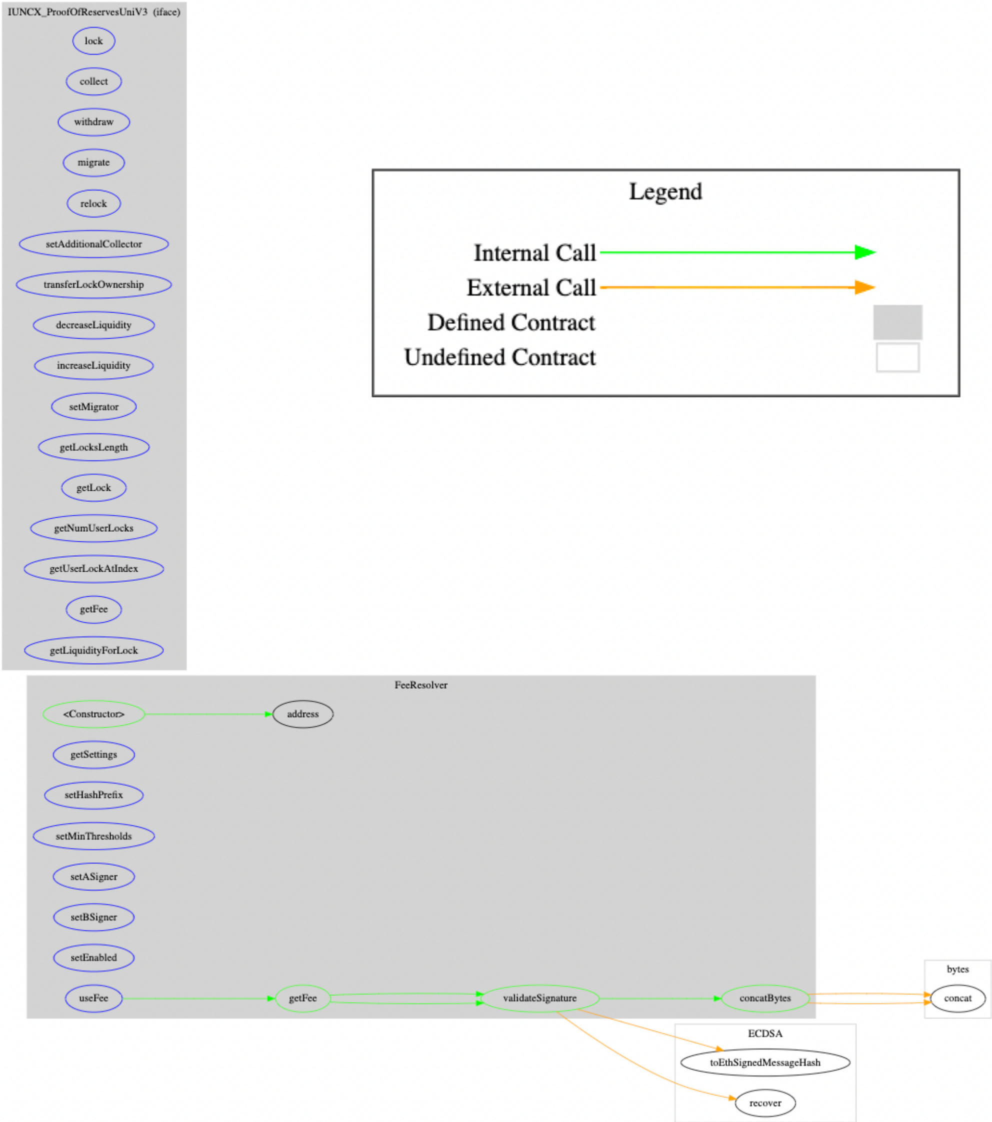
File	Fingerprint (MD5)
./contracts/uniswap-updated/LiquidityAmounts.sol	3f42eb6fd30782bcf58ff091cff9a835
./contracts/uniswap-updated/TickMath.sol	bfad8b11801942517f1e3c5bf564e90e
./contracts/uniswap-updated/INonfungiblePositionManager.sol	ec798d8225a6d51619fb2eba7e81ab74
./contracts/uniswap-updated/FullMath.sol	ee2b7f3eeb0b1386a8b0a433856fced3
./contracts/ICountryList.sol	442dfda2de615687a0ac89620eb34c58
./contracts/IUNCX_ProofOfReservesUniV3.sol	8bf448e710e9c343311678f490826503
./contracts/IMigrateV3NFT.sol	1e6d57c3907b308172457b8ea14a7a88
./contracts/FeeResolver.sol	7f79e8467888e862dbf0cb6afc4bb888
./contracts/MigrateV3NFT.sol	59c680db47157594223f0f390d5518f7
./contracts/UNCX_ProofOfReservesUniV3.sol	c9ccb4b86b01f9cedf90fce7de262a38

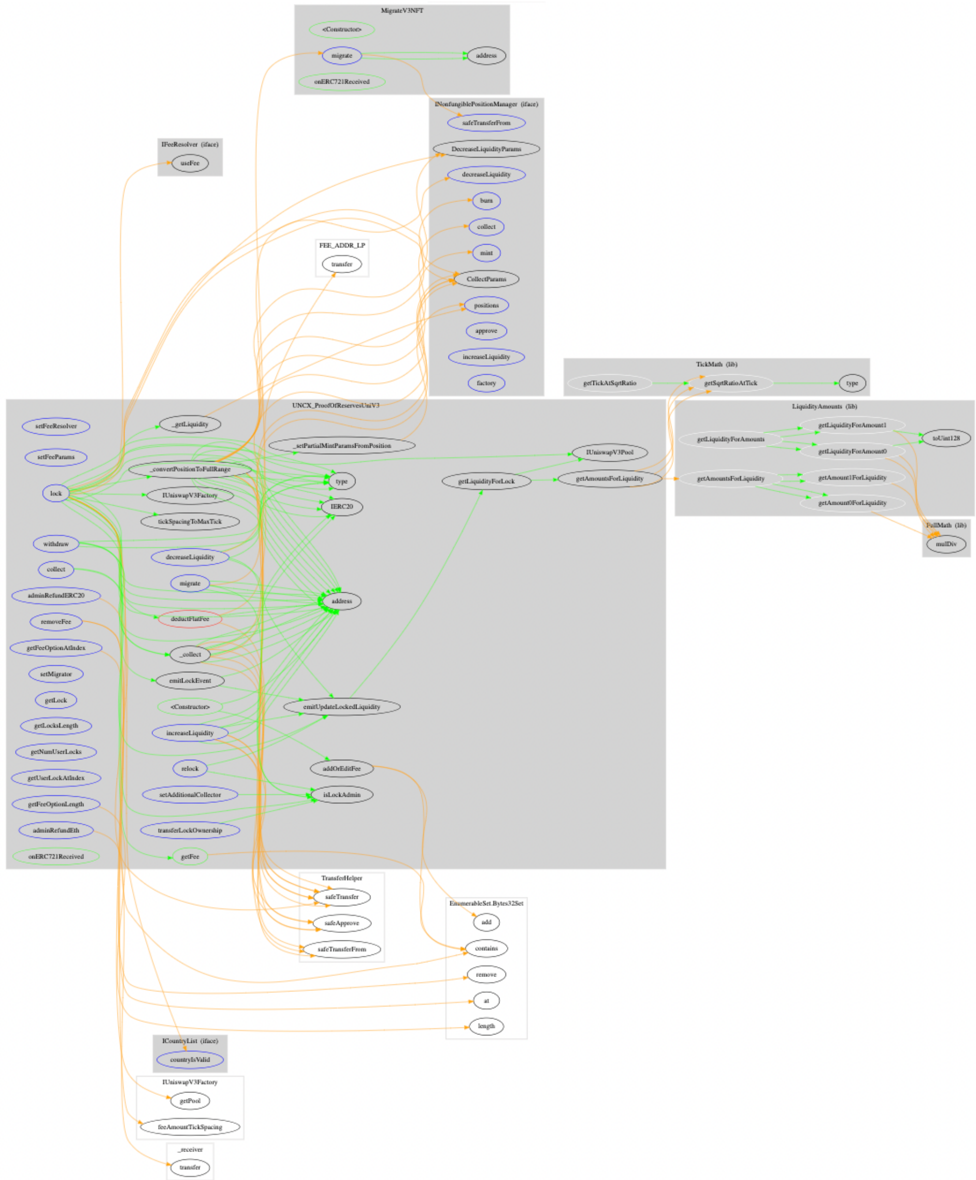
5.2 Used Code from other Frameworks/Smart Contracts (direct imports)

Dependency / Import Path	Source
@openzeppelin/contracts/access/Ownable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/access/Ownable.sol
@openzeppelin/contracts/security/ReentrancyGuard.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/security/ReentrancyGuard.sol
@openzeppelin/contracts/token/ERC20/IERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/token/ERC20/IERC20.sol
@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/token/ERC721/IERC721Receiver.sol
@openzeppelin/contracts/utils/cryptography/ECDSA.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/utils/cryptography/ECDSA.sol
@openzeppelin/contracts/utils/structs/EnumerableSet.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.2/contracts/utils/structs/EnumerableSet.sol
@uniswap/v3-core/contracts/interfaces/IUniswapV3Factory.sol	https://github.com/Uniswap/v3-core/tree/v1.0.0/contracts/interfaces/IUniswapV3Factory.sol
@uniswap/v3-core/contracts/interfaces/IUniswapV3Pool.sol	https://github.com/Uniswap/v3-core/tree/v1.0.0/contracts/interfaces/IUniswapV3Pool.sol
@uniswap/v3-core/contracts/libraries/FixedPoint96.sol	https://github.com/Uniswap/v3-core/tree/v1.0.0/contracts/libraries/FixedPoint96.sol

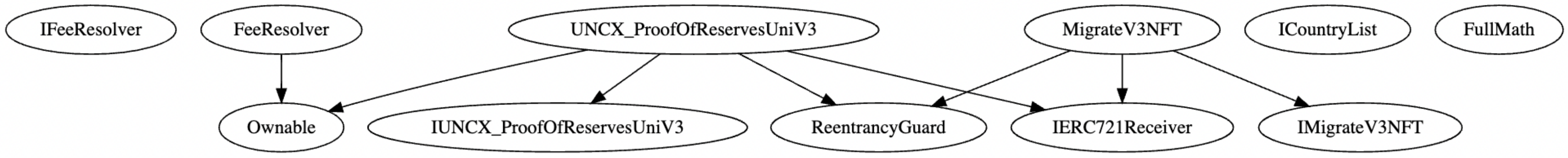
Dependency / Import Path	Source
@uniswap/v3-periphery/contracts/libraries/TransferHelper.sol	https://github.com/Uniswap/v3-periphery/tree/main/contracts/libraries/TransferHelper.sol

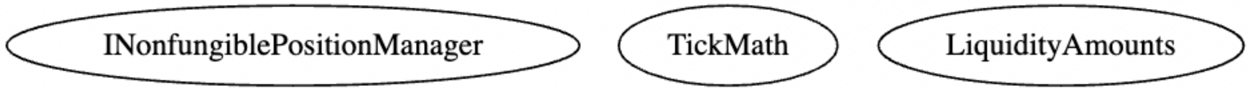
5.3 CallGraph



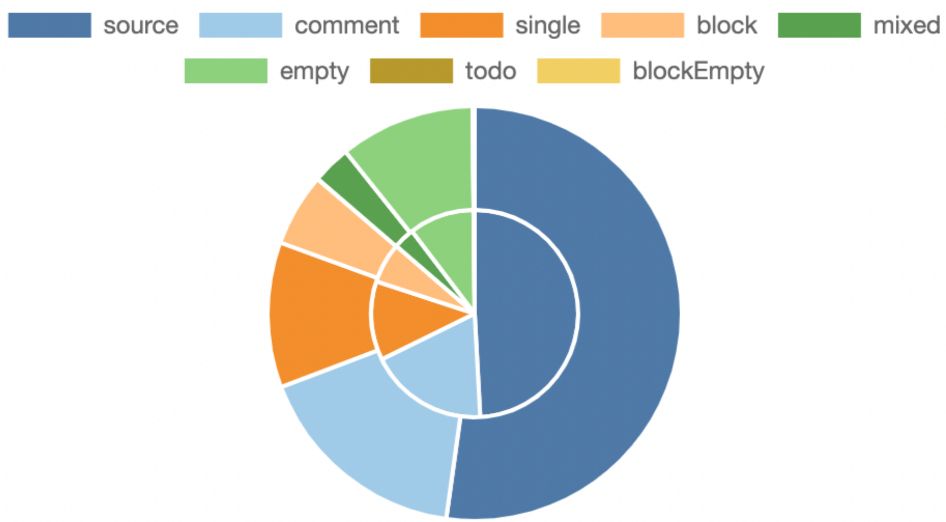


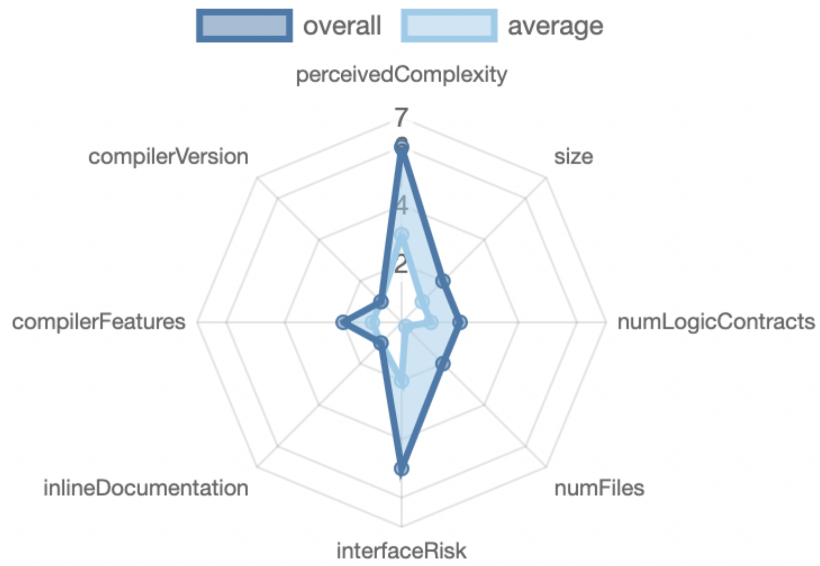
5.4 Inheritance Graph





5.5 Source Lines & Risk





5.6 Capabilities

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥 Uses Assembly	💣 Has Destroyable Contracts

`^0.8.9`
`^0.8.0`
`>=0.5.0`
		`yes`	`yes`
			(25 asm blocks)	

📤 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧮 Uses Hash Functions	🔖 ECRecover	🌀 New/Create/Create2
`yes`			`yes`		

Exposed Functions

This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

🌐Public	💰Payable
67	11

External	Internal	Private	Pure	View
57	48	9	14	24

5.7 Source Unites in Scope

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
📝🔍	./contracts/UNCX_ProofOfReservesUniV3.sol	1	1	545	537	358	119	313	💰📤🧮
📝	./contracts/MigrateV3NFT.sol	1		36	31	20	3	17	
📝	./contracts/FeeResolver.sol	1		137	133	88	25	65	🧮
🔍	./contracts/IMigrateV3NFT.sol		1	12	11	4	4	3	
🔍	./contracts/IUNCX_ProofOfReservesUniV3.sol		1	112	48	35	36	42	💰
🔍	./contracts/ICountryList.sol		1	10	9	3	4	3	🔆
📚	./contracts/uniswap-updated/FullMath.sol	1		96	96	40	49	83	🖥Σ
🔍	./contracts/uniswap-updated/INonfungiblePositionManager.sol		1	141	80	51	21	34	💰
📚	./contracts/uniswap-updated/TickMath.sol	1		254	246	207	24	584	🖥
📚	./contracts/uniswap-updated/LiquidityAmounts.sol	1		137	110	53	44	30	

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
📝📚🔍	**Totals**	**6**	**5**	**1480**	**1301**	**859**	**329**	**1174**	🖥💰📤🧮🔆 **Σ**

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments
- **Complexity Score**: a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...)

6. Scope of Work

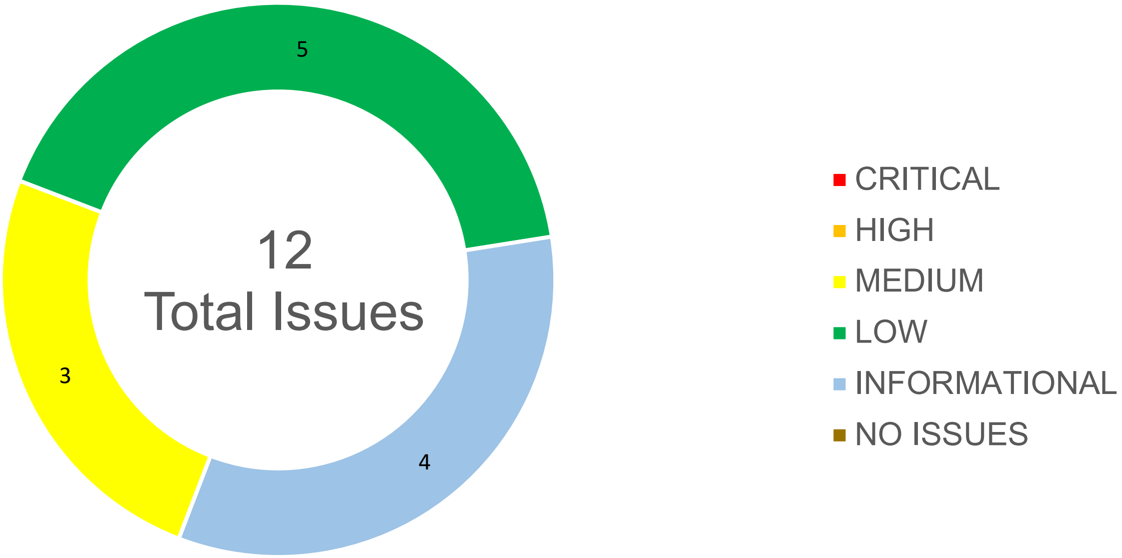
The Unicrypt Team provided us with the files that needs to be tested. The scope of the audit is the Locker v3 contract.

The team put forward the following assumptions regarding the security, usage of the contracts:

1. The contract creators have implemented thorough access control mechanisms to prevent unauthorized actions, such as using the Ownable contract and limiting certain actions to specific signers (A_SIGNER and B_SIGNER).
2. The contract has been designed to prevent common attack vectors such as reentrancy attacks, integer overflows/underflows, and front-running. The contracts rely on the proper configuration of the external contracts they interact with, such as the Uniswap V3 contracts, OpenZeppelin contracts, and any potential oracles.
3. The contracts assume that the signers (A_SIGNER and B_SIGNER) are trusted entities, and they will not act maliciously or collude with external parties. The nonce system is implemented effectively to prevent replay attacks and ensure that signed messages are unique and valid only for one-time usage.
4. The contracts assume that fees are set reasonably by A_SIGNER and B_SIGNER, and the system will not be exploited by adjusting the fees inappropriately.
5. The contract developers have made sure that the contract logic is gas-efficient and optimized to minimize the risk of running out of gas during contract execution.
6. The smart contract is coded according to the newest standards and in a secure way.

The main goal of this audit was to verify these claims. The auditors can provide additional feedback on the code upon the client’s request.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	No Fee Charged On Custom Fee Locking	MEDIUM	FIXED
6.2.2	Possible Front-Running Attack With Custom Fees	MEDIUM	FIXED
6.2.3	Bad Test Coverage	MEDIUM	ACKNOWLEDGED
6.2.4	Potential Loss Of Contract Ownership	LOW	FIXED
6.2.5	Potential Loss of Token Lock Ownership	LOW	FIXED
6.2.6	Missing Value Validation	LOW	ACKNOWLEDGED
6.2.7	Incorrect Return Value of getLocksLength Function	LOW	FIXED
6.2.8	Insecure Adjustment of Hash Prefix Value	LOW	FIXED
6.2.9	Floating Compiler Version	INFORMATIONAL	FIXED
6.2.10	Missing Natspec Documentation	INFORMATIONAL	ACKNOWLEDGED

6.2.11	Redundant Zero Address Checking	INFORMATIONAL	FIXED
6.2.12	Missing Unlock Date Time Check	INFORMATIONAL	FIXED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, Chainsulting's experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, Chainsulting's experts found **no High issues** in the code of the smart contract.

MEDIUM ISSUES

During the audit, Chainsulting's experts found 3 Medium issues in the code of the smart contract.

6.2.1 No Fee Charged On Custom Fee Locking

Severity: MEDUM
Status: FIXED
Code: CWE-862
File(s) affected: UNCX_ProofOfReservesUniV3.sol

Attack / Description	In the current implementation of the lock function, the caller can pass a custom fee object into the function. The signature of the passed custom fee object is checked and if it is valid, no fee will be charged during the whole locking process due to not setting the fee to the desired data.

Code	Line 149 – 154 (UNCX_ProofOfReservesUniV3.sol)
	`FeeStruct memory fee;`
	`        if (params.r.length > 0) {`

	<pre>FEE_RESOLVER.useFee(params.r);
	} else {
	    fee = getFee(params.feeName);
	}
	
	if (fee.flatFee > 0) {
	    deductFlatFee(fee);
	}</pre>

Result/Recommendation	It is recommended to set the *fee* variable to the returned fee struct of the *useFee* function, which checks the passed fee for validity.
	
	if (params.r.length > 0) {
	fee = FEE_RESOLVER.useFee(params.r);
	} else {

6.2.2 Possible Front-Running Attack With Custom Fees
Severity: MEDUM
Status: FIXED
Code: NA
File(s) affected: UNCX_ProofOfReservesUniV3.sol, FeeResolver.sol

Attack / Description	In the current implementation of the *lock* function, the caller can pass a custom fee object into the function. The signature of the passed custom fee object is checked and the fee will be applied to the specific lock. However, the signature signs only the fee related data and not any data verifying the caller or targeted token id. An attacker could wait for an unconfirmed transaction with a valid custom fee struct and front-run the transaction by calling the lock function with any other NFT and

	the custom fee type. Thus, the custom fee can be used by other NFTs before the actual NFT get locked.

Code	Line 94 -123 (FeeResolver.sol)
	function validateSignature (bytes[] memory args, address requiredSigner) public view {
	bytes32 messageHash = keccak256(concatBytes(args));
	bytes32 prefixedHash = ECDSA.toEthSignedMessageHash(messageHash);
	address signer = ECDSA.recover(prefixedHash, args[args.length - 1]);
	require(signer == requiredSigner, "R sig");
	}
	
	// public getter
	function getFee(bytes[] memory args) public view returns (IUNCX_ProofOfReservesUniV3.FeeStruct memory) {
	bytes32 nonce = abi.decode(args[0], (bytes32));
	
	require(args.length == 5, "R Length");
	require(NONCE_USED[nonce] != true, "R used");
	require(ENABLED, "R not enabled");
	
	uint256 lpFee = abi.decode(args[2], (uint256));
	uint256 collectFee = abi.decode(args[3], (uint256));
	
	if (abi.decode(args[1], (bool))) {
	validateSignature(args, A_SIGNER);
	} else {
	require(lpFee >= LP_MIN && collectFee >= COLLECT_MIN, "R threshold");
	validateSignature(args, B_SIGNER);
	}
	
	IUNCX_ProofOfReservesUniV3.FeeStruct memory newFee;
	newFee.lpFee = lpFee;

	newFee.collectFee = collectFee;
	return newFee;
	}

Result/Recommendation	It is recommended to include the token id of the NFT that should be locked with the discounted fee. The signature should only be valid for a fee modal for a specific NFT to prevent front-running attacks.

6.2.3 Bad Test Coverage
Severity: MEDIUM
Status: ACKNOWLEDGED
Code: NA
File(s) affected: All

Attack / Description	In the current implementation, several functions are rarely tested.
Code	NA
Result/Recommendation	It is recommended to enhance the test coverage to reach nearly 100%. Tests are crucial to validate that the contracts are behaving and functioning as expected. To check the total test coverage, it is recommended to use solidity-coverage hardhat plugin and test every branch of every function call with happy and sad path testing.

LOW ISSUES

During the audit, Chainsulting's experts found **5 Low issues** in the code of the smart contract

6.2.4 Potential Loss Of Contract Ownership

Severity: LOW
Status: FIXED
Code: CWE-862
File(s) affected: UNCX_ProofOfReservesUniV3.sol, FeeResolver.sol

Attack / Description	The owner of the contract can be changed by calling transferOwnership function of OpenZeppelin Ownable implementation. This function directly sets the owner to the given address, if the address is not the zero address. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.
Code	`import "@openzeppelin/contracts/access/Ownable.sol";`
Result/Recommendation	It is recommended to use the two step pattern by setting the new owner as potential owner in the first step and approving the ownership transfer by calling an approve function by the new owner. OpenZeppelin provides such a Ownable2Step implementation. This prevents transferring ownership to an invalid address.

6.2.5 Potential Loss of Token Lock Ownership

Severity: LOW
Status: FIXED
Code: CWE-862
File(s) affected: UNCX_ProofOfReservesUniV3.sol

Attack / Description	The owner of the token lock can be changed by calling transferLockOwnership function. This function directly sets the token lock owner to the given address, if the address is not the current

	owner address. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes. It is also possible to transfer the lock ownership to the zero address.

Code	Line 415 – 424 (UNCX_ProofOfReservesUniV3.sol)
	`function transferLockOwnership (uint256 _lockId, address _newOwner) external override nonReentrant {`
	`isLockAdmin(_lockId);`
	`require(msg.sender != _newOwner, "TRANSFER TO SAME OWNER");`
	`Lock storage userLock = LOCKS[_lockId];`
	`USER_LOCKS[userLock.owner].remove(_lockId);`
	`userLock.owner = _newOwner;`
	`USER_LOCKS[_newOwner].add(_lockId);`
	
	`emit onTransferLockOwnership(_lockId, msg.sender, _newOwner);`
	`}`

Result/Recommendation	It is recommended to use the two step pattern by setting the new token lock owner as potential owner in the first step and approving the ownership transfer by calling an approve function by the new owner. This prevents transferring ownership to an invalid address. It could be implemented by adding an extra field to the lock struct.

6.2.6 Missing Value Validation

Severity: LOW
Status: ACKNOWLEDGED
Code: CWE-20
File(s) affected: UNCX_ProofOfReservesUniV3.sol, FeeResolver.sol

Attack / Description	Several onlyOwner functions lack value safety checks. Therefore, only values that are consistent with the logic of the contract should be permitted. Missing address validation checks could lead to contract deployment without properly set addresses. Wrong dimensions of fees could lead to unintended behaviour of the contract.

Code	Line 77 – 79 (UNCX_ProofOfReservesUniV3.sol)
	`function setFeeResolver (IFeeResolver _resolver) external onlyOwner {`
	`FEE_RESOLVER = _resolver;`
	`}`
	
	Line 81 – 85
	`function setFeeParams (address _autoCollectAccount, address payable _lpFeeReceiver, address payable _collectFeeReceiver) external onlyOwner {`
	`AUTO_COLLECT_ACCOUNT = _autoCollectAccount;`
	`FEE_ADDR_LP = _lpFeeReceiver;`
	`FEE_ADDR_COLLECT = _collectFeeReceiver;`
	`}`
	
	Line 58 – 69 (FeeResolver.sol)
	`function setMinThresholds (uint256 _lpMin, uint256 _collectMin) external onlyOwner {`
	`LP_MIN = _lpMin;`
	`COLLECT_MIN = _collectMin;`
	`}`
	
	`function setASigner (address _signer) external onlyOwner {`
	`A_SIGNER = _signer;`
	`}`
	
	`function setBSigner (address _signer) external onlyOwner {`
	`B_SIGNER = _signer;`
	`}`

Result/Recommendation	It is recommended to check address values for correctness. This can be done in first stage to exclude zero address in a require statement. In second stage to check if an address is a contract. And most specific for contracts if an address implements a specified interface (EIP-165).

	Additionally, it is recommended to allow only variable values consistent with the contract logic (i.e. fee values not greater than denominator).

6.2.7 Incorrect Return Value of getLocksLength Function

Severity: LOW
Status: FIXED
Code: CWE-476
File(s) affected: UNCX_ProofOfReservesUniV3.sol

Attack / Description	The function *getLocksLength* is meant to represent the number of all unique locks in the contract but it only returns the number of all ever created locks including the deleted ones.

Code	Line 453 – 455 (UNCX_ProofOfReservesUniV3.sol)
	`function getLocksLength() external view override returns (uint256) {`
	`        return NONCE;`
	`    }`

Result/Recommendation	It is recommended to implement a correct counter of all currently existing locks or adjust the description of the function, as it is not implementing what it should.

6.2.8 Insecure Adjustment of Hash Prefix Value

Severity: LOW
Status: FIXED
Code: NA
File(s) affected: FeeResolver.sol

Attack / Description	In the current implementation the signatures to allow a custom fee for NFTs is including a HASH_PREFIX to prevent signature replay attacks on other contracts or blockchains. It initially is

	set in the constructor and includes the chain id as well as the contracts address. However, the owner of the contract is allowed to set this hash to arbitrary data, which can lead to same HASH_PREFIX values on different chains or in different contracts.

Code	Line 41 (FeeResolver.sol)
	`HASH_PREFIX = keccak256(abi.encode(block.chainid, address(this)));`
	
	Line 54 – 56 (FeeResolver.sol)
	`function setHashPrefix (bytes32 _hashPrefix) external onlyOwner {`
	`HASH_PREFIX = _hashPrefix;`
	`}`

Result/Recommendation	It is recommended to remove the option to change the HASH_PREFIX during runtime. If it is really needed to adjust the HASH_PREFIX it is recommended to initially include a version number and write a setter function, where you can change only the version number and the chain id as well as the contract address are included in any .
	
	For example:
	`HASH_PREFIX = keccak256(abi.encode(block.chainid, address(this), 1));`
	
	`function setHashPrefix (uint256 _version) external onlyOwner {`
	`HASH_PREFIX = keccak256(abi.encode(block.chainid, address(this), _version));`
	`}`

INFORMATIONAL ISSUES

During the audit, Chainsulting's experts found **4 Informational issues** in the code of the smart contract

6.2.9 Floating Compiler Version

Severity: INFORMATIONAL
Status: FIXED

Code: SWC-103
File(s) affected: All

Attack / Description	The current pragma Solidity directive is floating. It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.
Code	pragma solidity ^0.8.0; pragma solidity ^0.8.9;
Result/Recommendation	It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee. i.e. Pragma solidity 0.8.9 See SWC-103: https://swcregistry.io/docs/SWC-103

6.2.10 Missing Natspec Documentation

Severity: LOW
Status: ACKNOWLEDGED
Code: CWE-1059
File(s) affected: All

Attack / Description	Solidity contracts can use a special form of comments to provide rich documentation for function, return variables, and more. This special form is named Ethereum Natural Language Specification Format(NatSpec).
Code	NA

Result/Recommendation	It is recommended to include natspec documentation and follow the doxygen style including @author, @title, @notice, @dev, @param, @return and make it easier to review and understand your smart contract.

6.2.11 Redundant Zero Address Checking
Severity: INFORMATIONAL
Status: FIXED
Code: CWE-697
File(s) affected: UNCX_ProofOfReservesUniV3.sol

Attack / Description	The _collect function first checks if the caller is the userLock.additionalCollector and afterwards if the userLock.additionalCollector is not the zero address. If it is equal to the caller, it can never be the zero address and thus this check is redundant.
Code	Line 285 (UNCX_ProofOfReservesUniV3.sol) `require(userLock.owner == msg.sender || (userLock.additionalCollector == msg.sender && userLock.additionalCollector != address(0)) || collectorIsBot, "OWNER");`
Result/Recommendation	It is recommended to remove the additional check of non-zero address if the caller is already verified as the additional collector.

6.2.12 Missing Unlock Date Time Check
Severity: INFORMATIONAL
Status: FIXED
Code: NA

File(s) affected: UNCX_ProofOfReservesUniV3.sol

Attack / Description	The caller passed a unlock date into the lock function. This unlock date is checked for the format and if it is forever, but not if it is in the future. The user has no advantage by setting this date to a timestamp in the past, but it may be helpful to prevent locks with invalid unlock dates.
Code	Line 147 (UNCX_ProofOfReservesUniV3.sol) `require(params.unlockDate < 1e10 || params.unlockDate == ETERNAL_LOCK, 'TIMESTAMP: Milliseconds detected, Use seconds'); // prevents errors when timestamp entered in`
Result/Recommendation	It is recommended to check in a require check if the unlock date is in the future.

6.3 SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✅
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✅
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✅

ID	Title	Relationships	Test Result
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✅
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✅
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✅
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✅
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✅
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✅
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✅
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✅
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✅

ID	Title	Relationships	Test Result
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✅
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✅
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✅
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✅
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✅
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✅
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✅
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✅
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✅
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✅

ID	Title	Relationships	Test Result
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✅
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✅
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✅
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✅
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✅
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✅
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✅
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✅
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✅

6.4 Verify Claims

6.4.1 The contract has implemented thorough access control mechanisms to prevent unauthorized actions, such as using the Ownable contract and limiting certain actions to specific signers (A_SIGNER and B_SIGNER).
Status: tested and verified ✅

6.4.2 The contract has been designed to prevent common attack vectors such as reentrancy attacks, integer overflows/underflows, and front-running. The contracts rely on the proper configuration of the external contracts they interact with, such as the Uniswap V3 contracts and OpenZeppelin contracts.
Status: tested and verified ✅

6.4.3 The contracts assume that the signers (A_SIGNER and B_SIGNER) are trusted entities, and they will not act maliciously or collude with external parties. The nonce system is implemented effectively to prevent replay attacks and ensure that signed messages are unique and valid only for one-time usage.
Status: tested and verified ✅

6.4.4 The contracts assume that fees are set reasonably by A_SIGNER and B_SIGNER, and the system will not be exploited by adjusting the fees inappropriately.
Status: tested and verified ✅

6.4.5 The contract developers have made sure that the contract logic is gas-efficient and optimized to minimize the risk of running out of gas during contract execution.
Status: tested and verified ✅

6.4.6 The smart contract is coded according to the newest standards and in a secure way.
Status: tested and verified ✅

6.5 Unit Tests

Uniswap V3 Lockers

Lock a NFT on mainnet

---------- Liquidity NFT Before ----------

LIQUIDITY WETH 1.100392828184218327 81%

LIQUIDITY USDT 461.103937 19%

---------- Liquidity NFT After -----------

LIQUIDITY WETH 0.251227963003636972 50%

LIQUIDITY USDT 456.492897 50%

---------- Balances ----------------------

DUST WETH 0.846627208988863206

DUST USDT 0.0

LP_FEE WETH 0.0025376561919492944

LP_FEE USDT 4.611039

FLAT_FEE 1.5 ETH / GAS TOKEN

---------- SUM TOTAL ----------------------

SUM 1.100392828184218326

SUM 461.103936

CurrentTick Before -201271

CurrentTick After -201271

✔ Should lock a nft you enter from the mainnet (13183ms)

1 passing (13s)

7. Executive Summary

Two (2) independent Chainsulting experts performed an unbiased and isolated audit of the smart contract codebase.

The main goal of the audit was to verify the claims regarding the security and functions of the smart contract. During the audit, no critical, no high, three medium, five low and four informational issues have been found, after the manual and automated security testing.

We advise the Unicrypt team to implement the recommendations to further enhance the code's security and readability.

Update (28.04.2023): Unicrypt team fixed all necessary issues.

8. About the Auditor

Chainsulting is a professional software development firm, founded in 2017 and based in Germany. They show ways, opportunities, risks and offer comprehensive Web3 solutions. Their services include Web3 development, security and consulting.

Chainsulting conducts code audits on market-leading blockchains such as Solana, Tezos, Ethereum, Binance Smart Chain, and Polygon to mitigate risk and instil trust and transparency into the vibrant crypto community. They have also reviewed and secure the smart contracts of many top DeFi projects.

Chainsulting currently secures $100 billion in user funds locked in multiple DeFi protocols. The team behind the leading audit firm relies on their robust technical know-how in the web3 sector to deliver top-notch smart contract audit solutions, tailored to the clients' evolving business needs.

Check our website for further information: https://chainsulting.de