Live Art Market

Artwork & Exchange

SMART CONTRACT AUDIT

10.07.2021

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 Live Art Inc (liveart.market). 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 (20.06.2021)	Layout
0.4 (21.06.2021)	Automated Security Testing Manual Security Testing
0.5 (23.06.2021)	Verify Claims and Test Deployment
0.6 (22.06.2021)	Testing SWC Checks
0.9 (23.06.2021)	Summary and Recommendation
1.1 (24.06.2021)	Final document
1.2 (10.07.2021)	Added deployed contract

2. About the Project and Company

Company address:

Live Art Inc.
24A Trolley Square #2133
Wilmington, DE, 19806
USA

Website: https://liveart.market

Twitter: https://twitter.com/liveartmarket

LinkedIn: https://www.linkedin.com/company/liveartholdings

Instagram: https://www.instagram.com/liveartmarket

Facebook: https://www.facebook.com/LiveArtMarket

2.1 Project Overview

LiveArt Market began limited, invitation-only trading in 2021 and has already achieved sales approaching $5 million, with more than 1,000 works of art valued at approximately $120 million in the pipeline for sale.

Prices are ranging between $50,000 and $500,000, with works by Amoako Boafo and Ed Clark commanding six-figure sums. Early offerings available for purchase include works by Derrick Adams, Jean-Michel Basquiat, Yayoi Kusama, Pablo Picasso and Andy Warhol, among others. LiveArt puts collectors in control by providing participants with one destination for real-time information and an efficient and secure marketplace in which to privately transact. All LiveArt Market participants are extensively vetted and therefore can transact anonymously in virtual deal rooms. Additionally, sellers can control the visibility of their works of art and only share exact details and images once they are comfortable with a potential buyer – addressing two key concerns often raised by market participants.

Marisa Kayyem, Chief Content & Data Officer for LiveArt: “Privacy is a hallmark of LiveArt, critically important for those who want to pursue a potential sale or purchase without the risk of overexposing a work or revealing a collecting strategy. At the same time, LiveArt offers more transparency into the sale process than any other platform or venue – a single seller and a single buyer, and straight-forward and low fees. The virtual deal rooms empower both sellers and buyers to control the outcome and all-in price.”

Sellers upload works of art from their own collection to LiveArt’s AI-powered comprehensive data platform and instantly receive a LiveArt Estimate™, view price trends and comparable sales, and make informed decisions about a potential sale. Buyers discover works by browsing the LiveArt Market and viewing works listed publicly, as well as those listed privately – where comparable works are shown and details are only shared once the seller approves. Once there is commitment to move ahead with a sale, the work is shipped to a secure facility in Delaware for inspection before the sale is completed. Funds are held in escrow before being released to the seller, and a flat 10% fee is charged to successful purchasers.

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

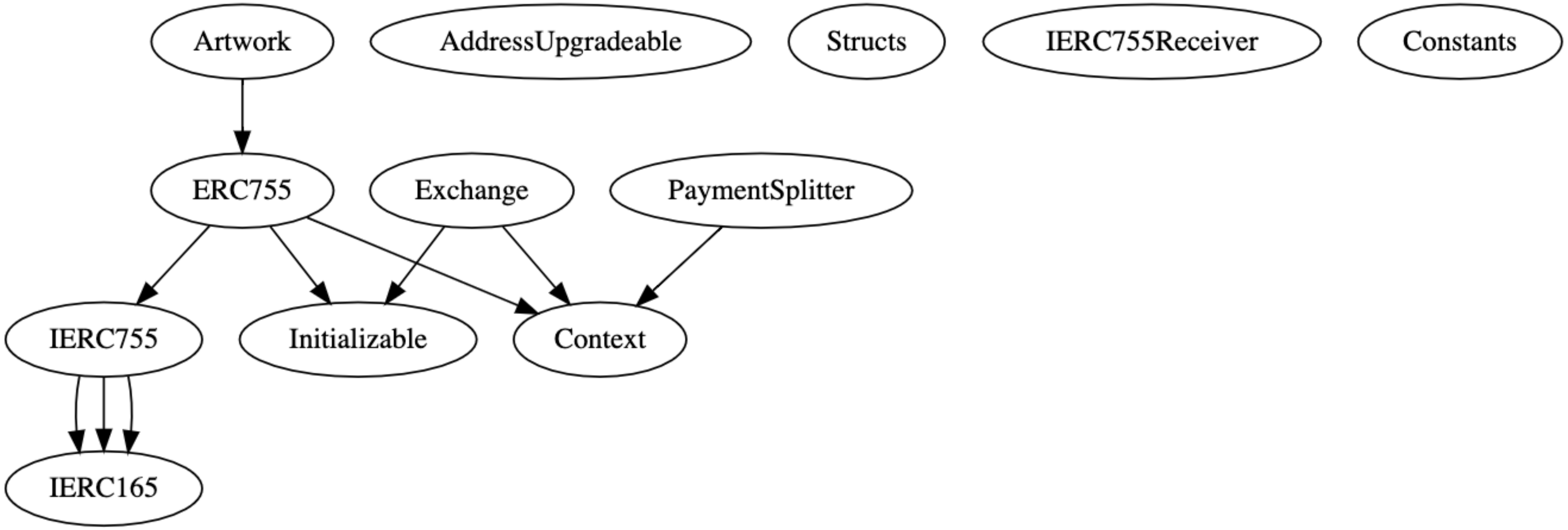
The auditing process follows a routine series of steps:

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.

4.2 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)
./Artwork.sol	9e2f5e4e5c06f728f0802fc6f4e5640c
./Exchange.sol	3b7454a80b4c1b76ca0875ba030f97cf
./IERC755.sol	946314b444461c8242e8c1895a38d1f7



4.3 Metrics / CallGraph (Artwork)

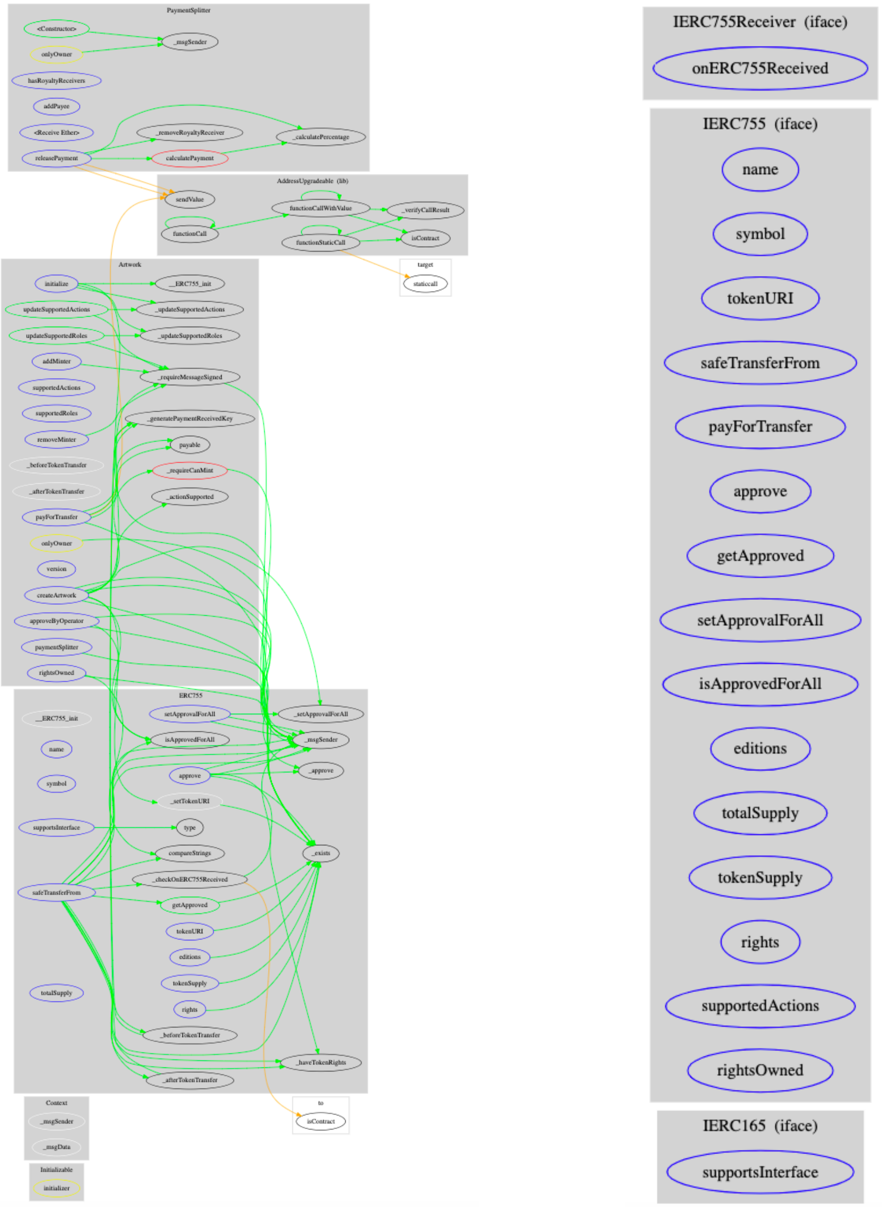
Legend

Internal Call →

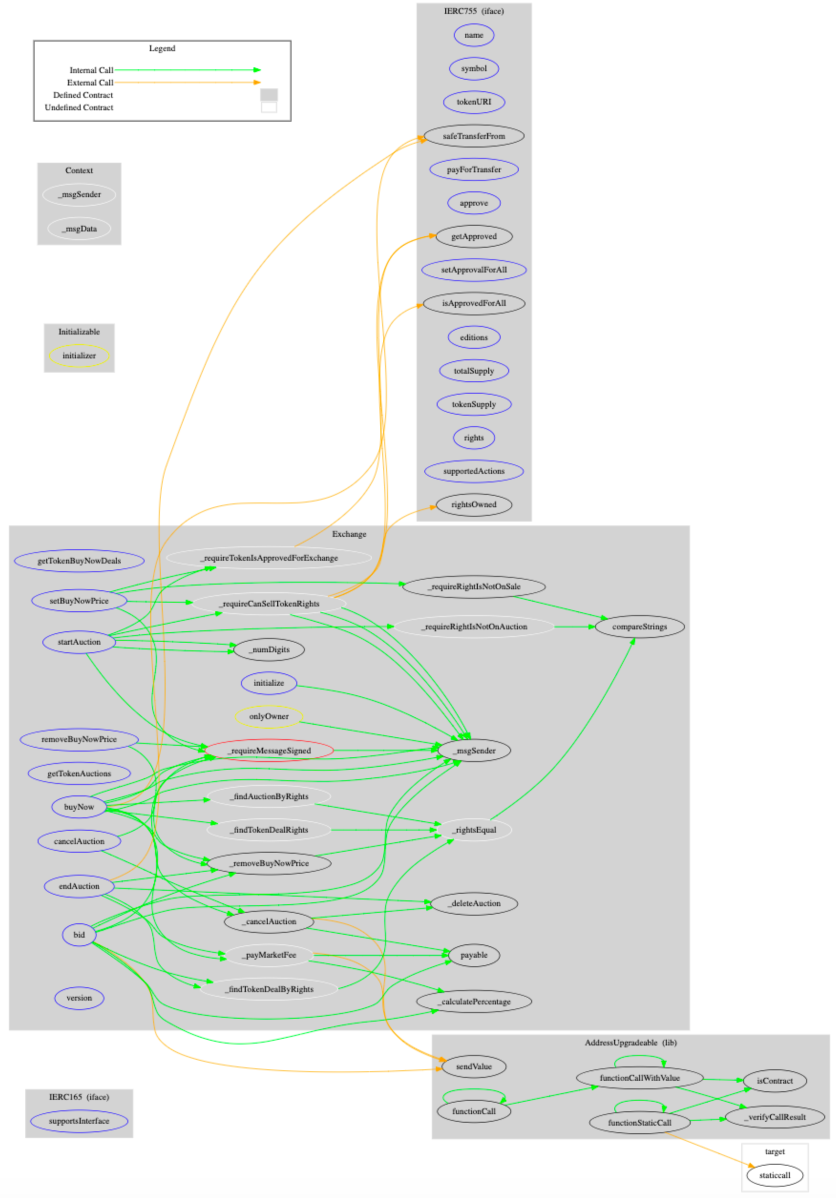
External Call →

Defined Contract ▨

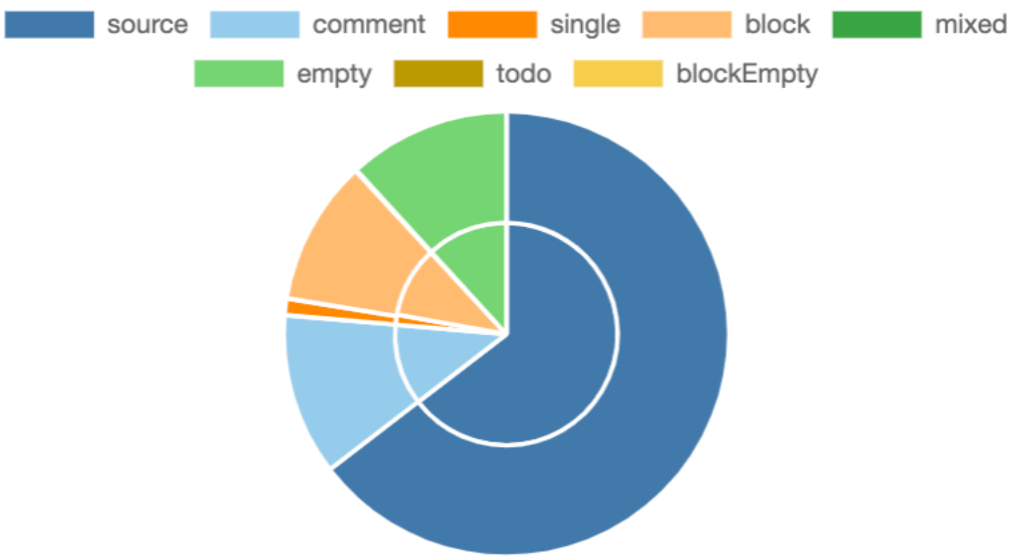
Undefined Contract □

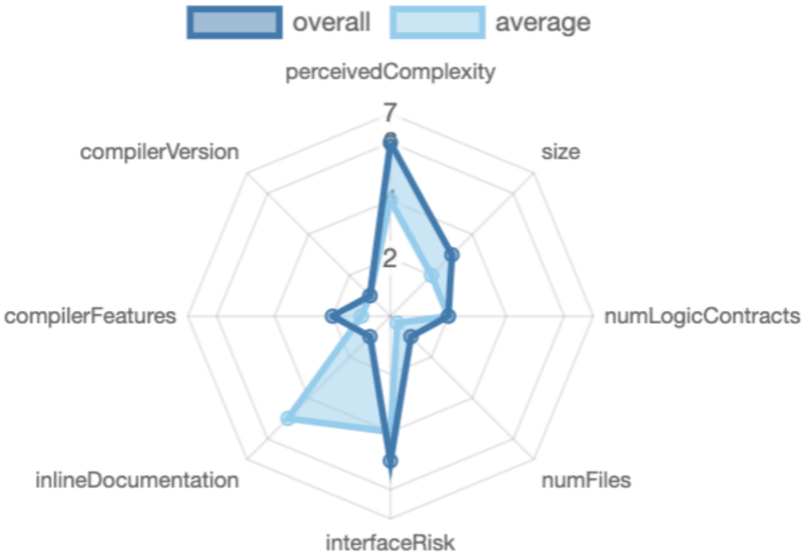


4.4 Metrics / CallGraph (Exchange)



4.5 Metrics / Source Lines & Risk





4.6 Metrics / Capabilities

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥 Uses Assembly	💣 Has Destroyable Contracts
`^0.8.0`		`yes`	`yes` (4 asm blocks)	

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

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
90	16

External	Internal	Private	Pure	View
86	90	16	13	74

StateVariables

Total	🌐Public
42	0

4.7 Metrics / Source Unites in Scope

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
📝📚🔍🎨	smart contracts/Artwork.sol	8	3	1218	1014	740	152	525	🖥💰🧮🔖🌀🔆
📚🔍	smart contracts/IERC755.sol	1	2	142	89	60	18	46	💰🔆
📝📚🔍🎨	smart contracts/Exchange.sol	6	2	1063	885	658	154	419	🖥💰🧮🔖🔆
📝📚🔍🎨	**Totals**	**15**	**7**	**2423**	**1988**	**1458**	**324**	**990**	🖥💰🧮🔖🌀🔆

Legend: [➖]

- **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, ...)

5. Scope of Work

The Live Art Market Team provided us with the file that needs to be tested. The scope of the audit is the Artwork and Exchange NFT contract.

Following contracts with the direct imports has been tested:

- Artwork.sol
- Exchange.sol
- IERC755.sol

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

Artwork smart contract (Artwork.sol/IERC755.sol)

Token minting

- only person with minting granted permission can mint a token
- caller signature is verified
- PaymentSplitter is deployed with royalty receivers config
- no more tokens than maxTokenSupply could be minted
- token can’t be minted without token rights

Token rights transferring

- transfer payment is correctly split between royalty receivers and seller
- token rights could not be transferred without a received payment
- user can’t transfer not owned rights (except rights that he is approved for or an operator for)

Exchange smart contract (Exchange.sol)

Fixed price

- caller signature is verified
- user can’t list same right twice
- user can’t list not owned rights (except rights that he is approved for or an operator for)
- auction with the same rights is cancelled on purchase
- token transfer is done on purchase

Auction

- caller signature is verified
- user can’t auction same right twice
- user can’t list not owned rights (except rights that he is approved for or an operator for)
- fixed price is removed when bid is >= 50% of the fixed price
- user can’t make a bid lower than the initial price and previous bid
- previous bidder funds are released on a new bid
- bidder funds are released on auction cancel
- token transfer is done on auction end if there is a winner bid
- auction end time is extended by 15 minutes on a bid when <= 15 minutes left

- 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.

5.1 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 **no Medium issues** in the code of the smart contract.

LOW ISSUES

5.1.1 Wrong Boolean checked

Severity: LOW
Status: Acknowledged
File(s) affected: Artwork.sol

Attack / Description	Code Snippet	Result/Recommendation

The current implementation are two require checks for the same variable directly after each other. Probably there is a typo and an other variable is meant. This could lead to unintended behaviour.	Line 899 & 900:
	`require(ownerIsSupported, "owner role should be supported");`
	`require(ownerIsSupported, "creator role should be supported");`
		It is recommended to change the second checked variable to *creatorIsSupported* to ensure that really the creator role is supported, as the message says.

5.1.2 A floating pragma is set.

Severity: LOW
Code: SWC-103
Status: Acknowledged
File(s) affected: Artwork.sol, Exchange.sol, IERC755.sol

Attack / Description	Code Snippet	Result/Recommendation
The current pragma Solidity directive is "^0.5.0". 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.	Line 1: `pragma solidity ^0.8.0;`	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 0.8.0

INFORMATIONAL ISSUES

5.1.3 Missing NatSpec documentation
Severity: INFORMATIONAL
Status: FIXED
File(s) affected: Artwork.sol, Exchange.sol, IERC755.sol

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

5.1.4 Public functions could be external
Severity: INFORMATIONAL
Status: Acknowledged
File(s) affected: Artwork.sol

Attack / Description	Code Snippet	Result/Recommendation

In the current implementation several functions are declared as public where they could be external. For public functions Solidity immediately copies array arguments to memory,	Line 832 - 838:
	`function updateSupportedActions(…)`
	`public onlyOwner {`
	Line 863 - 869:
		We recommend declaring functions as external if they are not used internally. This leads to lower gas consumption and better code readability.

while external functions can read directly from calldata. Because memory allocation is expensive, the gas consumption of public functions is higher.	`function updateSupportedRoles(…)`
	`public onlyOwner {`	

5.1.5 uint values can be smaller

Severity: INFORMATIONAL
Status: Acknowledged
File(s) affected: Artwork.sol

Attack / Description	Code Snippet	Result/Recommendation

Too big uint values can cause high gas cost for the end-user.	Line 256 - 261:
	`library Structs {`
	`struct RoyaltyReceiver {`
	`address payable wallet;`
	`string role;`
	`uint256 percentage;`
	`uint256 resalePercentage;`
	`uint256 CAPPS;`
	`uint256 fixedCut;`
	`}`
		uint values can be smaller to save gas for unused storage

5.1.6 Safe gas by avoiding large loops
Severity: INFORMATIONAL
Status: Acknowledged
File(s) affected: Exchange.sol

Attack / Description	Code Snippet	Result/Recommendation

Too large loops can cause high gas cost for end-user.	Line 647 - 660:
	`for (uint256 i = 0; i < _buyNowTokenDeals[tokenId].length; i++) {`
	`if (_buyNowTokenDeals[tokenId][i].price == price) {`
	`if (_rightsEqual(_buyNowTokenDeals[tokenId][i].rights, sellRights)) {`
	`if (i == _buyNowTokenDeals[tokenId].length - 1) {`
	`_buyNowTokenDeals[tokenId].pop();`
	`} else {`
	`for (uint256 j = i; j < _buyNowTokenDeals[tokenId].length - 1; j++) {`
	`_buyNowTokenDeals[tokenId][j] = _buyNowTokenDeals[tokenId][j + 1];`
	`}`
	`_buyNowTokenDeals[tokenId].pop();`
	`}`
	`}`
	…
		safe gas by avoiding large loops. Put last index to the removed index place instead of shifting all one position.

5.2. 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	✅
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	✅

ID	Title	Relationships	Test Result
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	✅
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')	✅

ID	Title	Relationships	Test Result
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	✅
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	✅

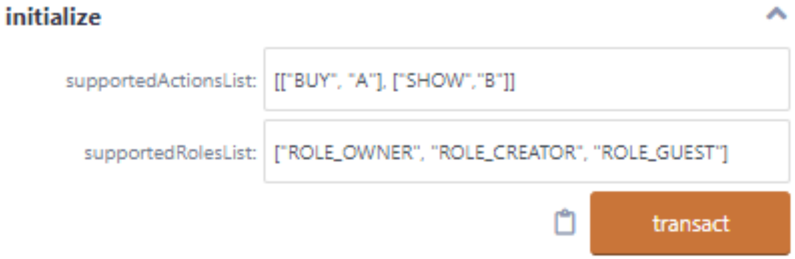
ID	Title	Relationships	Test Result
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	X
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	☑
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	☑
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	☑

5.3. Verifying claims

Deploy artwork contract
Tx: https://kovan.etherscan.io/tx/0xdc0f2304452819bb7c868bc0e29e1d226a3d9710519ba926d27eb227f4f2ba21
Contract: https://kovan.etherscan.io/address/0x3be2b0b8f97f9b6015d1cf73889f9e3b4b09f6e1

Deploy exchange contract
Tx: https://kovan.etherscan.io/tx/0x24a7e9a590c325b234dc9b232b42307ff11920daf82b95717e4c0187d1ba68c6
Contract: https://kovan.etherscan.io/address/0x4e522dC0925e70CE658C2e81B2bE5522a348a41e

Initialize deployed Artwork token



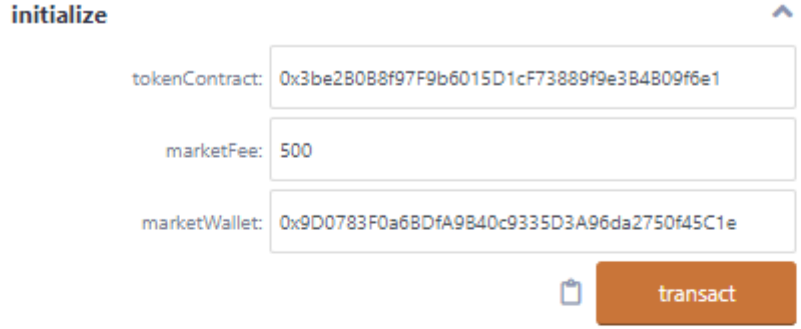
initialize

supportedActionsList: [["BUY", "A"], ["SHOW","B"]]

supportedRolesList: ["ROLE_OWNER", "ROLE_CREATOR", "ROLE_GUEST"]

Tx: https://kovan.etherscan.io/tx/0x84ff7ce8b0abe6fd3e91698adf71f614291d165e1dcc75c9ce0bf937f85268ff

Initialize deployed exchange contract
(with market fee of 5 %)



initialize

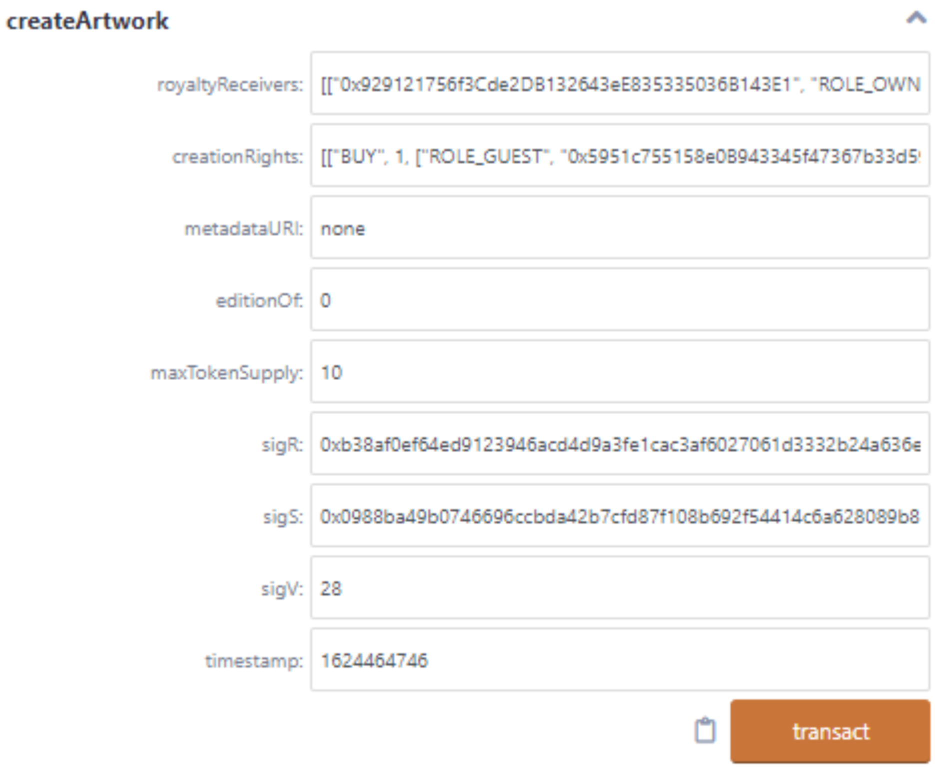
tokenContract: 0x3be2B0B8f97F9b6015D1cF73889f9e3B4B09f6e1

marketFee: 500

marketWallet: 0x9D0783F0a6BDfA9B40c9335D3A96da2750f45C1e

Tx: https://kovan.etherscan.io/tx/0x5ca26101258d3cf39f17a83de7bd164f6f835614171a5a01726deea6556ef0b7

Cannot call functions with unvalid signature



createArtwork

royaltyReceivers: [["0x929121756f3Cde2DB132643eE835335036B143E1", "ROLE_OWN

creationRights: [["BUY", 1, ["ROLE_GUEST", "0x5951c755158e0B943345f47367b33d5

metadataURI: none

editionOf: 0

maxTokenSupply: 10

sigR: 0xb38af0ef64ed9123946acd4d9a3fe1cac3af6027061d3332b24a636e

sigS: 0x0988ba49b0746696ccbda42b7cfd87f108b692f54414c6a628089b8

sigV: 28

timestamp: 1624464746

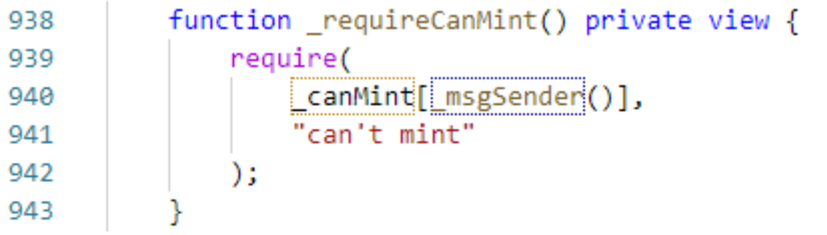
Tx: https://kovan.etherscan.io/tx/0x15c3374868e721e1b87d2d0fd91b6caaf67a54a50b9e4fba9094402acaa3fced

5.3.1 Artwork smart contract (Artwork.sol/IERC755.sol)

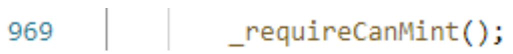
5.3.1.1 Token minting

5.3.1.1.1 only person with minting granted permission can mint a token

Minting a token can only be done by calling *createArtwork* function. This function checks if the caller is allowed to mint tokens (line 969). This allowance can be set and remove by calling *addMinter* or *removeMinter* function. This guarantees only added minters can mint a token.



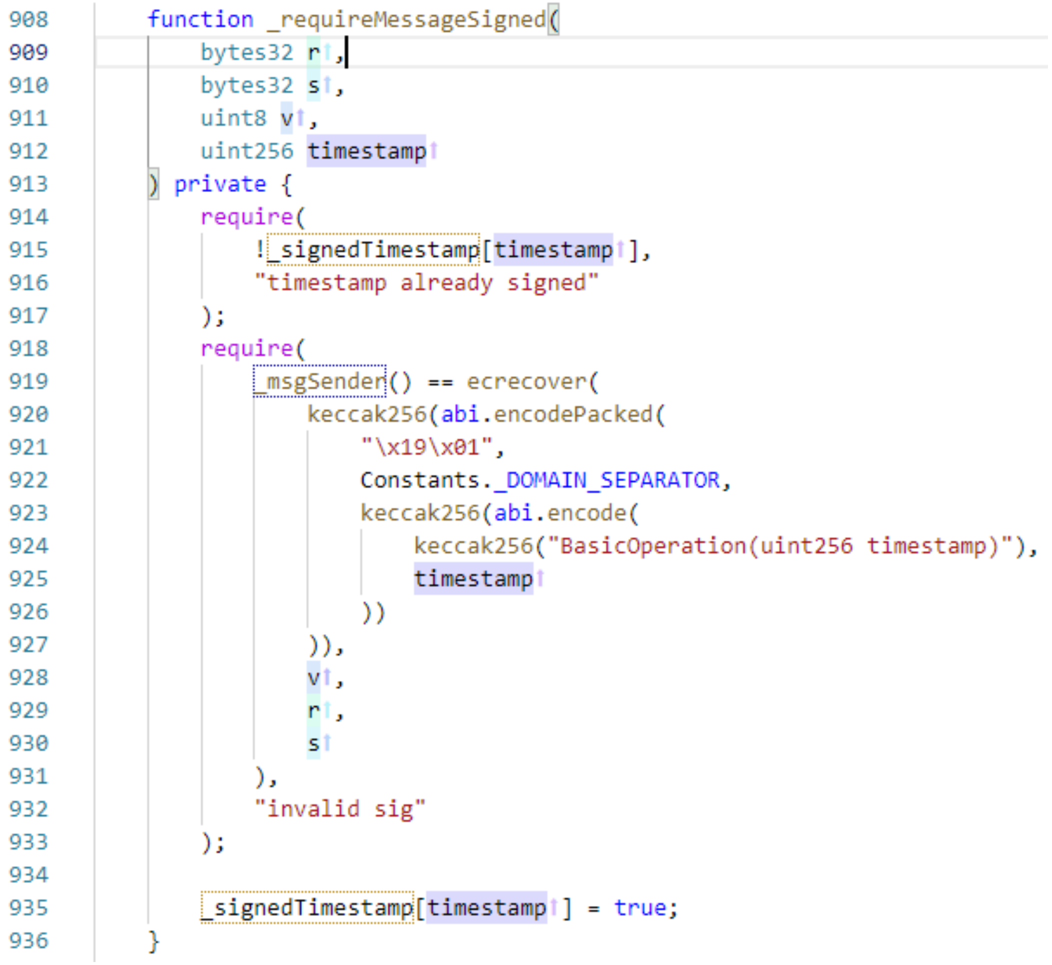
```
938    function _requireCanMint() private view {
939        require(
940            _canMint[_msgSender()],
941            "can't mint"
942        );
943    }
```



```
969            _requireCanMint();
```

5.3.1.1.2 caller signature is verified

The signature of the caller is verified in several functions. These functions are *updateSupportedActions* (line 839), *updateSupportedRoles* (line 870), *createArtwork* (line 968), *addMinter* (line 1201) and *removeMinter* (line 1213).
For verification are the parameters of the elliptic curve and a timestamp used, which are passed to the functions by calling them.

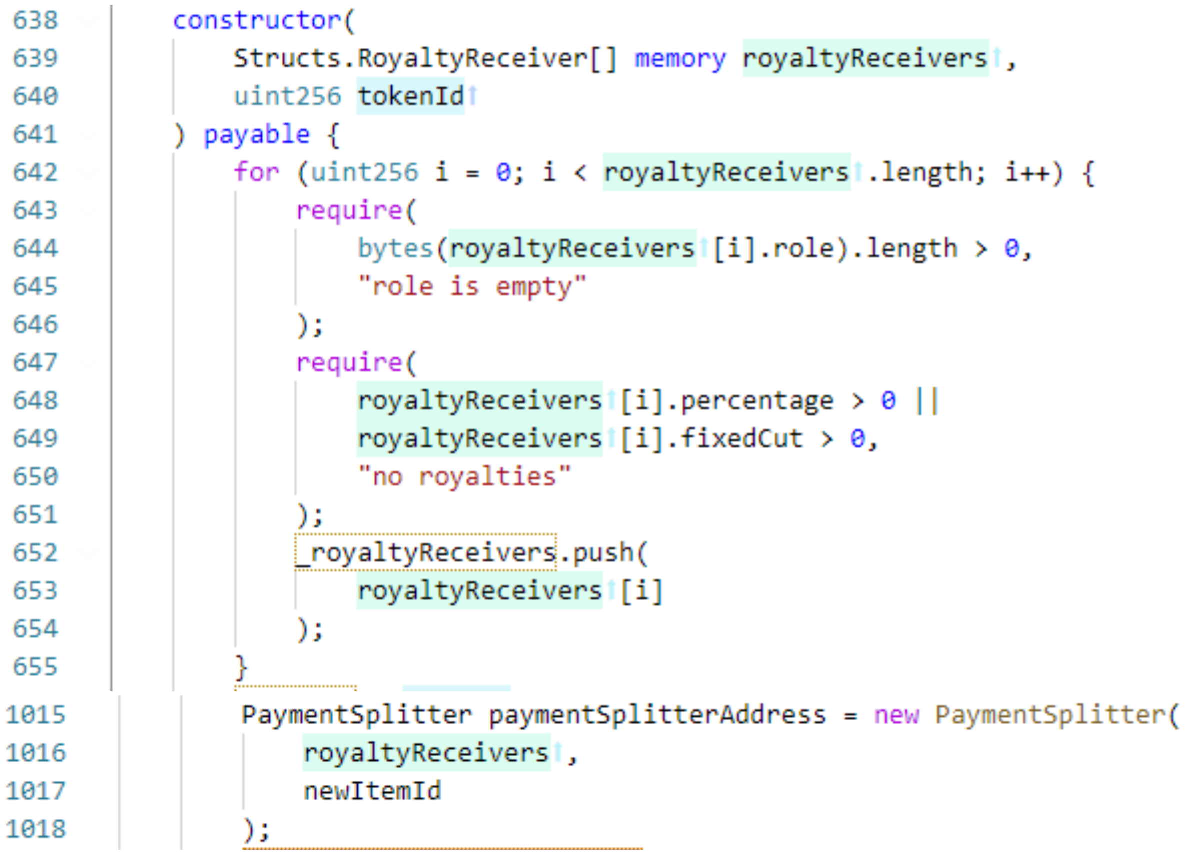


```
    function _requireMessageSigned(
        bytes32 r,
        bytes32 s,
        uint8 v,
        uint256 timestamp
    ) private {
        require(
            !_signedTimestamp[timestamp],
            "timestamp already signed"
        );
        require(
            _msgSender() == ecrecover(
                keccak256(abi.encodePacked(
                    "\x19\x01",
                    Constants._DOMAIN_SEPARATOR,
                    keccak256(abi.encode(
                        keccak256("BasicOperation(uint256 timestamp)"),
                        timestamp
                    ))
                )),
                v,
                r,
                s
            ),
            "invalid sig"
        );

        _signedTimestamp[timestamp] = true;
    }
```

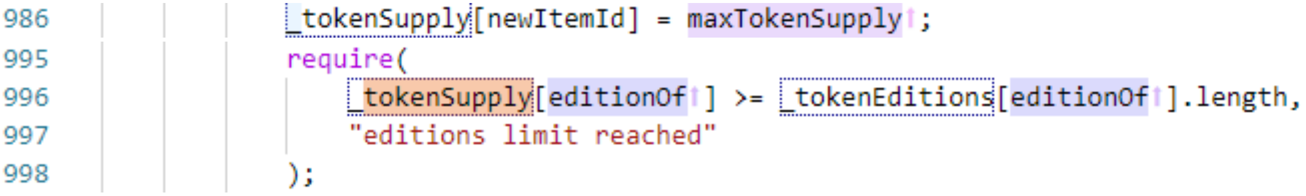
5.3.1.1.3 PaymentSplitter is deployed with royalty receivers config

On calling *createArtwork* the PaymentSplitter is created with passed in royalty receivers config (line 1015 – 1018). The PaymentSplitter checks if the configuration of each royalty receiver is valid and adds them accordingly to the list (line 642 – 655).



```
638      constructor(
639          Structs.RoyaltyReceiver[] memory royaltyReceivers,
640          uint256 tokenId
641      ) payable {
642          for (uint256 i = 0; i < royaltyReceivers.length; i++) {
643              require(
644                  bytes(royaltyReceivers[i].role).length > 0,
645                  "role is empty"
646              );
647              require(
648                  royaltyReceivers[i].percentage > 0 ||
649                  royaltyReceivers[i].fixedCut > 0,
650                  "no royalties"
651              );
652              _royaltyReceivers.push(
653                  royaltyReceivers[i]
654              );
655          }
1015         PaymentSplitter paymentSplitterAddress = new PaymentSplitter(
1016             royaltyReceivers,
1017             newItemId
1018         );
```

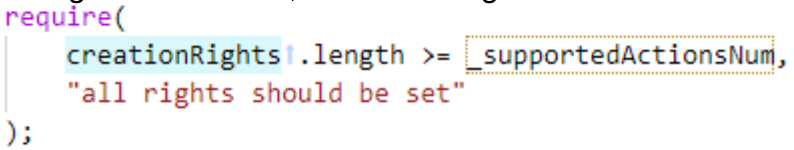
5.3.1.1.4 No more tokens than maxTokenSupply could be minted



```
_tokenSupply[newItemId] = maxTokenSupply;
require(
    _tokenSupply[editionOf] >= _tokenEditions[editionOf].length,
    "editions limit reached"
);
```

5.3.1.1.5 token can’t be minted without token rights

By calling *createArtwork* function it is checked if the creation rights are set with the same amount of rights the token is initialized with. If the rights are not set, the function gets reverted and tokens cannot be minted.

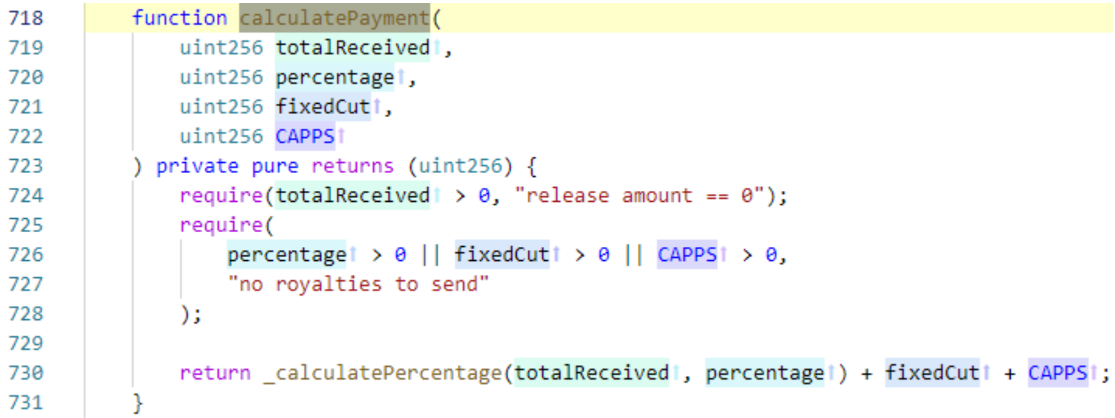


```
require(
    creationRights.length >= _supportedActionsNum,
    "all rights should be set"
);
```

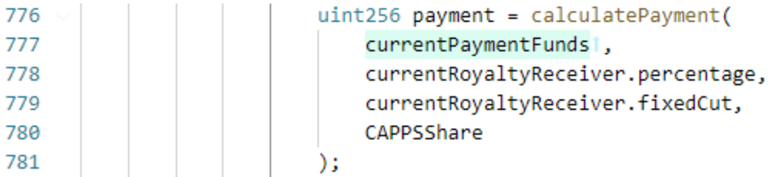
5.3.1.2 Token rights transferring

5.3.1.2.1 transfer payment is correctly split between royalty receivers and seller

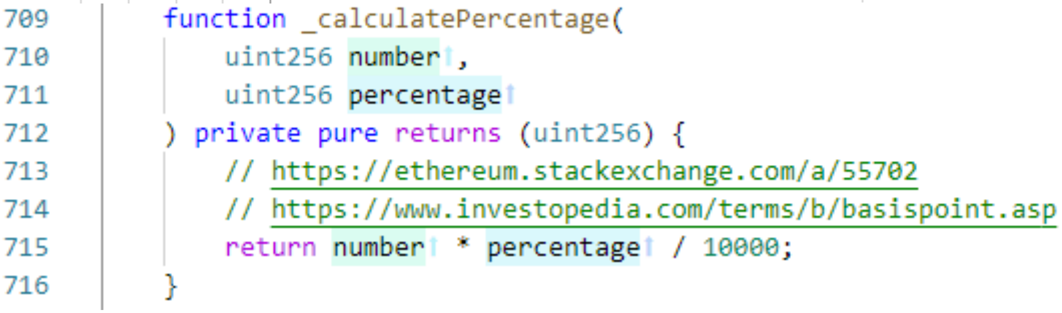
In the *releasePayment* function the released payment amount for each royalty receiver is calculated by calling the *calculatePayment* function (line 776 -781). This function calculates the total receiving amount for each royalty receiver by calling the _*calculatePercentage* function with the correct percentage value for each receiver (line 730). The percentages are calculated safely (line 715). In this way all the payments for the different royalties are calculated correctly.



```
718    function calculatePayment(
719        uint256 totalReceived,
720        uint256 percentage,
721        uint256 fixedCut,
722        uint256 CAPPS
723    ) private pure returns (uint256) {
724        require(totalReceived > 0, "release amount == 0");
725        require(
726            percentage > 0 || fixedCut > 0 || CAPPS > 0,
727            "no royalties to send"
728        );
729
730        return _calculatePercentage(totalReceived, percentage) + fixedCut + CAPPS;
731    }
```



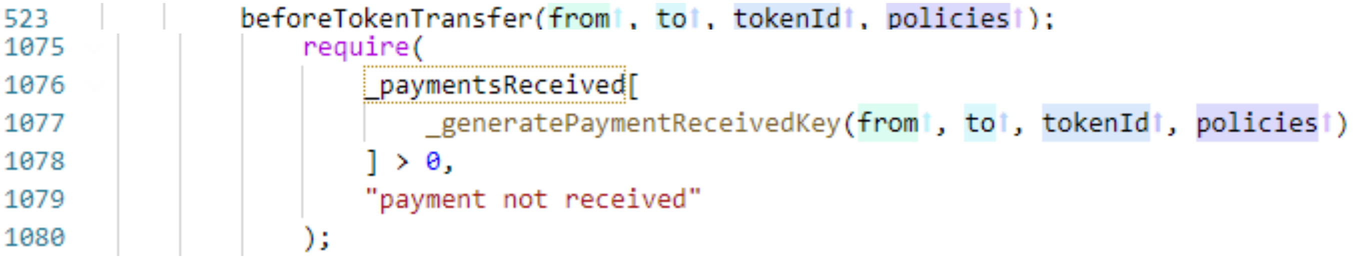
```
776                    uint256 payment = calculatePayment(
777                        currentPaymentFunds,
778                        currentRoyaltyReceiver.percentage,
779                        currentRoyaltyReceiver.fixedCut,
780                        CAPPSShare
781                    );
```



```
709    function _calculatePercentage(
710        uint256 number,
711        uint256 percentage
712    ) private pure returns (uint256) {
713        // https://ethereum.stackexchange.com/a/55702
714        // https://www.investopedia.com/terms/b/basispoint.asp
715        return number * percentage / 10000;
716    }
```

5.3.1.2.2 token rights could not be transferred without a received payment

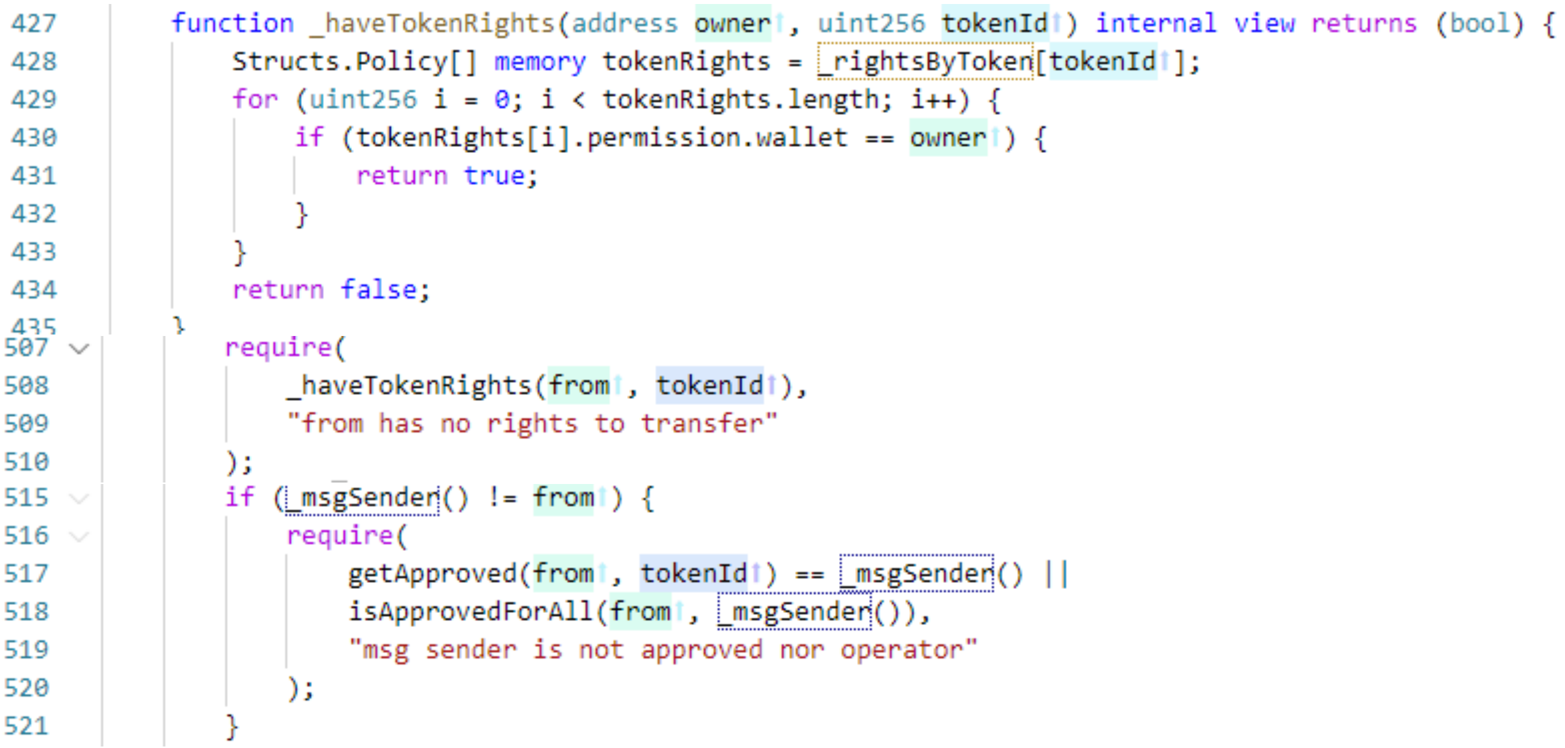
By calling the safeTransferFrom function the token rights are transferred (line 538- 541). Before the rights can be transferred it is checked if the user received a payment by calling _beforeTokenTransfer function (line 523 & line 1075 - 1080). This ensures the seller received the payment before transferring token rights.



```
523           beforeTokenTransfer(from, to, tokenId, policies);
1075              require(
1076                  _paymentsReceived[
1077                      _generatePaymentReceivedKey(from, to, tokenId, policies)
1078                  ] > 0,
1079                  "payment not received"
1080              );
```

5.3.1.2.3 user can’t transfer not owned rights (except rights that he is approved for or an operator for)

In the *safeTransferFrom* function of the token is checked if the owner has the rights to transfer the token (line 507 – 510). Therefore it is checked if the wallet address of the permissions for the *tokenRights* is the address of the sender(line 427 – 435). If a caller is not the sender, it is checked if he is approved for the transfer (line 515 – 521). Otherwise the function call gets reverted.



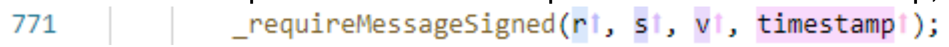
```
427     function _haveTokenRights(address owner, uint256 tokenId) internal view returns (bool) {
428         Structs.Policy[] memory tokenRights = _rightsByToken[tokenId];
429         for (uint256 i = 0; i < tokenRights.length; i++) {
430             if (tokenRights[i].permission.wallet == owner) {
431                 return true;
432             }
433         }
434         return false;
435     }
507         require(
508             _haveTokenRights(from, tokenId),
509             "from has no rights to transfer"
510         );
515         if (_msgSender() != from) {
516             require(
517                 getApproved(from, tokenId) == _msgSender() ||
518                 isApprovedForAll(from, _msgSender()),
519                 "msg sender is not approved nor operator"
520             );
521         }
```

5.3.2 Exchange smart contract (Exchange.sol)

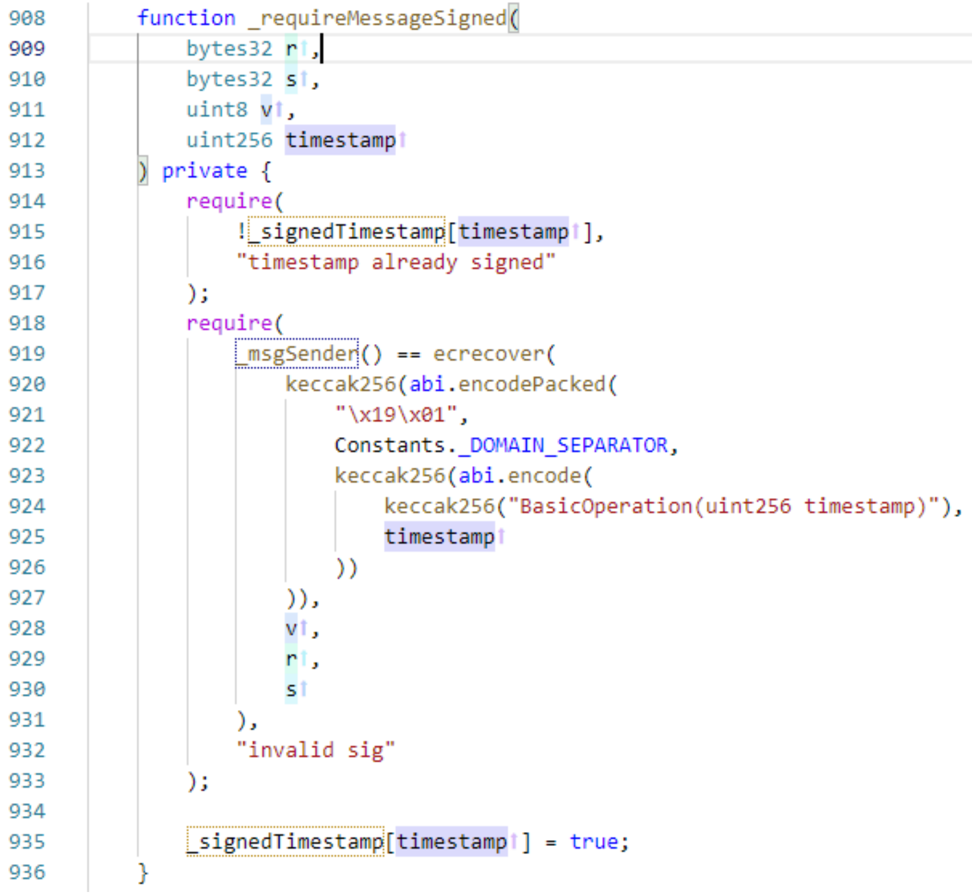
5.3.2.1 Fixed price

5.3.2.1.1 caller signature is verified

The signature of the caller is verified in *buyNow* function (line 771) by calling the *_requireMessageSigned* function. For verification this function uses the parameters of the elliptic curve and a timestamp, which are passed to the *buyNow* function by calling.



```
771            _requireMessageSigned(r, s, v, timestamp);
```

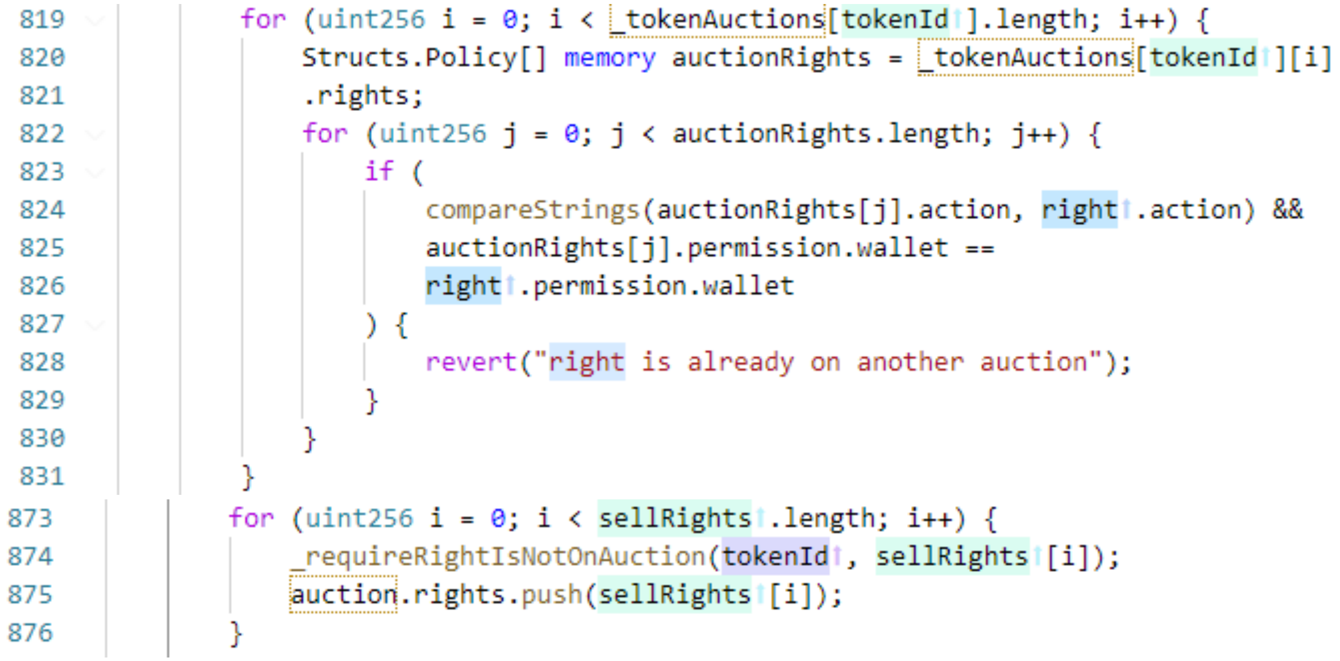


```
    function _requireMessageSigned(
        bytes32 r,
        bytes32 s,
        uint8 v,
        uint256 timestamp
    ) private {
        require(
            !_signedTimestamp[timestamp],
            "timestamp already signed"
        );
        require(
            _msgSender() == ecrecover(
                keccak256(abi.encodePacked(
                    "\x19\x01",
                    Constants._DOMAIN_SEPARATOR,
                    keccak256(abi.encode(
                        keccak256("BasicOperation(uint256 timestamp)"),
                        timestamp
                    ))
                )),
                v,
                r,
                s
            ),
            "invalid sig"
        );

        _signedTimestamp[timestamp] = true;
    }
```

5.3.2.1.2 user can’t list same right twice

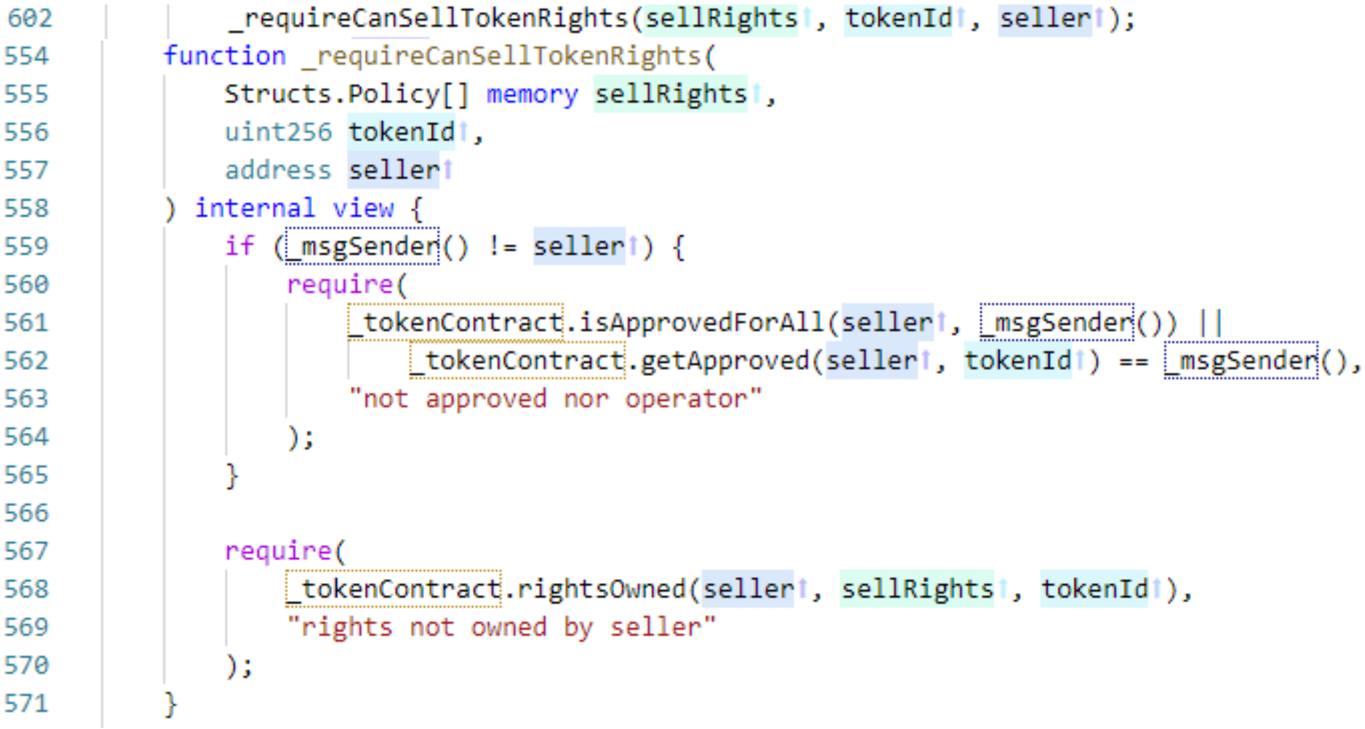
In the *startAuction* function for every selling right it is checked if the right is already on an auction by calling *_requireRightIsNotOnAuction* function (line 874). This function checks all running auctions, is the right is already listed (line 819 - 831). If that is the case, the function gets reverted (line 828). In this way a user cannot list the same right multiple times.



```
819     for (uint256 i = 0; i < _tokenAuctions[tokenId].length; i++) {
820         Structs.Policy[] memory auctionRights = _tokenAuctions[tokenId][i]
821         .rights;
822         for (uint256 j = 0; j < auctionRights.length; j++) {
823             if (
824                 compareStrings(auctionRights[j].action, right.action) &&
825                 auctionRights[j].permission.wallet ==
826                 right.permission.wallet
827             ) {
828                 revert("right is already on another auction");
829             }
830         }
831     }
873     for (uint256 i = 0; i < sellRights.length; i++) {
874         _requireRightIsNotOnAuction(tokenId, sellRights[i]);
875         auction.rights.push(sellRights[i]);
876     }
```

5.3.2.1.3 user can’t list not owned rights (except rights that he is approved for or an operator for)

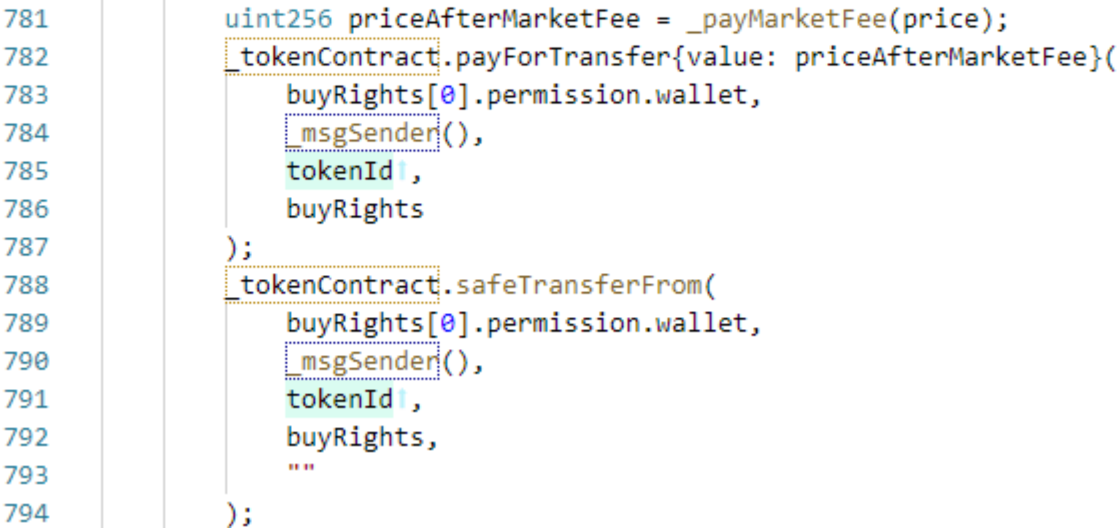
In the *setBuyNowPrice* function is checked if the seller has the right to sell the token by calling *_requireCanSellTokenRights* (line 602). This function checks if the seller is owner of the rights or if he is approved for the rights (line 559 – 570). If he has not the right, the token will not be listed.



```
602             _requireCanSellTokenRights(sellRights, tokenId, seller);
554         function _requireCanSellTokenRights(
555             Structs.Policy[] memory sellRights,
556             uint256 tokenId,
557             address seller
558         ) internal view {
559             if (_msgSender() != seller) {
560                 require(
561                     _tokenContract.isApprovedForAll(seller, _msgSender()) ||
562                         _tokenContract.getApproved(seller, tokenId) == _msgSender(),
563                     "not approved nor operator"
564                 );
565             }
566
567             require(
568                 _tokenContract.rightsOwned(seller, sellRights, tokenId),
569                 "rights not owned by seller"
570             );
571         }
```

5.3.2.1.4 auction with the same rights is cancelled on purchase
Cannot find something to prove in the code. Auctions for an already listed right cannot be made.

5.3.2.1.5 token transfer is done on purchase
In the *buyNow* function are token transferred after paying the market fee by calling *_payMarketFee* function (line 781) and paying for the transfer by calling *payForTransfer* function (line 782). The tokens are only transferred, if the payments for market fee and for transfer are successful (line 788).

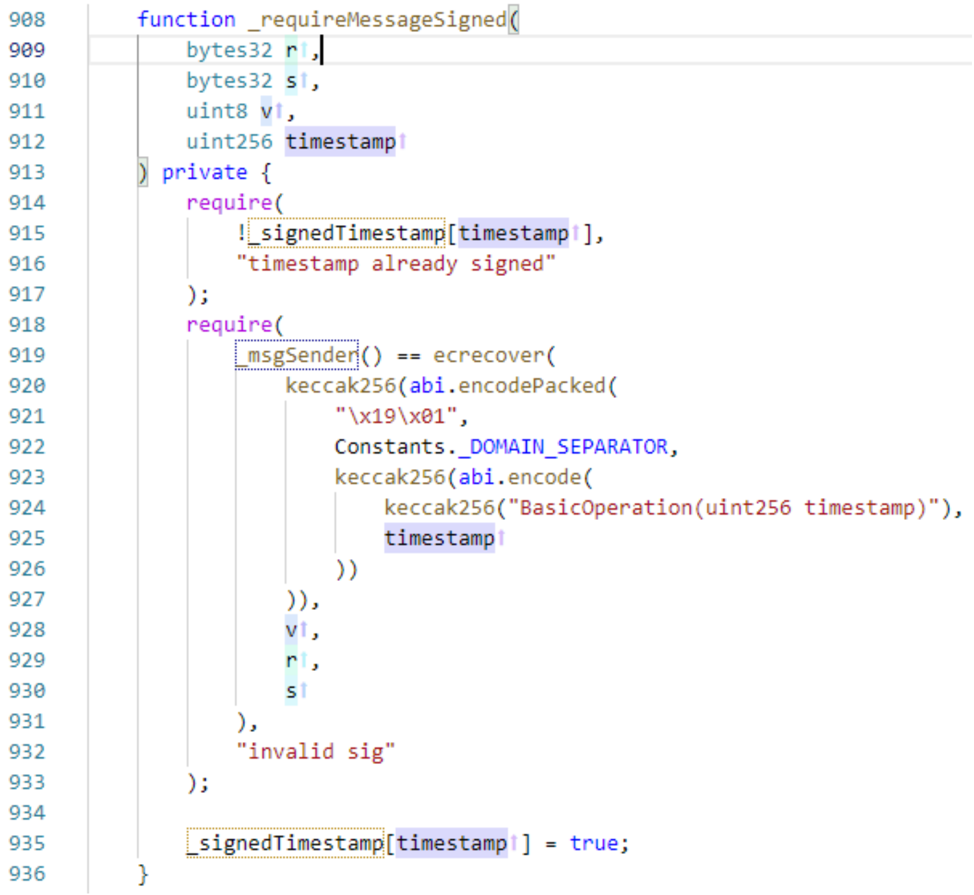


```
781         uint256 priceAfterMarketFee = _payMarketFee(price);
782         _tokenContract.payForTransfer{value: priceAfterMarketFee}(
783             buyRights[0].permission.wallet,
784             _msgSender(),
785             tokenId,
786             buyRights
787         );
788         _tokenContract.safeTransferFrom(
789             buyRights[0].permission.wallet,
790             _msgSender(),
791             tokenId,
792             buyRights,
793             ""
794         );
```

5.3.2.2 Auction

5.3.2.2.1 caller signature is verified

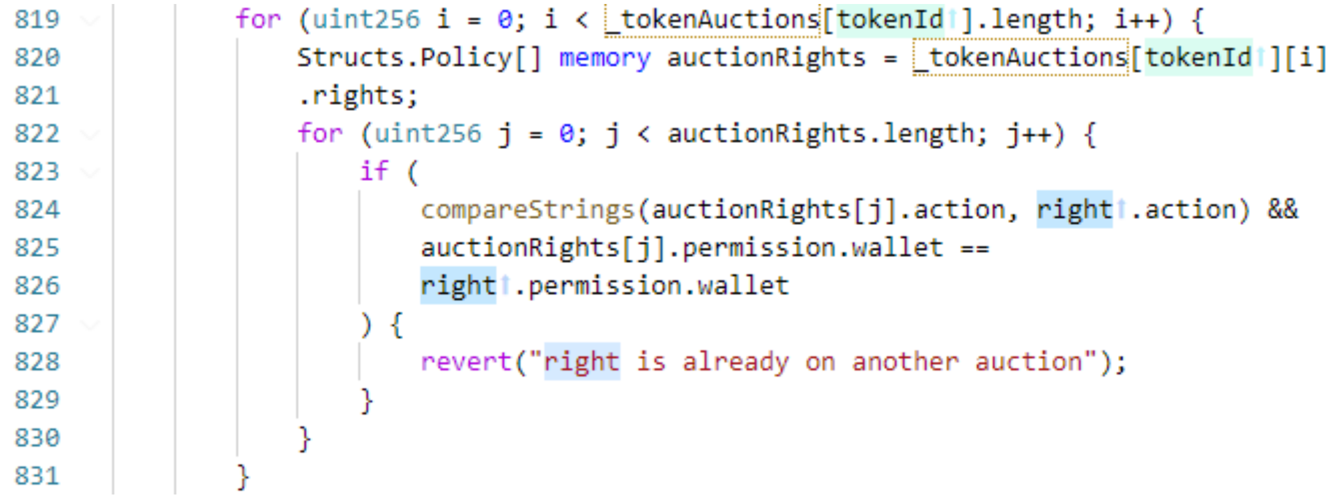
The signature of the caller is verified in *startAuction* (line 853), *cancelAuction* (line 908) and *bid* (line 954) function by calling the *_requireMessageSigned* function. For verification this function uses the parameters of the elliptic curve and a timestamp, which are passed to the functions by calling.



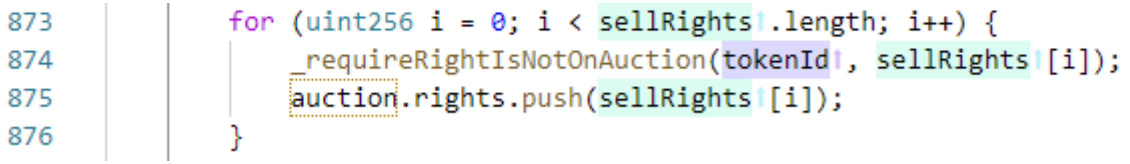
```
908      function _requireMessageSigned(
909          bytes32 r,
910          bytes32 s,
911          uint8 v,
912          uint256 timestamp
913      ) private {
914          require(
915              !_signedTimestamp[timestamp],
916              "timestamp already signed"
917          );
918          require(
919              _msgSender() == ecrecover(
920                  keccak256(abi.encodePacked(
921                      "\x19\x01",
922                      Constants._DOMAIN_SEPARATOR,
923                      keccak256(abi.encode(
924                          keccak256("BasicOperation(uint256 timestamp)"),
925                          timestamp
926                      ))
927                  )),
928                  v,
929                  r,
930                  s
931              ),
932              "invalid sig"
933          );
934
935          _signedTimestamp[timestamp] = true;
936      }
```

5.3.2.2.2 user can’t auction same right twice

In the *startAuction* function for every selling right it is checked if the right is already on an auction by calling *_requireRightIsNotOnAuction* function (line 874). This function checks all running auctions, is the right is already listed (line 819 - 831). If that is the case, the function gets reverted (line 828). In this way a user cannot list the same right multiple times.



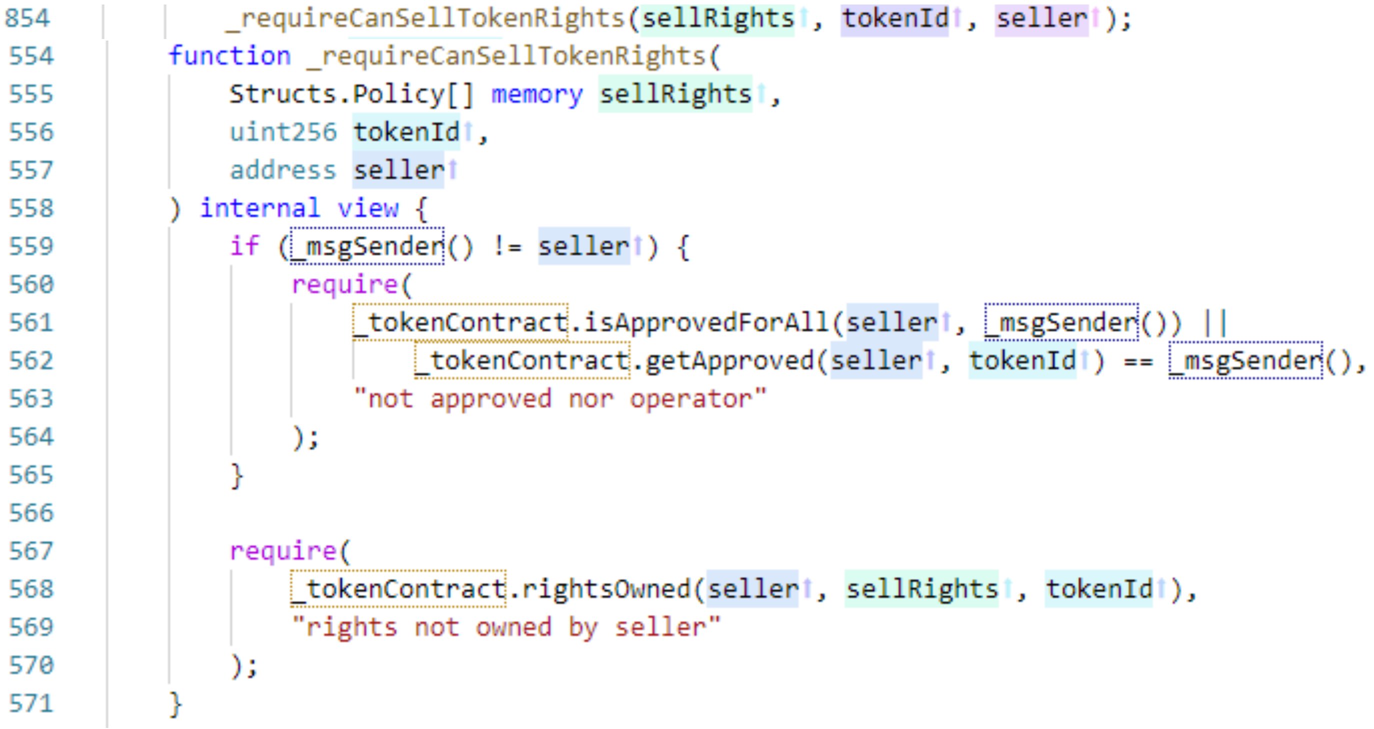
```
819     for (uint256 i = 0; i < _tokenAuctions[tokenId].length; i++) {
820         Structs.Policy[] memory auctionRights = _tokenAuctions[tokenId][i]
821         .rights;
822         for (uint256 j = 0; j < auctionRights.length; j++) {
823             if (
824                 compareStrings(auctionRights[j].action, right.action) &&
825                 auctionRights[j].permission.wallet ==
826                 right.permission.wallet
827             ) {
828                 revert("right is already on another auction");
829             }
830         }
831     }
```



```
873     for (uint256 i = 0; i < sellRights.length; i++) {
874         _requireRightIsNotOnAuction(tokenId, sellRights[i]);
875         auction.rights.push(sellRights[i]);
876     }
```

5.3.2.2.3 user can’t list not owned rights (except rights that he is approved for or an operator for)

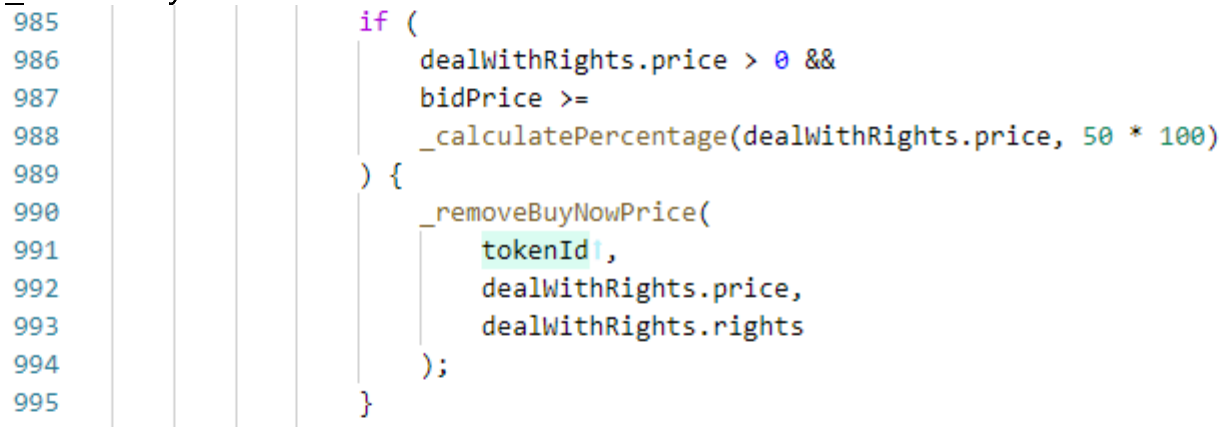
In the *startAuction* function is checked if the seller has the right to sell the token by calling *_requireCanSellTokenRights* (line 854). This function checks if the seller is owner of the rights or if he is approved for the rights (line 559 – 570). If he has not the right, the token will not be listed.



```
854          _requireCanSellTokenRights(sellRights, tokenId, seller);
554      function _requireCanSellTokenRights(
555          Structs.Policy[] memory sellRights,
556          uint256 tokenId,
557          address seller
558      ) internal view {
559          if (_msgSender() != seller) {
560              require(
561                  _tokenContract.isApprovedForAll(seller, _msgSender()) ||
562                      _tokenContract.getApproved(seller, tokenId) == _msgSender(),
563                  "not approved nor operator"
564              );
565          }
566
567          require(
568              _tokenContract.rightsOwned(seller, sellRights, tokenId),
569              "rights not owned by seller"
570          );
571      }
```

5.3.2.2.4 fixed price is removed when bid is >= 50% of the fixed price

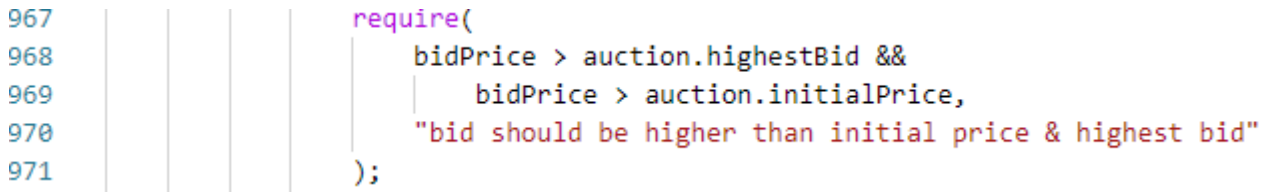
In the *bid* function the fixed price is removed if it is set and the bid is higher than 50 percent of the fixed price by calling *_removeBuyNowPrice* function.



```
if (
    dealWithRights.price > 0 &&
    bidPrice >=
    _calculatePercentage(dealWithRights.price, 50 * 100)
) {
    _removeBuyNowPrice(
        tokenId,
        dealWithRights.price,
        dealWithRights.rights
    );
}
```

5.3.2.2.5 user can’t make a bid lower than the initial price and previous bid

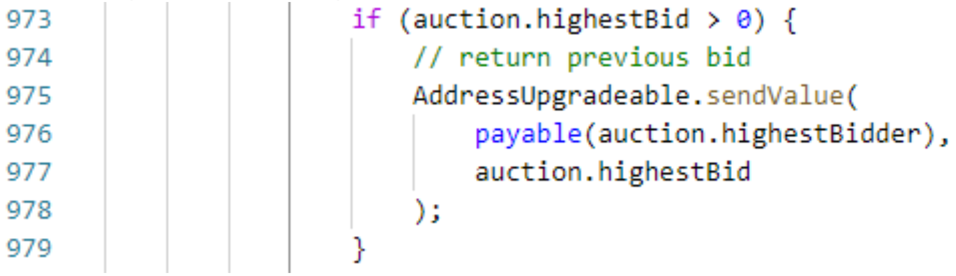
In the *bid* function is checked if the entered bidding price is higher than the initial price and higher than the current highest bid (line 967 – 971).



```
967  require(
968      bidPrice > auction.highestBid &&
969          bidPrice > auction.initialPrice,
970      "bid should be higher than initial price & highest bid"
971  );
```

5.3.2.2.6 previous bidder funds are released on a new bid

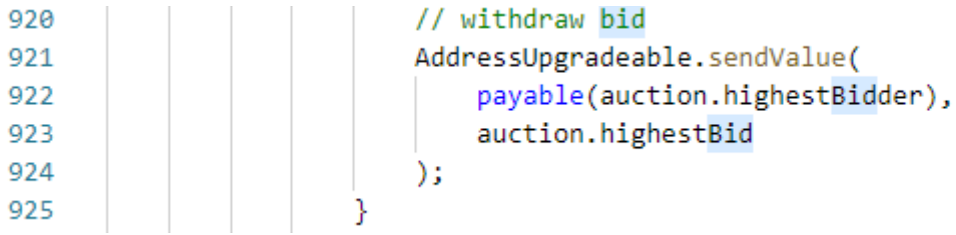
If the entered bid is higher than the previous one and there is a previous bid, the previous bidder is refunded by calling *sendValue* function (line 973 – 979).



```
973                 if (auction.highestBid > 0) {
974                     // return previous bid
975                     AddressUpgradeable.sendValue(
976                         payable(auction.highestBidder),
977                         auction.highestBid
978                     );
979                 }
```

5.3.2.2.7 bidder funds are released on auction cancel

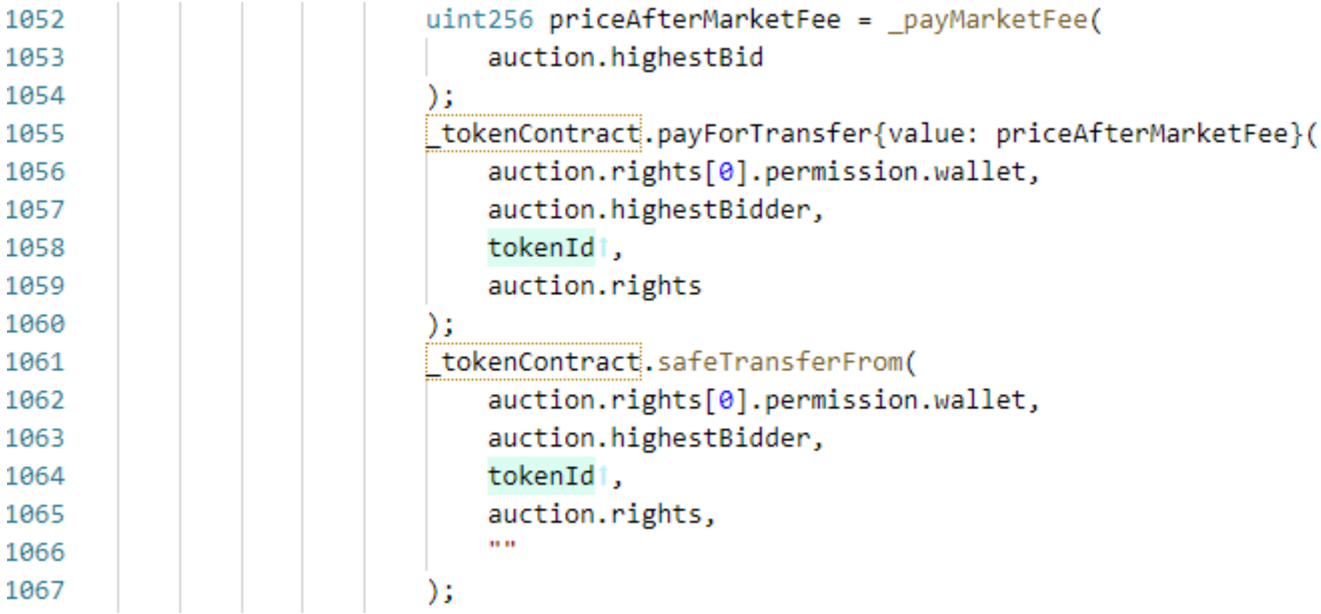
In the *_cancelAuction* function are funds sent back to the currently highest bidder by calling *sendValue* function (line 921 – 924).



```
920                     // withdraw bid
921                     AddressUpgradeable.sendValue(
922                         payable(auction.highestBidder),
923                         auction.highestBid
924                     );
925                 }
```

5.3.2.2.8 token transfer is done on auction end if there is a winner bid

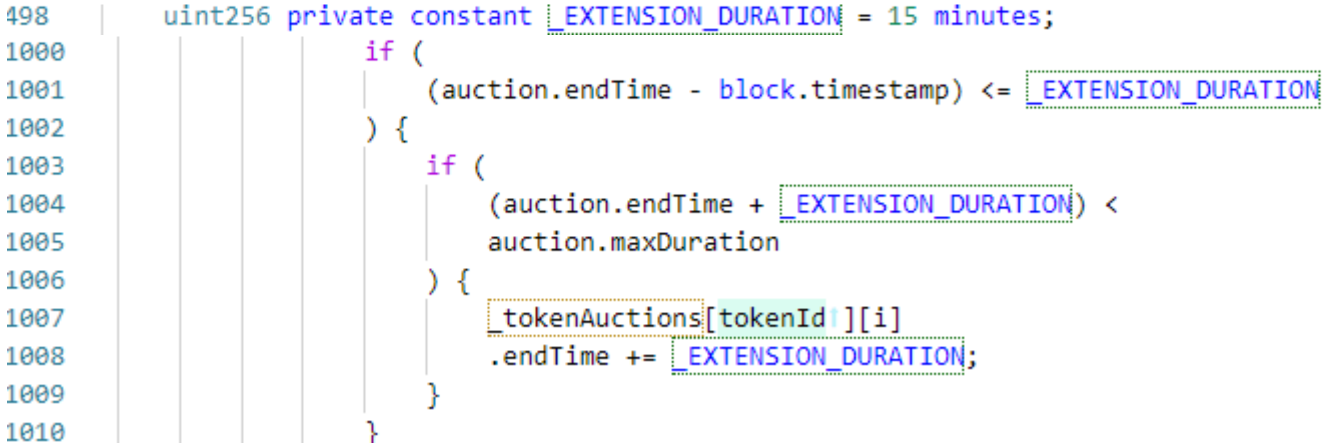
If the auction is ended, everyone can call *endAuction* function to end an auction. If there is a bid greater than 0 the token is transferred to the winning bid address after sending market fee and paying for transfer (line 1040 – 1067).



```
1052    uint256 priceAfterMarketFee = _payMarketFee(
1053        auction.highestBid
1054    );
1055    _tokenContract.payForTransfer{value: priceAfterMarketFee}(
1056        auction.rights[0].permission.wallet,
1057        auction.highestBidder,
1058        tokenId,
1059        auction.rights
1060    );
1061    _tokenContract.safeTransferFrom(
1062        auction.rights[0].permission.wallet,
1063        auction.highestBidder,
1064        tokenId,
1065        auction.rights,
1066        ""
1067    );
```

5.3.2.2.9 auction end time is extended by 15 minutes on a bid when <= 15 minutes left

The auction end time is extended by 15 minutes if the bid is made less than 15 minutes before auction end time (line 1000 – 1010). The auction end time can only be extended to given max duration of auction (line 1003 - 1005.



```
498       uint256 private constant _EXTENSION_DURATION = 15 minutes;
1000                        if (
1001                            (auction.endTime - block.timestamp) <= _EXTENSION_DURATION
1002                        ) {
1003                            if (
1004                                (auction.endTime + _EXTENSION_DURATION) <
1005                                auction.maxDuration
1006                            ) {
1007                                _tokenAuctions[tokenId ][i]
1008                                .endTime += _EXTENSION_DURATION;
1009                            }
1010                        }
```

6. 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 of the smart contract and the functions. During the audit, no critical issues were found, after the manual and automated security testing. Only informational and low issues were found, to increase the code quality. Please make sure to add more in-line documentation within the codebase, to make the functions way easier to understand. Overall, everything worked as it was supposed to be, we have been satisfied with the code quality and security measures, that has been taken.

7. Deployed Smart Contract

VERIFIED

Artwork:
proxy - https://etherscan.io/address/0xcB1E67a4ce9AB2aE1b16C9FDFdd30D34Db25672c#code
impl - https://etherscan.io/address/0xE95BC8ebb552C43F48a7271bB1963250571ffa0b#code

Exchange:
proxy - https://etherscan.io/address/0x41cF8cfA6889886Ed6A5F67c1322Ecb4D7ef5070#code
impl - https://etherscan.io/address/0x7C2F63ad74E4D6E77c3aB13726A1B0ae1ce9B304#code