SAMPLE SECURITY ASSESSMENT CHECKLIST

for Smart Contract based Applications

Until recent times, Cryptocurrency was the only prominent use case of Blockchain. Emergence of Decentralized financing solutions such as lending or crowdfunding applications, Digital Identity solutions, NFT based art and collectibles, supply chain applications, Anti-piracy solutions, real estate software, and Gaming Applications in the market has made it apparent that Blockchain is more than just Cryptocurrency. Several elements in the Blockchain ecosystem need attention from a security perspective which essentially includes the following:

- Layer 1 - Covers Blockchain protocols like Bitcoin Core or Geth.
- Layer 2 - Facilitates efficient transactions using micropayment channels.
- Smart Contracts – Complies with the general contractual conditions of Blockchain-based systems, such as charges, payment terms, enforcement, and confidentiality. Smart contracts are used heavily for developing decentralized applications.
- Custodial or Non-custodial software wallets
- Hardware wallets for storing private keys
- Mining Software
- Centralized and Decentralized Exchanges

As the Blockchain-based solutions are immutable, the elements deployed with smart contracts remains permanent with no option to reverse the action unless chosen to destroy the smart contract or create a proxy contract with a pseudo upgrade. Additionally, as the exploitation of security vulnerabilities in smart contracts-based applications often results in significant financial impact, tracing (after an attack) becomes a tedious task and challenges the patching or workarounds.

To help our enterprise customers effectively assess the inherent and associated security risks related to the development, adoption, and deployment of Blockchain-based applications and services, NST Cyber Consultants have rolled-out a curated sample list of audit checklists that can be used as a base reference standard. These audit questions are meticulously prepared based on recommendations from standards like NIST, Smart Chain Verification Standard (SCVS). NST Cyber also includes it rich industry experience, learnings from real-world attacks against Blockchain-based systems, and general secure coding practices into the checklist. Some of the security controls listed may not be directly applicable to all use cases as validation methods tend to vary based on application type and design. However most of controls can be seamlessly applied to Layer 1 Blockchain protocols, Layer 2 channels, and Smart contract-based applications or services (DeFi).

Sr.No	Category
	Blockchain Deployment Infrastructure Security
1	Ensure Blockchain solutions are separated by channels and namespaces are in place, to ensure private communication between the members of a ledger and easy management of assets
2	Ensure Endorsement policies are in place and adequately enforced, to aid security of digital assets, Business associations, and contracts.
3	Ensure adequate Access Control mechanisms are in place. Access control should follow the, 'Principle of least privilege', and, 'Need to know Principle' to ensure only the required and rightful access is provided to any given authenticated user to a service or data.
4	Ensure a standardised authorization mechanisms like Oauth, OIDC, SAML2, etc is used, in case an organization is working as an Identity Supplier. Using custom authentication, verification, and authorization mechanisms will lead to Access Control Bypass due to improper implementation or code induced vulnerabilities.
5	Ensure HSMs are used to secure and manage Cryptographic keysand avoid leakage of cryptographic keys.
6	Ensure Privilege Access Mechanisms are in place to monitor Privilege Access to the servers/services to prevent inadvertent privileged access to an attacker or a normal user.
7	Ensure APIs are secured from attacks mentioned in OWASP API Top 10 in addition to contextual vulnerabilities, lack of incorporating security controls (Authentication, Identfication ,and Authorization) will lead to unauthorized access/transaction.
8	Ensure Encryption of data at rest and data in transit of sensitive data such as keys, tokens, certificates , to avoid compromise of sensitive information in case of a compromise
9	Ensure Data Classification Controls are in place to adequately categorize data and information in order to apply appropriate controls.
10	Ensure Privacy Preserving mechanisms such as Permissioned Ledgers are used, to protect/hide the participating members' sensitive information related to transactions.
11	Ensure periodic Vulnerability Assessment and Penetration testing, and Configuration reviews are conducted against the solution to identify any security gaps and fix it,. before an attacker leverages the same. It is also recommended to incorporate Secure Code Review concepts along with the CI/CD pipeline.
12	Ensure Trusted Platform Modules are used inorder to preserve privacy and chain codes, and untampered execution of sensitive codes.
13	Ensure the entire channel is secured by TLSv1.2 and above end-to-end, to avoid leakage of sensitive data while in transit
14	Ensure Security standards, policies and procedures are in place to ensure consistency and operational efficiency.
15	Ensure a robust mechanism is in place for handling and managing internal and external TLS certificates to avoid certificate leakage and keeping the TLS environment current.

Sr.No	Category
16	Ensure the presence of security controls built in to the application, to address vulnerabilities mentioned in OWASP top 10 in addition to contextual vulnerabilities, that could compromise the system and let an attacker to gain unauthorized access and expose unintended permissions
17	Ensure the infrastructure housing the data and the solution is secured and hardened as per industry security best practices, so as to ensure a consistent operation and prevent attacks and compromises.
18	Ensure a legal framework and control is in place, hugging the organization's context, inorder to prevent any deviation that might land the product/organization in a battle to fight non-conformance to compliance standards/laws/guidelines
Smart Contract Application	
19	Verify that no miner-influenced values like block hashes or timestamps are used as a source of randomness in a smart contract.
20	Validate that no random numbers are stored in the contract until all lottery entries are stored as any number that the contract could generate can potentially be precalculated off-chain before the end of the block.
21	Verify that verifiable delay functions which produce pseudorandom number and take a fixed amount of sequential time to evaluate are in use.
22	Verify that a commit reveal scheme is in use where users must stake wei to participate. A decentralised autonomous organization with predefined participation rules should be used in a contract to generate random numbers.
23	Verify that contracts that perform bulk transactions or updates are not using for loop that can be DoS'd if a call to another contract or transfer fails during the loop.
24	Verify that while loops that exits when the gas drops below a threshold with an iterator stored in a private variable is used to handle iterating over dynamically sized data structure.
25	Validate that a mechanism that favour pull over push is used for external calls.
26	Validate that contracts cannot be forced to receive ether without triggering any code.
27	Verify that proper checks are implemented to handle the increases in balances of contracts.
28	Verify that all bookkeeping state variables before transferring execution to an external contract are updated effectively to prevent chances of reentrancy attacks.
29	Validate the chance of an attacker performing reentrancy attacks using a fallback option after transfer to execute the vulnerable function again before the state variable change (with call. value).
30	Validate that, contracts do not define functions with a different type of signature than the implementation, causing two different method id's to be created which results in the fallback method to be executed when an interface is called.
31	Verify that type signatures are identical between interfaces and implementations.
32	Validate that, arithmetic methods used in a contract does not cause overflow or underflow (add and sub can cause overflow/underflow on any type of integers).
33	Verify that pop methods used in dynamic arrays in a contract are safe as attackers could underflow the length of an array to alter other variables in a contract.

Sr.No	Category
34	Validate that users can only approve transactions when the requires who requires approval is approved for 0 tokens.
35	Verify that the gap in creation of a transaction and the time when it is accepted in the blockchain does not help attackers to put contracts in state that take advantage of it.
36	Verify that ERC20 standard's approve and transfer form functions which are vulnerable to a race condition is used with mechanisms to mitigate the race condition.
37	Verify the risk from unchecked external calls in smart contracts and validate the effectiveness of process-level controls for prevention like manual validation and code level practices (Eg: Use of address.transfer()).
38	Validate that all modifiers on a functions are specific and in use to avoid contracts being modified by attackers.
39	Validate that variables declared within a certain scope (decision block, method, or inner class) are not named same as variables in outer scope.
40	Verify the use of ‘named’ constructor (ensure the existence of unnamed “constructor”) which may act as a runtime bytecode instead of a constructor that an attacker can use for changing the state variables initialized in the function.
Layer 1/Consensus Protocol	
41	Validate that Historical Weighted Difficulty based Proof of work mechanisms are in place to protect the consensus Protocol.
42	Verify that Random Mining Group Selection based mechanisms are in place to protect the consensus Protocol against 51% Attack.
43	Verify that Indegree and Outdegree based mechanisms are in place to protect the consensus Protocol against Eclipse Attack.
44	Verify that Self-Registration based mechanisms are in place to protect the consensus Protocol against Sybil Attack.
45	Validate that Backward-Incompatible Defense mechanisms are in place to protect the consensus Protocol.
46	Verify that Tie Breaking Defense based mechanisms are in place to protect the consensus protocol against Selfish Mining Attack.
47	Verify that Dynamic and Auto Responsive Approach is used to protect the consensus protocol against DDoS Attack
Layer 1/Sharding Protocol	
48	Verify that proper mechanisms to prevent against shrad attacks are in place. A potential shrad attack may be prevented by randomly assigning nodes to certain shards and constantly reassigning them at random intervals. This random sampling would make it difficult for hackers to know when and where to corrupt a shard.
SCVS - Architecture, Design and Threat Modeling	
49	Verify that the every introduced design change is preceded by an earlier threat modelling.
50	Verify that the documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).

Sr.No	Category
51	Verify that the SCSVS, security requirements or policy is available to all developers and testers.
52	Verify that there exists an upgrade process for the contract which allows to deploy the security fixes or it is clearly stated that the contract is not upgradeable.
53	Verify that the events for the (state changing/crucial for business) operations are defined
54	Verify that there is a component that monitors the contract activity using events.
55	Verify that there exists a mechanism that can temporarily stop the sensitive functionalities of the contract in case of a new attack. This mechanism should not block access to the assets (e.g. tokens) for the owners.
56	Verify that there is a policy to track new security bugs and to update the libraries to the latest secure version.
57	Verify that the value of cryptocurrencies kept on contract is controlled and at the minimal acceptable level.
58	Verify that if the fallback function can be called by anyone it is included in the threat modelling.
59	Verify that the business logic in contracts is consistent. Important changes in the logic should be allowed for all or none of the contracts.
60	Verify that code analysis tools are in use that can detect potentially malicious code.
61	Verify that the latest version of the major Solidity release is used.
62	Verify that, when using the external implementation of contract, you use the current version which has not been superseded.
63	Verify that there are no vulnerabilities associated with system architecture and design.
SCVS - Access Control	
64	Verify that the principle of least privilege exists, other contracts should only be able to access functions and data for which they possess specific authorization.
65	Verify that new contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permission until access to the new features is explicitly granted.
66	Verify that the creator of the contract complies with the rule of least privilege and their rights strictly follow the documentation.
67	Verify that the contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present (as the client-side access control can be easily bypassed).
68	Verify that there is a centralized mechanism for protecting access to each type of protected resource.
69	Verify that the calls to external contracts are allowed only if necessary.
70	Verify that visibility of all functions is specified.
71	Verify that the initialization functions are marked internal and cannot be executed twice.
72	Verify that the code of modifiers is clear and simple. The logic should not contain external calls to untrusted contracts.

Sr.No	Category
73	Verify that the contract relies on the data provided by right sender and contract does not rely on tx.origin value.
74	Verify that all user and data attributes used by access controls are kept in trusted contract and cannot be manipulated by other contracts unless specifically authorized.
75	Verify that the access controls fail securely including when a revert occurs.
76	Verify that there are no vulnerabilities associated with access control.
SCVS - Blockchain Data	
77	Verify that any data saved in the contracts is not considered safe or private (even private variables).
78	Verify that no confidential data is stored in the blockchain (passwords, personal data, token etc.).
79	Verify that the list of sensitive data processed by the smart contract is identified, and that there is an explicit policy for how access to this data must be controlled and enforced under relevant data protection directives.
80	Verify that there is a component that monitors access to sensitive contract data using events.
81	Verify that contract does not use string literals as keys for mappings. Verify that global constants are used instead to prevent Homoglyph attack.
82	Verify that there are no vulnerabilities associated with blockchain data.
SCVS - Communications	
83	Verify that libraries which are not part of the application (but the smart contract relies on to operate) are identified.
84	Verify that contract does not use hard-coded addresses unless necessary. If the hard coded address is used, make sure that its contract has been audited.
85	Verify that contracts and libraries which call external security services have a centralized implementation.
86	Verify that delegatecall is not used with untrusted contracts.
87	Verify that re-entrancy attack is mitigated by blocking recursive calls from other contracts. Do not use call and send function unless it is a must.
88	Verify that the result of low-level function calls (e.g. send, delegatecall, call) from another contracts is checked.
89	Verify that third party contracts do not shadow special functions (e.g. revert).
90	Verify that there are no vulnerabilities associated with communications.
SCVS - Arithmetic	
91	Verify that the values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
92	Verify that the unchecked code snippets from Solidity 0.8.* do not introduce integer under/overflows.
93	Verify that the extreme values (e.g. maximum and minimum values of the variable type) are considered and does change the logic flow of the contract.

Sr.No	Category
94	Verify that non-strict inequality is used for balance equality.
95	Verify that there is a correct order of magnitude in the calculations.
96	Verify that in calculations, multiplication is performed before division for accuracy.
97	Verify that there are no vulnerabilities associated with arithmetics.
SCVS - Malicious input handling	
98	Verify that if the input (function parameters) is validated, the positive validation approach (allowlisting) is used where possible.
99	Verify that the length of the address being passed is determined and validated by smart contract.
100	Verify that there are no vulnerabilities associated with malicious input handling.
SCVS - Gas usage & limitations	
101	Verify that the usage of gas in the smart contract is anticipated, defined and has clear limitations that cannot be exceeded. Both, code structure and malicious input should not cause gas exhaustion.
102	Verify that two types of the addresses are considered when using the send function. Sending Ether to the contract address costs more than sending Ether to the personal address.
103	Verify that the contract does not iterate over unbound loops.
104	Verify that the contract does not check whether the address is a contract using *extcodesize* opcode.
105	Verify that the contract does not generate pseudorandom numbers trivially basing on the information from blockchain (e.g. seeding with the block number).
106	Verify that the contract does not assume fixed-point precision but uses a multiplier or store both the numerator and denominator instead.
107	Verify that, if signed transactions are used for relaying, the signatures are created in the same way for every possible flow to prevent replay attacks.
108	Verify that there exists a mechanism that protects the contract from a replay attack in case of a hard-fork.
109	Verify that all library functions that should be upgradeable are not internal.
110	Verify that the *external* keyword is used for functions that can be called externally only to save gas.
111	Verify that there is no hard-coded amount of gas assigned to the function call (the gas prices may vary in the future).
112	Verify that there are no vulnerabilities associated with gas usage & limitations.
SCVS - Business logic	
113	Verify that the contract logic implementation corresponds to the documentation.
114	Verify that the business logic flows of smart contracts proceed in a sequential step order and it is not possible to skip any part of it or to do it in a different order than designed.
115	Verify that the contract has business limits and correctly enforces it.

Sr.No	Category
116	Verify that the business logic of contract does not rely on the values retrieved from untrusted contracts with multiple calls of the same function.
117	Verify that the contract logic does not rely on the balance of contract (e.g. balance == 0).
118	Verify that the sensitive operations of contract do not depend on the block data (i.e. block hash, timestamp).
119	Verify that the contract uses mechanisms that mitigate transaction-ordering dependence (front-running) attacks (e.g. pre-commit scheme).
120	Verify that the contract does not send funds automatically but it lets users withdraw funds on their own in separate transaction instead.
121	Verify that the inherited contracts do not contain identical functions or the order of inheritance is carefully specified.
122	Verify that the business logic does not compare the extcodehash return value with 0 to check whether another address is contract (the hash of empty data is returned in such case).
123	Verify that there are no vulnerabilities associated with business logic.
SCVS - Denial of service	
124	Verify that the self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
125	Verify that the business logic does not block its flows when any of the participants is absent forever.
126	Verify that the contract logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher that the profit).
127	Verify that the expressions of functions assert or require to have a passing variant.
128	Verify that if the fallback function is not callable by anyone, it is not blocking the functionalities of contract and the contract is not vulnerable to Denial of Service attacks.
129	Verify that the function calls to external contracts (e.g. *send*, *call*) are not the arguments of *require* and *assert* functions.
130	Verify that the function declarations are callable by the used compiler version (see the *Uncallable function example* link below).
131	Verify that there are no vulnerabilities associated with availability.
SCVS - Token	
132	Verify that the token contract follows a tested and stable token standard.
133	Use the approve function from the ERC-20 standard to change allowed amount only to 0 or from 0.
134	Verify that the contract does not allow to transfer tokens to zero address.
135	Verify that the re-entracy attack has been considered when using the token contracts with callbacks (e.g. ERC-777).
136	Verify that the transfer business logic is consistent, especially when re-sending tokens to the same address (msg.sender == destination).

Sr.No	Category
137	Verify that there are no vulnerabilities associated with Token.
	SCVS - Code clarity
139	Verify that the logic is clear and modularized in multiple simple contracts and functions.
140	Verify that the inheritance order is considered for contracts that use multiple inheritance and shadow functions.
141	Verify that the contract uses existing and tested code (e.g. token contract or mechanisms like *ownable*) instead of implementing its own.
142	Verify that the same rules for variable naming are followed throughout all the contracts (e.g. use the same variable name for the same object).
143	Verify that variables with similar names are not used.
144	Verify that all variables are defined as storage or memory variable.
145	Verify that all storage variables are initialised.
146	Verify that the constructor keyword is used for Solidity version greater than 0.4.24. For older versions of Solidity make sure the constructor name is the same as contract's name.
147	Verify that the functions which specify a return type return the value.
148	Verify that all functions are used. Unused ones should be removed.
149	Verify that the *require* function is used instead of the *revert* function in *if* statement.
150	Verify that the *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition on the input from users and external contracts.
151	Verify that assembly code is used only if necessary.
152	Verify that there is a description in the form of 1-2 short sentences of what the contract is for at the beginning of the contract.
153	Verify that if the system uses a ready-made implementation of the contract, it has been marked in the comment. If it contains changes from the original, those have been specified.
	SCVS - Test coverage
154	Verify that considered as sensitive functions of verified contract are covered with tests in the development phase.
155	Verify that the implementation of verified contract has been checked for security vulnerabilities using static and dynamic analysis.
156	Verify that the specification of smart contract has been formally verified.
157	Verify that the specification and the result of formal verification is included in the documentation.
	SCVS - Known attacks
158	Verify that the contract is not vulnerable to Integer Overflow and Underflow attacks.
	[5.1] Verify that the values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations.

Sr.No	Category
	[5.2] Verify that the extreme values (e.g. maximum and minimum values of the variable type) are considered and does change the logic flow of the contract.
159	Verify that the contract is not vulnerable to Reentrancy attack.
	[4.5] Verify that the re-entrancy attack is mitigated by blocking recursive calls from the other contracts. Do not use call and send functions unless it is a must.
160	Verify that the contract is not vulnerable to Access Control issues.
	[2.1] Verify that the principle of least privilege exists - other contracts should only be able to access functions or data for which they possess specific authorization.
	[2.2] Verify that new contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permission until access to the new features is explicitly granted.
	[2.3] Verify that the creator of the contract complies with the rule of least privilege and his rights strictly follow the documentation.
	[2.4] Verify that the contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present (as the client-side access control can be easily bypassed).
	[2.5] Verify that there is a centralized mechanism for protecting access to each type of protected resource.
	[2.6] Verify that the calls to external contracts are allowed only if necessary.
	[2.7] Verify that visibility of all functions is specified.
	[2.8] Verify that the initialization functions are marked internal and cannot be executed twice.
	[2.9] Verify that the code of modifiers is clear and simple. The logic should not contain external calls to untrusted contracts.
	[2.10] Verify that the contract relies on the data provided by right sender and contract does not rely on tx.origin value.
	[2.11] Verify that all user and data attributes used by access controls are kept in trusted contract and cannot be manipulated by other contracts unless specifically authorized.
	[2.12] Verify that the access controls fail securely including when a revert occurs.
	[3.4] Verify that there is a component that monitors access to sensitive contract data using events.
	[7.4] Verify that the contract does not check whether the address is a contract using extcodesize opcode.
161	Verify that the contract is not vulnerable to Silent Failing Sends and Unchecked-Send attacks.
	[4.6] Verify that the result of low-level function calls (e.g. send, delegatecall, call) from another contracts is checked.
	[4.7] Verify that the third party contracts do not shadow special functions (e.g. revert).
162	Verify that the contract is not vulnerable to Denial of Service attacks.
	[7.3] Verify that the contract does not iterate over unbound loops.

Sr.No	Category
	[9.1] Verify that the self-destruct functionality is used only if necessary.
	[9.2] Verify that the business logic does not block its flows when any of the participants is absent forever.
	[9.3] Verify that the contract logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher that the profit).
	[9.4] Verify that the expressions of functions assert or require have a passing variant.
	[9.5] Verify that if the fallback function is not callable by anyone, it is not blocking the functionalities of contract and the contract is not vulnerable to Denial of Service attacks.
	[9.6] Verify that the function calls to external contracts (e.g. send, call) are not the arguments of require and assert functions.
163	Verify that the contract is not vulnerable to Bad Randomness issues.
	[7.5] Verify that the contract does not generate pseudorandom numbers trivially basing on the information from blockchain (e.g. seeding with the block number).
164	Verify that the contract is not vulnerable to Front-Running attacks.
	[8.7] Verify that the contract uses mechanisms that mitigate transaction-ordering dependence (front-running) attacks (e.g. pre-commit scheme).
165	Verify that the contract is not vulnerable to Time Manipulation issues.
	[8.6] Verify that the sensitive operations of contract do not depend on the block data (i.e. block hash, timestamp).
166	Verify that the contract is not vulnerable to Short Address Attack.
	[6.2] Verify that the length of passed address is determined and validated by smart contract.
167	Verify that the contract is not vulnerable to Insufficient Gas Griefing attack.
	[7.1] Verify that the usage of gas in smart contracts is anticipated, defined and have clear limitations that cannot be exceeded. Both, code structure and malicious input should not cause gas exhaustion.
	[7.2] Verify that two types of the addresses are considered when using the send function. Sending Ether to contract address costs more than sending Ether to personal address.
	[7.3] Verify that the contract does not iterate over unbound loops.
	[7.4] Verify that the contract does not check whether the address is a contract using extcodesize opcode.
	[7.10] Verify that the external keyword is used for functions that can be called externally only to save gas
SCVS - Decentralized Finance	
168	Verify that the lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
169	Verify that the functions which change lender's balance and lend cryptocurrency are non-re-entrant if the smart contract allows to borrow the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.

Sr.No	Category
170	Verify that the flash loan function can call only a predefined function on the receiver contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sender (borrower) contract is the one to be called back.
171	Verify that the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiver's owner or other trusted source (e.g. multisig), if it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed).
172	Verify that the calculations of the share in a liquidity pool are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 decimals precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimals (e.g. dividend * 10**18 / divisor).
173	Verify that the rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). That protects from the momentary fluctuations in shares.
174	Verify that the governance contracts are protected from the attacks that use flash loans. One possible security is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
175	Verify that, when using an on-chain oracles, the smart contract is able to pause the operations based on the oracle's result (in case of oracle has been compromised).
176	Verify that the external contracts (even trusted) that are allowed to change the attributes of the smart contract (e.g. token price) have the following limitations implemented: a thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
177	Verify that the smart contract attributes that can be updated by the external contracts (even trusted) are monitored (e.g. using events) and the procedure of incident response is implemented (e.g. the response to an ongoing attack).
178	Verify that the complex math operations that consist of both multiplication and division operations firstly perform the multiplications and then division.
179	Verify that, when calculating conversion price (e.g. price in ETH for selling a token), the numerator and denominator are multiplied by the reserves (see the getInputPrice function in UniswapExchange contract as an example).

+971-557391463 info@netsentries.com www.netsentries.com

NetSentries Technologies, Techno Hub 2,Dtec, Dubai Silicon Oasis,
P.O. Box: 644945, Dubai, United Arab Emirates