securing

www.securing.biz

Smart Contract Security Verification Standard

SMART CONTRACT SECURITY VERIFICATION STANDARD

Authors

Damian Rusinek, Paweł Kuryłowicz

Introduction

Smart Contract Security Verification Standard (v1.1) is a FREE 14-part checklist created to standardize the security of smart contracts for developers, architects, security reviewers and vendors. This list helps to avoid the majority of known security problems and vulnerabilities by providing guidance at every stage of the development cycle of the smart contracts (from designing to implementation).

Objectives

- Help to develop high quality code of the smart contracts.
- Help to mitigate known vulnerabilities by design.
- Provide a checklist for security reviewers.
- Provide a clear and reliable assessment of how secure a smart contract is in the relation to the percentage of SCSVS coverage.

Use cases

You can use the SCSVS checklist in multiple ways:

- As a starting point for formal threat modelling exercise.
- As a measure of your smart contract security and maturity.
- As a scoping document for penetration test or security audit of a smart contract.
- As a formal security requirement list for developers or third parties developing the smart contract for you.
- As a self-check for developers.
- To point areas which need further development in regards to security.

The entire checklist is in a form similar to OWASP APPLICATION SECURITY VERIFICATION STANDARD v4.0. Every category has a brief description of the control objectives and a list of security verification requirements.

Key areas that have been included

- V1: Architecture, Design and Threat Modelling
- V2: Access Control
- V3: Blockchain Data
- V4: Communications
- V5: Arithmetic
- V6: Malicious Input Handling
- V7: Gas Usage & Limitations
- V8: Business Logic
- V9: Denial of Service
- V10: Token
- V11: Code Clarity
- V12: Test Coverage
- V13: Known Attacks
- V14: Decentralized Finance

Severity of the risk

Threat modelling and risk analysis are important parts of the security assessment. Threat modelling allows to discover the potential threats and their risk impact. The aim of risk analysis is to identify security risks and determine their severity which allows to prioritize them in the mitigation process.

The SCSVS does not include the severity of the risks related to the requirements. Even though there are multiple methodologies to assess the severity, each application is unique and so are the threat actors, their goals, and the impact of a breach.

Moreover, the requirements cannot be uniquely linked to the security risks as many risks can refer to one requirement and many requirements can refer to one risk.

We recommend to determine the severity of the risks related with the requirements when performing the security assessment using SCSVS standard.
We recommend Common Vulnerability Scoring System (CVSS), a free and open industry standard for assessing the severity of security vulnerabilities.

License

1. V1: ARCHITECTURE, DESIGN AND THREAT MODELLING

1.1. Control Objective

Architecture, design and threat modelling in the context of creating secure smart contracts. Consider all possible threats before the implementation of the smart contract.

Ensure that a verified contract satisfies the following high-level requirements:

- All related smart contracts are identified and used properly,
- Specific smart contracts security assumptions are considered during the design phase.

Category "V1" lists requirements related to the architecture, design and threat modelling of the smart contracts.

1.2. Security Verification Requirements

#	Description
1.1	Verify that the every introduced design change is preceded by an earlier threat modelling.
1.2	Verify that the documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
1.3	Verify that the SCSVS, security requirements or policy is available to all developers and testers.
1.4	Verify that there exists an upgrade process for the contract which allows to deploy the security fixes.
1.5	Verify that there is a component that monitors the contract activity using events.
1.6	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.
1.7	Verify that there is a policy to track new security bugs and to update the libraries to the latest secure version.
1.8	Verify that the value of cryptocurrencies kept on contract is controlled and at the minimal acceptable level.
1.9	Verify that if the fallback function can be called by anyone it is included in the threat modelling.
1.10	Verify that the business logic in contracts is consistent. Important changes in the logic should be allowed for all or none of the contracts.
1.11	Verify that code analysis tools are in use that can detect potentially malicious code.
1.12	Verify that the latest version of Solidity is used.

1.3. References

For more information, see also:

- OWASP Threat Modeling Cheat Sheet
- OWASP Attack Surface Analysis Cheat Sheet
- OWASP Threat modeling
- Microsoft SDL
- NIST SP 800-57
- Use events to monitor contract activity

2. V2: ACCESS CONTROL

2.1. Control Objective

Access control is the process of granting or denying specific requests to obtain and use information and related information processing services.

Ensure that a verified contract satisfies the following high-level requirements:

- Users and other contracts are associated with a well-defined set of roles and privileges,
- Access is granted only to the privileged users and contracts.

Category "V2" lists requirements related to the access control mechanisms of the smart contracts.

2.2. Security Verification Requirements

#	Description
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.

2.3. References

For more information, see also:

- OWASP Cheat Sheet: Access Control
- OpenZeppelin: Role-Based Access Control

3. V3: BLOCKCHAIN DATA

3.1. Control Objective

Smart contracts in public blockchains have no built-in mechanism to store secret data securely. It is important to protect sensitive data from reading by an untrusted actor.

Ensure that a verified contract satisfies the following high-level requirements:

- Data stored in smart contract is identified and protected,
- Secret data is not kept in blockchain as plaintext,
- Smart contract is not vulnerable due to data misrepresentation.

Category "V3" lists requirements related to the blockchain data of the smart contracts.

3.2. Security Verification Requirements

#	Description
3.1	Verify that any data saved in the contracts is not considered safe or private (even private variables).
3.2	Verify that no confidential data is stored in the blockchain (passwords, personal data, token etc.).
3.3	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.
3.4	Verify that there is a component that monitors access to sensitive contract data using events.
3.5	Verify that contract does not use string literals as keys for mappings. Verify that global constants are used instead to prevent Homoglyph attack.

3.3. References

For more information, see also:

- Homoglyph attack

4. V4: COMMUNICATIONS

4.1. Control Objective

Communications includes the topic of the relations between smart contracts and their libraries.

Ensure that a verified contract satisfies the following high-level requirements:

- The external calls from and to other contracts have considered abuse case and are authorized,
- Used libraries are safe and the state-of-the-art security libraries are used.

Category "V4" lists requirements related to the function calls between the verified contracts and other contracts out of the scope of the application.

4.2. Security Verification Requirements

#	Description
4.1	Verify that the libraries which are not part of the application (but the smart contract relies on to operate) are identified.
4.2	Verify that the contract does not use hardcoded addresses unless necessary. If the hard-coded address is used, make sure that its contract has been audited.
4.3	Verify that the contracts and libraries which call external security services have a centralized implementation.
4.4	Verify that *delegatecall* is not used with untrusted contracts.
4.5	Verify that the re-entrancy attack is mitigated by blocking the recursive calls from the other contracts. Do not use *call* and *send* function unless it is a must.
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*).

4.3. References

For more information, see also:

- Security Considerations: Sending and Receiving Ether
- DASP 10: Unchecked Return Values For Low Level Calls
- SWC-107 Reentrancy
- SWC-112 Delegatecall to Untrusted Callee

5. V5: ARITHMETIC

5.1. Control Objective

Ensure that a verified contract satisfies the following high-level requirements:

- All mathematical operations and extreme variable values are handled in a safe and predictable manner.

Category "V5" lists requirements related to the arithmetic operations of the smart contracts.

5.2. Security Verification Requirements

#	Description
5.1	Verify that the values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations.
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.
5.3	Verify that non-strict inequality is used for balance equality.

5.3. References

For more information, see also:

- DASP 10: Artithmetic Issues
- OpenZeppelin: SafeMath
- SWC-101 Integer Overflow and Underflow

6. V6: MALICIOUS INPUT HANDLING

6.1. Control Objective

Values obtained as parameters by smart contracts should be validated.

Ensure that a verified contract satisfies the following high-level requirements:

- The function parameters being passed are handled in a safe and predictable manner.

Category "V6" lists requirements related to the malicious input to the functions of smart contracts.

6.2. Security Verification Requirements

#	Description
6.1	Verify that if the input (function parameters) is validated, the positive validation approach (whitelisting) is used.
6.2	Verify that the length of the address being passed is determined and validated by smart contract.

6.3. References

For more information, see also:

- Security Considerations - Sending and Receiving Ether
- OpenZeppelin: SafeMath

7. V7: GAS USAGE & LIMITATIONS

7.1. Control Objective

The efficiency of gas consumption should be taken into account not only in terms of optimization, but also in terms of safety to avoid possible exhaustion. Smart contracts by nature have different limitations that, if they are not taken into account, can cause different vulnerabilities.

Ensure that a verified contract satisfies the following high-level requirements:

- The use of gas is optimized and unnecessary losses are prevented,
- The implementation is in line with the smart contracts' limitations.

Category "V7" lists requirements related to gas and smart contract limitations.

7.2. Security Verification Requirements

#	Description
7.1	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.
7.2	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.
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.
7.5	Verify that the contract does not generate pseudorandom numbers trivially basing on the information from blockchain (e.g. seeding with the block number).
7.6	Verify that the contract does not assume fixed-point precision but uses a multiplier or store both the numerator and denominator instead.
7.7	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.
7.8	Verify that there exists a mechanism that protects the contract from a replay attack in case of a hard-fork.
7.9	Verify that all library functions that should be upgradeable are not internal.
7.10	Verify that the *external* keyword is used for functions that can be called externally only to save gas.

7.3. References

For more information, see also:

- DASP 10: Bad Randomness
- Gas Limit and Loops
- Insufficient gas griefing
- SWC-121 Missing Protection against Signature Replay Attacks

8. V8: BUSINESS LOGIC

8.1. Control Objective

To build secure smart contracts, we need to consider security of their business logic. Some of the smart contracts are influenced by vulnerabilities by their design. The components used should not be considered safe without verification. Business assumptions should meet with the principle of minimal trust, which is essential in building smart contracts.

Ensure that a verified contract satisfies the following high-level requirements:

- The business logic flow is sequential and in order.
- Business limits are specified and enforced.
- Cryptocurrency and token business logic flows have considered abuse cases and malicious actors.

Category “V8” lists requirements related to the business logic of the smart contracts.

8.2. Security Verification Requirements

#	Description
8.1	Verify that the contract logic implementation corresponds to the documentation.
8.2	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.
8.3	Verify that the contract has business limits and correctly enforces it.
8.4	Verify that the business logic of contract does not rely on the values retrieved from untrusted contracts with multiple calls of the same function.
8.5	Verify that the contract logic does not rely on the balance of contract (e.g. balance == 0).
8.6	Verify that the sensitive operations of contract do not depend on the block data (i.e. block hash, timestamp).
8.7	Verify that the contract uses mechanisms that mitigate transaction-ordering dependence (front-running) attacks (e.g. pre-commit scheme).
8.8	Verify that the contract does not send funds automatically but it lets users withdraw funds on their own in separate transaction instead.
8.9	Verify that the inherited contracts do not contain identical functions or the order of inheritance is carefully specified.

8.3. References

For more information, see also:

- Timestamp Dependence
- DASP 10: Front-Running
- Front-Running (AKA Transaction-Ordering Dependence)
- Solidity Recommendations
- SWC-125 Incorrect Inheritance Order

9. V9: DENIAL OF SERVICE

9.1. Control Objective

Ensure that a verified contract satisfies the following high-level requirements:

- The contract logic prevents influencing the availability of the contract.

Category "V9" lists requirements related to the possible denial of service of the smart contracts.

9.2. Security Verification Requirements

#	Description
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 to 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.

9.3. References

For more information, see also:

- DASP 10: Denial of Service
- Gas Limit and Loops
- Gas Limit DoS on the Network via Block Stuffing
- DoS with Block Gas Limit
- SWC-128 DoS With Block Gas Limit
- SWC-113 DoS with Failed Call

10. V10: TOKEN

10.1. Control Objective

Ensure that a verified contract satisfies the following high-level requirements:

- The implemented token follows the security standards.

Category “V10” lists requirements related to the token of the smart contracts.

10.2. Security Verification Requirements

#	Description
10.1	Verify that the token contract follows a tested and stable token standard.
10.2	Use the approved function of the ERC-20 standard to change allowed amount only to 0 or from 0.
10.3	Verify that the contract does not allow to transfer tokens to zero address.

10.3. References

For more information, see also:

- Token Implementation Best Practice

11. V11: CODE CLARITY

11.1. Control Objective

Ensure that a verified contract satisfies the following high-level requirements:

- The code is clear and easy to read,
- There are no unwanted or unused parts of the code,
- Variable names are in line with good practices,
- The functions being used are secure.

Category “V11” lists requirements related to the code clarity of the smart contracts.

11.2. Security Verification Requirements

#	Description
11.1	Verify that the logic is clear and modularized in multiple simple contracts and functions.
11.2	Verify that the inheritance order is considered for contracts that use multiple inheritance and shadow functions.
11.3	Verify that the contract uses existing and tested code (e.g. token contract or mechanisms like *ownable*) instead of implementing its own.
11.4	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).
11.5	Verify that variables with similar names are not used.
11.6	Verify that all variables are defined as storage or memory variable.
11.7	Verify that all storage variables are initialised.
11.8	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.
11.9	Verify that the functions which specify a return type return the value.
11.10	Verify that all functions are used. Unused ones should be removed.
11.11	Verify that the *require* function is used instead of the *revert* function in *if* statement.
11.12	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.
11.13	Verify that assembly code is used only if necessary.

11.3. References

For more information, see also:

- Solidity: Security Considerations & Recommendations

12. V12: TEST COVERAGE

12.1. Control Objective

Ensure that a verified contract satisfies the following high-level requirements:

- The specification has been formally tested,
- The implementation has been tested statically and dynamically,
- The implementation has been tested using symbolic execution.

Category "V12" lists requirements related to the testing process of the smart contracts.

12.2. Security Verification Requirements

#	Description
12.1	Verify that all functions of verified contract are covered with tests in the development phase.
12.2	Verify that the implementation of verified contract has been checked for security vulnerabilities using static and dynamic analysis.
12.3	Verify that the specification of smart contract has been formally verified.
12.4	Verify that the specification and the result of formal verification is included in the documentation.

12.3. References

For more information, see also:

- Formal Verification
- Code coverage for Solidity testing
- Remix IDE
- MythX Plugin for Truffle
- Securify
- SmartCheck
- Oyente
- Security Tools

13. V13: KNOWN ATTACKS

13.1. Control Objective

The *Known attacks* category has a different form from the other categories. It covers all known attacks and link them to the requirements from other categories.

Category "V13" aims to ensure that a verified contract is protected from the known attacks.

13.2. Security Verification Requirements

#	Description
13.1	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.
	[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.
13.2	Verify that the contract is not vulnerable to Reentrancy attack.
	[4.5] Verify that the re-entrancy attack is mitigated by blocking the recursive calls from the other contracts. Do not use *call* and *send* function unless it is a must.
13.3	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.
13.4	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*).
13.5	Verify that the contract is not vulnerable to Denial of Service attacks.
	[7.3] Verify that the contract does not iterate over unbound loops.
	[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 to 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.
13.6	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).
13.7	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).
13.8	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).

13.9	Verify that the contract is not vulnerable to Short Address Attack.
	[6.2] Verify that the length of the address being passed is determined and validated by smart contract.
13.10	Verify that the contract is not vulnerable to Insufficient Gas Griefing attack.
	[7.1] 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.
	[7.2] 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.
	[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.

13.3. References

For more information, see also:

- DASP - Top 10
- Known Attacks

14. V14: DECENTRALIZED FINANCE

14.1. Control Objective

The Decentralized Finance (DeFi) is a concept with various financial applications deployed on blockchain using smart contracts. This category covers the security requirements for the constructions used by the DeFi applications such as lending pools, flash loans, governance, on-chain oracles, etc.

Ensure that a verified contract satisfies the following high-level requirements:

- The assets controlled by decentralized finance are safe and cannot be withdrawed by an unauthorized person,
- The data sources for decentralized finance (e.g. oracles) cannot be manipulated.

Category “V14” lists requirements related to the contructions used by the decentralized finance solutions.

14.2. Security Verification Requirements

#	Description
14.1	Verify that the lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
14.2	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.
14.3	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.
14.4	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).
14.5	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).
14.6	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.

14.7	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.
14.8	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).
14.9	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).
14.10	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).
14.11	Verify that the complex math operations that consist of both multiplication and division operations firstly perform the multiplications and then division.
14.12	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).

14.3. References

For more information, see also:

- Damn vulnerable #DeFi
- Damn vulnerable #DeFi Write-ups & lessons learned
- Uniswap's *getInputPrice* function

15. ACKNOWLEDGEMENTS

15.1. Contributors

We would like to thank aftermentioned people for the initial feedback on SCSVS:
- Gabriel Garrido Calvo (Lightstreams Network)
- Bernhard Mueller (ConsenSys)
- Tomasz Szymański (SoftwareMill)

15.2. Community

We would like to thank the creators of the following websites, which are a great source of information about the security of smart contracts:
- Decentralized Application Security Project Top 10
- Ethereum Smart Contract Security Best Practices
- Smart Contract Weakness Classification and Test Cases
- Solidity - Security Considerations