ITUPublications
Recommendations

International Telecommunication Union
Standardization Sector

Recommendation

ITU-T X.1412 (04/2023)

SERIES X: Data networks, open system communications and security

Secure applications and services (2) – Distributed ledger technology (DLT) security

---

Security requirements for smart contract management based on the distributed ledger technology

ITU-T X-SERIES RECOMMENDATIONS

Data networks, open system communications and security

PUBLIC DATA NETWORKS	X.1-X.199
OPEN SYSTEMS INTERCONNECTION	X.200-X.299
INTERWORKING BETWEEN NETWORKS	X.300-X.399
MESSAGE HANDLING SYSTEMS	X.400-X.499
DIRECTORY	X.500-X.599
OSI NETWORKING AND SYSTEM ASPECTS	X.600-X.699
OSI MANAGEMENT	X.700-X.799
SECURITY	X.800-X.849
OSI APPLICATIONS	X.850-X.899
OPEN DISTRIBUTED PROCESSING	X.900-X.999
INFORMATION AND NETWORK SECURITY	X.1000-X.1099
SECURE APPLICATIONS AND SERVICES (1)	X.1100-X.1199
CYBERSPACE SECURITY	X.1200-X.1299
SECURE APPLICATIONS AND SERVICES (2)	X.1300-X.1499
Emergency communications	X.1300-X.1309
Ubiquitous sensor network security	X.1310-X.1319
Smart grid security	X.1330-X.1339
Certified mail	X.1340-X.1349
Internet of things (IoT) security	X.1350-X.1369
Intelligent transportation system (ITS) security	X.1370-X.1399
Distributed ledger technology (DLT) security	**X.1400-X.1429**
Application Security (2)	X.1450-X.1459
Web security (2)	X.1470-X.1489
CYBERSECURITY INFORMATION EXCHANGE	X.1500-X.1599
CLOUD COMPUTING SECURITY	X.1600-X.1699
QUANTUM COMMUNICATION	X.1700-X.1729
DATA SECURITY	X.1750-X.1799
IMT-2020 SECURITY	X.1800-X.1819

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T X.1412

Security requirements for smart contract management based on the distributed ledger technology

Summary

Recommendation ITU-T X.1412 analyses security threats and challenges and provides security requirements for smart contract management in distributed ledger technology (DLT) systems. As smart contracts are widely used in DLT systems, they are faced with a lot of security threats and challenges.

As such this Recommendation can be used by smart contract designers, developers, and managers to manage smart contracts, including design and development, compilation and deployment, invocation and execution, maintenance and management in DLT systems. This Recommendation does not deal with the security issues of wallets or distributed applications related to smart contracts.

History *

Edition	Recommendation	Approval	Study Group	Unique ID
1.0	ITU-T X.1412	2023-04-29	17	11.1002/1000/15545

Keywords

Blockchain, distributed ledger technology, security requirements, smart contract.

* To access the Recommendation, type the URL https://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents/software copyrights, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the appropriate ITU-T databases available via the ITU-T website at http://www.itu.int/ITU-T/ipr/.

Table of Contents

Recommendation ITU-T X.1412

Security requirements for smart contract management based on the distributed ledger technology

1 Scope

This Recommendation analyses the security threats and challenges of the smart contract in distributed ledger technology (DLT) systems and specifies the security requirements for smart contract management.

This Recommendation can be referred to in smart contract lifecycle management as a starting point for security. This Recommendation does not deal with the security issues of wallets or distributed applications related to smart contracts.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T X.1401] Recommendation ITU-T X.1401 (2019), *Security threats of distributed ledger technology*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 block [b-ITU-T X.1400]: Individual data unit of a blockchain (see clause 3.1.2), composed of a collection of transactions and a block header.

3.1.2 blockchain [b-ITU-T X.1400]: A type of distributed ledger which is composed of digitally recorded data arranged as a successively growing chain of blocks with each block cryptographically linked and hardened against tampering and revision.

3.1.3 blockchain system [b-ITU-T X.1400]: A system that implements a blockchain.

3.1.4 consensus [b-ITU-T X.1400]: Agreement that a set of transactions is valid.

3.1.5 consensus mechanism [b-ITU-T X.1400]: Rules and procedures by which consensus is reached.

3.1.6 distributed ledger [b-ITU-T X.1400]: A type of ledger that is shared, replicated, and synchronized in a distributed and decentralized manner.

3.1.7 distributed ledger technology (DLT) [b-ITU-T X.1400]: Technology that enables the operation and use of distributed ledgers.

3.1.8 DLT system [b-ITU-T X.1400]: A system that implements a distributed ledger.

3.1.9 on-chain [b-ITU-T X.1400]: Located, performed, or run inside a blockchain system.

3.1.10 off-chain [b-ITU-T X.1400]: Related to a blockchain system, but located, performed, or run outside that blockchain system.

3.1.11 personally identifiable information (PII) [b-ITU-T X.1252]: Any information:

a) that identifies or can be used to identify, contact, or locate the person to whom such information pertains;

b) from which identification or contact information of an individual person can be derived; or

c) that is or can be linked to a natural person directly or indirectly.

3.1.12 public DLT system [b-ITU-T X.1400]: A distributed ledger technology (DLT) system which is accessible to the public for use.

3.1.13 private DLT system [b-ITU-T X.1400]: A distributed ledger technology (DLT) system which is accessible for use only to a limited group of DLT users.

3.1.14 sensitive data [b-ITU-T X.1040]: Data with potentially harmful effects in the event of disclosure or misuse.

3.1.15 smart contract [b-ITU-T X.1400]: A program written on a distributed ledger system which encodes the rules for specific types of distributed ledger system transactions in a way that can be validated and triggered by specific conditions.

3.1.16 transaction [b-ITU-T X.1400]: Whole of the exchange of information between nodes. A transaction is uniquely identified by a transaction identifier.

3.2 Terms defined in this Recommendation

None.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

CPU Central Processing Unit

DLT Distributed Ledger Technology

DoS Denial of Service

EVM Ethereum Virtual Machine

PII Personally Identifiable Information

SCSC Smart Contract Security Challenge

5 Conventions

This Recommendation applies the following verbal forms for the expression of provisions:

a) "**Shall**" indicates a requirement,

b) "**Should**" indicates a recommendation,

c) "**May**" indicates a permission,

d) "**Can**" indicates a possibility and a capability.

6 Security threats of smart contract management

[ITU-T X.1401] categorizes the security threats to smart contracts into the following four types: timestamp dependence attack, mishandled exceptions attack, integer overflow attack, and predictable random number attack. Given below is a view of the attack method.

Smart contract codes can include errors and vulnerabilities such as intentionally or accidentally, lack of education and training, lack of awareness, simple human error (with/without tiredness), or by technology evolution.

Malicious users can exploit smart contract vulnerabilities to attack the DLT system and cause unexpected economic losses.

As smart contracts are hard to patch once they are deployed on a DLT system, the attack impact and the consequences caused by the vulnerabilities could be long lasting. If smart contracts are used to manage digital assets, any vulnerabilities and errors can cause problems such as asset loss and misappropriation.

Figure 1 shows the components related to smart contracts in the DLT system and the challenges considered to securely manage smart contracts.

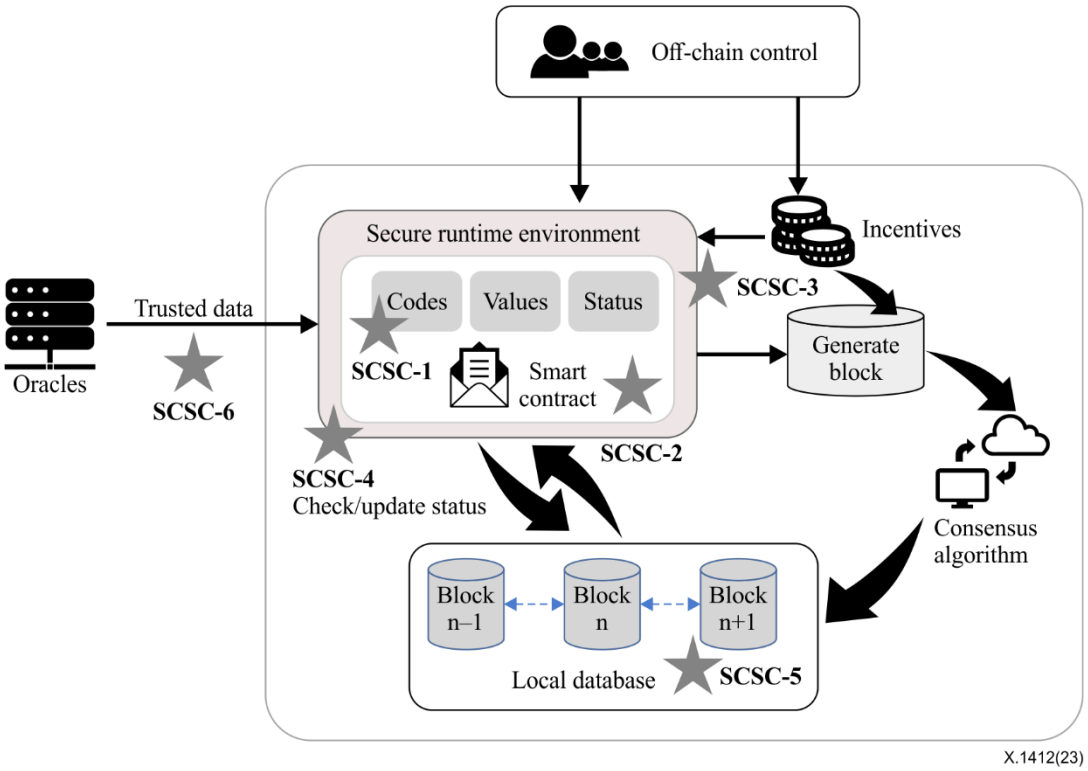

Figure 1 – Security challenges of smart contract based on DLT

The security challenges of smart contracts based on the DLT are listed below:

- SCSC-1: Smart contract codes can include vulnerabilities, which cannot be changed directly. It makes insecure codes difficult to modify and increases the likelihood of exploitation.
- SCSC-2: Smart contract codes are typically public. It makes them easy to analyse and increases the likelihood of finding vulnerabilities. If the code involves sensitive data, it is also exposed.
- SCSC-3: Evolution of the underlying DLT platform can make new vulnerabilities, reveal unknown vulnerabilities, or make existing smart contract codes obsolete.
- SCSC-4: A DLT system's operating environment can have security flaws that possibly lead to denial of service (DoS), virtual machine escaping, any code execution, privilege escalation, etc.
- SCSC-5: A smart contract can consume excessive environmental resources, such as central processing unit (CPU), memory, and disk storage, which will adversely affect the performance of the DLT system.

- SCSC-6: When smart contracts access external data, they can encounter errors forged, or inconsistent data, which might cause incorrect results, misuse, or exploitation.

More detailed information on the security analysis of smart contracts can be found at [b-ACM], [b-ARXIV], [b-Kaiser], and [b-SWCR].

7 Security framework of smart contract management

Throughout the smart contract development lifecycle, all security requirements will be introduced to avoid relevant security threats and challenges, which include but are not limited to security activities across the design and development, compilation and deployment, invocation and execution, and lastly maintenance and management phases [b-IEEE-scs].

The security framework of smart contract lifecycle management is shown in the Figure 2.

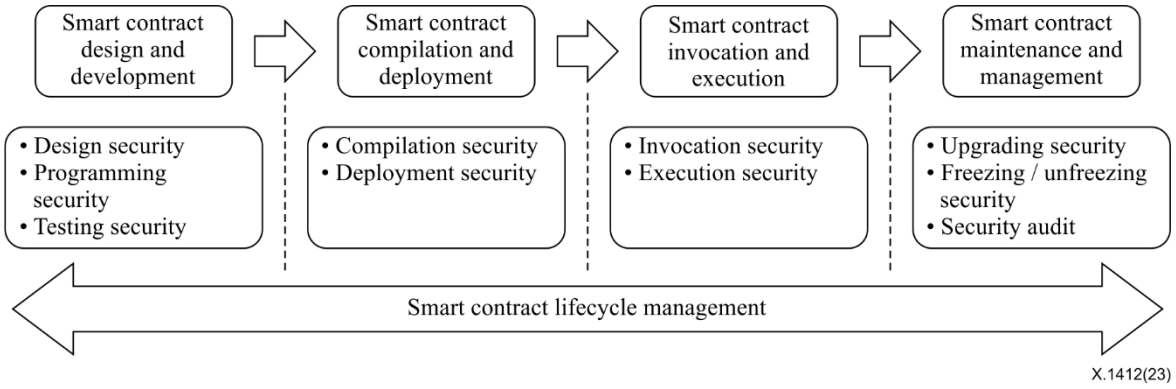

Figure 2 – Security framework of smart contract management

In the design and development phase, it includes design security, programming security, and testing security.

In the compilation and deployment phase, it includes compilation security and deployment security.

In the invocation and execution phase, it includes invocation security and execution security.

In the maintenance and management phase, it includes upgrading security, freezing/unfreezing security and security audit.

8 Security requirements for smart contract management

Developers should have appropriate DLT related knowledge and skills, through ongoing security updates and education. The knowledge and skills include DLT platforms and the smart contract they support, particularly when it evolves continuously and rapidly, to reflect the technology changes and to respond to newly reported vulnerabilities and threats.

Different vulnerabilities might exist in different DLT systems. The smart contracts in different DLT systems will probably be programmed with different programming languages, and will be executed in different environments.

8.1 Security requirements in the design and development phase

8.1.1 Design security

It is critically important to consider smart contract security issues carefully during the design and development phase. Mitigation of security issues is much less expensive when performed in the design phase compared with later phases during the smart contract lifecycle.

Before writing the smart contract code, the functions of the entire contract and external interactions should be designed, and the security problems caused by design flaws should be avoided.

In the design phase, the security requirements are as follows:

a) Smart contracts shall be designed to have defect mitigation capability. The defect mitigation capability includes early termination capability, online update capability, and other capabilities to handle or mitigate defects after a smart contract's deployment.

b) Smart contracts, in principle, shall not be designed to store, use, or transfer sensitive information on-ledger. The sensitive information includes personally identifiable information (PII) and other sensitive data which should be protected from unauthorized access.

c) When a smart contract receives sensitive information, the information shall be encrypted.

d) Smart contracts shall be based upon proven safe encryption algorithms to reduce algorithm attacks.

e) Smart contracts should be designed to avoid deriving operation results including random data or action.

f) Smart contracts should be designed to avoid predictable random numbers of attacks.

g) Smart contracts shall be designed with reasonable sequence logic to avoid different execution results between nodes.

h) Smart contracts shall be designed to avoid timestamp dependency attacks.

i) When designing a smart contract, there should be no dependence on the order, such as the order of payment, application, etc., to avoid security problems caused by the competition of conditions.

j) For the private DLT system, each role and its authorization should be designed reasonably.

k) A smart contract should be developed in a targeted manner according to the differences between different versions of the platforms.

l) When the platform is upgraded, the operating differences of the smart contract on the new version of the platform should be evaluated, and appropriate adjustments should be made to avoid security issues.

8.1.2 Programming security

8.1.2.1 Development environment security

The development and compilation tools and dependencies for smart contracts should be updated to respond to newly reported vulnerabilities and threats. Keeping tools and dependencies up to date is very important to ensure the security of the smart contract development environment.

It should be noted that there may be new security flaws in the latest tools, and supply chain security shall be considered when using third-party tools and codes.

8.1.2.2 Coding security

For the coding security of smart contracts, the security requirements are as follows:

a) When developing smart contracts, developers shall clearly define the types of variables to determine the range of values for variables and prevent them from overflow, underflow, or uninitialized variables.

b) Developers should not use programming languages which can cause security failure that advanced languages can avoid.

c) When performing mathematical operations, developers shall be aware of the possible boundary conditions, for example, array index bound or bound for integer size, or any value should not be divided by zero.

d) When implementing the functionality or logic of the smart contract, developers shall avoid introducing unreliable input.

e) When the smart contract calls other functions outside of a smart contract, developers shall deal with possible errors. This includes but is not limited to, catching exceptions, handling error return values, and so on.

8.1.2.3 Logical security

Even if the code itself does not have security issues, there may be security flaws in the code logic.

Regarding the logical security of smart contracts, the security requirements are as follows:

a) When multiple users are included in the smart contract or multiple users are allowed to interact with it, the permissions of each user should be restricted. Only specific users should be allowed to use destruction operations and other sensitive operations.

b) Developers shall ensure that the overall logical design of smart contracts has no obvious defects.

c) Developers shall make smart contract implementation to be consistent with the logic design to avoid ambiguity or deviation.

d) Developers shall ensure that there are no obvious loopholes, especially in the game design.

e) Developers should take measures to avoid the risk of "DoS".

f) Developers should take measures to avoid the pre-emptive attack on transaction risk and transaction sequence dependence issues.

g) Smart contracts shall be developed in a targeted manner by complying with a specific type and/or version of the platform.

h) When designing applications based on smart contracts, developers should guarantee security for the logic of the smart contract itself and the logic of the off-chain.

8.1.3 Testing security

Smart contracts are hard to patch once they are deployed on the DLT system, so it is important to test the smart contracts before deployment. Developers shall perform security testing to ensure the security of the codes before they are deployed.

The security requirements for smart contract testing include:

a) The release of smart contracts shall be subject to security analysis and testing.

b) Security testing should cover general vulnerability detection and security verification of contract code or bytecode.

c) Static security analysis of smart contract code shall be performed to discover security issues in the code itself.

d) Dynamic security analysis should be performed to check whether there are security issues according to the running status.

e) Formal verification should be performed to use mathematical methods to determine whether the program's operating results meet expectations. Details of formal verification can be found at [b-Abdellatif].

8.2 Security requirements in the compilation and deployment phase

8.2.1 Compilation security

Smart contract compilation is the process of converting the smart contract code into an executable format of the operating environment.

For the compilation security of smart contracts, the requirements are as follows:

a) Mature and secure compilation tools shall be selected and used.

b) When a defect in the compiler is found, it shall be updated in time to eliminate the defect.

c) The logic of the smart contract source code shall be consistent after being compiled into bytecode.

d) It shall support the compilation and detection of smart contracts on the blockchain platform.

8.2.2 Deployment security

During this phase, a smart contract is made available to the blockchain system with a transaction containing the raw or (pre-)compiled code.

The security requirements for smart contract deployment include:

a) Consensus shall be achieved by the DLT system before smart contract deployment, to avoid evil nodes to deploy the smart contract.

b) The integrity of the smart contract shall be verified before deployment.

c) The authentication and authorization of the smart contract deployment shall be verified before deployment.

8.3 Security requirements in the invocation and execution phase

8.3.1 Invocation security

In the invocation step, the code of the smart contract is activated by sending a transaction to the ledger, calling its primarily assigned address, and optionally transferring invocation parameters. Invocation methods include direct interface invocation, invocation between smart contracts, oracle machine invocation, etc.

The security requirements for smart contract invocation include:

a) In case of interface invocation, the smart contract shall have clear declarations for the interfaces to facilitate the interface invocation.

b) In case of invocation between smart contracts, risks and errors caused by untrusted external smart contracts should be prevented.

c) When interacting with external smart contracts, it should be clearly marked that the interaction is external by means of variable names, method names, or interface names.

d) In case of oracle machine invocation, the interface protocol shall be based on the secure transmission protocol, and the oracle machine itself should pass a third-party security and reliability assessment.

e) The smart contract shall specify and verify the invocation scope.

f) The invocation parameters shall be checked and verified by the nodes in the DLT system.

8.3.2 Execution security

8.3.2.1 Execution environment security

The execution environment security includes the following requirements:

a) The operating environment of the smart contract shall correctly handle exceptions, support real-time monitoring, and ensure that the smart contract can be terminated normally.

b) When the platform is upgraded, smart contracts targeted to the previous version of the platform shall be evaluated and provided appropriate adjustments and/or backward compatibility to avoid security issues.

8.3.2.2 Data security

The data security includes the following requirements:

a) The status data of other users or other contracts shall not be tampered with without any security control.

b) Users' data shall be protected by security controls, e.g., using cryptography algorithms, and isolation of contract status data.

8.3.2.3 Resource security

The execution environment should guarantee resource security for smart contract execution, e.g., a trusted execution environment (TEE).

8.4 Security requirements in the maintenance and management phase

8.4.1 Upgrading security

The security requirements for smart contract upgrading include:

a) It should support online upgrades of smart contracts.

b) After the smart contract is upgraded, the historical version should be retained.

c) When the smart contract upgrade fails, it should be able to roll back to the original smart contract.

d) The upgrade operation of the smart contract should be recorded in the block.

8.4.2 Freezing / unfreezing security

Smart contract freezing and unfreezing is the process of stopping and reopening the calling function of smart contracts.

The security requirements for smart contract freezing / unfreezing include:

a) The DLT system shall be able to freeze the smart contract, in case the smart contract has bugs or security issues.

b) The DLT system shall be able to unfreeze the smart contract after the bugs in the smart contract are fixed.

c) When freezing or unfreezing the smart contract, user authentication and authorization shall be performed.

8.4.3 Security audit

The security requirements for smart contract audit include:

a) Smart contract auditors should be able to audit the source code. They can audit and analyse the code security of the smart contract by manually reviewing the source code and static code audit tools.

b) Smart contract auditors should be able to audit the compilation environment security of smart contracts.

c) Smart contract auditors should be able to audit the execution environment security of smart contracts.

d) The audit records shall be kept within a specified timeframe.

Appendix I

Smart contract weakness classification

(This appendix does not form an integral part of this Recommendation.)

This appendix is intended to facilitate understanding of this Recommendation by providing examples of various smart security vulnerabilities. The SWE registry publicizes this smart contract weakness classification data so anyone can register new smart contract vulnerabilities according to the weakness classification scheme proposed in [b-EIP-1470].

ID	Title	Related CWE	Description
SWC-136	Unencrypted private data on-chain	CWE-767: Access to critical private variable via public method	It is a common misconception that private type variables cannot be read. Even if your contract is not published, attackers can look at contract transactions to determine values stored in the state of the contract. For this reason, it is important that unencrypted private data is not stored in the contract code or state.
SWC-135	Code with no effects	CWE-1164: Irrelevant code	In solidity, it is possible to write code that does not produce the intended effects. Currently, the solidity compiler will not return a warning for effect-free code. This can lead to the introduction of "dead" code that does not properly perform an intended action.
SWC-134	Message call with the hardcoded gas amount	CWE-655: Improper initialization	The transfer() and send() functions forward a fixed amount of 2 300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against re-entrancy attacks. However, the gas cost of ethereum virtual machine (EVM) instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example, EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.
SWC-133	Hash collisions with multiple variable length arguments	CWE-294: Authentication bypass by capture-replay	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision. Since abi.encodePacked() packs all elements in order regardless of whether they are part of an array, you can move elements between arrays and, so long as all elements are in the same order, it will return the same encoding. In a signature verification situation, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization.
SWC-132	Unexpected ether balance	CWE-667: Improper locking	Contracts can behave erroneously when they strictly assume a specific ether balance. It is always possible to forcibly send ether to a contract (without triggering its fallback function), using self-destruct, or by mining to the account. In the worst-case scenario, this could lead to DOS conditions that might render the contract unusable.
SWC-131	Presence of unused variables	CWE-1164: Irrelevant code	Unused variables are allowed in solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can.

ID	Title	Related CWE	Description
SWC-130	Right-to-left-override control character (U+202E)	CWE-451: User interface (UI) misrepresentation of critical information	Malicious actors can use the right-to-left-override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.
SWC-129	Typographical error	CWE-480: Use of incorrect operator	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable (+=) but it has accidentally been defined in a wrong way (=+), introducing a typo that happens to be a valid operator. Instead of calculating the sum it initializes the variable again.
SWC-128	DoS with block gas limit	CWE-400: Uncontrolled resource consumption	When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas, based on how much computation is needed to complete them. The ethereum network specifies a block gas limit and the sum of all transactions included in a block cannot exceed the threshold.
SWC-127	Arbitrary jump with function type variable	CWE-695: Use of low-level functionality	Solidity supports function types. That is, a variable of function type can be assigned with a reference to a function with a matching signature. The function saved to such a variable can be called just like a regular function.
SWC-126	Insufficient gas griefing	CWE-691: Insufficient control flow management	Insufficient gas griefing attacks can be performed on contracts that accept data and use it in a sub-call on another contract. If the sub-call fails, either the whole transaction is reverted, or execution is continued. In the case of a relayed contract, the user who executes the transaction, the 'forwarder', can effectively censor transactions by using just enough gas to execute the transaction, but not enough for the sub-call to succeed.
SWC-125	Incorrect inheritance order	CWE-696: Incorrect behaviour order	Solidity supports multiple inheritance, meaning that one contract can inherit several contracts. Multiple inheritance introduces ambiguity called diamond problem: if two or more base contracts define the same function, which one should be called in the child contract? Solidity deals with this ambiguity by using reverse C3 linearization, which sets a priority between base contracts.
SWC-124	Write to arbitrary storage location	CWE-123: Write-what-where condition	A smart contract's data (e.g., storing the owner of the contract) is persistently stored at some storage location (i.e., a key or address) on the EVM level. The contract is responsible for ensuring that only authorized users or contract accounts may write to sensitive storage locations. If an attacker is able to write to arbitrary storage locations of a contract, the authorization checks may easily be circumvented. This can allow an attacker to corrupt the storage; for instance, by overwriting a field that stores the address of the contract owner.
SWC-123	Requirement violation	CWE-573: Improper following of specification by caller	The solidity require() construct is meant to validate external inputs of a function. In most cases, such external inputs are provided by callers, but they may also be returned by callees. In the former case, we refer to them as precondition violations. Violations of a requirement can indicate one of two possible issues.

ID	Title	Related CWE	Description
SWC-122	Lack of proper signature verification	CWE-345: Insufficient verification of data authenticity	It is a common pattern for smart contract systems to allow users to sign messages off-chain instead of directly requesting users to do an on-chain transaction because of the flexibility and increased transferability that this provides. Smart contract systems that process signed messages have to implement their own logic to recover the authenticity of the signed messages before they process them further. A limitation of such systems is that smart contracts cannot directly interact with them because they cannot sign messages. Some signature verification implementations attempt to solve this problem by assuming the validity of a signed message based on other methods that do not have this limitation. An example of such a method is to rely on msg.sender and assume that if a signed message originated from the sender address then it has also been created by the sender address. This can lead to vulnerabilities, especially in scenarios where proxies can be used to relay transactions.
SWC-121	Missing protection against signature replay attacks	CWE-347: Improper verification of cryptographic signature	It is sometimes necessary to perform signature verification in smart contracts to achieve better usability or to save gas costs. A secure implementation needs to protect against signature replay attacks by for example keeping track of all processed message hashes and only allowing new message hashes to be processed. A malicious user could attack a contract without such control and get a message hash that was sent by another user processed multiple times.
SWC-120	Weak sources of randomness from chain attributes	CWE-330: Use of insufficiently random values	Ability to generate random numbers is very helpful in all kinds of applications. One obvious example is gambling distributed applications, where a pseudo-random number generator is used to pick the winner. However, creating a strong enough source of randomness in ethereum is very challenging. For example, the use of block.timestamp is insecure, as a miner can choose to provide any timestamp within a few seconds and still get their block accepted by others. The use of blockhash, block.difficulty and other fields is also insecure, as they are controlled by the miner. If the stakes are high, the miner can mine lots of blocks in a short time by renting hardware, pick the block that has the required block hash for the miner to win, and drop all the others.
SWC-119	Shadowing state variables	CWE-710: Improper adherence to coding standards	Solidity allows for ambiguous naming of state variables when inheritance is used. Contract A with a variable x could inherit contract B which also has a state variable x defined. This would result in two separate versions of x, one of them being accessed from contract A and the other one from contract B. In more complex contract systems this condition could go unnoticed and subsequently lead to security issues.

ID	Title	Related CWE	Description
SWC-118	Incorrect constructor name	CWE-665: Improper initialization	Constructors are special functions that are called only once during the contract creation. They often perform critical, privileged actions such as setting the owner of the contract. Before solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract class containing it. A function meant to become a constructor becomes a normal, callable function if its name does not exactly match the contract name. This behaviour sometimes leads to security issues, in particular when smart contract code is re-used with a different name, but the name of the constructor function is not changed accordingly.
SWC-117	Signature malleability	CWE-347: Improper verification of cryptographic signature	The implementation of a cryptographic signature system in Ethereum contracts often assumes that the signature is unique, but signatures can be altered without the possession of the private key and still be valid. The EVM specification defines several so-called 'precompiled' contracts one of them being ecrecover, which executes the elliptic curve public key recovery. A malicious user can slightly modify the three values *v*, *r* and *s* to create other valid signatures. A system that performs signature verification on the contract level might be susceptible to attacks if the signature is part of the signed message hash. Valid signatures could be created by a malicious user to replay previously signed messages.
SWC-116	Block values as a proxy for time	CWE-829: Inclusion of functionality from untrusted control sphere	Contracts often need access to time values to perform certain types of functionalities. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.
SWC-115	Authorization through tx.origin	CWE-477: Use of obsolete function	tx.origin is a global variable in solidity that returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.
SWC-114	Transaction order dependence	CWE-362: Concurrent execution using shared resource with improper synchronization ('Race condition')	The Ethereum network processes transactions in blocks with new blocks getting confirmed around every 17 seconds. The miners look at transactions they have received and select which transactions to include in a block, based on who has paid a high enough gas price to be included. Additionally, when transactions are sent to the Ethereum network they are forwarded to each node for processing. Thus, a person who is running an Ethereum node can tell which transactions are going to occur before they are finalized. A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.
SWC-113	DoS with failed call	CWE-703: Improper check or handling of exceptional conditions	External calls can fail accidentally or deliberately, which can cause a DoS condition in the contract. To minimize the damage caused by such failures, it is better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically (this also reduces the chance of problems with the gas limit).

ID	Title	Related CWE	Description
SWC-112	Delegatecall to untrusted callee	CWE-829: Inclusion of functionality from untrusted control sphere	There exists a special variant of a message call, named delegatecall which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract, and msg.sender and msg.value do not change their values. This allows a smart contract to dynamically load code from a different address at runtime. Storage, current address, and balance still refer to the calling contract.
SWC-111	Use of deprecated solidity functions	CWE-477: Use of obsolete function	Several functions and operators in solidity are deprecated. Using them leads to reduced code quality. With new major versions of the solidity compiler, deprecated functions, and operators may result in side effects and compile errors.
SWC-110	Assert violation	CWE-670: Always-incorrect control flow implementation	The solidity assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement. A reachable assertion can mean one of two things: 1. A bug exists in the contract that allows it to enter an invalid state. 2. The assert statement is used incorrectly, e.g., to validate inputs.
SWC-109	Uninitialized storage pointer	CWE-824: Access of uninitialized pointer	Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to intentional or unintentional vulnerabilities.
SWC-108	State variable default visibility	CWE-710: Improper adherence to coding standards	Labelling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.
SWC-107	Reentrancy	CWE-841: Improper enforcement of behavioural workflow	One of the major dangers of calling external contracts is that they can take over the control flow. In the reentrancy attack (a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.
SWC-106	Unprotected SELFDESTRUCT instruction	CWE-284: Improper access control	Due to missing or insufficient access controls, malicious parties can self-destruct the contract.
SWC-105	Unprotected ether withdrawal	CWE-284: Improper access control	Due to missing or insufficient access controls, malicious parties can withdraw some or all ether from the contract account.
SWC-104	Unchecked call return value	CWE-252: Unchecked return value	The return value of a message call is not checked. Execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic.
SWC-103	Floating pragma	CWE-664: Improper control of a resource through its lifetime	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
SWC-102	Outdated compiler version	CWE-937: Using components with known vulnerabilities	Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.
SWC-101	Integer overflow and underflow	CWE-682: Incorrect calculation	An overflow / underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance, if a number is stored in the uint8 type, it means that

ID	Title	Related CWE	Description
			the number is stored in an 8 bits unsigned number ranging from 0 to 2^8-1. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.
SWC-100	Function default visibility	CWE-710: Improper adherence to coding standards	Functions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgets to set the visibility and a malicious user is able to make unauthorized or unintended state changes.

Bibliography

[b-ITU-T X.1040] Recommendation ITU-T X.1040 (2017), *Security reference architecture for lifecycle management of e-commerce business data*.

[b-ITU-T X.1252] Recommendation ITU-T X.1252 (2021), *Baseline identity management terms and definitions*.

[b-ITU-T X.1400] Recommendation ITU-T X.1400 (2020), *Terms and definitions for distributed ledger technology*.

[b-IEEE-scs] Huang, Y., Bian, Y., Li, R., Zhao, J.L., and Shi, P. (2019), *Smart contract security: a software lifecycle perspective*. IEEE Access.
<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8864988>

[b-Abdellatif] Abdellatif, T. and Brousmiche, K.-L. (2018), *Formal verification of smart contracts based on users and blockchain behaviors models*.
<https://www.researchgate.net/publication/324175498_Formal_Verification_of_Smart_Contracts_Based_on_Users_and_Blockchain_Behaviors_Models>

[b-ACM] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F. and Vechev, M. (2018), *SECURIFY: Practical security analysis of smart contracts*. In Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Toronto, pp. 67-82.
<https://dl.acm.org/doi/10.1145/3243734.3243780>

[b-ARXIV] Praitheeshan, P., Pan, L., Yu, J., Liu, J. and Doss R. (2019), *Security analysis methods on Ethereum smart contract vulnerabilities: A survey*. arXiv:1908.08605.
<https://arxiv.org/abs/1908.08605>

[b-EIP-1470] G. Wagner (2018), *EIP-1470: Smart Contract Weakness Classification (SWC)*. Ethereum Improvement Proposals.
<https://eips.ethereum.org/EIPS/eip-1470>

[b-Kaiser] Kaiser, T. (2019), *Chaincode Scanner: Automated Security Analysis of Chaincode*. ChainSecurity.
<https://www.youtube.com/watch?v=aFRfojcctDY>

[b-SWCR] SWC Registry (2020), *Smart Contract Weakness Classification and Test Cases*.
<https://swcregistry.io>

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks, open system communications and security**
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	Languages and general software aspects for telecommunication systems

Published in Switzerland
Geneva, 2023