**Security Pitfalls & Best Practices 101**

101 Security Pitfalls & Best Practices

1. **Solidity versions**: Using very old versions of Solidity prevents benefits of bug fixes and newer security checks. Using the latest versions might make contracts susceptible to undiscovered compiler bugs. Consider using one of these versions: *0.7.5, 0.7.6 or 0.8.4 .*(see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity))
2. **Unlocked pragma**: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using *^* in *pragma solidity 0.5.10)* ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. (see [here](https://swcregistry.io/docs/SWC-103))
3. **Multiple Solidity pragma**: It is better to use one Solidity compiler version across all contracts instead of different versions with different bugs and security checks. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used))
4. **Incorrect access control**: Contract functions executing critical logic should have appropriate access control enforced via address checks (e.g. owner, controller etc.) typically in modifiers. Missing checks allow attackers to control critical logic. (see [here](https://docs.openzeppelin.com/contracts/3.x/api/access) and [here](https://dasp.co/#item-2))
5. **Unprotected withdraw function**: Unprotected (*external*/*public*) function calls sending Ether/tokens to user-controlled addresses may allow users to withdraw unauthorized funds. (see [here](https://swcregistry.io/docs/SWC-105))
6. **Unprotected call to *selfdestruct***: A user/attacker can mistakenly/intentionally kill the contract. Protect access to such functions. (see [here](https://swcregistry.io/docs/SWC-106))
7. **Modifier side-effects:**Modifiers should only implement checks and not make state changes and external calls which violates the [checks-effects-interactions](https://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern) pattern. These side-effects may go unnoticed by developers/auditors because the modifier code is typically far from the function implementation. (see [here](https://consensys.net/blog/blockchain-development/solidity-best-practices-for-smart-contract-security/))
8. **Incorrect modifier**: If a modifier does not execute*\_*or *revert*, the function using that modifier will return the default value causing unexpected behavior. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-modifier))
9. **Constructor names:**Before *solc 0.4.22*, constructor names had to be the same name as the contract class containing it. Misnaming it wouldn’t make it a constructor which has security implications. *Solc 0.4.22* introduced the *constructor* keyword. Until *solc 0.5.0*, contracts could have both old-style and new-style constructor names with the first defined one taking precedence over the second if both existed, which also led to security issues. *Solc 0.5.0* forced the use of the *constructor* keyword. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#multiple-constructor-schemes) and [here](https://swcregistry.io/docs/SWC-118))
10. **Void constructor:**Calls to base contract constructors that are unimplemented leads to misplaced assumptions. Check if the constructor is implemented or remove call if not. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#void-constructor))
11. **Implicit constructor callValue check**: The creation code of a contract that does not define a constructor but has a base that does, did not revert for calls with non-zero callValue when such a constructor was not explicitly payable. This is due to a compiler bug introduced in *v0.4.5* and fixed in *v0.6.8*. Starting from Solidity 0.4.5 the creation code of contracts without explicit payable constructor is supposed to contain a callvalue check that results in contract creation reverting, if non-zero value is passed. However, this check was missing in case no explicit constructor was defined in a contract at all, but the contract has a base that does define a constructor. In these cases it is possible to send value in a contract creation transaction or using inline assembly without revert, even though the creation code is supposed to be non-payable. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
12. **Controlled delegatecall:***delegatecall()* or *callcode()* to an address controlled by the user allows execution of malicious contracts in the context of the caller’s state. Ensure trusted destination addresses for such calls. (see [here](https://swcregistry.io/docs/SWC-112))
13. **Reentrancy vulnerabilities**: Untrusted external contract calls could callback leading to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards. (see [here](https://swcregistry.io/docs/SWC-107))
14. **ERC777 callbacks and reentrancy:**ERC777 tokens allow arbitrary callbacks via hooks that are called during token transfers. Malicious contract addresses may cause reentrancy on such callbacks if reentrancy guards are not used. (see [here](https://quantstamp.com/blog/how-the-dforce-hacker-used-reentrancy-to-steal-25-million))
15. **Avoid *transfer()*/*send()*as reentrancy mitigations**: Although *transfer()* and *send()* have been recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts. Use *call()* instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection. (see [here](https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/) and [here](https://swcregistry.io/docs/SWC-134))
16. **Private on-chain data**: Marking variables *private* does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain. (see [here](https://swcregistry.io/docs/SWC-136))
17. **Weak PRNG**: PRNG relying on *block.timestamp*, *now* or *blockhash*can be influenced by miners to some extent and should be avoided. (see [here](https://swcregistry.io/docs/SWC-120))
18. **Block values as time proxies:***block.timestamp* and *block.number*are not good proxies (i.e. representations, not to be confused with smart contract proxy/implementation pattern) for time because of issues with synchronization, miner manipulation and changing block times. (see [here](https://swcregistry.io/docs/SWC-116))
19. **Integer overflow/underflow**: Not using OpenZeppelin’s SafeMath (or similar libraries) that check for overflows/underflows may lead to vulnerabilities or unexpected behavior if user/attacker can control the integer operands of such arithmetic operations. *Solc v0.8.0* introduced default overflow/underflow checks for all arithmetic operations. (see [here](https://swcregistry.io/docs/SWC-101) and [here](https://blog.soliditylang.org/2020/10/28/solidity-0.8.x-preview/))
20. **Divide before multiply:**Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply))
21. **Transaction order dependence:**Race conditions can be forced on specific Ethereum transactions by monitoring the mempool. For example, the classic ERC20 *approve()* change can be front-run using this method. Do not make assumptions about transaction order dependence. (see [here](https://swcregistry.io/docs/SWC-114))
22. **ERC20 *approve()* race condition:**Use *safeIncreaseAllowance()* and *safeDecreaseAllowance()* from OpenZeppelin’s *SafeERC20* implementation to prevent race conditions from manipulating the allowance amounts. (see [here](https://swcregistry.io/docs/SWC-114))
23. **Signature malleability**: The *ecrecover* function is susceptible to signature malleability which could lead to replay attacks. Consider using OpenZeppelin’s[ECDSA library](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol). (see [here](https://swcregistry.io/docs/SWC-117), [here](https://swcregistry.io/docs/SWC-121) and [here](https://medium.com/cryptronics/signature-replay-vulnerabilities-in-smart-contracts-3b6f7596df57))
24. **ERC20 transfer() does not return boolean:**Contracts compiled with *solc >= 0.4.22* interacting with such functions will revert. Use OpenZeppelin’s SafeERC20 wrappers. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-erc20-interface) and [here](https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca))
25. **Incorrect return values for ERC721 *ownerOf()*:**Contracts compiled with *solc >= 0.4.22* interacting with ERC721 *ownerOf()* that returns a *bool* instead of *address* type will revert. Use OpenZeppelin’s ERC721 contracts. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-erc721-interface))
26. **Unexpected Ether and this.balance**: A contract can receive Ether via *payable* functions, *selfdestruct(), coinbase*transaction or pre-sent before creation. Contract logic depending on *this.balanc*e can therefore be manipulated. (see [here](https://github.com/sigp/solidity-security-blog#3-unexpected-ether-1) and [here](https://swcregistry.io/docs/SWC-132))
27. ***fallback* vs *receive()***: Check that all precautions and subtleties of *fallback*/*receive* functions related to visibility, state mutability and Ether transfers have been considered.  (see [here](https://docs.soliditylang.org/en/latest/contracts.html#fallback-function) and [here](https://docs.soliditylang.org/en/latest/contracts.html#receive-ether-function))
28. **Dangerous strict equalities:**Use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using *>=* or *<=* instead of *==* for such variables depending on the contract logic. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities))
29. **Locked Ether**: Contracts that accept Ether via *payable* functions but without withdrawal mechanisms will lock up that Ether. Remove *payable* attribute or add withdraw function. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether))
30. **Dangerous usage of *tx.origin***: Use of *tx.origin* for authorization may be abused by a MITM malicious contract forwarding calls from the legitimate user who interacts with it. Use *msg.sender* instead. (see [here](https://swcregistry.io/docs/SWC-115))
31. **Contract check:**Checking if a call was made from an Externally Owned Account (EOA) or a contract account is typically done using *extcodesize* check which may be circumvented by a contract during construction when it does not have source code available. Checking if *tx.origin == msg.sender*is another option. Both have implications that need to be considered. (see [here](https://consensys.net/blog/blockchain-development/solidity-best-practices-for-smart-contract-security/))
32. **Deleting a *mapping* within a *struct***: Deleting a *struct* that contains a *mapping* will not delete the *mapping* contents which may lead to unintended consequences. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#deletion-on-mapping-containing-a-structure))
33. **Tautology or contradiction:**Tautologies (always true) or contradictions (always false) indicate potential flawed logic or redundant checks. e.g. *x >= 0* which is always true if *x* is *uint*. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#tautology-or-contradiction))
34. **Boolean constant**: Use of Boolean constants (*true*/*false*) in code (e.g. conditionals) is indicative of flawed logic. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#misuse-of-a-boolean-constant))
35. **Boolean equality**: Boolean variables can be checked within conditionals directly without the use of equality operators to *true*/*false*. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality))
36. **State-modifying functions**: Functions that modify state (in assembly or otherwise) but are labelled *constant*/*pure*/*view* revert in *solc >=0.5.0* (work in prior versions) because of the use of *STATICCALL* opcode. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#constant-functions-using-assembly-code))
37. **Return values of low-level calls**: Ensure that return values of low-level calls (*call*/*callcode*/*delegatecall*/*send*/etc.) are checked to avoid unexpected failures. (see [here](https://swcregistry.io/docs/SWC-104))
38. **Account existence check for low-level calls**: Low-level calls *call*/*delegatecall*/*staticcall* return true even if the account called is non-existent (per EVM design). Account existence must be checked prior to calling if needed. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls))
39. **Dangerous shadowing:**Local variables, state variables, functions, modifiers, or events with names that shadow (i.e. override) builtin Solidity symbols e.g. *now*or other declarations from the current scope are misleading and may lead to unexpected usages and behavior. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#builtin-symbol-shadowing))
40. **Dangerous state variable shadowing**: Shadowing state variables in derived contracts may be dangerous for critical variables such as contract owner(for e.g. where modifiers in base contracts check on base variables but shadowed variables are set mistakenly) and contracts incorrectly use base/shadowed variables. Do not shadow state variables. (see [here](https://swcregistry.io/docs/SWC-119))
41. **Pre-declaration usage of local variables**: Usage of a variable before its declaration (either declared later or in another scope) leads to unexpected behavior in *solc < 0.5.0*but *solc >= 0.5.0* implements C99-style scoping rules where variables can only be used after they have been declared and only in the same or nested scopes. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#pre-declaration-usage-of-local-variables))
42. **Costly operations inside a loop**: Operations such as state variable updates (use SSTOREs) inside a loop cost a lot of gas, are expensive and may lead to out-of-gas errors. Optimizations using local variables are preferred. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop))
43. **Calls inside a loop:**Calls to external contracts inside a loop are dangerous (especially if the loop index can be user-controlled) because it could lead to DoS if one of the calls reverts or execution runs out of gas. Avoid calls within loops, check that loop index cannot be user-controlled or is bounded. (see [here](https://swcregistry.io/docs/SWC-113))
44. **DoS with block gas limit**: Programming patterns such as looping over arrays of unknown size may lead to DoS when the gas cost of execution exceeds the block gas limit. (see [here](https://swcregistry.io/docs/SWC-128))
45. **Missing events**: Events for critical state changes (e.g. owner and other critical parameters) should be emitted for tracking this off-chain. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-access-control))
46. **Unindexed event parameters**: Parameters of certain events are expected to be indexed (e.g. ERC20 Transfer/Approval events) so that they’re included in the block’s bloom filter for faster access. Failure to do so might confuse off-chain tooling looking for such indexed events. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#unindexed-erc20-event-oarameters))
47. **Incorrect event signature in libraries**: Contract types used in events in libraries cause an incorrect event signature hash. Instead of using the type `address` in the hashed signature, the actual contract name was used, leading to a wrong hash in the logs. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.8*. (see [here](https://docs.soliditylang.org/en/v0.8.1/bugs.html))
48. **Dangerous unary expressions**: Unary expressions such as *x =+ 1* are likely errors where the programmer really meant to use *x += 1*. Unary *+* operator was deprecated in *solc v0.5.0*. (see [here](https://swcregistry.io/docs/SWC-129))
49. **Missing zero address validation**: Setters of address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation))
50. **Critical address change**: Changing critical addresses in contracts should be a two-step process where the first transaction (from the old/current address) registers the new address (i.e. grants ownership) and the second transaction (from the new address) replaces the old address with the new one (i.e. claims ownership). This gives an opportunity to recover from incorrect addresses mistakenly used in the first step. If not, contract functionality might become inaccessible. (see [here](https://github.com/OpenZeppelin/openzeppelin-contracts/issues/1488) and [here](https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2369))
51. **assert()/require() state change**: Invariants in *assert()* and *require()* statements should not modify the state per best practices. (see [here](https://swcregistry.io/docs/SWC-110))
52. ***require()* vs *assert()*:***require()* should be used for checking error conditions on inputs and return values while *assert()* should be used for invariant checking. Between *solc 0.4.10*and *0.8.0*, *require()* used *REVERT* (*0xfd*) opcode which refunded remaining gas on failure while *assert()* used INVALID (*0xfe*) opcode which consumed all the supplied gas. (see [here](https://docs.soliditylang.org/en/v0.8.1/control-structures.html#error-handling-assert-require-revert-and-exceptions))
53. **Deprecated keywords**: Use of deprecated functions/operators such as *block.blockhash()* for *blockhash()*, *msg.gas* for *gasleft(), throw* for *revert()*, *sha3()* for *keccak256()*, *callcode()* for *delegatecall(),* *suicide()* for *selfdestruct(), constant*for*view*or*var*for*actual type name* should be avoided to prevent unintended errors with newer compiler versions. (see [here](https://swcregistry.io/docs/SWC-111))
54. **Function default visibility***:*Functions without a visibility type specifier are *public* by default in *solc < 0.5.0*. This can lead to a vulnerability where a malicious user may make unauthorized state changes. *solc >= 0.5.0* requires explicit function visibility specifiers. (see [here](https://swcregistry.io/docs/SWC-100))
55. **Incorrect inheritance order**: Contracts inheriting from multiple contracts with identical functions should specify the correct inheritance order i.e. more general to more specific to avoid inheriting the incorrect function implementation. (see [here](https://swcregistry.io/docs/SWC-125))
56. **Missing inheritance**: A contract might appear (based on name or functions implemented) to inherit from another interface or abstract contract without actually doing so. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#missing-inheritance))
57. **Insufficient gas griefing**: Transaction relayers need to be trusted to provide enough gas for the transaction to succeed. (see [here](https://swcregistry.io/docs/SWC-126))
58. **Modifying reference type parameters**: Structs/Arrays/Mappings passed as arguments to a function may be by value (memory) or reference (storage) as specified by the data location (optional before *solc 0.5.0*). Ensure correct usage of memory and storage in function parameters and make all data locations explicit. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#modifying-storage-array-by-value))
59. **Arbitrary jump with function type variable:**Function type variables should be carefully handled and avoided in assembly manipulations to prevent jumps to arbitrary code locations. (see [here](https://swcregistry.io/docs/SWC-127))
60. **Hash collisions with multiple variable length arguments**: Using *abi.encodePacked()* with multiple variable length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in *abi.encodePacked()*, use fixed length arrays or use *abi.encode()*. (see [here](https://swcregistry.io/docs/SWC-133) and [here](https://docs.soliditylang.org/en/v0.5.3/abi-spec.html#non-standard-packed-mode))
61. **Malleability risk from dirty high order bits**: Types that do not occupy the full 32 bytes might contain “dirty higher order bits” which does not affect operation on types but gives different results with *msg.data*. (see [here](https://docs.soliditylang.org/en/v0.8.1/security-considerations.html#minor-details))
62. **Incorrect shift in assembly**: Shift operators (*shl(x, y)*, *shr(x, y)*, *sar(x, y)*) in Solidity assembly apply the shift operation of *x* bits on *y*and not the other way around, which may be confusing. Check if the values in a shift operation are reversed. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-shift-in-assembly))
63. **Assembly usage**: Use of EVM assembly is error-prone and should be avoided or double-checked for correctness. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage))
64. **Right-To-Left-Override control character (U+202E)**: 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. U+202E character should not appear in the source code of a smart contract. (see [here](https://swcregistry.io/docs/SWC-130))
65. **Constant state variables**: Constant state variables should be declared constant to save gas. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant))
66. **Similar variable names**: Variables with similar names could be confused for each other and therefore should be avoided. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar))
67. **Uninitialized state/local variables**: Uninitialized state/local variables are assigned zero values by the compiler and may cause unintended results e.g. transferring tokens to zero address. Explicitly initialize all state/local variables. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables) and [here](https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables))
68. **Uninitialized storage pointers:**Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to vulnerabilities. *Solc 0.5.0* and above disallow such pointers. (see [here](https://swcregistry.io/docs/SWC-109))
69. **Uninitialized function pointers in constructors:**Calling uninitialized function pointers in constructors of contracts compiled with *solc* versions *0.4.5-0.4.25* and *0.5.0-0.5.7* lead to unexpected behavior because of a compiler bug. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-function-pointers-in-constructors))
70. **Long number literals**: Number literals with many digits should be carefully checked as they are prone to error. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits))
71. **Out-of-range enum:***Solc < 0.4.5*produced unexpected behavior with out-of-range enums*.*Check enum conversion or use a newer compiler.(see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-enum-conversion))
72. **Uncalled public functions**: *Public* functions that are never called from within the contracts should be declared *external* to save gas. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external))
73. **Dead/Unreachable code**: Dead code may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see [here](https://en.wikipedia.org/wiki/Dead_code))
74. **Unused return values**: Unused return values of function calls are indicative of programmer errors which may have unexpected behavior. (see [here](https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return))
75. **Unused variables**: Unused state/local variables may be indicative of programmer error, missing logic or potential optimization opportunity, which needs to be flagged for removal or addressed appropriately. (see [here](https://swcregistry.io/docs/SWC-131))
76. **Redundant statements**: Statements with no effects that do not produce code may be indicative of programmer error or missing logic, which needs to be flagged for removal or addressed appropriately. (see [here](https://swcregistry.io/docs/SWC-135))
77. **Storage array with signed Integers with ABIEncoderV2**: Assigning an array of signed integers to a storage array of different type can lead to data corruption in that array. This is due to a compiler bug introduced in *v0.4.7* and fixed in *v0.5.10*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
78. **Dynamic constructor arguments clipped with ABIEncoderV2**: A contract's constructor which takes structs or arrays that contain dynamically sized arrays reverts or decodes to invalid data when ABIEncoderV2 is used. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.9*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
79. **Storage array with multiSlot element with ABIEncoderV2**: Storage arrays containing structs or other statically sized arrays are not read properly when directly encoded in external function calls or in *abi.encode()*. This is due to a compiler bug introduced in *v0.4.16* and fixed in *v0.5.10*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
80. **Calldata structs with statically sized and dynamically encoded members with ABIEncoderV2**: Reading from calldata structs that contain dynamically encoded, but statically sized members can result in incorrect values. This is due to a compiler bug introduced in *v0.5.6* and fixed in *v0.5.11*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
81. **Packed storage with ABIEncoderV2:**Storage structs and arrays with types shorter than 32 bytes can cause data corruption if encoded directly from storage using ABIEncoderV2. This is due to a compiler bug introduced in *v0.5.0* and fixed in *v0.5.7*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
82. **Incorrect loads with Yul optimizer and ABIEncoderV2**: The Yul optimizer incorrectly replaces *MLOAD* and *SLOAD* calls with values that have been previously written to the load location. This can only happen if ABIEncoderV2 is activated and the experimental Yul optimizer has been activated manually in addition to the regular optimizer in the compiler settings. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.5.15*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
83. **Array slice dynamically encoded base type with ABIEncoderV2**: Accessing array slices of arrays with dynamically encoded base types (e.g. multi-dimensional arrays) can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.0* and fixed in *v0.6.8*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
84. **Missing escaping in formatting with ABIEncoderV2:**String literals containing double backslash characters passed directly to external or encoding function calls can lead to a different string being used when ABIEncoderV2 is enabled. This is due to a compiler bug introduced in *v0.5.14* and fixed in *v0.6.8*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
85. **Double shift size overflow**: Double bitwise shifts by large constants whose sum overflows 256 bits can result in unexpected values. Nested logical shift operations whose total shift size is *2\*\*256* or more are incorrectly optimized. This only applies to shifts by numbers of bits that are compile-time constant expressions. This happens when the optimizer is used and *evmVersion >= Constantinople.*This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.6*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
86. **Incorrect byte instruction optimization:**The optimizer incorrectly handles byte opcodes whose second argument is 31 or a constant expression that evaluates to 31. This can result in unexpected values. This can happen when performing index access on *bytesNN* types with a compile time constant value (not index) of 31 or when using the byte opcode in inline assembly. This is due to a compiler bug introduced in *v0.5.5* and fixed in *v0.5.7*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
87. **Essential assignments removed with Yul Optimizer** : The Yul optimizer can remove essential assignments to variables declared inside *for* loops when Yul's *continue* or *break* statement is used mostly while using inline assembly with *for* loops and *continue* and *break* statements. This is due to a compiler bug introduced in *v0.5.8*/*v0.6.0* and fixed in *v0.5.16*/*v0.6.1*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
88. **Private methods overridden**: While private methods of base contracts are not visible and cannot be called directly from the derived contract, it is still possible to declare a function of the same name and type and thus change the behaviour of the base contract's function. This is due to a compiler bug introduced in *v0.3.0* and fixed in *v0.5.17*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
89. **Tuple assignment multi stack slot components**: Tuple assignments with components that occupy several stack slots, i.e. nested tuples, pointers to external functions or references to dynamically sized calldata arrays, can result in invalid values. This is due to a compiler bug introduced in *v0.1.6* and fixed in *v0.6.6*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
90. **Dynamic array cleanup**: When assigning a dynamically sized array with types of size at most 16 bytes in storage causing the assigned array to shrink, some parts of deleted slots were not zeroed out. This is due to a compiler bug fixed in *v0.7.3*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
91. **Empty byte array copy**: Copying an empty byte array (or string) from memory or calldata to storage can result in data corruption if the target array's length is increased subsequently without storing new data. This is due to a compiler bug fixed in *v0.7.4*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
92. **Memory array creation overflow**: The creation of very large memory arrays can result in overlapping memory regions and thus memory corruption. This is due to a compiler bug introduced in *v0.2.0* and fixed in *v0.6.5*. (see [here](https://solidity.ethereum.org/2020/04/06/memory-creation-overflow-bug/))
93. **Calldata** ***using for***: Function calls to internal library functions with calldata parameters called via “*using for”* can result in invalid data being read. This is due to a compiler bug introduced in *v0.6.9* and fixed in *v0.6.10*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
94. **Free function redefinition**: The compiler does not flag an error when two or more free functions (functions outside of a contract) with the same name and parameter types are defined in a source unit or when an imported free function alias shadows another free function with a different name but identical parameter types. This is due to a compiler bug introduced in *v0.7.1* and fixed in *v0.7.2*. (see [here](https://docs.soliditylang.org/en/v0.8.9/bugs.html))
95. **Unprotected initializers in proxy-based upgradeable contracts**: Proxy-based upgradeable contracts need to use *public* initializer functions instead of constructors that need to be explicitly called only once. Preventing multiple invocations of such initializer functions (e.g. via *initializer* modifier from OpenZeppelin’s *Initializable* library) is a must. (see [here](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#initializers) and [here](https://github.com/crytic/slither/wiki/Upgradeability-Checks#initializer-is-not-called))
96. **Initializing state-variables in proxy-based upgradeable contracts**: This should be done in initializer functions and not as part of the state variable declarations in which case they won’t be set. (see [here](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#avoid-initial-values-in-field-declarations))
97. **Import upgradeable contracts in proxy-based upgradeable contracts**: Contracts imported from proxy-based upgradeable contracts should also be upgradeable where such contracts have been modified to use initializers instead of constructors. (see [here](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#use-upgradeable-libraries))
98. **Avoid *selfdestruct* or *delegatecall* in proxy-based upgradeable contracts**: This will cause the logic contract to be destroyed and all contract instances will end up delegating calls to an address without any code. (see [here](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#potentially-unsafe-operations))
99. **State variables in proxy-based upgradeable contracts**: The declaration order/layout and type/mutability of state variables in such contracts should be preserved exactly while upgrading to prevent critical storage layout mismatch errors. (see [here](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#modifying-your-contracts))
100. **Function ID collision between proxy/implementation in proxy-based upgradeable contracts**: Malicious proxy contracts may exploit function ID collision to invoke unintended proxy functions instead of delegating to implementation functions. Check for function ID collisions. (see [here](https://github.com/crytic/slither/wiki/Upgradeability-Checks#functions-ids-collisions) and [here](https://forum.openzeppelin.com/t/beware-of-the-proxy-learn-how-to-exploit-function-clashing/1070))
101. **Function shadowing between proxy/contract in proxy-based upgradeable contracts**: Shadow functions in proxy contract prevent functions in logic contract from being invoked. (see [here](https://github.com/crytic/slither/wiki/Upgradeability-Checks#functions-shadowing))

# Ethereum 101

### 101 key aspects of Ethereum

1. Ethereum is “A Next-Generation Smart Contract and Decentralized Application Platform” (See [here](https://ethereum.org/en/whitepaper/))
2. Ethereum is a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. (See [here](https://ethereum.org/en/whitepaper/))
3. Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system’s state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs. It is often described as "the world computer.” (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
4. The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing high availability, auditability, transparency, and neutrality, it also reduces or eliminates censorship and reduces certain counterparty risks. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
5. Ethereum’s purpose is not primarily to be a digital currency payment network. While the digital currency ether is both integral to and necessary for the operation of Ethereum, ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
6. Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code of arbitrary and unbounded complexity. Where Bitcoin’s Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum’s language is Turing complete, meaning that Ethereum can straightforwardly function as a general-purpose computer. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
7. The original blockchain, namely Bitcoin’s blockchain, tracks the state of units of bitcoin and their ownership. You can think of Bitcoin as a distributed consensus state machine, where transactions cause a global state transition, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined. Ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store, i.e., a store that can hold any data expressible as a key–value tuple. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
8. Ethereum’s core components (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc)):
   1. P2P network: Ethereum runs on the Ethereum main network, which is addressable on TCP port 30303, and runs a protocol called ÐΞVp2p.
   2. Transactions: Ethereum transactions are network messages that include (among other things) a sender, recipient, value, and data payload.
   3. State machine: Ethereum state transitions are processed by the Ethereum Virtual Machine (EVM), a stack-based virtual machine that executes bytecode (machine-language instructions). EVM programs, called "smart contracts," are written in high-level languages (e.g., Solidity or Vyper) and compiled to bytecode for execution on the EVM.
   4. Data structures: Ethereum’s state is stored locally on each node as a database (usually Google’s LevelDB), which contains the transactions and system state in a serialized hashed data structure called a Merkle Patricia Tree.
9. Ethereum’s core components (continued):
   1. Consensus algorithm: Ethereum uses Bitcoin’s consensus model, Nakamoto Consensus, which uses sequential single-signature blocks, weighted in importance by Proof-of-Work (PoW) to determine the longest chain and therefore the current state.
   2. However, this is being transitioned to a Proof-of-Stake (PoS) algorithm in Ethereum 2.0.
   3. Economic security: Ethereum currently uses a PoW algorithm called Ethash, but this is being transitioned to a PoS algorithm in Ethereum 2.0.
   4. Clients: Ethereum has several interoperable implementations of the client software, the most prominent of which are Go-Ethereum (Geth) and OpenEthereum. The others are Erigon, Nethermind and Turbo-geth. OpenEthereum is being deprecated to transition to Erigon, which is the former Turbo-geth. (See [here](https://www.ethernodes.org/))
10. Ethereum’s ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing-complete system. Turing-complete systems face the challenge of the halting problem i.e. given an arbitrary program and its input, it is not solvable to determine whether the program will eventually stop running. So Ethereum cannot predict if a smart contract will terminate, or how long it will run. Therefore, to constrain the resources used by a smart contract, Ethereum introduces a metering mechanism called gas. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
11. As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a predetermined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of what can be consumed running the smart contract. The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing-complete computation while limiting the resources that any program can consume. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
12. Ether needs to be sent along with a transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable gas price. Just like at the pump, the price of gas is not fixed. Gas is purchased for the transaction, the computation is executed, and any unused gas is refunded back to the sender of the transaction. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/01what-is.asciidoc))
13. A Decentralized Application, abbreviated as ÐApp, is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services and typically combines smart contracts with a web interface.
14. ÐApps represent a transition from “Web 2.0” where applications are centrally owned and managed to “Web 3.0” where applications are built on decentralised peer-to-peer protocols for compute (i.e. blockchain), storage and messaging.
15. Ethereum blockchain represents the decentralized compute part of Web 3.0. Swarm represents the decentralized storage and Whisper (now Waku) represents the decentralized messaging protocol.
16. Decentralization can be considered as three types (See [here](https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274)):
    1. Architectural decentralization
    2. Political decentralization
    3. Logical decentralization
17. Ethereum’s currency unit is called ether or “ETH.” Ether is subdivided into smaller units and the smallest unit is named wei. 10\*\*3 wei is 1 Babbage, 10\*\*6 wei is 1 Lovelace, 10\*\*9 wei is 1 Shannon and 10\*\*18 wei is 1 Ether.
18. Ethereum uses public key cryptography to create public–private key pairs (considered a "pair" because the public key is derived from the private key) which are not used for encryption but for digital signatures.
19. Ethereum uses Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signatures (SECP-256k1 curve) which is based on Elliptic-curve cryptography (ECC), an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields.  (See [here](https://en.wikipedia.org/wiki/Elliptic-curve_cryptography))
20. An Ethereum private key is a 256-bit random number that uniquely determines a single Ethereum address also known as an account
21. An Ethereum public key is a point on an elliptic curve calculated from the private key using elliptic curve multiplication. One cannot calculate the private key from the public key.
22. Ethereum state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. (See [here](https://ethereum.org/en/whitepaper/#ethereum-accounts))
23. Ethereum account contains four fields:
    1. The nonce, a counter used to make sure each transaction can only be processed once
    2. The account's current ether balance
    3. The account's contract code, if present
    4. The account's storage (empty by default)
24. Ethereum has two different types of accounts:
    1. Externally Owned Accounts (EOAs) controlled by private keys
    2. Contract Accounts controlled by their contract code
25. Ownership of ether by EOAs is established through private keys, Ethereum addresses, and digital signatures. Anyone with a private key has control of the corresponding EOA account and any ether it holds.
26. An EOA has no code, and one can send messages from an EOA by creating and signing a transaction
27. A contract account has code and associated storage and every time it receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.
28. Smart contracts can be thought of as "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to keep track of persistent variables.
29. Ethereum uses Keccak-256 as its cryptographic hash function. Keccak-256 was the winning candidate for the SHA-3 competition held by NIST but is different from the finally adopted SHA-3 standard. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/04keys-addresses.asciidoc))
30. Ethereum address of an EOA account is the last 20 bytes (least significant bytes) of the Keccak-256 hash of the public key of the EOA’s key pair.
31. Transactions are signed messages originated by an externally owned account (EOA), transmitted by the Ethereum network, and recorded on the Ethereum blockchain. Only transactions can trigger a change of state. Ethereum is a transaction-based state machine. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/06transactions.asciidoc))
32. Transaction properties:
    1. Atomic: it is all or nothing i.e. cannot be divided or interrupted by other transactions
    2. Serial: Transactions are processed sequentially one after the other without any overlapping by other transactions
    3. Inclusion: Transaction inclusion is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine inclusion.
    4. Order: Transaction order is not guaranteed and depends on network congestion and gasPrice among other things. Miners determine order.
33. A transaction is a serialized binary message that contains the following components (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/06transactions.asciidoc)):
    1. nonce: A sequence number, issued by the originating EOA, used to prevent message replay
    2. gasPrice: The amount of ether (in wei) that the originator is willing to pay for each unit of gas
    3. gasLimit: The maximum amount of gas the originator is willing to pay for this transaction
    4. recipient: The destination Ethereum address
    5. value: The amount of ether (in wei) to send to the destination
    6. data: The variable-length binary data payload
    7. v,r,s: The three components of an ECDSA digital signature of the originating EOA
34. Nonce: A scalar value equal to the number of transactions sent from the EOA account or, in the case of Contract accounts, it is the number of contract-creations made by the account. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/06transactions.asciidoc#the-transaction-nonce))
35. Gas price: The price a transaction originator is willing to pay in exchange for gas. The price is measured in wei per gas unit. The higher the gas price, the faster the transaction is likely to be confirmed on the blockchain. The suggested gas price depends on the demand for block space at the time of the transaction.
36. Gas limit: The maximum number of gas units the transaction originator is willing to pay in order to complete the transaction
37. Recipient:  The 20-byte Ethereum address of the transaction’s recipient which can be an EOA or a Contract account.
    1. The Ethereum protocol does not validate recipient addresses in transactions. One can send a transaction to an address that has no corresponding private key or contract. Validation should be done at the user interface level.
    2. Note that there is no from address in the transaction because the EOA’s public key can be derived from the v,r,s components of the ECDSA signature and the transaction originator’s address can be derived from this public key
38. Value: The value of ether sent to the transaction recipient. If the recipient is an EOA then that account’s balance will be increased by this value. If the recipient is a contract address then the result depends on any data that is sent as part of this transaction. If there is no data, the recipient contract’s receive or fallback function is called if they are present. Depending on the implementation of those functions, the ether value is added to the contract account’s balance or an exception occurs and this ether remains with the originator’s account.
39. Data: The information (typically) sent to a contract account indicating the contract’s function to be called and the arguments to that function.
40. v,r,s: r and s are the two parts of the ECDSA signature produced by the transaction originator using the private key. v is the recovery identifier which is calculated as either one of 27 or 28, or as the chain ID (Ethereum mainnet chainID is 1) doubled plus 35 or 36. (See [here](https://github.com/ethereumbook/ethereumbook/blob/develop/06transactions.asciidoc#digital-signatures))
41. A digital signature serves three purposes in Ethereum: 1) proves that the owner of the private key, who is by implication the owner of an Ethereum account, has authorized the spending of ether, or execution of a contract 2) guarantees non-repudiation: the proof of authorization is undeniable 3) proves that the transaction data has not been and cannot be modified by anyone after the transaction has been signed.
42. Contract creation transactions are sent to a special destination address called the zero address i.e. 0x0. A contract creation transaction contains a data payload with the compiled bytecode to create the contract. An optional ether amount in the value field will create the new contract with a starting balance.
43. Transactions vs Messages:
    1. A transaction is produced by an EOA where an external actor sends a signed data package which either: 1) triggers a message to another EOA where it leads to a transfer of value or 2) triggers a message to a contract account where it leads to the recipient contract account running its code
    2. A message is either: 1) triggered by a transaction to another EOA or contract account or 2) triggered internally within the EVM by a contract account when it executes the CALL family of opcodes and leads to the recipient contract account running its code or value transfer to the recipient EOA
44. Transactions are grouped together into blocks. A blockchain contains a series of such blocks that are chained together.
45. Blocks: are batches of transactions with a hash of the previous block in the chain. This links blocks together (in a chain) because hashes are cryptographically derived from the block data. This prevents fraud, because one change in any block in history would invalidate all the following blocks as all subsequent hashes would change and everyone running the blockchain would notice. To preserve the transaction history, blocks are strictly ordered (every new block created contains a reference to its parent block), and transactions within blocks are strictly ordered as well. (See [here](https://ethereum.org/en/developers/docs/blocks/))
46. Ethereum node/client: A node is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum nodes. A client is a specific implementation of Ethereum node. The two most common client implementations are Geth and OpenEthereum. Ethereum transactions are sent to Ethereum nodes to be broadcast across the peer-to-peer network. (See [here](https://www.ethernodes.org/))
47. Miners: are entities running Ethereum nodes that validate and execute these transactions and combine them into blocks. The process of validating each block by having a miner provide a mathematical proof is known as a “proof of work.” Miners are rewarded for blocks accepted into the blockchain with a block reward in ether (currently 2 ETH). A miner also gets fees which is the ether spent on gas by all the transactions included in the block.
48. Block gas limit is set by miners and refers to the cap on the total amount of gas expended by all transactions in the block, which ensures that blocks can’t be arbitrarily large. Blocks therefore are not a fixed size in terms of the number of transactions because different transactions consume different amounts of gas. See [here](https://etherscan.io/chart/gaslimit) for historical block gas limits.
49. Blocks take time to propagate through the network and multiple miners are simultaneously producing valid blocks. This leads to the blockchain considering multiple blocks at the same level but ultimately choosing only one block at any level that creates the canonical blockchain. This choice is dictated by Ethereum’s Greedy Heaviest Observed Subtree (GHOST) protocol which includes stale blocks up to seven levels in the calculation of the longest chain. Stale blocks are called uncles or ommers.
50. Consensus: Decentralized consensus in the context of Ethereum refers to the process of determining which miner’s block should be appended next to the blockchain. This involves two key components of Proof-of-Work (PoW) and the Longest-chain Rule. Miners apply these rules to build on the canonical blockchain. This is referred to as "Nakamoto Consensus” and is adapted from Bitcoin.
51. State is a mapping between addresses and account states implemented as a modified Merkle Patricia tree or trie. A Merkle tree or trie is a type of binary tree composed of a set of nodes with:
    1. Leaf nodes at the bottom of the tree that contain the underlying data
    2. Intermediate nodes, where each node is the hash of its two child nodes
    3. A single root node formed from the hash of its two child nodes representing the top of the tree
52. Ethereum’s proof-of-work algorithm is called “Ethash” (previously known as Dagger-Hashimoto).
    1. The algorithm is formally defined as m = Hm ∧ n <= 2\*\*256/Hd with (m, n) = PoW(Hn’, Hn, d) where Hn’ is the new block’s header but without the nonce and mix-hash components; Hn is the nonce of the header; d is a large data set needed to compute the mixHash and Hd is the new block’s difficulty value
    2. PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudorandom number cryptographically dependent on H and d.
53. Blocks contain block header, transactions and ommers’ block headers. Block header contains (See [here](https://ethereum.github.io/yellowpaper/paper.pdf)):
    1. parentHash: The Keccak 256-bit hash of the parent block’s header, in its entirety
    2. ommersHash: The Keccak 256-bit hash of the ommers list portion of this block
    3. beneficiary: The 160-bit address to which all fees collected from the successful mining of this block be transferred
    4. stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
    5. transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block
    6. receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block
    7. logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list
    8. difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block’s difficulty level and the timestamp
    9. number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero;
    10. gasLimit: A scalar value equal to the current limit of gas expenditure per block
    11. gasUsed: A scalar value equal to the total gas used in transactions in this block
    12. timestamp: A scalar value equal to the reasonable output of Unix’s time() at this block’s inception
    13. extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer
    14. mixHash: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block
    15. nonce: A 64-bit value which, combined with the mixhash, proves that a sufficient amount of computation has been carried out on this block
54. stateRoot, transactionsRoot and receiptsRoot are 256-bit hashes of the root nodes of modified Merkle-Patricia trees. The leaves of stateRoot are key-value pairs of all Ethereum address-account pairs, where each respective account consists of:
    1. nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account
    2. balance: A scalar value equal to the number of Wei owned by this address
    3. storageRoot: A 256-bit hash of the root node of a modified Merkle-Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values.
    4. codeHash: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction.
55. Transaction receipt is a tuple of four items comprising:
    1. The cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened
    2. The set of logs created through execution of the transaction
    3. The Bloom filter composed from information in those logs
    4. The status code of the transaction
56. Gas refund and beneficiary: Any unused gas in a transaction (gasLimit minus gas used by the transaction) is refunded to the sender’s account at the same gasPrice. Ether used to purchase gas used for the transaction is credited to the beneficiary address (specified in the block header), the address of an account typically under the control of the miner. This is the transaction “fees” paid to the miner.
57. EVM is a quasi Turing complete machine where the quasi qualification comes from the fact that the computation is intrinsically bounded through a parameter, gas, which limits the total amount of computation done. EVM is the runtime environment for smart contracts.
58. The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes (hence called bytecode), where each byte represents an operation.
59. The EVM is a simple stack-based architecture consisting of the stack, volatile memory, non-volatile storage with a word size of 256-bit (chosen to facilitate the Keccak256 hash scheme and elliptic-curve computations) and Calldata.
60. Stack is made up of 1024 256-bit elements. EVM instructions can operate with the top 16 stack elements. Most EVM instructions operate with the stack (stack-based architecture) and there are also stack-specific operations e.g. PUSH, POP, SWAP, DUP etc.
61. Memory is a linear byte-array addressable at a byte-level and is volatile. All locations are well-defined initially as zero. This is accessed with MLOAD, MSTORE and MSTORE8 instructions.
62. Storage is a 256-bit to 256-bit key-value store. Unlike memory, which is volatile, storage is non-volatile and is maintained as part of the system state. All locations are well-defined initially as zero. This is accessed with SLOAD/SSTORE instructions.
63. Calldata is a read-only byte-addressable space where the data parameter of a transaction or call is held. This is accessed with CALLDATASIZE/CALLDATALOAD/CALLDATACOPY instructions.
64. EVM does not follow the standard von Neumann architecture. Rather than storing program code in generally accessible memory or storage, it is stored separately in a virtual ROM accessible only through a specialized instruction.
65. EVM uses big-endian ordering where the most significant byte of a word is stored at the smallest memory address and the least significant byte at the largest
66. EVM instruction set can be classified into 11 categories:
    1. Stop and Arithmetic Operations
    2. Comparison & Bitwise Logic Operations
    3. SHA3
    4. Environmental Information
    5. Block Information
    6. Stack, Memory, Storage and Flow Operations
    7. Push Operations
    8. Duplication Operations
    9. Exchange Operations
    10. Logging Operations
    11. System Operations
67. Stop and Arithmetic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x00 STOP 0 0 Halts execution
    2. 0x01 ADD 2 1 Addition operation
    3. 0x02 MUL 2 1 Multiplication operation
    4. 0x03 SUB 2 1 Subtraction operation
    5. 0x04 DIV 2 1 Integer division operation
    6. 0x05 SDIV 2 1 Signed integer division operation (truncated)
    7. 0x06 MOD 2 1 Modulo remainder operation
    8. 0x07 SMOD 2 1 Signed modulo remainder operation
    9. 0x08 ADDMOD 3 1 Modulo addition operation
    10. 0x09 MULMOD 3 1 Modulo multiplication operation
    11. 0x0a EXP 2 1 Exponential operation
    12. 0x0b SIGNEXTEND 2 1 Extend length of two’s complement signed integer
68. Comparison & Bitwise Logic Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x10 LT 2 1 Less-than comparison
    2. 0x11 GT 2 1 Greater-than comparison
    3. 0x12 SLT 2 1 Signed less-than comparison
    4. 0x13 SGT 2 1 Signed greater-than comparison
    5. 0x14 EQ 2 1 Equality comparison
    6. 0x15 ISZERO 1 1 Simple not operator
    7. 0x16 AND 2 1 Bitwise AND operation
    8. 0x17 OR 2 1 Bitwise OR operation
    9. 0x18 XOR 2 1 Bitwise XOR operation
    10. 0x19 NOT 1 1 Bitwise NOT operation
    11. 0x1a BYTE 2 1 Retrieve single byte from word
    12. 0x1b SHL 2 1 Left shift operation
    13. 0x1c SHR 2 1 Logical right shift operation
    14. 0x1d SAR 2 1 Arithmetic (signed) right shift operation
69. SHA3 (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x20 SHA3 2 1 Compute Keccak-256 hash
70. Environmental Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x30 ADDRESS 0 1 Get address of currently executing account
    2. 0x31 BALANCE 1 1 Get balance of the given account
    3. 0x32 ORIGIN 0 1 Get execution origination address
    4. 0x33 CALLER 0 1 Get caller address
    5. 0x34 CALLVALUE 0 1 Get deposited value by the instruction/transaction responsible for this execution
    6. 0x35 CALLDATALOAD 1 1 Get input data of current environment
    7. 0x36 CALLDATASIZE 0 1 Get size of input data in current environment
    8. 0x37 CALLDATACOPY 3 0 Copy input data in current environment to memory
    9. 0x38 CODESIZE 0 1 Get size of code running in current environment
    10. 0x39 CODECOPY 3 0 Copy code running in current environment to memory
    11. 0x3a GASPRICE 0 1 Get price of gas in current environment
    12. 0x3b EXTCODESIZE 1 1 Get size of an account’s code
    13. 0x3c EXTCODECOPY 4 0 Copy an account’s code to memory
    14. 0x3d RETURNDATASIZE 0 1 Get size of output data from the previous call from the current environment
    15. 0x3e RETURNDATACOPY 3 0 Copy output data from the previous call to memory
    16. 0x3f EXTCODEHASH 1 1 Get hash of an account’s code
71. Block Information (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x40 BLOCKHASH 1 1 Get the hash of one of the 256 most recent complete blocks
    2. 0x41 COINBASE 0 1 Get the block’s beneficiary address
    3. 0x42 TIMESTAMP 0 1 Get the block’s timestamp
    4. 0x43 NUMBER 0 1 Get the block’s number
    5. 0x44 DIFFICULTY 0 1 Get the block’s difficulty
    6. 0x45 GASLIMIT 0 1 Get the block’s gas limit
72. Stack, Memory, Storage and Flow Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x50 POP 1 0 Remove item from stack
    2. 0x51 MLOAD 1 1 Load word from memory
    3. 0x52 MSTORE 2 0 Save word to memory
    4. 0x53 MSTORE8 2 0 Save byte to memory
    5. 0x54 SLOAD 1 1 Load word from storage
    6. 0x55 SSTORE 2 0 Save word to storage
    7. 0x56 JUMP 1 0 Alter the program counter
    8. 0x57 JUMPI 2 0 Conditionally alter the program counter
    9. 0x58 PC 0 1 Get the value of the program counter prior to the increment corresponding to this instruction
    10. 0x59 MSIZE 0 1 Get the size of active memory in bytes
    11. 0x5a GAS 0 1 Get the amount of available gas, including the corresponding reduction for the cost of this instruction
    12. 0x5b JUMPDEST 0 0 Mark a valid destination for jumps. This operation has no effect on machine state during execution.
73. Push Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x60 PUSH1 0 1 Place 1 byte item on stack
    2. 0x61 PUSH2 0 1 Place 2-byte item on stack
    3. PUSH3, PUSH4, PUSH5…PUSH31 place 3, 4, 5..31 byte items on stack respectively
    4. 0x7f PUSH32 0 1 Place 32-byte (full word) item on stack
74. Duplication Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x80 DUP1 1 2 Duplicate 1st stack item
    2. DUP2, DUP3..DUP15 duplicate 2nd, 3rd..15th stack item respectively
    3. 0x8f DUP16 16 17 Duplicate 16th stack item
75. Exchange Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0x90 SWAP1 2 2 Exchange 1st and 2nd stack items
    2. 0x91 SWAP2 3 3 Exchange 1st and 3rd stack items
    3. SWAP3, SWAP4..SWAP15 exchange 1st and 4th..15th stack items respectively
    4. 0x9f SWAP16 17 17 Exchange 1st and 17th stack items
76. Logging Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0xa0 LOG0 2 0 Append log record with no topics
    2. 0xa1 LOG1 3 0 Append log record with one topic
    3. 0xa2 LOG2 4 0 Append log record with two topics
    4. 0xa3 LOG3 5 0 Append log record with three topics
    5. 0xa4 LOG4 6 0 Append log record with four topics
77. System Operations (Opcode, Mnemonic, Stack items removed, Stack items placed, Description):
    1. 0xf0 CREATE 3 1 Create a new account with associated code
    2. 0xf1 CALL 7 1 Message-call into an account
    3. 0xf2 CALLCODE 7 1 Message-call into this account with an alternative account’s code
    4. 0xf3 RETURN 2 0 Halt execution returning output dat
    5. 0xf4  DELEGATECALL 6 1 Message-call into this account with an alternative account’s code, but persisting the current values for sender and value
    6. 0xf5 CREATE2 4 1 Create a new account with associated code
    7. 0xfa STATICCALL 6 1 Static message-call into an account
    8. 0xfd REVERT 2 0 Halt execution reverting state changes but returning data and remaining gas
    9. 0xfe INVALID ∅ ∅ Designated invalid instruction
    10. 0xff SELFDESTRUCT 1 0 Halt execution and register account for later deletion
78. Gas costs for different instructions are different depending on their computational/storage load on the client. Examples are:
    1. STOP, INVALID and REVERT are 0 gas
    2. Most arithmetic, logic and stack operations are 3-5 gas
    3. CALL\*, BALANCE and EXT\* are 2600 gas
    4. MLOAD/MSTORE/MSTORE8 are 3 gas
    5. SLOAD is 2100 gas and SSTORE is 20,000 gas to set a storage slot from 0 to non-0 and 5,000 gas otherwise
    6. CREATE is 32000 gas and SELFDESTRUCT is 5000 gas
79. A transaction reverts for different exceptional conditions such as running out of gas, invalid instructions etc. in which case all state changes made so far are discarded and the original state of the account is restored as it was before this transaction executed.
80. A transaction with a contract address destination has the contract’s function target and the required arguments in the data field of the transaction. These are encoded according to the Application Binary Interface (ABI):
81. Application Binary Interface (ABI): The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction.
    1. Interface functions of a contract are strongly typed, known at compilation time and static.
    2. Contracts will have the interface definitions of any contracts they call available at compile-time.
82. Function Selector: The first four bytes of the call data for a function call specifies the function to be called.
    1. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function.
    2. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.
    3. Function Arguments: The encoded arguments follow the function selector from the fifth byte onwards.
83. Block explorers: are portals that allow anyone to see real-time data on blocks, transactions, accounts, contract interactions etc. A popular Ethereum block explorer is [etherscan.io](http://etherscan.io/).
84. Mainnet: Short for "main network," this is the main public Ethereum blockchain. There are other Ethereum “testnets” where protocol or smart contract developers test their protocol upgrades or contracts. While mainnet uses real ETH, testnets use test ETH that can be obtained from faucets. The popular testnets are:
    1. Görli: A proof-of-authority (a small number of nodes are allowed to validate transactions and create blocks) testnet that works across clients
    2. Kovan: A proof-of-authority testnet for those running OpenEthereum clients
    3. Rinkeby: A proof-of-authority testnet for those running Geth client
    4. Ropsten: A proof-of-work testnet. This means it's the best representation of mainnet Ethereum
85. Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards. Standards Track EIPs are separated into a number of types: (See [here](https://eips.ethereum.org/))
    1. Core: Improvements requiring a consensus fork as well as changes that are not necessarily consensus critical but may be relevant to “core dev” discussions
    2. Networking: Includes improvements around devp2p and Light Ethereum Subprotocol, as well as proposed improvements to network protocol specifications of whisper and swarm
    3. Interface: Includes improvements around client API/RPC specifications and standards, and also certain language-level standards like method names and contract ABIs. The label “interface” aligns with the interfaces repo and discussion should primarily occur in that repository before an EIP is submitted to the EIPs repository
    4. ERC: Application-level standards and conventions, including contract standards such as token standards (ERC-20), name registries, URI schemes, library/package formats, and wallet formats
    5. Meta: Describes a process surrounding Ethereum or proposes a change to (or an event in) a process
    6. Informational: Describes a Ethereum design issue, or provides general guidelines or information to the Ethereum community, but does not propose a new feature
86. Eth2 or Ethereum 2.0: refers to a set of interconnected upgrades that will make Ethereum more scalable, more secure, and more sustainable (See [here](https://ethereum.org/en/eth2/))
87. Immutable code: Once a contract's code is deployed, it becomes immutable (with exceptions noted below). Standard software development practices that rely on being able to fix bugs and add new features to deployed code do not apply here. This represents a significant security challenge for smart contract development. There are three exceptions:
    1. The modified contract can be deployed at a new address (and old state carried over) but all interacting entities should be notified/enabled to interact with the updated contract at the new address. This is typically considered impractical.
    2. The modified contract can be deployed as a new implementation in a proxy pattern where the proxy points to the modified contract after the update. This is the most commonly used approach to update/add functionality.
    3. CREATE2 opcode allows updating in place using init\_code
88. Web3: is a permissionless, trust-minimized and censorship-resistant network for transfer of value and information.
    1. The popular approach to realise Web3 is to build it over a foundation of peer-to-peer network of nodes for compute, communication and storage.
    2. In the Ethereum ecosystem, this is a combination of the Ethereum blockchain, Waku (previously Whisper) and Swarm respectively.
    3. Privacy and anonymity are big motivating factors in Web3.
    4. Most of the foundational security design principles and development practices from Web2 still apply to Web3. But Web3 security is indeed a paradigm shift along many frontiers.
89. Languages: Web2 programming languages such as JavaScript, Go, Rust and Nim are used extensively in Web3. But the entire domain of smart contracts is new and specific to Web3. Languages such as Solidity and Vyper were created exclusively for Web3.
90. On-chain vs Off-chain: Smart contracts are “on-chain” Web3 components and they interact with “off-chain” components that are very similar to Web2 software. So the major differences in security perspectives between Web3 and Web2 mostly narrow down to security considerations of smart contracts vis-a-vis Web2 software.
91. Open-source & Transparent: Given the emphasis on trust-minimization, Web3 software, especially smart contracts, are expected to be open-source by default.
    1. The deployed bytecode is also expected to be source code verified (on a service such as Etherscan). Security by obscurity with proprietary code is not part of Web3's ethos.
    2. All interactions with smart contracts are recorded on the blockchain as transactions. This includes the transactions’ senders, data and outcome. Having complete visibility into the entire history of transactions and state transitions is akin to having a publicly accessible audit log of a system since inception.
    3. Furthermore, transactions that are still “in flight” and are yet to be confirmed on the blockchain are also publicly visible in pending transaction queues (i.e. mempools) and lend to front-running attacks.
92. Unstoppable & Immutable: Web3 applications, popularly known as Decentralized Applications ( ÐApps), are expected to be unstoppable and immutable because they run on a decentralized blockchain network.
    1. There should not be any one entity that can unilaterally decide to stop a running  ÐApp or make changes to it. Transactions and data on the blockchain are guaranteed to be immutable unless a majority of the network decides otherwise.
    2. Smart contracts, in general, are expected (by users) to not have kill switches controlled by deployers. They are also expected to not be arbitrarily upgradeable. Both these stem from the Web3 goal of trust-minimization, i.e. lack of need to trust potentially malicious  ÐApp developers. However, this makes fixing security vulnerabilities in deployed code and responding to exploits very challenging.
93. Pseudonymous Teams & DAOs: Perhaps inspired by Bitcoin’s Satoshi Nakamoto, there is a trend among some project teams in Web3 to be pseudonymous and known only by their online handles.
    1. One reason for this could be to avoid any potential legal implications in future, given the regulatory uncertainty in this space. This makes it harder to associate any social reputation as it pertains to perceived security trustworthiness of the product or the processes behind its development. It also makes it tricky to hold anyone legally/socially liable or accountable.
    2. “Trust software not wetware” (i.e. people) is the mantra here. While this may be an extreme view, there are still social processes around rollout and governance of projects which affect security posture.
    3. To minimise the role and influence of a few privileged individuals in the lifecycle of projects, there is an increasing trend towards governance by token-holding community members — a Decentralized Autonomous Organization (DAO) of pseudonymous token-holding blockchain addresses making voting-based decisions on project treasury spending and protocol changes. While this reduces centralized points of wetware failure, it potentially slows down decision-making on security-critical aspects and may even lead to project forks.
94. New Architecture, Language & Toolchains: Ethereum has a new virtual machine (EVM) architecture which is a stack-based machine with 256-bit words and associated gas semantics.
    1. Solidity language continues to dominate smart contracts without much real competition (except Vyper perhaps).
    2. The associated toolchains which include development environments (e.g. Truffle, Brownie, Hardhat), libraries (e.g. OpenZeppelin), security tools (e.g. Slither, MythX, Securify) and wallets (e.g. Metamask) are maturing but still playing catch up to the exponential growth of the space.
95. Byzantine Threat Model: The Web3 threat model is based on byzantine faults dealing with arbitrary malicious behavior and governed by mechanism design.
    1. Given the aspirational absence of trusted intermediaries, everyone and everything is meant to be untrusted by default. Participants in this model include developers, miners/validators, infrastructure providers and users, all of whom could potentially be adversaries.
    2. This is a fundamentally different threat model from that of Web2 where there are generalized notions of trusted insiders with authorized access to resources/assets that have to be protected against untrusted outsiders (and malicious insiders). Web3 is the ultimate zero-trust scenario.
96. Keys & Tokens: While “crypto” may indeed mean cryptocurrencies to some non-technical observers, it factually refers to cryptography which is a fundamental bedrock of Web3. As much as we unknowingly use cryptography in the Web2 world, Web3 is taking it to the masses. Cryptographic keys are first-class members of the Web3 world.
    1. Without the presence of Web2 trusted intermediaries who can otherwise reset passwords or restore accounts/assets from their centralized databases, Web3 ideologically pushes the onus of managing keys (and the assets they control) to end users in their wallets. Loss of private keys (or seed phrases) is irreversible and many assets have been lost to such incidents. This is a significant mindset shift from the Web2 world where passwords have become far too common, security pundits are tired of bemoaning the use of commonly reused simple passwords, password databases continue to be dumped and password-killing technologies continue to evade us. Web2 passwords here symbolize the role of trusted centralized intermediaries that Web3 is seeking to replace.
    2. Web2 security breaches targeting financial assets (i.e. excluding ransomware and botnets for DDoS) typically involve stealing of financial or personal data which is then sold on the dark web and used for monetary gain. This is getting much harder because of various checks and measures (both technical and regulatory) being put in place (at centralized intermediaries) to reduce such cybersecurity incidents and prevent anomalous asset transfers. When such unauthorised asset transfers do happen, the involved intermediaries may even cooperate to reverse such transactions and make good.
    3. The notion of assets in Web3 is fundamentally different. Cryptoassets are borderless digital tokens whose accounting ledger is managed by consensus on the blockchain and ownership is determined by access to corresponding cryptographic keys. If someone gets access to your private keys controlling cryptoassets, they can transfer those assets to blockchain addresses controlled by their keys. In a perfectly decentralized world, no intermediary (e.g. centralized exchange) should exist that can reverse such a loss — transactions are immutable. Because there are limited response options, preventive security measures become more critical in the Web3 space.
97. Composability by Design: Permissionless innovation and censorship-resistance are core aspirational goals of Web3.
    1. There are numerous stories of Web2 companies that initially enticed developers to build on their platforms only to shut them out later when they were perceived as a competitive threat.
    2. Web3 applications, especially smart contracts, are open by design and can be accessed permissionlessly by end users and other smart contracts alike.
    3. This composability lends itself to applications that can be layered on top of others like legos, which is great if everything holds up and new lego toys are reliably built on others. However, this unconstrained composability introduces unexpected cross-systemic dependencies that may trigger invalid assumptions across components (likely built by different teams with different constraints in mind) and expose attack surfaces or modes previously unconsidered.
    4. This makes characterizing Web3 vulnerabilities and exploit scenarios very challenging without deep knowledge of all interacting components, constraints and configurations.
98. Compressed Timescales: It feels like innovation in the Web3 space moves at warp speed. Aspects of transparent-development and composability-by-design are strong catalysts to accelerating permissionless and borderless participation which is further incentivized by Internet-native cryptoeconomic tokens — a perfect storm.
    1. This shrinks innovation timescales by orders of magnitude where new waves of experiments happen over weeks or months instead of the years it typically takes within the walled gardens of Web2. It may seem like the only moat here is the speed of execution.
    2. This compressed timescale has a tangible impact on security considerations during design, development and deployment. Corners are cut and shortcuts taken to ride new waves of hype. The end result is a poorly tested system that holds millions of dollars worth of tokens but is vulnerable to exploits.
99. Test-in-Prod: A combination of compressed timescale, unrestricted composability, byzantine threat model and challenges of replicating full state for predicting failure modes of interacting components built with rapidly evolving experimental software/tools in many ways forces realistic testing to happen only in production, i.e. on the “mainnet”. This implies that complex technical and cryptoeconomic exploits may only be discoverable upon production deployment.
100. Audit-as-a-Silver-Bullet: Secure Software Development Lifecycle (SSDLC) processes for Web2 products have evolved over several decades to a point where they are expected to meet some minimum requirements of a combination of internal validation, external assessments (e.g. product/process audits, penetration testing) and certifications depending on the value of managed assets, anticipated risk, threat model and the market domain of products (e.g. financial sector has stricter regulatory compliance requirements).
101. Web3 projects seem to increasingly rely on external audits as a stamp of security approval. This is typically justified by the lack of sufficient in-house security expertise. While the optics of this approach seems to falsely convince speculators, this approach is untenable for several reasons:
     1. Audits currently are very expensive because demand is much greater than supply for top-rated audit teams that have the experience and reputation to analyze complex projects
     2. Audits are typically commissioned once at the end of project development just before production release
     3. Upgrades to projects go unaudited for commercial or logistical reasons
     4. The expectation (from the project team and users) is that audits are a panacea for all vulnerabilities and that the project is “bug-free” after a short audit (typically few weeks)

# Solidity 101

### 101 key aspects of Solidity

1. Solidity is a high-level language for implementing smart contracts on Ethereum (and the blockchains) targeting the EVM. Solidity was proposed in 2014 by Gavin Wood and was later developed by Ethereum’s Solidity team, led by Christian Reitwiessner, Alex Beregszaszi & others. (See [here](https://en.wikipedia.org/wiki/Solidity))
2. It is influenced mainly by C++, a little from Python and early-on from JavaScript. The syntax and OOP concepts are from C++.  Solidity’s modifiers, multiple inheritance, C3 linearization and the super keyword are influences from Python. Function-level scoping and var keyword were JavaScript influences early-on but those have been reduced since v0.4.0. (See [here](https://docs.soliditylang.org/en/v0.8.9/language-influences.html))
3. Solidity is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language.
4. The layout of a Solidity source file can contain an arbitrary number of pragma directives, import directives and struct/enum/contract definitions. The best-practices for layout within a contract is the following order: state variables, events, modifiers, constructor and functions.
5. SPDX License Identifier: Solidity source files are recommended to start with a comment indicating its license e.g.:“// SPDX-License-Identifier: MIT”, where the compiler includes the supplied string in the bytecode metadata to make it machine readable. SPDX stands for Software Package Data Exchange (See [SPDX](https://spdx.org/)).
6. Pragmas: The pragma keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project. If you import another file, the pragma from that file does not automatically apply to the importing file. There are two types: 1) Version: a)  Compiler version b) ABI Coder version 2) Experimental: a) SMTChecker
7. Version Pragma: This indicates the specific Solidity compiler version to be used for that source file and is used as follows: “pragma solidity x.y.z;” where x.y.z indicates the version of the compiler.
   1. Using the version pragma does not change the version of the compiler. It also does not enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.
   2. The latest compiler versions are in the 0.8.z range
   3. A different y in x.y.z indicates breaking changes e.g. 0.6.0 introduces breaking changes over 0.5.z. A different z in x.y.z indicates bug fixes.
   4. A ‘^’ symbol prefixed to x.y.z in the pragma indicates that the source file may be compiled only from versions starting with x.y.z until x.(y+1).z. For e.g., “pragma solidity ^0.8.3;” indicates that source file may be compiled with compiler version starting from 0.8.3 until any 0.8.z but not 0.9.z. This is known as a “floating pragma.”
   5. Complex pragmas are also possible using ‘>’,’>=‘,’<‘ and ‘<=‘ symbols to combine multiple versions e.g. “pragma solidity >=0.8.0 <0.8.3;”
8. ABI Coder Pragma: This indicates the choice between the two implementations of the ABI encoder and decoder: “pragma abicoder v1;” or “pragma abicoder v2;”
   1. The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. It might produce less optimal code and has not received as much testing as the old encoder. This is activated by default.
   2. The set of types supported by the new encoder is a strict superset of the ones supported by the old one. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-abicoder v2 contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling abicoder v2 for your contract is enough to make the error go away.
   3. This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.
9. Experimental Pragma: This can be used to enable features of the compiler or language that are not yet enabled by default
   1. SMTChecker: The use of “pragma experimental SMTChecker;” performs additional safety checks which are obtained by querying an SMT solver (See [SMTChecker](https://docs.soliditylang.org/en/v0.8.9/smtchecker.html" \l "formal-verification))
   2. The SMTChecker module automatically tries to prove that the code satisfies the specification given by require and assert statements. That is, it considers require statements as assumptions and tries to prove that the conditions inside assert statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.
   3. Other checks: Arithmetic underflow and overflow, Division by zero, Trivial conditions and unreachable code, Popping an empty array, Out of bounds index access, Insufficient funds for a transfer.
10. Imports: Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on) e.g. “import “filename”;"
11. Comments: Single-line comments (//) and multi-line comments (/\*...\*/) are possible. Comments are recommended as in-line documentation of what contracts, functions, variables, expressions, control and data flow are expected to do as per the implementation, and any assumptions/invariants made/needed. They help in readability and maintainability.
12. NatSpec Comments: NatSpec stands for “Ethereum Natural Language Specification Format.” These are written with a triple slash (///) or a double asterisk block(/\*\* ... \*/) directly above function declarations or statements to generate documentation in JSON format for developers and end-users. It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). These comments contain different types of tags:
    1. @title: A title that should describe the contract/interface
    2. @author: The name of the author (for contract, interface)
    3. @notice: Explain to an end user what this does (for contract, interface, function, public state variable, event)
    4. @dev: Explain to a developer any extra details (for contract, interface, function, state variable, event)
    5. @param: Documents a parameter (just like in doxygen) and must be followed by parameter name (for function, event)
    6. @return: Documents the return variables of a contract’s function (function, public state variable)
    7. @inheritdoc: Copies all missing tags from the base function and must be followed by the contract name (for function, public state variable)
    8. @custom…: Custom tag, semantics is application-defined (for everywhere)
13. Contracts: They are similar to classes in object-oriented languages in that they contain persistent data in state variables and functions that can modify these variables. Contracts can inherit from other contracts
14. Contracts can contain declarations of State Variables, Functions, Function Modifiers, Events, Errors, Struct Types and Enum Types
15. State Variables: They are variables that can be accessed by all functions of the contract and whose values are permanently stored in contract storage
16. State Visibility Specifiers: State variables have to be specified as being public, internal or private:
    1. public: Public state variables are part of the contract interface and can be either accessed internally or via messages. An automatic getter function is generated.
    2. internal: Internal state variables can only be accessed internally from within the current contract or contracts deriving from it
    3. private: Private state variables can only be accessed from the contract they are defined in and not even in derived contracts. Everything that is inside a contract is visible to all observers external to the blockchain. Making variables private only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.
17. State Variables: Constant & Immutable
    1. State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time i.e. in the constructor or point of declaration.
    2. For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. block.timestamp, address(this).balance or block.number) or execution data (msg.value or gasleft()) or makes calls to external contracts is disallowed.
    3. Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They cannot be read during construction time and can only be assigned once.
    4. The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.
18. Compared to regular state variables, the gas costs of constant and immutable variables are much lower:
    1. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations.
    2. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, constant values can sometimes be cheaper than immutable values.
    3. The only supported types are strings (only for constants) and value types.
19. Functions: Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts. They have different levels of visibility towards other contracts.
20. Function parameters: Function parameters are declared the same way as variables, and the name of unused parameters can be omitted. Function parameters can be used as any other local variable and they can also be assigned to.
21. Function Return Variables: Function return variables are declared with the same syntax after the returns keyword.
    1. The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.
    2. You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or multiple ones) directly with the return statement
    3. If you use an early return to leave a function that has return variables, you must provide return values together with the return statement
    4. When a function has multiple return types, the statement return (v0, v1, ..., vn) can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an implicit conversion
22. Function Modifiers: They can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function’s control flow continues after the “\_” in the preceding modifier. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The \_ symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.
23. Function Visibility Specifiers: Functions have to be specified as being public, external, internal or private:
    1. public: Public functions are part of the contract interface and can be either called internally or via messages.
    2. external: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function f cannot be called internally (i.e. f() does not work, but this.f() works).
    3. internal: Internal functions can only be accessed internally from within the current contract or contracts deriving from it
    4. private: Private functions can only be accessed from the contract they are defined in and not even in derived contracts
24. Function Mutability Specifiers: Functions can be specified as being pure or view:
    1. view functions can read contract state but cannot modify it. This is enforced at runtime via STATICCALL opcode. The following are considered state modifying: 1) Writing to state variables 2) Emitting events 3) Creating other contracts 4) Using selfdestruct 5) Sending Ether via calls 6) Calling any function not marked view or pure 7) Using low-level calls 8) Using inline assembly that contains certain opcodes.
    2. pure functions can neither read contract state nor modify it. The following are considered reading from state: 1) Reading from state variables 2) Accessing address(this).balance or <address>.balance 3) Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data) 4) Calling any function not marked pure 5) Using inline assembly that contains certain opcodes.
    3. It is not possible to prevent functions from reading the state at the level of the EVM. It is only possible to prevent them from writing to the state via STATICCALL. Therefore,  only view can be enforced at the EVM level, but not pure.
25. Function Overloading: A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading.”
    1. Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call.
    2. Return parameters are not taken into account for overload resolution.
26. Free Functions: Functions that are defined outside of contracts are called “free functions” and always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.
27. Events: They are an abstraction on top of the EVM’s logging functionality. Emitting events cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible. The Log and its event data is not accessible from within contracts (not even from the contract that created them). Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.
28. Indexed Event Parameters: Adding the attribute indexed for up to three parameters adds them to a special data structure known as “topics” instead of the data part of the log. If you use arrays (including string and bytes) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes). All parameters without the indexed attribute are ABI-encoded into the data part of the log. Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.
29. Emit: Events are emitted using `emit`, followed by the name of the event and the arguments e.g. “emit Deposit(msg.sender, \_id, msg.value);”
30. Struct Types: They are custom defined types that can group several variables of same/different types together to create a custom data structure. The struct members are accessed using ‘.’ e.g.: struct s {address user; uint256 amount} where s.user and s.amount access the struct members.
31. Enums: They can be used to create custom types with a finite set of constant values to improve readability. They need a minimum of one member and can have a maximum of 256. They can be explicitly converted to/from integers. The options are represented by unsigned integer values starting from 0. The default value is the first member.
32. Constructor: Contracts can be created “from outside” via Ethereum transactions or from within Solidity contracts. When a contract is created, its constructor (a function declared with the constructor keyword) is executed once. A constructor is optional and only one constructor is allowed. After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.
33. Receive Function: A contract can have at most one receive function, declared using receive() external payable { ... } without the function keyword. This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability.
    1. The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers via .send() or .transfer().
    2. In the worst case, the receive function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging
    3. A contract without a receive Ether function can receive Ether as a recipient of a coinbase transaction (aka miner block reward) or as a destination of a selfdestruct. A contract cannot react to such Ether transfers and thus also cannot reject them. This means that address(this).balance can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).
34. Fallback Function: A contract can have at most one fallback function, declared using either fallback () external [payable] or fallback (bytes calldata \_input) external [payable] returns (bytes memory \_output), both without the function keyword. This function must have external visibility.
    1. The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function. The fallback function always receives data, but in order to also receive Ether it must be marked payable.
    2. In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available
35. Solidity is a statically-typed language, which means that the type of each variable (state and local) needs to be specified in code at compile-time. This is unlike dynamically-typed languages where types are required only with runtime values. Statically-typed languages perform compile-time type-checking according to the language rules. Other examples are C, C++, Java, Rust, Go, Scala.
36. Solidity has two categories of types: Value Types and Reference Types. Value Types are called so because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments. In contrast, Reference Types can be modified through multiple different names i.e. references to the same underlying variable.
37. Value Types: Types that are passed by value, i.e. they are always copied when they are used as function arguments or in assignments — Booleans, Integers, Fixed Point Numbers, Address, Contract, Fixed-size Byte Arrays (bytes1, bytes2, …, bytes32), Literals (Address, Rational, Integer, String, Unicode, Hexadecimal), Enums, Functions.
38. Reference Types: Types that can be modified through multiple different names. Arrays (including Dynamically-sized bytes array bytes and string), Structs, Mappings.
39. Default Values: A variable which is declared will have an initial default value whose byte-representation is all zeros. The “default values” of variables are the typical “zero-state” of whatever the type is. For example, the default value for a bool is false. The default value for the uint or int types is 0. For statically-sized arrays and bytes1 to bytes32, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, bytes and string, the default value is an empty array or string. For the enum type, the default value is its first member.
40. Scoping: Scoping in Solidity follows the widespread scoping rules of C99
    1. Variables are visible from the point right after their declaration until the end of the smallest { }-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.
    2. Variables that are parameter-like (function parameters, modifier parameters, catch parameters, …) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.
    3. Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.
41. Boolean: bool Keyword and the possible values are constants true and false.
    1. Operators are ! (logical negation) && (logical conjunction, “and”) || (logical disjunction, “or”)== (equality) and != (inequality).
    2. The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.
42. Integers: int / uint: Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively. Operators are:
    1. Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
    2. Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
    3. Shift operators: << (left shift), >> (right shift)
    4. Arithmetic operators: +, -, unary - (only for signed integers), \*, /, % (modulo), \*\* (exponentiation)
43. Integers in Solidity are restricted to a certain range. For example, with uint32, this is 0 up to 2\*\*32 - 1. There are two modes in which arithmetic is performed on these types: The “wrapping” or “unchecked” mode and the “checked” mode. By default, arithmetic is always “checked”, which means that if the result of an operation falls outside the value range of the type, the call is reverted through a failing assertion. You can switch to “unchecked” mode using unchecked { ... }. This was introduced in compiler version 0.8.0.
44. Fixed Point Numbers: Fixed point numbers using keywords fixed / ufixed are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from. There are fixed-point libraries that are widely used for this such as DSMath, PRBMath, ABDKMath64x64 etc.
45. Address Type: The address type comes in two types: (1) address: Holds a 20 byte value (size of an Ethereum address) (2) address payable: Same as address, but with the additional members transfer and send. address payable is an address you can send Ether to, while a plain address cannot be sent Ether.
    1. Operators are <=, <, ==, !=, >= and >
    2. Conversions: Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(<address>). Explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types.
    3. Only expressions of type address and contract-type can be converted to the type address payable via the explicit conversion payable(...). For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a receive or a payable fallback function.
46. Members of Address Type:
    1. <address>.balance (uint256): balance of the Address in Wei
    2. <address>.code (bytes memory): code at the Address (can be empty)
    3. <address>.codehash (bytes32): the codehash of the Address
    4. <address payable>.transfer(uint256 amount): send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable
    5. <address payable>.send(uint256 amount) returns (bool): send given amount of Wei to Address, returns false on failure, forwards 2300 gas stipend, not adjustable
    6. <address>.call(bytes memory) returns (bool, bytes memory): issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable
    7. <address>.delegatecall(bytes memory) returns (bool, bytes memory): issue low-level DELEGATECALL with the given payload, returns success condition and return data, forwards all available gas, adjustable
    8. <address>.staticcall(bytes memory) returns (bool, bytes memory): issue low-level STATICCALL with the given payload, returns success condition and return data, forwards all available gas, adjustable
47. Transfer: The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure. The code in receive function or if not present then in fallback function is executed with the transfer call. If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.
48. Send: The send function is the low-level counterpart of transfer. If the execution fails then send only returns false and does not revert unlike transfer. So the return value of send must be checked by the caller.
49. Call/Delegatecall/Staticcall: In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory). The functions abi.encode, abi.encodePacked, abi.encodeWithSelector and abi.encodeWithSignature can be used to encode structured data.
    1. gas and value modifiers can be used with these functions (delegatecall doesn’t support value) to specify the amount of gas and Ether value passed to the callee.
    2. With delegatecall, only the code of the given address is used but all other aspects (storage, balance, msg.sender etc.) are taken from the current contract. The purpose of delegatecall is to use library/logic code which is stored in callee contract but operate on the state of the caller contract
    3. With staticcall, the execution will revert if the called function modifies the state in any way
50. Contract Type: Every contract defines its own type. Contracts can be explicitly converted to and from the address type. Contract types do not support any operators. The members of contract types are the external functions of the contract including any state variables marked as public.
51. Fixed-size Byte Arrays: The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32. The type byte[] is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the bytes type instead.
52. Literals: They can be of 5 types:
    1. Address Literals: Hexadecimal literals that pass the address checksum test are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. The mixed-case address checksum format is defined in [EIP-55](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md).
    2. Rational and Integer Literals: Integer literals are formed from a sequence of numbers in the range 0-9. Decimal fraction literals are formed by a . with at least one number on one side. Scientific notation is also supported, where the base can have fractions and the exponent cannot. Underscores can be used to separate the digits of a numeric literal to aid readability and are semantically ignored.
    3. String Literals: String literals are written with either double or single-quotes ("foo" or ‘bar’). They can only contain printable ASCII characters and a set of escape characters
    4. Unicode Literals: Unicode literals prefixed with the keyword unicode can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.
    5. Hexadecimal Literals: Hexadecimal literals are hexadecimal digits prefixed with the keyword hex and are enclosed in double or single-quotes e.g. hex”001122FF”, hex'0011\_22\_FF'.
53. Enums: Enums are one way to create a user-defined type in Solidity. They require at least one member and its default value when declared is the first member. They cannot have more than 256 members.
54. Function Types: Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls.  They come in two flavours - internal and external functions. Internal functions can only be called inside the current contract. External functions consist of an address and a function signature and they can be passed via and returned from external function calls.
55. Reference Types & Data Location: Every reference type has an additional annotation — the data location where it is stored. There are three data locations: memory, storage and calldata.
    1. memory: whose lifetime is limited to an external function call
    2. storage: whose lifetime is limited to the lifetime of a contract and the location where the state variables are stored
    3. calldata: which is a non-modifiable, non-persistent area where function arguments are stored and behaves mostly like memory. It is required for parameters of external functions but can also be used for other variables.
56. Data Location & Assignment: Data locations are not only relevant for persistence of data, but also for the semantics of assignments.
    1. Assignments between storage and memory (or from calldata) always create an independent copy.
    2. Assignments from memory to memory only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
    3. Assignments from storage to a local storage variable also only assign a reference.
    4. All other assignments to storage always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.
57. Arrays: Arrays can have a compile-time fixed size, or they can have a dynamic size
    1. The type of an array of fixed size k and element type T is written as T[k], and an array of dynamic size as T[].
    2. Indices are zero-based
    3. Array elements can be of any type, including mapping or struct.
    4. Accessing an array past its end causes a failing assertion
58. Array members:
    1. length: returns number of elements in array
    2. push(): appends a zero-initialised element at the end of the array and returns a reference to the element
    3. push(x):  appends a given element at the end of the array and returns nothing
    4. pop: removes an element from the end of the array and implicitly calls delete on the removed element
59. Variables of type bytes and string are special arrays
    1. bytes is similar to byte[], but it is packed tightly in calldata and memory
    2. string is equal to bytes but does not allow length or index access
    3. Solidity does not have string manipulation functions, but there are third-party string libraries
    4. Use bytes for arbitrary-length raw byte data and string for arbitrary-length string (UTF-8) data
    5. Use bytes over byte[] because it is cheaper, since byte[] adds 31 padding bytes between the elements
    6. If you can limit the length to a certain number of bytes, always use one of the value types bytes1 to bytes32 because they are much cheaper
60. Memory Arrays: Memory arrays with dynamic length can be created using the new operator
    1. As opposed to storage arrays, it is not possible to resize memory arrays i.e. the .push member functions are not available
    2. You either have to calculate the required size in advance or create a new memory array and copy every element
61. Array Literals: An array literal is a comma-separated list of one or more expressions, enclosed in square brackets ([…])
    1. It is always a statically-sized memory array whose length is the number of expressions
    2. The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.
    3. Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays
62. Gas costs of push and pop: Increasing the length of a storage array by calling push() has constant gas costs because storage is zero-initialised, while decreasing the length by calling pop() has a cost that depends on the “size” of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling delete on them.
63. Array Slices: Array slices are a view on a contiguous portion of an array. They are written as x[start:end], where start and end are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is x[start] and the last element is x[end - 1]
    1. If start is greater than end or if end is greater than the length of the array, an exception is thrown
    2. Both start and end are optional: start defaults to 0 and end defaults to the length of the array
    3. Array slices do not have any members
    4. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice
    5. Array slices do not have a type name which means no variable can have an array slices as type and they only exist in intermediate expressions
    6. Array slices are only implemented for calldata arrays.
    7. Array slices are useful to ABI-decode secondary data passed in function parameters
64. Struct Types: Structs help define new aggregate types by combining other value/reference types into one unit. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays. It is not possible for a struct to contain a member of its own type
65. Mapping Types: Mappings define key-value pairs and are declared using the syntax mapping(\_KeyType => \_ValueType) \_VariableName.
    1. The \_KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. \_ValueType can be any type, including mappings, arrays and structs.
    2. Key data is not stored in a mapping, only its keccak256 hash is used to look up the value
    3. They do not have a length or a concept of a key or value being set
    4. They can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions
    5. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.
    6. You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that.
66. Operators Involving LValues (i.e. a variable or something that can be assigned to)
    1. a += e is equivalent to a = a + e. The operators -=, \*=, /=, %=, |=, &= and ^= are defined accordingly
    2. a++ and a-- are equivalent to a += 1 / a -= 1 but the expression itself still has the previous value of a
    3. In contrast, --a and ++a have the same effect on a but return the value after the change
67. delete
    1. delete a assigns the initial value for the type to a
    2. For integers it is equivalent to a = 0
    3. For arrays, it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value
    4. delete a[x] deletes the item at index x of the array and leaves all other elements and the length of the array untouched
    5. For structs, it assigns a struct with all members reset
    6. delete has no effect on mappings. So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.
    7. For mappings, individual keys and what they map to can be deleted: If a is a mapping, then delete a[x] will delete the value stored at x
68. Implicit Conversions: An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators
    1. implicit conversion between value-types is possible if it makes sense semantically and no information is lost
    2. For example, uint8 is convertible to uint16 and int128 to int256, but int8 is not convertible to uint256, because uint256 cannot hold values such as -1
69. Explicit Conversions: If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler e.g. int to uint
    1. If an integer is explicitly converted to a smaller type, higher-order bits are cut off
    2. If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end)
    3. Fixed-size bytes types while explicitly converting to a smaller type and will cut off the bytes to the right
    4. Fixed-size bytes types while explicitly converting to a larger type and will pad bytes to the right.
70. Conversions between Literals and Elementary Types
    1. Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation
    2. Decimal number literals cannot be implicitly converted to fixed-size byte arrays
    3. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type
    4. String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the bytes type
71. A literal number can take a suffix of wei, gwei (1e9) or ether (1e18) to specify a sub-denomination of Ether
72. Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit where 1 == 1 seconds,1 minutes == 60 seconds, 1 hours == 60 minutes, 1 days == 24 hours and 1 weeks == 7 days
    1. Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds
    2. These suffixes cannot be applied directly to variables but can be applied by multiplication
73. Block and Transaction Properties:
    1. blockhash(uint blockNumber) returns (bytes32): hash of the given block - only works for 256 most recent, excluding current, blocks
    2. block.chainid (uint): current chain id
    3. block.coinbase (address payable): current block miner’s address
    4. block.difficulty (uint): current block difficulty
    5. block.gaslimit (uint): current block gaslimit
    6. block.number (uint): current block number
    7. block.timestamp (uint): current block timestamp as seconds since unix epoch
    8. msg.data (bytes calldata): complete calldata
    9. msg.sender (address): sender of the message (current call)
    10. msg.sig (bytes4): first four bytes of the calldata (i.e. function identifier)
    11. msg.value (uint): number of wei sent with the message
    12. tx.gasprice (uint): gas price of the transaction
    13. gasleft() returns (uint256): remaining gas
    14. tx.origin (address): sender of the transaction (full call chain)
74. The values of all members of msg, including msg.sender and msg.value can change for every external function call. This includes calls to library functions.
75. Do not rely on block.timestamp or blockhash as a source of randomness. Both the timestamp and the block hash can be influenced by miners to some degree. The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.
76. The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.
77. ABI Encoding and Decoding Functions:
    1. abi.decode(bytes memory encodedData, (...)) returns (...): ABI-decodes the given data, while the types are given in parentheses as second argument.
    2. abi.encode(...) returns (bytes memory): ABI-encodes the given arguments
    3. abi.encodePacked(...) returns (bytes memory): Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!
    4. abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory): ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
    5. abi.encodeWithSignature(string memory signature, ...) returns (bytes memory): Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), …)
78. Error Handling:
    1. assert(bool condition): causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.
    2. require(bool condition): reverts if the condition is not met - to be used for errors in inputs or external components.
    3. require(bool condition, string memory message): reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.
    4. revert(): abort execution and revert state changes
    5. revert(string memory reason): abort execution and revert state changes, providing an explanatory string
79. Mathematical and Cryptographic Functions:
    1. addmod(uint x, uint y, uint k) returns (uint): compute (x + y) % k where the addition is performed with arbitrary precision and does not wrap around at 2\*\*256. Assert that k != 0 starting from version 0.5.0.
    2. mulmod(uint x, uint y, uint k) returns (uint): compute (x \* y) % k where the multiplication is performed with arbitrary precision and does not wrap around at 2\*\*256. Assert that k != 0 starting from version 0.5.0.
    3. keccak256(bytes memory) returns (bytes32): compute the Keccak-256 hash of the input
    4. sha256(bytes memory) returns (bytes32): compute the SHA-256 hash of the input
    5. ripemd160(bytes memory) returns (bytes20): compute RIPEMD-160 hash of the input
    6. ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address): recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature: r = first 32 bytes of signature, s = second 32 bytes of signature, v = final 1 byte of signature. ecrecover returns an address, and not an address payable.
80. If you use ecrecover, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin has a ECDSA helper library that you can use as a wrapper for ecrecover without this issue.
81. Contract Related:
    1. this (current contract’s type): the current contract, explicitly convertible to Address
    2. selfdestruct(address payable recipient): Destroy the current contract, sending its funds to the given Address and end execution.
82. selfdestruct has some peculiarities: the receiving contract’s receive function is not executed and the contract is only really destroyed at the end of the transaction and revert’s might “undo” the destruction.
83. Type Information: The expression type(X) can be used to retrieve information about the type X, where X can be either a contract or an integer type. For a contract type C, the following type information is available:
    1. type(C).name: The name of the contract.
    2. type(C).creationCode: Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the create2 opcode. This property cannot be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.
    3. type(C).runtimeCode: Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of C. If C has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with .creationCode also apply for this property.
    4. For an interface type I, the following type information is available: type(I).interfaceId: A bytes4 value containing the EIP-165 interface identifier of the given interface I. This identifier is defined as the XOR of all function selectors defined within the interface itself - excluding all inherited functions.
84. For an integer type T, , the following type information is available:
    1. type(T).min: The smallest value representable by type T.
    2. type(T).max: The largest value representable by type T.
85. Control Structures: Solidity has if, else, while, do, for, break, continue, return, with the usual semantics known from C or JavaScript
    1. Parentheses can not be omitted for conditionals, but curly braces can be omitted around single-statement bodies
    2. Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so if (1) { ... } is not valid Solidity.
86. Exceptions: Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller
    1. When exceptions happen in a sub-call, they “bubble up” (i.e., exceptions are rethrown) automatically. Exceptions to this rule are send and the low-level functions call, delegatecall and staticcall: they return false as their first return value in case of an exception instead of “bubbling up”.
    2. Exceptions in external calls can be caught with the try/catch statement
    3. Exceptions can contain data that is passed back to the caller. This data consists of a 4-byte selector and subsequent ABI-encoded data. The selector is computed in the same way as a function selector, i.e., the first four bytes of the keccak256-hash of a function signature - in this case an error signature.
    4. Solidity supports two error signatures: Error(string) and Panic(uint256). The first (“error”) is used for “regular” error conditions while the second (“panic”) is used for errors that should not be present in bug-free code.
87. The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.
88. The assert function creates an error of type Panic(uint256). Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input.
89. A Panic exception is generated in the following situations. The error code supplied with the error data indicates the kind of panic:
    1. 0x01: If you call assert with an argument that evaluates to false.
    2. 0x11: If an arithmetic operation results in underflow or overflow outside of an unchecked { ... } block.
    3. 0x12; If you divide or modulo by zero (e.g. 5 / 0 or 23 % 0).
    4. 0x21: If you convert a value that is too big or negative into an enum type.
    5. 0x22: If you access a storage byte array that is incorrectly encoded.
    6. 0x31: If you call .pop() on an empty array.
    7. 0x32: If you access an array, bytesN or an array slice at an out-of-bounds or negative index (i.e. x[i] where i >= x.length or i < 0).
    8. 0x41: If you allocate too much memory or create an array that is too large.
    9. 0x51: If you call a zero-initialized variable of internal function type.
90. The require function either creates an error of type Error(string) or an error without any error data and it should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts. You can optionally provide a message string for require, but not for assert.
91. A Error(string) exception (or an exception without data) is generated in the following situations:
    1. Calling require with an argument that evaluates to false.
    2. If you perform an external function call targeting a contract that contains no code
    3. If your contract receives Ether via a public function without payable modifier (including the constructor and the fallback function)
    4. If your contract receives Ether via a public getter function
92. revert: A direct revert can be triggered using the revert statement and the revert function. The revert statement takes a custom error as a direct argument without parentheses: revert CustomError(arg1, arg2). The revert() function is another way to trigger exceptions from within other code blocks to flag an error and revert the current call. The function takes an optional string message containing details about the error that is passed back to the caller and it will create an Error(string) exception. Using a custom error instance will usually be much cheaper than a string description, because you can use the name of the error to describe it, which is encoded in only four bytes. A longer description can be supplied via NatSpec which does not incur any costs.
93. try/catch: The try keyword has to be followed by an expression representing an external function call or a contract creation (new ContractName()). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The returns part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract’s execution continues inside the first success block. If the end of the success block is reached, execution continues after the catch blocks.
94. Solidity supports different kinds of catch blocks depending on the type of error:
    1. catch Error(string memory reason) { ... }: This catch clause is executed if the error was caused by revert("reasonString") or require(false, "reasonString") (or an internal error that causes such an exception).
    2. catch Panic(uint errorCode) { ... }: If the error was caused by a panic, i.e. by a failing assert, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
    3. catch (bytes memory lowLevelData) { ... }: This clause is executed if the error signature does not match any other clause, if there was an error while decoding the error message, or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.
    4. catch { ... }: If you are not interested in the error data, you can just use catch { ... } (even as the only catch clause) instead of the previous clause.
95. If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).
96. The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: The caller always retains 63/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left
97. Programming style: coding conventions for writing solidity code. Style is about consistency. Consistency with style is important. Consistency within a project is more important. Consistency within one module or function is most important. Two main categories: 1) Layout 2) Naming Conventions. Programming style affects readability and maintainability, both of which affect security.
98. Code Layout:
    1. Indentation: Use 4 spaces per indentation level
    2. Tabs or Spaces: Spaces are the preferred indentation method. Mixing tabs and spaces should be avoided.
    3. Blank Lines: Surround top level declarations in solidity source with two blank lines.
    4. Maximum Line Length: Keeping lines to a maximum of 79 (or 99) characters helps readers easily parse the code.
    5. Wrapped lines should conform to the following guidelines: The first argument should not be attached to the opening parenthesis. One, and only one, indent should be used. Each argument should fall on its own line. The terminating element, );, should be placed on the final line by itself.
    6. Source File Encoding: UTF-8 or ASCII encoding is preferred.
    7. Imports: Import statements should always be placed at the top of the file.
    8. Order of Functions: Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. Functions should be grouped according to their visibility and ordered: constructor, receive function (if exists), fallback function (if exists), external, public, internal, private. Within a grouping, place the view and pure functions last.
99. More Code Layout:
    1. Whitespace in Expressions: Avoid extraneous whitespace in the following situations —  Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.
    2. Control Structures:  The braces denoting the body of a contract, library, functions and structs should: open on the same line as the declaration, close on their own line at the same indentation level as the beginning of the declaration. The opening brace should be preceded by a single space.
    3. Function Declaration: For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration. The closing brace should be at the same indentation level as the function declaration. The opening brace should be preceded by a single space.
    4. Mappings: In variable declarations, do not separate the keyword mapping from its type by a space. Do not separate any nested mapping keyword from its type by whitespace.
    5. Variable Declarations: Declarations of array variables should not have a space between the type and the brackets.
    6. Strings should be quoted with double-quotes instead of single-quotes.
    7. Operators: Surround operators with a single space on either side. Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence.This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator
    8. Layout contract elements in the following order: Pragma statements, Import statements, Interfaces, Libraries, Contracts. Inside each contract, library or interface, use the following order: Type declarations, State variables, Events, Functions
100. Naming Convention:
     1. Types: lowercase, lower\_case\_with\_underscores, UPPERCASE, UPPER\_CASE\_WITH\_UNDERSCORES, CapitalizedWords, mixedCase, Capitalized\_Words\_With\_Underscores
     2. Names to Avoid: l - Lowercase letter el, O - Uppercase letter oh, I - Uppercase letter eye. Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.
     3. Contracts and libraries should be named using the CapWords style. Contract and library names should also match their filenames. If a contract file includes multiple contracts and/or libraries, then the filename should match the core contract. This is not recommended however if it can be avoided. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
     4. Structs should be named using the CapWords style. Examples: MyCoin, Position, PositionXY.
     5. Events should be named using the CapWords style. Examples: Deposit, Transfer, Approval, BeforeTransfer, AfterTransfer.
     6. Functions should use mixedCase. Examples: getBalance, transfer, verifyOwner, addMember, changeOwner.
101. More Naming Convention:
     1. Function arguments should use mixedCase. Examples: initialSupply, account, recipientAddress, senderAddress, newOwner.
     2. Local and state variable names should use mixedCase. Examples: totalSupply, remainingSupply, balancesOf, creatorAddress, isPreSale, tokenExchangeRate.
     3. Constants should be named with all capital letters with underscores separating words. Examples: MAX\_BLOCKS, TOKEN\_NAME, TOKEN\_TICKER, CONTRACT\_VERSION.
     4. Modifier names should use mixedCase. Examples: onlyBy, onlyAfter, onlyDuringThePreSale.
     5. Enums, in the style of simple type declarations, should be named using the CapWords style. Examples: TokenGroup, Frame, HashStyle, CharacterLocation.
     6. Avoiding Naming Collisions: single\_trailing\_underscore\_. This convention is suggested when the desired name collides with that of a built-in or otherwise reserved name.

# Audit Techniques & Tools 101

### 101 Audit Techniques & Tools

1. Audit: is an external security assessment of a project codebase, typically requested and paid-for by the project team
   1. It detects and describes (in a report) security issues with underlying vulnerabilities, severity/difficulty, potential exploit scenarios and recommended fixes.
   2. It also provides subjective insights into code quality, documentation and testing.
   3. The scope/depth/format of audit reports varies across auditing teams but they generally cover similar aspects.
2. Audit Scope: For Ethereum-based smart-contract projects, the scope is typically the on-chain smart contract code and sometimes includes the off-chain components that interact with the smart contracts.
3. Audit Goal: The goal of audits is to assess project code (with any associated specification, documentation) and alert project team, typically before launch, of potential security-related issues that need to be addressed to improve security posture, decrease attack surface and mitigate risk.
4. Audit Non-goal: Audit is *not* a security guarantee of “bug-free” code by any stretch of imagination but a best-effort endeavour by trained security experts operating within reasonable constraints of time, understanding, expertise and of course, decidability.
5. Audit Target: Security companies execute audits for clients who pay for their services. Engagements are therefore geared towards priorities of project owners and *not*project users/investors. Audits are *not* intended to alert potential project users of any inherent risk. That is not their business/technical goal.
6. Audit Need: Smart contract based projects  do not have sufficient in-house Ethereum smart contract security expertise and/or time to perform internal security assessments and therefore rely on external experts who have domain expertise in those areas. Even if projects have some expertise in-house, they would still benefit from an unbiased external team with supplementary/complementary skill sets that can review the assumptions, design, specification and implementation of the project codebase.
7. Audit Types: depend on the scope/nature/status of projects but generally fall into the following categories:
   1. New audit: for a new project that is being launched
   2. Repeat audit: for a new version of an existing project being revised with new/fixed features
   3. Fix audit: for reviewing the fixes made to the findings from a current/prior audit
   4. Retainer audit: for constantly reviewing project updates
   5. Incident audit: for reviewing an exploit incident, root causing the incident, identifying the underlying vulnerabilities and proposing fixes.
8. Audit Timeline: depends on the scope/nature/status of the project to be assessed and the type of audit. This may vary from a few days for a fix/retainer audit to several weeks for a new/repeat/incident audit.
9. Audit Effort: typically involves more than one auditor simultaneously for getting independent, redundant or supplementary/complementary assessment expertise on the project.
10. Audit Costs: depends on the type/scope of audit but typically costs upwards of USD $10K/week depending on the complexity of the project, market demand/supply for audits and the strength/reputation of the auditing firm.
11. Audit Prerequisites should include:
    1. Clear definition of the scope of the project to be assessed typically in the form of a specific  commit hash of project files/folders on a github repository
    2. Public/private repository
    3. Public/anonymous team
    4. Specification of the project’s design and architecture
    5. Documentation of the project’s implementation and business logic
    6. Threat models and specific areas of concern
    7. Prior testing, tools used, other audits
    8. Timeline, effort and costs/payments
    9. Engagement dynamics/channels for questions/clarifications, findings communication and reports
    10. Points of contact on both sides
12. Audit Limitations: Audits are necessary (for now at least) but not sufficient:
    1. There is risk reduction but residual risk exists because of several factors such as limited amount of audit time/effort, limited insights into project specification/implementation, limited security expertise in the new and fast evolving technologies, limited audit scope, significant project complexity and limitations of automated/manual analysis.
    2. Not all audits are equal — it greatly depends on the expertise/experience of auditors, effort invested vis-a-vis project complexity/quality and tools/processes used.
    3. Audits provide a project’s security snapshot over a brief (typically few weeks) period. However, smart contracts need to evolve over time to add new features, fix bugs or optimize. Relying on external audits after every change is impractical.
13. Audit Reports: include details of the scope, goals, effort, timeline, approach, tools/techniques used, findings summary, vulnerability details, vulnerability classification, vulnerability severity/difficulty/likelihood, vulnerability exploit scenarios, vulnerability fixes and informational recommendations/suggestions on programming best-practices.
14. Audit Findings Classification: The vulnerabilities found during the audit are typically classified into different categories which helps to understand the nature of the vulnerability, potential impact/severity, impacted project components/functionality and exploit scenarios. Trail of Bits, for example, uses the below classification:
    1. Access Controls: Related to authorization of users and assessment of rights
    2. Auditing and Logging: Related to auditing of actions or logging of problems
    3. Authentication: Related to the identification of users
    4. Configuration: Related to security configurations of servers, devices or software
    5. Cryptography: Related to protecting the privacy or integrity of data
    6. Data Exposure: Related to unintended exposure of sensitive information
    7. Data Validation: Related to improper reliance on the structure or values of data
    8. Denial of Service: Related to causing system failure
    9. Error Reporting: Related to the reporting of error conditions in a secure fashion
    10. Patching: Related to keeping software up to date
    11. Session Management: Related to the identification of authenticated users
    12. Timing: Related to race conditions, locking or order of operations
    13. Undefined Behavior: Related to undefined behavior triggered by the program
15. Audit Findings Likelihood/Difficulty: Per [OWASP](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology), likelihood or difficulty is a rough measure of how likely or difficult this particular vulnerability is to be uncovered and exploited by an attacker. OWASP proposes three Likelihood levels of Low, Medium and High. Trail of Bits, for example, classifies every finding into four difficulty levels:
    1. Undetermined: The difficulty of exploit was not determined during this engagement
    2. Low: Commonly exploited, public tools exist or can be scripted that exploit this flaw
    3. Medium: Attackers must write an exploit, or need an in-depth knowledge of a complex system
    4. High: The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue
16. Audit Findings Impact: Per OWASP, this estimates the magnitude of the technical and business impact on the system if the vulnerability were to be exploited. OWASP proposes three Impact levels of Low, Medium and High.
17. Audit Findings Severity: Per OWASP, the Likelihood estimate and the Impact estimate are put together to calculate an overall Severity for this risk. This is done by figuring out whether the Likelihood is Low, Medium, or High and then do the same for impact.
    1. OWASP proposes a 3x3 Severity Matrix which combines the three Likelihood levels with the three Impact levels
    2. Severity Matrix (Likelihood-Impact = Severity): Low-Low = Note; Low-Medium = Low; Low-High = Medium; Medium-Low = Low; Medium-Medium = Medium; Medium-High = High; High-Low = Medium; High-Medium = High; High-High = Critical;
    3. Trail of Bits uses:
       1. Informational: The issue does not pose an immediate risk, but is relevant to security best practices or Defence in Depth
       2. Undetermined: The extent of the risk was not determined during this engagement
       3. Low: The risk is relatively small or is not a risk the customer has indicated is important
       4. Medium: Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal implications for client
       5. High: Large numbers of users, very bad for client’s reputation, or serious legal or financial implications
    4. ConsenSys uses:
       1. Minor: issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
       2. Medium: issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
       3. Major: issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
       4. Critical: issues are directly exploitable security vulnerabilities that need to be fixed.
18. Audit Checklist For Projects (See [here](https://blog.trailofbits.com/2018/04/06/how-to-prepare-for-a-security-audit/) for Trail of Bits recommendations):
    1. Resolve the easy issues: 1) Enable and address every last compiler warning 2) Increase unit and feature test coverage 3) Remove dead code, stale branches, unused libraries, and other extraneous weight.
    2. Document: 1) Describe what your product does, who uses it, why, and how it delivers. 2) Add comments about intended behavior in-line with the code. 3) Label and describe your tests and results, both positive and negative. 4) Include past reviews and bugs.
    3. Deliver the code batteries included: 1) Document the steps to create a build environment from scratch on a computer that is fully disconnected from your internal network 2) Include external dependencies 3) Document the build process, including debugging and the test environment 4) Document the deployment process and environment, including all the specific versions of external tools and libraries for this process.
19. Audit Techniques: involve a combination of different methods that are applied to the project codebase with accompanying specification and documentation. Many are automated analyses performed with tools and some require manual assistance.
    1. Specification analysis (manual)
    2. Documentation analysis (manual)
    3. Testing (automated)
    4. Static analysis (automated)
    5. Fuzzing (automated)
    6. Symbolic checking (automated)
    7. Formal verification (automated)
    8. Manual analysis (manual)

One may also think of these as manual/semi-automated/fully-automated, where the distinction between semi-automated and fully-automated is the difference between a tool that requires a user to define properties versus a tool that requires (almost) no user configuration except to triage results. Fully-automated tools tend to be straightforward to use, while semi-automated tools require some human assistance and are therefore more resource-expensive.

1. Specification analysis: Specification describes in detail what (and sometimes why) the project and its various components are supposed to do functionally as part of their design and architecture.
   1. From a security perspective, it specifies what the assets are, where they are held, who are the actors, privileges of actors, who is allowed to access what and when, trust relationships, threat model, potential attack vectors, scenarios and mitigations.
   2. Analysing the specification of a project provides auditors with the above details and lets them evaluate the assumptions made and indicate any shortcomings
   3. Very few smart contract projects have detailed specifications at their first audit stage. At best, they have some documentation about what is implemented. Auditors spend a lot of time inferring specification from documentation/implementation which leaves them with less time for vulnerability assessment.
2. Documentation analysis: Documentation is a description of what has been implemented based on the design and architectural requirements.
   1. Documentation answers ‘how’ something has been designed/architected/implemented without necessarily addressing the ‘why’ and the design/requirement goals
   2. Documentation is typically in the form of Readme files in the Github repository describing individual contract functionality combined with functional NatSpec and individual code comments.
   3. Documentation in many cases serves as a substitute for specification and provides critical insights into the assumptions, requirements and goals of the project team
   4. Understanding the documentation before looking at the code helps auditors save time in inferring the architecture of the project, contract interactions, program constraints, asset flow, actors, threat model and risk mitigation measures
   5. Mismatches between the documentation and the code could indicate stale/poor documentation, software defects or security vulnerabilities
   6. Auditors are expected to encourage the project team to document thoroughly so that they do not need to waste their time inferring this by reading code
3. Testing: Software testing or validation is a well-known fundamental software engineering primitive to determine if software produces expected outputs when executed with different chosen inputs.
   1. Smart contract testing has a similar motivation but is arguably more complicated despite their relatively smaller sizes (in lines of code) compared to Web2 software
   2. Smart contract development platforms (Truffle, Embark, Brownie, Waffle, Hardhat etc.) are relatively new with different levels of support for testing
   3. Projects, in general, have very little testing done at the audit stage. Testing integrations and composability with mainnet contracts and state is non-trivial
   4. Test coverage and test cases give a good indication of project maturity and also provide valuable insights to auditors into assumptions/edge-cases for vulnerability assessments
   5. Auditors should expect a high-level of testing and test coverage because this is a must-have software-engineering discipline, especially when smart contracts that are by-design exposed to everyone on the blockchain end up holding assets worth tens of millions of dollars
   6. "Program testing can be used to show the presence of bugs, but never to show their absence!” - E.W. Dijkstra
4. Static analysis: is a technique of analyzing program properties without actually executing the program.
   1. This is in contrast to software testing where programs are actually executed/run with different inputs
   2. For smart contracts, static analysis can be performed on the Solidity code or on the EVM bytecode. [Slither](https://github.com/crytic/slither) performs static analysis at the Solidity level while [Mythril](https://github.com/ConsenSys/mythril) analyzes EVM bytecode.
   3. Static analysis typically is a combination of control flow and data flow analyses
5. [Fuzzing](https://en.wikipedia.org/wiki/Fuzzing): or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks
   1. Fuzzing is especially relevant to smart contracts because anyone can interact with them on the blockchain with random inputs without necessarily having a valid reason or expectation (arbitrary byzantine behaviour)
   2. [Echidna](https://github.com/crytic/echidna) and [Harvey](https://mariachris.github.io/Pubs/FSE-2020-Harvey.pdf) are two popular tools for smart contract fuzzing
6. [Symbolic checking](https://en.wikipedia.org/wiki/Model_checking#Symbolic_model_checking): is a technique of checking for program correctness, i.e. proving/verifying, by using symbolic inputs to represent set of states and transitions instead of enumerating individual states/transitions separately
   1. Model checking or property checking is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness)
   2. In order to solve such a problem algorithmically, both the model of the system and its specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a structure satisfies a given logical formula.
   3. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure
   4. Instead of enumerating reachable states one at a time, the state space can sometimes be traversed more efficiently by considering large numbers of states at a single step. When such state space traversal is based on representations of a set of states and transition relations as logical formulas, binary decision diagrams (BDD) or other related data structures, the model-checking method is symbolic.
   5. Model-checking tools face a combinatorial blow up of the state-space, commonly known as the state explosion problem, that must be addressed to solve most real-world problems
   6. Symbolic algorithms avoid explicitly constructing the graph for the finite state machines (FSM); instead, they represent the graph implicitly using a formula in quantified propositional logic
7. [Formal verification](https://en.wikipedia.org/wiki/Formal_verification): is the act of proving or disproving the correctness of intended  algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics
   1. Formal verification is effective at detecting complex bugs which are hard to detect manually or using simpler automated tools
   2. Formal verification needs a specification of the program being verified and techniques to translate/compare the specification with the actual implementation
   3. [Certora’s](https://www.certora.com/) Prover and ChainSecurity’s [VerX](http://verx.ch/) are examples of formal verification tools for smart contracts. [KEVM](https://github.com/kframework/evm-semantics) from Runtime Verification Inc is a formal verification framework that models EVM semantics.
8. Manual analysis: is complimentary to automated analysis using tools and serves a critical need in smart contract audits
   1. Automated analysis using tools is cheap (typically open-source free software), fast, deterministic and scalable (varies depending on the tool being semi-/fully-automated) but however is only as good as the properties it is made aware of, which is typically limited to Solidity and EVM related constraints
   2. Manual analysis with humans, in contrast, is expensive, slow, non-deterministic and not scalable because human expertise in smart contact security is a rare/expensive skill set today and we are slower, prone to error and inconsistent.
   3. Manual analysis is however the only way today to infer and evaluate business logic and application-level constraints which is where a majority of the serious vulnerabilities are being found
9. False Positives: are findings which indicate the presence of vulnerabilities but which in fact are not vulnerabilities. Such false positives could be due to incorrect assumptions or simplifications in analysis which do not correctly consider all the factors required for the actual presence of vulnerabilities.
   1. False positives require further manual analysis on findings to investigate if they are indeed false or true positives
   2. High number of false positives increases manual effort in verification and lowers the confidence in the accuracy of the earlier automated/manual analysis
   3. True positives might sometimes be classified as false positives which leads to vulnerabilities being exploited instead of being fixed
10. False Negatives: are missed findings that should have indicated the presence of vulnerabilities but which are in fact are not reported at all. Such false negatives could be due to incorrect assumptions or inaccuracies in analysis which do not correctly consider the minimum factors required for the actual presence of vulnerabilities.
    1. False negatives, per definition, are not reported or even realised unless a different analysis reveals their presence or the vulnerabilities are exploited
    2. High number of false negatives lowers the confidence in the effectiveness of the earlier manual/automated analysis.
11. Audit Firms (representative; not exhaustive): [ABDK](https://www.abdk.consulting/), [Arcadia](https://arcadiamgroup.com/), [Beosin](https://www.cybersecurityintelligence.com/beosin-5834.html), [Blockchain Consilium](https://www.blockchainconsilium.com/), [BlockSec](https://www.blocksecteam.com/), [CertiK](https://certik.io/), [ChainSafe](https://chainsafe.io/), [ChainSecurity](https://chainsecurity.com/), [Chainsulting](https://chainsulting.de/smart-contract-audit/), [CoinFabrik](https://www.coinfabrik.com/services/smart-contract-audits/), [ConsenSys Diligence](https://consensys.net/diligence/), [Dedaub](https://www.dedaub.com/), [G0](https://github.com/g0-group), [Hacken](https://hacken.io/), [Haechi](https://audit.haechi.io/), [Halborn](https://halborn.com/), [HashEx](https://hashex.org/smart_contracts_and_audit/), [Iosiro](https://iosiro.com/services/smart-contract-auditing), [Least Authority](https://leastauthority.com/), [MixBytes](https://mixbytes.io/audit), [NCC](https://www.nccgroup.com/us/our-services/cyber-security/specialist-practices/cryptography-services/blockchain-security/), [NewAlchemy](https://audits.newalchemy.io/), [OpenZeppelin](https://openzeppelin.com/), [PeckShield](https://peckshield.com/en), [Pessimistic](https://pessimistic.io/), [PepperSec](https://peppersec.com/smart-contract-audit.html), [Pickle](https://pickle.solutions/security-audits/), [Quantstamp](https://quantstamp.com/), [QuillHash](https://audits.quillhash.com/smart-contract-audit), [Runtime Verification](https://runtimeverification.com/), [Sigma Prime](https://sigmaprime.io/), [SlowMist](https://www.slowmist.com/en/), [SmartDec](https://smartdec.net/), [Solidified](https://solidified.io/), [Somish](https://www.somish.com/blockchain/smart-contract-audit/), [Trail of Bits](https://www.trailofbits.com/) and [Zokyo](https://www.zokyo.io/).
12. Smart contract security tools: are critical in assisting smart contract developers and auditors with showcasing (potentially) exploitable vulnerabilities, highlighting dangerous programming styles or surfacing common patterns of misuse. None of these however replace the need for manual review/validation to evaluate contract-specific business logic and other complex control-flow, data-flow & value-flow aspects.
13. Categories of security tools: tools for testing, test coverage, linting, disassembling, visualization, static analysis, dynamic analysis and formal verification of smart contracts.
14. [Slither](https://github.com/crytic/slither) *is a Solidity static analysis framework written in Python 3. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.*It implements [74 detectors](https://github.com/crytic/slither#detectors) in the publicly available free version (with [trophies](https://github.com/crytic/slither/blob/master/trophies.md) that showcase Slither findings in real-world contracts).
15. Slither features:
    1. Detects vulnerable Solidity code with low false positives
    2. Identifies where the error condition occurs in the source code
    3. Easily integrates into continuous integration and Truffle builds
    4. Built-in 'printers' quickly report crucial contract information
    5. Detector API to write custom analyses in Python
    6. Ability to analyze contracts written with Solidity >= 0.4
    7. Intermediate representation (SlithIR) enables simple, high-precision analyses
    8. Correctly parses 99.9% of all public Solidity code
    9. Average execution time of less than 1 second per contract
16. Slither bugs and optimizations detection can run on a Truffle/Embark/Dapp/Etherlime/Hardhat application or on a single Solidity file:
    1. Slither runs all its detectors by default. To run only selected detectors, use *--detect detector1,detector2*. To exclude detectors, use *--exclude detector1,detector2*.
    2. To exclude detectors with an informational or low severity, use *--exclude-informational* or *--exclude-low*
    3. *--list-detectors* lists available detectors
17. Slither printers allow printing contract information with *--print* and following options (with contract-summary, human-summary, and inheritance-graph for quick review, and others such as call-graph, cfg, function-summary and vars-and-auth for in-depth review):
    1. call-graph: Export the call-graph of the contracts to a dot file
    2. cfg: Export the CFG of each functions
    3. constructor-calls: Print the constructors executed
    4. contract-summary: Print a summary of the contracts
    5. data-dependency: Print the data dependencies of the variables
    6. echidna: Export Echidna guiding information
    7. evm: Print the evm instructions of nodes in functions
    8. function-id: Print the keccack256 signature of the functions
    9. function-summary: Print a summary of the functions
    10. human-summary: Print a human-readable summary of the contracts
    11. inheritance: Print the inheritance relations between contracts
    12. inheritance-graph: Export the inheritance graph of each contract to a dot file
    13. modifiers: Print the modifiers called by each function
    14. require: Print the require and assert calls of each function
    15. slithir: Print the slithIR representation of the functions
    16. slithir-ssa: Print the slithIR representation of the functions
    17. variable-order: Print the storage order of the state variables
    18. vars-and-auth: Print the state variables written and the authorization of the functions
18. Slither upgradeability checks helps review contracts that use the delegatecall proxy pattern using *slither-check-upgradeability* tool with following options:
    1. became-constant: Variables that should not be constant
    2. function-id-collision: Functions ids collision
    3. function-shadowing: Functions shadowing
    4. missing-calls: Missing calls to init functions
    5. missing-init-modifier: initializer() is not called
    6. multiple-calls: Init functions called multiple times
    7. order-vars-contracts: Incorrect vars order with the v2
    8. order-vars-proxy: Incorrect vars order with the proxy
    9. variables-initialized: State variables with an initial value
    10. were-constant: Variables that should be constant
    11. extra-vars-proxy: Extra vars in the proxy
    12. missing-variables: Variable missing in the v2
    13. extra-vars-v2: Extra vars in the v2
    14. init-inherited: Initializable is not inherited
    15. init-missing: Initializable is missing
    16. initialize-target: Initialize function that must be called
    17. initializer-missing: initializer() is missing
19. Slither [code similarity detector](https://blog.trailofbits.com/2020/10/23/efficient-audits-with-machine-learning-and-slither-simil/) (a research-oriented tool) uses state-of-the-art machine learning to detect similar (vulnerable) Solidity functions
    1. It uses a pre-trained model from etherscan\_verified\_contracts with 60,000 contracts and more than 850,000 functions
    2. It uses FastText, a vector embedding technique, to generate compact numerical representations of every function
    3. It has four modes: (1) *test* - finds similar functions to your own in a dataset of contracts (2) *plot* - provide a visual representation of similarity of multiple sampled functions (3) *train* - builds new models of large datasets of contracts (4) *info* - inspects the internal information of the pre-trained model or the assessed code
20. Slither contract flattening tool *slither-flat* produces a flattened version of the codebase with the following features:
    1. Supports three strategies: 1) MostDerived: Export all the most derived contracts (every file is standalone) 2) OneFile: Export all the contracts in one standalone file 3) LocalImport: Export every contract in one separate file, and include import ".." in their preludes
    2. Supports circular dependency
    3. Supports all the compilation platforms (Truffle, embark, buidler, etherlime, ...).
21. Slither format tool *slither-format* generates automatically patches. The patches are compatible with git. Patches should be carefully reviewed before applying. Detectors supported with this tool are:
    1. unused-state
    2. solc-version
    3. pragma
    4. naming-convention
    5. external-function
    6. constable-states
    7. constant-function
22. Slither ERC conformance tool *slither-check-erc* checks the following for ERC's conformance for ERC20, ERC721, ERC777, ERC165, ERC223 and ERC1820:
    1. All the functions are present
    2. All the events are present
    3. Functions return the correct type
    4. Functions that must be view are view
    5. Events' parameters are correctly indexed
    6. The functions emit the events
    7. Derived contracts do not break the conformance
23. Slither property generation tool *slither-prop* generates code properties (e.g., invariants) that can be tested with unit tests or Echidna, entirely automatically. The ERC20 scenarios that can be tested are:
    1. Transferable - Test the correct tokens transfer
    2. Pausable - Test the pausable functionality
    3. NotMintable - Test that no one can mint tokens
    4. NotMintableNotBurnable - Test that no one can mint or burn tokens
    5. NotBurnable - Test that no one can burn tokens
    6. Burnable - Test the burn of tokens. Require the "burn(address) returns()" function
24. Slither new detectors: Slither’s plugin architecture lets you integrate new detectors that run from the command line. The skeleton for a detector has:
    1. *ARGUMENT*: lets you run the detector from the command line
    2. *HELP*: is the information printed from the command line
    3. *IMPACT*: indicates the impact of the issue. Allowed values are INFORMATIONAL|LOW|MEDIUM|HIGH
    4. *CONFIDENCE*: indicates your confidence in the analysis. Allowed values are LOW|MEDIUM|HIGH
    5. *WIKI*: constants are used to generate automatically the documentation.
    6. *\_detect()* is the function that implements the detector logic and needs to return a list of findings.
25. [Manticore](https://github.com/trailofbits/manticore) is a symbolic execution tool for analysis of Ethereum smart contracts (besides Linux binaries & WASM modules). See [tutorial](https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/manticore) for details.
    1. Program Exploration: Manticore can execute a program with symbolic inputs and explore all the possible states it can reach
    2. Input Generation: Manticore can automatically produce concrete inputs that result in a given program state
    3. Error Discovery: Manticore can detect crashes and other failure cases in binaries and smart contracts
    4. Instrumentation: Manticore provides fine-grained control of state exploration via event callbacks and instruction hooks
    5. Programmatic Interface: Manticore exposes programmatic access to its analysis engine via a Python API
26. [Echidna](https://github.com/crytic/echidna) is *a Haskell program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions.*
27. Echidna Features:
    1. Generates inputs tailored to your actual code
    2. Optional corpus collection, mutation and coverage guidance to find deeper bugs
    3. Powered by Slither to extract useful information before the fuzzing campaign
    4. Source code integration to identify which lines are covered after the fuzzing campaign
    5. Curses-based retro UI, text-only or JSON output
    6. Automatic test case minimization for quick triage
    7. Seamless integration into the development workflow
    8. Maximum gas usage reporting of the fuzzing campaign
    9. Support for a complex contract initialization with Etheno and Truffle
28. Echidna Usage (see [tutorial](https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/echidna#echidna-tutorial) for details):
    1. Executing the test runner: The core Echidna functionality is an executable called echidna-test. echidna-test takes a contract and a list of invariants (properties that should always remain true) as input. For each invariant, it generates random sequences of calls to the contract and checks if the invariant holds. If it can find some way to falsify the invariant, it prints the call sequence that does so. If it can't, you have some assurance the contract is safe.
    2. Writing invariants: Invariants are expressed as Solidity functions with names that begin with echidna\_, have no arguments, and return a boolean.
    3. Collecting and visualizing coverage: After finishing a campaign, Echidna can save a coverage maximizing corpus in a special directory specified with the corpusDir config option. This directory will contain two entries: (1) a directory named coverage with JSON files that can be replayed by Echidna and (2) a plain-text file named covered.txt, a copy of the source code with coverage annotations.
29. [Eth-security-toolbox](https://github.com/crytic/eth-security-toolbox) is *a Docker container preinstalled and preconfigured with all of Trail of Bits’ Ethereum security tools*. This includes:
    1. Echidna property-based fuzz tester
    2. Etheno integration tool and differential tester
    3. Manticore symbolic analyzer and formal contract verifier
    4. Slither static analysis tool
    5. Rattle EVM lifter
    6. Not So Smart Contracts repository
30. [Ethersplay](https://github.com/crytic/ethersplay) is *a* *Binary Ninja plugin which enables an EVM disassembler and related analysis tools*.
    1. Takes as input the evm bytecode in raw binary format
    2. Renders control flow graph of all functions
    3. Shows Manticore coverage
31. [Pyevmasm](https://github.com/crytic/pyevmasm) is *an assembler and disassembler library for the Ethereum Virtual Machine (EVM)*. It includes a command line utility and a Python API.
32. [Rattle](https://github.com/crytic/rattle) is *an EVM binary static analysis framework designed to work on deployed smart contracts*(not actively developed anymore).
    1. Takes EVM byte strings and uses a flow-sensitive analysis to recover the original control flow graph
    2. Lifts the control flow graph into an SSA/infinite register form, and optimizes the SSA – removing DUPs, SWAPs, PUSHs, and POPs
    3. The conversion from a stack machine to SSA form removes 60%+ of all EVM instructions and presents a much friendlier interface to those who wish to read the smart contracts they’re interacting with
33. [Evm\_cfg\_builder](https://github.com/crytic/evm_cfg_builder) is *a tool used to extract a control flow graph (CFG) from EVM bytecode and used by Ethersplay, Manticore, and other tools from Trail of Bits*.
    1. Reliably recovers a Control Flow Graph (CFG) from EVM bytecode using a dedicated Value Set Analysis
    2. Recovers functions names
    3. Recovers attributes (e.g., payable, view, pure)
    4. Outputs the CFG to a dot file
    5. Library API
34. [Crytic-compile](https://github.com/crytic/crytic-compile) is a smart contract compilation library which is used in Trail of Bits’ security tools and supports Truffle, Embark, Etherscan, Brownie, Waffle, Hardhat and other development environments. The plugin is used in Crytic tools, including:
    1. Slither
    2. Echidna
    3. Manticore
    4. evm-cfg-builder
35. [Solc-select](https://github.com/crytic/solc-select) is *a script to quickly switch between Solidity compiler versions*.
    1. solc-select: manages installing and setting different solc compiler versions
    2. solc: wrapper around solc which picks the right version according to what was set via solc-select
    3. solc binaries are downloaded from https://binaries.soliditylang.org/ which contains official artifacts for many historical and modern solc versions for Linux and macOS
36. [Etheno](https://github.com/crytic/etheno) is *the Ethereum testing Swiss Army knife. It’s a JSON RPC multiplexer, analysis tool wrapper, and test integration tool*.
    1. JSON RPC Multiplexing: Etheno runs a JSON RPC server that can multiplex calls to one or more clients: 1) API for filtering and modifying JSON RPC calls 2) Enables differential testing by sending JSON RPC sequences to multiple Ethereum clients 3) Deploy to and interact with multiple networks at the same time
    2. Analysis Tool Wrapper: Etheno provides a JSON RPC client for advanced analysis tools like Manticore 1) Lowers barrier to entry for using advanced analysis tools 2) No need for custom scripts to set up account and contract state 3) Analyze arbitrary transactions without Solidity source code
    3. Integration with Test Frameworks like Ganache and Truffle: 1) Run a local test network with a single command 2) Use Truffle migrations to bootstrap Manticore analyses 3) Symbolic semantic annotations within unit tests
37. [MythX](https://mythx.io/) is *a powerful security analysis service that finds Solidity vulnerabilities in your Ethereum smart contract code during your development life cycle.*It is a [paid API-based service](https://mythx.io/plans/) which uses [several tools](https://mythx.io/tools/) on the backend including a static analyzer (Maru), symbolic analyzer (Mythril) and a greybox fuzzer (Harvey) to implement a total of [46 detectors](https://mythx.io/detectors/). [Mythril](https://github.com/ConsenSys/mythril) is the open-source component of MythX.
38. MythX process:
    1. Submit your code: The analysis requests are encrypted with TLS and the code you submit is accessed only by you. Submit both the source code and the compiled bytecode of your smart contracts for best results.
    2. Activate a full suite of analysis techniques: The longer MythX runs, the more it can detect more security weaknesses.
    3. Receive a detailed analysis report: MythX detects a majority of vulnerabilities listed in the SWC Registry. The report will return a listing of all the weaknesses found in your code, including the exact position of the issue and its SWC ID. Reports generated can be only accessed by you. MythX offers 3 scan modes, quick, standard and deep. You can see the differences here.
39. MythX tools: When you submit your code to the API it gets analyzed by multiple microservices in parallel where these tools cooperate to return the more comprehensive results in the execution time provided.
    1. A static analyzer that parses the Solidity AST
    2. a symbolic analyzer that detects possible vulnerable states, and
    3. a greybox fuzzer that detects vulnerable execution paths
40. MythX coverage: extends to most SWCs found in the [SWC Registry](https://swcregistry.io/) with the 46 detectors listed [here](https://mythx.io/detectors/) along with the type of analysis used.
41. MythX is based on a SaaS (Security as a Service) platform based on the premise that:
    1. Higher performance compared to running security tools locally
    2. Higher vulnerability coverage than any standalone tool
    3. Benefit from continuous improvements to our security analysis technology with new and improved security tests as the smart contract security landscape evolves.
42. MythX privacy guarantee for the smart contract code submitted using their SaaS APIs:
    1. Code analysis requests are encrypted with TLS
    2. To provide comprehensive reports and improve performance, it stores some of the contract data in our database, including parts of the source code and bytecode but that data never leaves their secure server and is not shared with any outside parties.
    3. It keeps the results of your analysis so you can retrieve them later, but the report can be accessed by you only.
43. MythX running time: Quick scan runs for 5 minutes, Standard scan runs for 30 minutes, and Deep scan runs for 90 minutes.
44. MythX official integrations, tools and libraries include:
    1. MythX CLI: Unified tool to use MythX as a Command Line Interface (CLI) now with full Truffle Projects support.
    2. MythX-JS: Typescript library to integrate MythX in your JS or TS projects.
    3. PythX: Python library to integrate MythX in your Python projects.
    4. MythX VSCode: VSCode extension which allows you to scan smart-contracts and view results directly from your code editor.
45. MythX pricing:
    1. On Demand (US$9.99/3 scans): All scan modes and Prepaid scan packs
    2. Developer (US$49/mo): Quick and Standard scan modes; 500 scans/month
    3. Professional (US$249/mo): All scan modes; 10 000 scans/month
    4. Enterprise (Custom pricing): Custom plans for your team's specific needs; Custom Verification Service; Retainer for Custom Support
46. [Scribble](https://github.com/consensys/scribble) is *a verification language and runtime verification tool that translates high-level specifications into solidity code. It allows you to annotate a solidity smart contract with properties*(See[here](https://docs.scribble.codes/)*)*.
    1. Principles/Goals: 1) Specifications are easy to understand by developers and auditors 2) Specifications are simple to reason about 3) Specifications can be efficiently checked using off-the-shelf analysis tools 4) A small number of core specification constructs are sufficient to express and reason about more advanced constructs
    2. Transforms annotations in the Scribble specification language into concrete assertions
    3. With these instrumented but equivalent contracts, one can then use Mythril, Harvey, MythX
47. [Fuzzing-as-a-Service](https://consensys.net/diligence/fuzzing/): is a service recently launched by ConsenSys Diligence where projects can submit their smart contracts along with embedded inlined specifications or properties written using the Scribble language. These contracts are run through the Harvey fuzzer which uses the specified properties to optimize fuzzing campaigns. Any violations from fuzzing are reported back from the service for the project to fix.
48. [Karl](https://github.com/cleanunicorn/karl) is*a monitor for smart contracts that checks for security vulnerabilities using the Mythril detection engine. It can be used to monitor the Ethereum blockchain for newly deployed vulnerable smart contracts in real-time.*
49. [Theo](https://github.com/cleanunicorn/theo) is*an exploitation tool with a Metasploit-like interface, drops you into a Python REPL console, where you can use the available features to do smart contract reconnaissance, check the storage, run exploits or frontrun or backrun transactions targeting a specific smart contract.*Features:
    1. Automatic smart contract scanning which generates a list of possible exploits
    2. Sending transactions to exploit a smart contract
    3. Transaction pool monitor
    4. Web3 console
    5. Frontrunning and backrunning transactions
    6. Waiting for a list of transactions and sending out others
    7. Estimating gas for transactions means only successful transactions are sent
    8. Disabling gas estimation will send transactions with a fixed gas quantity.
50. Visual Auditoris*a Visual Studio Code extension that provides security-aware syntax and semantic highlighting for*[Solidity](https://marketplace.visualstudio.com/items?itemName=tintinweb.solidity-visual-auditor)*and*[Vyper](https://marketplace.visualstudio.com/items?itemName=tintinweb.vscode-vyper)*.*
    1. Syntax Highlighting: access modifiers (external, public, payable, …), security relevant built-ins, globals, methods and user/miner-tainted information, (address.call(), tx.origin, msg.data, block.\*, now), storage access modifiers (memory, storage), developer notes in comments (TODO, FIXME, HACK, …), custom function modifiers, contract creation / event invocations, easily differentiate between arithmetics vs. logical operations, make Constructor and Fallback function more prominent
    2. Semantic Highlighting: highlights StateVars (constant, inherited), detects and alerts about StateVar shadowing, highlights function arguments in the function body
    3. Review Features: audit annotations/bookmarks - @audit - <msg> @audit-ok - <msg> (see below), generic interface for importing external scanner results - cdili json format (see below), codelens inline action: graph, report, dependencies, inheritance, parse, ftrace, flatten, generate unit test stub, function signature hashes, uml
    4. Graph- and Reporting Features: access your favorite Sūrya features from within vscode, interactive call graphs with call flow highlighting and more, auto-generate UML diagrams from code to support your threat modelling exercises or documentation
    5. Code Augmentation: Hover over Ethereum Account addresses to download the byte-code, source-code or open it in the browser, Hover over ASM instructions to show their signatures, Hover over keywords to show basic Security Notes, Hover over StateVar's to show declaration information
    6. Views: Cockpit vs Outline
51. [Surya](https://github.com/ConsenSys/surya) *aids auditors in understanding and visualizing Solidity smart contracts by providing information about the contracts’ structure and generates call graphs and inheritance graphs. It also supports querying the function call graph in multiple ways to aid in the manual inspection of contracts.*
    1. Integrated with Visual Auditor
    2. Commands: graph, ftrace, flatten, describe, inheritance, dependencies, parse, mdreport
52. [SWC Registry](https://github.com/SmartContractSecurity/SWC-registry): The Smart Contract Weakness Classification Registry (SWC Registry) is an implementation of the weakness classification scheme proposed in EIP-1470.
    1. It is loosely aligned to the terminologies and structure used in the Common Weakness Enumeration (CWE) while overlaying a wide range of weakness variants that are specific to smart contracts
    2. The goals of this project are as follows: 1) Provide a straightforward way to classify security issues in smart contract systems. 2) Define a common language for describing security issues in smart contract systems' architecture, design, or code. 3) Serve as a way to train and increase performance for smart contract security analysis tools.
    3. This repository is maintained by the team behind MythX and currently contains 37 entries
53. [Securify](https://github.com/eth-sri/securify2): is a security scanner for Ethereum smart contracts which Implements static analysis written in Datalog and supports 38 vulnerabilities
54. [VerX](https://verx.ch/): is a verifier that can automatically prove temporal safety properties of Ethereum smart contracts. The verifier is based on a careful combination of three ideas: reduction of temporal safety verification to reachability checking, an efficient symbolic execution engine used to compute precise symbolic states within a transaction, and delayed abstraction which approximates symbolic states at the end of transactions into abstract states.
55. [SmartCheck](https://github.com/smartdec/smartcheck): is an extensible static analysis tool for discovering vulnerabilities and other code issues in Ethereum smart contracts written in the Solidity programming language. It translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns.
56. [K-Framework](https://kframework.org/) based analysis, modelling and verification tools from [Runtime Verification](https://runtimeverification.com/smartcontract/) (RV): provides [KEVM](https://github.com/kframework/evm-semantics) which is a model of EVM in the K-Framework. It is the first executable specification of the EVM that completely passes official test-suites and serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM.
57. [Certora](https://www.certora.com/)[Prover](https://www.certora.com/pubs/QuickGuide.pdf): checks that a smart contract satisfies a set of rules written in a language called Specify. Each rule is checked on all possible transactions, though of course this is not done by explicitly enumerating transactions, but rather through symbolic techniques.
    1. The Certora Prover provides complete path coverage for a set of safety rules provided by the user. For example, a rule might check that only a bounded number of tokens can be minted in an ERC20 contract. The prover either guarantees that a rule holds on all paths and all inputs or produces a test input that demonstrates a violation of the rule.
    2. The problem addressed by the Certora Prover is known to be undecidable which means that there will always be pathological programs and rules for which the Certora prover will time out without a definitive answer
    3. The Certora Prover takes as input the smart contract (either as EVM bytecode or Solidity source code) and a set of rules, written in Certora’s specification language. The Prover then automatically determines whether or not the contract satisfies all the rules using a combination of two computer science techniques: abstract interpretation and constraint solving
58. DappHub’s [Hevm](http://dapp.tools/hevm/): is an implementation of the EVM made specifically for unit testing and debugging smart contracts. It can run unit tests, property tests, interactively debug contracts while showing the Solidity source, or run arbitrary EVM code.
59. Capture the Flag (CTF): are fun and educational challenges where participants have to hack different (dummy) smart contracts that have vulnerabilities in them. They help understand the complexities around how vulnerabilities may be exploited in the wild. Popular ones include:
    1. [Capture The Ether](https://capturetheether.com/): is a set of twenty challenges created by [Steve Marx](https://twitter.com/smarx) which test knowledge of Ethereum concepts of contracts, accounts and math among other things.
    2. [Ethernaut](https://ethernaut.openzeppelin.com/): is a Web3/Solidity based war game from OpenZeppelin that is played in the Ethereum Virtual Machine. Each level is a smart contract that needs to be ‘hacked'. The game is 100% open source and all levels are contributions made by other players
    3. [Damn Vulnerable DeFi v2](https://www.damnvulnerabledefi.xyz/v2-release.html): is a set of 12 DeFi related challenges created by [tinchoabbate](https://twitter.com/tinchoabbate). Depending on the challenge, you should either stop the system from working, steal as much funds as you can, or do some other unexpected things.
    4. [Paradigm CFT](https://ctf.paradigm.xyz/): is a set of seventeen [challenges](https://github.com/paradigm-operations/paradigm-ctf-2021) created by [samczsun](https://twitter.com/samczsun) at Paradigm.
60. Smart contract security tools are useful in assisting auditors while reviewing smart contracts. They automate many of the tasks that can be codified into rules with different levels of coverage, correctness and precision. They are fast, cheap, scalable and deterministic compared to manual analysis. But they are also susceptible to false positives. They are especially well-suited currently to detect common security pitfalls and best-practices at the Solidity and EVM level. With varying degrees of manual assistance, they can also be programmed to check for application-level, business-logic constraints.
61. Audit Process could be thought of as a ten-step process as follows:
    1. Read specification/documentation of the project to understand the requirements, design and architecture
    2. Run fast automated tools such as linters or static analyzers to investigate common Solidity pitfalls or missing smart contract best-practices
    3. Manual code analysis to understand business logic and detect vulnerabilities in it
    4. Run slower but more deeper automated tools such as symbolic checkers, fuzzers or formal verification analyzers which typically require formulation of properties/constraints beforehand, hand holding during the analyses and some post-run evaluation of their results
    5. Discuss (with other auditors) the findings from above to identify any false positives or missing analyses
    6. Convey status to project team for clarifying questions on business logic or threat model
    7. Iterate the above for the duration of the audit leaving some time for report writing
    8. Write report summarizing the above with details on findings and recommendations
    9. Deliver the report to the project team and discuss findings, severity and potential fixes
    10. Evaluate fixes from the project team and verify that they indeed remove the vulnerabilities identified in findings.
62. Reading specification/documentation: For projects that have a specification of the design and architecture of their smart contracts, this is the recommended starting point. Very few new projects have a specification at the audit stage. Some of them have documentation in parts. Some key points:
    1. Specification starts with the project’s technical and business goals and requirements. It describes how the project’s design and architecture help achieve those goals.
    2. The actual implementation of smart contracts is a functional manifestation of the goals, requirements, specification, design and architecture, understanding of which is critical in evaluating if the implementation indeed meets the goals and requirements
    3. Documentation is a description of what has been implemented based on the design and architectural requirements.
    4. Specification answers ‘why’ something needs to be designed/architected/implemented the way it has been done. Documentation answers ‘how’ something has been designed/architected/implemented without necessarily addressing the ‘why’ and leaves it up to the auditors to speculate on the reasons.
    5. Documentation is typically in the form of Readme files describing individual contract functionality combined with functional NatSpec and individual code comments. Encouraging projects to provide a detailed specification and documentation saves a lot of time and effort for the auditors in understanding the project goals/structure and prevents them from making the same assumptions as the implementation which is a leading cause of vulnerabilities.
    6. In the absence of both specification and documentation, auditors are forced to infer goals, requirements, design and architecture from reading code and using tools such as Surya and Slither printers. This takes up a lot of time leaving less time for deeper/complex security analyses.
63. Running static analyzers: Automated tools such as linters or static analyzers help investigate common Solidity pitfalls or missing smart contract best-practices
    1. Tools such as Slither and MythX perform control-flow and data-flow analyses on the smart contracts in the context of their detectors which encode common security pitfalls and best-practices.
    2. Evaluating their findings, which are usually available in seconds/minutes, is a good starting point to detect common vulnerabilities based on well-known constraints/properties of Solidity language, EVM or Ethereum blockchain.
    3. False positives are possible among some of the detector findings and need to be verified manually if they are true/false positives
64. Manual code review: is required to understand business logic and detect vulnerabilities in it.
    1. Automated analyzers do not understand application-level logic and their constraints. They are limited to constraints/properties of Solidity language, EVM or Ethereum blockchain.
    2. Manual analysis of the code is required to detect security-relevant deviations in implementation vis-a-vis the specification or documentation.
    3. Auditors may need to infer business logic and their implied constraints directly from the code or from discussions with the project team and thereafter evaluate if those constraints/properties hold in all parts of the codebase.
65. Running deeper automated tools: such as fuzzers e.g. Echidna, symbolic checkers such as Manticore, tool suite such as MythX and formally verifying custom properties with Scribble or Certora Prover takes more setup and preparation time but helps run deeper analyses to discover edge-cases in application-level properties and mathematical errors, among other things.
    1. Given these require understanding of the project’s application logic, they are recommended to be used at least after an initial manual code review or sometimes after deeper discussion about the specification/implementation with the project team
    2. Analyzing the output of these tools requires significant expertise with the tools themselves, their domain-specific language and sometimes even their inner workings
    3. Evaluating false-positives is sometimes challenging with these tools but the true positives they discover are significant and extreme corner cases missed even by the best manual analyses
66. Brainstorming with other auditors: [Linus’s law](https://en.wikipedia.org/wiki/Linus's_law): ”Given enough eyeballs, all bugs are shallow” might apply with auditors too if they brainstorm on the smart contract implementation, assumptions, findings and vulnerabilities.
    1. While some audit firms encourage active/passive discussion, there are others whose approach is to let auditors separately perform the assessment to encourage independent thinking instead of group thinking. The premise is that group thinking might bias the audit team to focus on certain aspects while missing some vulnerabilities.
    2. A hybrid approach might be interesting where the audit team initially brainstorms to discuss the project’s goals, specification/documentation and implementation but later firewall themselves to independently pursue the assessments and finally come together to compile their findings.
67. Discussion with project team: Having an open communication channel with the project team is  useful to clarify any assumptions in specification, documentation, implementation, or discuss interim findings.
    1. Findings may also be shared with the project team immediately on a private repository to discuss impact, fixes and other implications.
    2. If the audit spans multiple weeks, it may help to have a weekly sync up call. A counterpoint to this is to independently perform the entire assessment so as to not get biased by the project team’s inputs and opinions.
68. Report writing: The audit report is a final compilation of the entire assessment and presents all aspects of the audit including the audit scope/coverage, timeline, team/effort, summaries, tools/techniques, findings, exploit scenarios, suggested fixes, short-/long-term recommendations and any appendices with further details on tools and rationale.
    1. An executive summary typically gives an overview of the audit report with highlights/lowlights illustrating the number/type/severity of vulnerabilities found and an overall assessment of risk. It may also include a description of the smart contracts, (inferred) actors, assets, roles, permissions, access control, interactions, threat model and existing risk mitigation measures
    2. The bulk of the report focuses on the findings from the audit, their type/category, likelihood/impact, severity, justifications for these ratings, potential exploit scenarios, affected parts of smart contracts and potential remediations
    3. It may also address subjective aspects of code quality, readability/auditability and other software-engineering best practices related to documentation, code structure, function/variable naming conventions, test coverage etc. that do not pose an imminent security risk but are indicators of anti-patterns and processes influencing the introduction and persistence of security vulnerabilities
69. Report delivery: The delivery of the report to the project team is a critical deliverable and milestone. Unless interim findings/status is shared, this will be the first time the project team will have access to the assessment details.
    1. The delivery typically happens via a shared online document and is accompanied with a readout where the auditors present the report highlights to the project team for discussion and any debate on findings and their severity ratings
    2. The project team typically takes some time to review the audit report and respond back with any counterpoints on findings, severities or suggested fixes
    3. Depending on the prior agreement, the project team and audit firm might release the audit report publicly (after all required fixes have been made) or the project may decide to keep it private for some reason
70. Evaluating fixes: Post audit, the project team may work on any required fixes for reported findings and request the audit firm for reviewing their responses
    1. Fixes may be applied for a majority of the findings and the review may need to confirm that applied fixes (could be different from audit’s recommended fixes) indeed mitigate the risk reported by the findings
    2. Findings may be contested as not being relevant, outside the project’s threat model or simply acknowledged as being within the project’s acceptable risk model
    3. Audit firms may evaluate the specific fixes applied and confirm/deny their risk mitigation. Unless it is a fix/retainer type audit, this phase typically takes not more than a day because it would usually be outside the agreed upon duration of the audit.
71. Manual review approaches: Auditors have different approaches to manual reviewing smart contract code for vulnerabilities.
    1. Starting with access control
    2. Starting with asset flow
    3. Starting with control flow
    4. Starting with data flow
    5. Inferring constraints
    6. Understanding dependencies
    7. Evaluating assumptions
    8. Evaluating security checklists
72. Starting with access control: Access control is the most fundamental security primitive which addresses ‘who’ has authorised access to ‘what.’ (In a formal access control model, the ‘who’ refers to subjects, ’what’ refers to objects and an access control matrix indicates the permissions between subjects and objects.)
    1. While the overall philosophy might be that smart contracts are permissionless, in reality, they do indeed have different permissions/roles for different actors who interact/use them.
    2. The general classification is that of users and admin(s). For purposes of guarded launch or otherwise, many smart contracts have an admin role that is typically the address that deployed the contract. Admins typically have control over critical configuration and application parameters including (emergency) transfers/withdrawals of contract funds.
    3. Starting with understanding the access control implemented by the smart contracts and checking if they have applied correctly, completely and consistently is a good approach to understanding access flow and detecting violations
73. Starting with asset flow: Assets are Ether or ERC20/ERC721/other tokens managed by smart contracts. Given that exploits target assets of value, it makes sense to start evaluating the flow of assets into/outside/within/across smart contracts and their dependencies.
    1. Who: Assets should be withdrawn/deposited only by authorised/specified addresses as per application logic
    2. When: Assets should be withdrawn/deposited only in authorised/specified time windows or under  authorised/specified  conditions as per application logic (when)
    3. Which: Assets, only those authorised/specified types, should be withdrawn/deposited as per application logic
    4. Why: Assets should be withdrawn/deposited only for authorised/specified reasons as per application logic
    5. Where: Assets should be withdrawn/deposited only to authorised/specified addresses as per application logic
    6. What type: Assets, only of authorised/specified types, should be withdrawn/deposited as per application logic
    7. How much: Assets, only in authorised/specified amounts, should be withdrawn/deposited as per application logic
74. Evaluating control flow: Control flow analyzes the transfer of control, i.e. execution order, across and within smart contracts.
    1. Interprocedural (procedure is just another name for a function) control flow is typically indicated by a call graph which shows which functions (callers) call which other functions (callees), across or within smart contracts
    2. Intraprocedural (i.e. within a function) control flow is dictated by conditionals (if/else), loops (for/while/do/continue/break) and return statements.
    3. Both intra and interprocedural control flow analysis help track the flow of execution and data in smart contracts
75. Evaluating data flow: Data flow analyzes the transfer of data across and within smart contracts
    1. Interprocedural data flow is evaluated by analyzing the data (variables/constants) used as argument values for function parameters at call sites
    2. Intraprocedural data flow is evaluated by analyzing the assignment and use of (state/memory/calldata) variables/constants along the control flow paths within functions.
    3. Both intra and interprocedural data flow analysis help track the flow of global/local storage/memory changes in smart contracts
76. Inferring constraints: Program constraints are basically rules that should be followed by the program. Language-level and EVM-level security constraints are well-known because they are part of the language and EVM specification. However, application-level constraints are rules that are implicit to the business logic implemented and may not be explicitly described in the specification e.g. mint an ERC-721 token to the address when it makes a certain deposit of ERC-20 tokens to the smart contract and burn it when it withdraws the earlier deposit. Such constraints may have to be inferred by the auditors while manually analyzing the smart contract code.
    1. One approach to inferring program constraints is to evaluate what is being done on most program paths related to a particular logic and treat it as a constraint. If such a constraint is missing on one or very few program paths then it could be an indicator of a vulnerability (assuming the constraint is security-related) or those program paths are exceptional conditions where the constraints do not need to hold.
    2. Program constraints can also be verified using a symbolic checker which generates counter-examples or witnesses along execution paths where such constraints do not hold.
77. Understanding dependencies: Dependencies exist when the correct compilation or functioning of program code relies on code/data from other smart contracts that were not necessarily developed by the project team.
    1. Explicit program dependencies are captured in the import statements and the inheritance hierarchy. For e.g., many projects use the community-developed, audited and time-tested smart contracts from OpenZeppelin for tokens, access control, proxy, security etc.
    2. Composability is expected and encouraged via smart contracts interfacing with other protocols and vice-versa, which results in emergent or implicit dependencies on the state/logic of external smart contracts via oracles for example.
    3. This is especially of interest/concern in DeFi protocols that rely on other related protocols for stablecoins, yield generation, borrowing/lending, derivatives, oracles etc.
78. Evaluating assumptions: Many security vulnerabilities result from faulty assumptions e.g. who can access what and when, under what conditions, for what reasons etc. Identifying the assumptions made by the program code and evaluating if they are indeed correct can be the source of many audit findings. Some common examples of faulty assumptions are:
    1. Only admins can call these functions
    2. Initialization functions will only be called once by the contract deployer (e.g. for upgradeable contracts)
    3. Functions will always be called in a certain order (as expected by the specification)
    4. Parameters can only have non-zero values or values within a certain threshold e.g. addresses will never be zero valued
    5. Certain addresses or data values can never be attacker controlled. They can never reach program locations where they can be misused. (In program analysis literature, this is known as taint analysis)
    6. Function calls will always be successful and so checking for return values is not required
79. Evaluating security checklists: Checklists are lists of itemized points that can be quickly and methodically followed (and referenced later by their list number) to make sure all listed items have been processed according to the domain of relevance.
    1. This checklist-based approach was made popular in the book “The Checklist Manifesto. How to Get Things Right” by Atul Gawande who is a noted surgeon, writer and public health leader. In his review of this book, Malcolm Gladwell writes that: “Gawande begins by making a distinction between errors of ignorance (mistakes we make because we don’t know enough), and errors of ineptitude (mistakes we made because we don’t make proper use of what we know). Failure in the modern world, he writes, is really about the second of these errors, and he walks us through a series of examples from medicine showing how the routine tasks of surgeons have now become so incredibly complicated that mistakes of one kind or another are virtually inevitable: it’s just too easy for an otherwise competent doctor to miss a step, or forget to ask a key question or, in the stress and pressure of the moment, to fail to plan properly for every eventuality. Gawande then visits with pilots and the people who build skyscrapers and comes back with a solution. Experts need checklists–literally–written guides that walk them through the key steps in any complex procedure. In the last section of the book, Gawande shows how his research team has taken this idea, developed a safe surgery checklist, and applied it around the world, with staggering success.”
    2. Given the mind-boggling complexities of the fast-evolving Ethereum infrastructure (new platforms, new languages, new tools and new protocols) and the risks associated with deploying smart contracts managing millions of dollars, there are so many things to get right with smart contracts that it is easy to miss a few checks, make incorrect assumptions or fail to consider potential situations. Smart contract experts therefore need checklists too.
    3. Smart contract security checklists (such as the articles in this series) help in navigating the vast number of key aspects to be remembered and applied. They help in going over the itemized features, concepts, pitfalls, best-practices and examples in a methodical manner without missing any items. Checklists are known to increase retention and have a faster recall. They also help in referencing specific items of interest e.g. #42 in Security Pitfalls & Best Practices 101 or #98 in Audit Techniques & Tools 101.
80. Presenting proof-of-concept exploits: Exploits are incidents where vulnerabilities are triggered by malicious actors to misuse smart contracts resulting, for example, in stolen/frozen assets
    1. Presenting proof-of-concepts of such exploits either in code or written descriptions of hypothetical scenarios make audit findings more realistic and relatable by illustrating specific exploit paths and justifying severity of findings
    2. Codified exploits should always be on a testnet, kept private and responsibly disclosed to project teams without any risk of being actually executed on live systems resulting in real loss of funds or access
    3. Descriptive exploit scenarios should make realistic assumptions on roles/powers of actors, practical reasons for their actions and sequencing of events that trigger vulnerabilities and illustrate the paths to exploitation
81. Estimating the likelihood and impact: Likelihood indicates the probability of a vulnerability being discovered by malicious actors and triggered to successfully exploit the underlying weakness. Impact indicates the magnitude of implications on the technical and business aspects of the system if the vulnerability were to be exploited. Estimating if likelihood/impact are low/medium/high is non-trivial in many cases.
    1. If the exploit can be triggered by a few transactions manually without requiring much resources/access (e.g. not admin) and without assuming many conditions to hold true then the likelihood is evaluated as High. Exploits that require deep knowledge of the system workings, privileged roles, large resources or multiple edge conditions to hold true are evaluated as Medium likelihood. Others that require even harder assumptions to hold true, miner collusion, chain forks or insider collusion for e.g., are considered as Low likelihood.
    2. If there is any loss or locking up of funds then the impact is evaluated as High. Exploits that do not affect funds but disrupt the normal functioning of the system are typically evaluated as Medium. Anything else is of Low impact.
    3. Many likelihood and impact evaluations are contentious and debatable between the audit and project teams, typically with security-conscious audit teams pressing for higher likelihood and impact and project teams downplaying the risks.

Estimating the severity: Severity, per OWASP, is a combination of likelihood and impact. With reasonable evaluations of those two, severity estimates from the OWASP matrix should be straightforward.

1. Summary: Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities but not their absence.