## Secureum Solidity 201/102

Solidity supports multiple inheritance including polymorphism:

## Secureum Solidity 201/103

Contract Types:

## Secureum Solidity 201/104

Using For: The directive using A for B; can be used to attach library functions (from the library A) to any type (B) in the context of a contract. These functions will receive the object they are called on as their first parameter.

## Secureum Solidity 201/105

Base Class Functions: It is possible to call functions further up in the inheritance hierarchy internally by explicitly specifying the contract using ContractName.functionName() or using super.functionName() if you want to call the function one level higher up in the flattened inheritance hierarchy

## Secureum Solidity 201/106

## Secureum Solidity 201/107

State Variable Shadowing: This is considered as an error. A derived contract can only declare a state variable x, if there is no visible state variable with the same name in any of its bases.

Function Overriding Changes: The overriding function may only change the visibility of the overridden function from external to public. The mutability may be changed to a more strict one following the order: nonpayable can be overridden by view and pure. view can be overridden by pure. payable is an exception and cannot be changed to any other mutability.

## Secureum Solidity 201/108

Virtual Functions: Functions without implementation have to be marked virtual outside of interfaces. In interfaces, all functions are automatically considered virtual. Functions with private visibility cannot be virtual.

## Secureum Solidity 201/109

Public State Variable Override: Public state variables can override external functions if the parameter and return types of the function matches the getter function of the variable. While public state variables can override external functions, they themselves cannot be overridden.

## Secureum Solidity 201/110

## Secureum Solidity 201/111

Modifier Overriding: Function modifiers can override each other. This works in the same way as function overriding (except that there is no overloading for modifiers). The virtual keyword must be used on the overridden modifier and the override keyword must be used in the overriding modifier

Base Constructors: The constructors of all the base contracts will be called following the linearization rules. If the base constructors have arguments, derived contracts need to specify all of them either in the inheritance list or in the derived constructor.

## Secureum Solidity 201/112

Name Collision Error: It is an error when any of the following pairs in a contract have the same name due to inheritance: 1) a function and a modifier 2) a function and an event 3) an event and a modifier

## Secureum Solidity 201/113

Library Restrictions: In comparison to contracts, libraries are restricted in the following ways:

## Secureum Solidity 201/114

EVM Storage: Storage is a key-value store that maps 256-bit words to 256-bit words and is

## Secureum Solidity 201/115

Storage Layout: State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and

accessed with EVM's SSTORE/SLOAD instructions. All locations in storage are initialized as zero.

mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0

## Secureum Solidity 201/116

Storage Layout Packing: For each state variable, a size in bytes is determined according to its type. Multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

## Secureum Solidity 201/117

Storage Layout & Structs/Arrays:

## Secureum Solidity 201/118

Storage Layout & Inheritance: For contracts that use inheritance, the ordering of state variables is determined by the C3-linearized order of contracts starting with the most base-ward contract. If allowed by the above rules, state variables from different contracts do share the same storage slot.

## Secureum Solidity 201/119

Storage Layout & Types: It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus,

combine multiple reads or writes into a single operation.

## Secureum Solidity 201/120

Storage Layout & Ordering: Ordering of storage variables and struct members affects how they can be packed tightly. For example, declaring your storage variables in the order of uint128, uint128, uint256 instead of uint128, uint256, uint128, as the former will only take up two slots of storage whereas the latter will take up three.

## Secureum Solidity 201/121

Storage Layout for Mappings & Dynamically-sized Arrays: Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored “in between” the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the rules above and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

## Secureum Solidity 201/122

Storage Layout for  Dynamic Arrays: If the storage location of the array ends up being a slot p after applying the storage layout rules, this slot stores the number of elements in the array (byte arrays and strings are an exception). Array data is located starting at keccak256(p) and it is laid out in the same way as statically-sized array data would: One element after the other, potentially sharing storage slots if the elements are not longer than 16 bytes. Dynamic arrays of dynamic arrays apply this rule recursively.

## Secureum Solidity 201/123

Storage Layout for Mappings: For mappings, the slot stays empty, but it is still needed to ensure that even if there are two mappings next to each other, their content ends up at different storage locations. The value corresponding to a mapping key k is located at keccak256(h(k) . p) where . is concatenation and h is a function that is applied to the key depending on its type: 1) for value types, h pads the value to 32 bytes in the same way as when storing the value in memory. 2) for strings and byte arrays, h computes the keccak256 hash of the unpadded data. If the mapping value is a non-value type, the computed slot marks the start of the data. If the value is of struct type, for

example, you have to add an offset corresponding to the struct member to reach the member.

## Secureum Solidity 201/124

Storage Layout for bytes and string: bytes and string are encoded identically. In general, the encoding is similar to byte1[], in the sense that there is a slot for the array itself and a data area that is computed using a keccak256 hash of that slot's position. However, for short values (shorter than 32 bytes) the array elements are stored together with the length in the same slot.

## Secureum Solidity 201/125

EVM Memory: EVM memory is linear and can be addressed at byte level and accessed with MSTORE/MSTORE8/MLOAD instructions. All locations in memory are initialized as zero.

## Secureum Solidity 201/126

Memory Layout: Solidity places new memory objects at the free memory pointer and memory is never freed. The free memory pointer points to 0x80 initially.

## Secureum Solidity 201/127

Reserved Memory: Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

## Secureum Solidity 201/128

Memory Layout & Arrays: Elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for byte[], but not for bytes and string).

## Secureum Solidity 201/129

Free Memory Pointer: There is a “free memory pointer” at position 0x40 in memory. If you want to allocate memory, use the memory starting from where this pointer points at and update it. Considering the reserved memory, allocatable memory starts at 0x80, which is the initial value of the free memory pointer.

## Secureum Solidity 201/130

Zeroed Memory: There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory.

## Secureum Solidity 201/131

Reserved Keywords: These keywords are reserved in Solidity. They might become part of the syntax in the future: *after, alias, apply, auto, case, copyof, default, define, final, immutable, implements, in, inline, let, macro, match, mutable, null, of, partial, promise, reference, relocatable, sealed, sizeof, static, supports, switch, typedef, typeof, unchecked*

## Secureum Solidity 201/132

## Secureum Solidity 201/133

Inline Assembly: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

Inline Assembly Access to External Variables, Functions and Libraries:

## Secureum Solidity 201/134

Yul Syntax: Yul parses comments, literals and identifiers in the same way as Solidity. Inside a code block, the following elements can be used:

## Secureum Solidity 201/135

Solidity v0.6.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

## Secureum Solidity 201/136

Solidity v0.6.0 Explicitness Requirements:

## Secureum Solidity 201/137

Solidity v0.6.0 Semantic and Syntactic Changes:

## Secureum Solidity 201/138

Solidity v0.6.0 New Features:

## Secureum Solidity 201/139

Solidity v0.7.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

## Secureum Solidity 201/140

Solidity v0.7.0 Changes to the Syntax - changes that might cause existing contracts to not compile

## Secureum Solidity 201/141

Solidity v0.7.0 Removal of Unused or Unsafe Features

anymore:

## Secureum Solidity 201/142

Solidity v0.8.0 Breaking Semantic Changes - changes where existing code changes its behaviour without the compiler notifying you about it:

## Secureum Solidity 201/143

Solidity v0.8.0 New Restrictions - changes that might cause existing contracts to not compile anymore:

## Secureum Solidity 201/144

Zero Address Check: address(0) which is 20-bytes of 0's is treated specially in Solidity contracts because the private key corresponding to this address is unknown. Ether and tokens sent to this address cannot be retrieved and setting access control roles to this address also won't work (no private key to sign transactions). Therefore zero addresses should be used with care and checks should be implemented for user-supplied address parameters.

## Secureum Solidity 201/145

*tx.origin* Check: Recall that Ethereum has two types of accounts: Externally Owned Account (EOA) and Contract Account. Transactions can originate only from EOAs. In situations where contracts would like to determine if the *msg.sender* was a contract or not, checking if *msg.sender* is equal to *tx.origin* is an effective check.

## Secureum Solidity 201/146

Overflow/Underflow Check: Until Solidity version 0.8.0 which introduced checked arithmetic by default, arithmetic was unchecked and therefore susceptible to overflows and underflows which could lead to critical vulnerabilities. The recommended best-practice for such contracts is to use OpenZeppelin's SafeMath library for arithmetic.

## Secureum Solidity 201/147

OpenZeppelin Libraries: OpenZeppelin's smart contract libraries are perhaps the most commonly used libraries in smart contract projects. These include contracts for popular token standards, access control, security, safe math, proxies and other utilities.

## Secureum Solidity 201/148

OpenZeppelin ERC20: Implements the popular ERC20 token standard. The functions are:

## Secureum Solidity 201/149

OpenZeppelin SafeERC20: Wrappers around ERC20 operations that throw on failure when the token contract implementation returns false. Tokens that return no value and instead revert or throw on failure are also supported with non-reverting calls assumed to be successful. Adds *safeTransfer*, *safeTransferFrom*, *safeApprove*, *safeDecreaseAllowance*, and *safeIncreaseAllowance*.

## Secureum Solidity 201/150

OpenZeppelin TokenTimelock: A token holder contract that will allow a beneficiary to extract the tokens after a given release time. Useful for simple vesting schedules like "advisors get all of their tokens after 1 year".

## Secureum Solidity 201/151

OpenZeppelin ERC721: Implements the popular ERC721 Non-Fungible Token Standard. The functions are:

## Secureum Solidity 201/152

OpenZeppelin ERC777: Like ERC20, ERC777 is a standard for fungible tokens with improvements such as getting rid of the confusion around decimals, minting and burning with proper events, among others, but its killer feature is receive hooks. ERC777 is backwards compatible with ERC20 (See here)

## Secureum Solidity 201/153

OpenZeppelin ERC1155: is a novel token standard that aims to take the best from previous standards to create a fungibility-agnostic and gas-efficient token contract.

## Secureum Solidity 201/154

OpenZeppelin Ownable: provides a basic access control mechanism, where there is an account (an owner) that can be granted exclusive access to specific functions. By default, the owner account will be the one that deploys the contract. This can later be changed with *transferOwnership*. This module is used through inheritance. It will make available the modifier *onlyOwner*, which can be applied to your functions to restrict their use to the owner.

## Secureum Solidity 201/155

OpenZeppelin AccessControl: provides a general role based access control mechanism. Multiple hierarchical roles can be created and assigned each to multiple accounts. Roles can be used to represent a set of permissions. *hasRole* is used to restrict access to a function call. Roles can be granted and revoked dynamically via the *grantRole* and *revokeRole* functions which can only be called by the role's associated admin accounts.

## Secureum Solidity 201/156

OpenZeppelin Pausable: provides an emergency stop mechanism using functions pause and unpause that can be triggered by an authorized account. This module is used through inheritance. It will make available the modifiers *whenNotPaused* and *whenPaused*, which can be applied to the functions of your contract. Only the functions using the modifiers will be affected when the contract is paused or unpaused.

## Secureum Solidity 201/157

OpenZeppelin ReentrancyGuard: prevents reentrant calls to a function. Inheriting from ReentrancyGuard will make the *nonReentrant* modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

## Secureum Solidity 201/158

## Secureum Solidity 201/159

OpenZeppelin PullPayment: provides a pull-payment strategy, where the paying contract doesn't invoke any functions on the receiver account which must withdraw its payments itself. Pull-payments are often considered the best practice when it comes to sending Ether, security-wise. It prevents recipients from blocking execution and eliminates reentrancy concerns.

## Secureum Solidity 201/160

OpenZeppelin Arrays: Collection of functions related to array types:

OpenZeppelin Address: Collection of functions related to the address type:

## Secureum Solidity 201/161

OpenZeppelin Context: Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via *msg.sender* and *msg.data*, they should not be accessed in such a direct manner, since when dealing with meta-transactions the account sending and paying for execution may not be the actual sender (as far as an application is concerned). This contract is only required for intermediate, library-like contracts.

## Secureum Solidity 201/162

## Secureum Solidity 201/163

OpenZeppelin Counters: Provides counters that can only be incremented or decremented by one. This can be used e.g. to track the number of elements in a mapping, issuing ERC721 ids, or counting request ids. Functions are:

OpenZeppelin Create2: makes usage of the CREATE2 EVM opcode easier and safer. CREATE2 can be used to compute in advance the address where a smart contract will be deployed, which allows for interesting new mechanisms known as 'counterfactual interactions'.

## Secureum Solidity 201/164

OpenZeppelin Multicall: Provides a function to batch together multiple calls in a single external call

## Secureum Solidity 201/165

OpenZeppelin Strings: String operations:

## Secureum Solidity 201/166

OpenZeppelin ECDSA: provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via *web3.eth.sign*, and are a 65 byte array (of type bytes in Solidity) arranged the following way: [[v (1)], [r (32)], [s (32)]]. The data signer can be recovered

## Secureum Solidity 201/167

OpenZeppelin MerkleProof: This deals with verification of Merkle Trees proofs.

with *ECDSA.recover*, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix '\x19Ethereum Signed Message:\n', so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use *toEthSignedMessageHash*.

## Secureum Solidity 201/168

OpenZeppelin SignatureChecker: Provide a single mechanism to verify both private-key (EOA) ECDSA signature and ERC1271 contract signatures. Using this instead of ECDSA.recover in your contract will make them compatible with smart contract wallets such as Argent and Gnosis.

## Secureum Solidity 201/169

OpenZeppelin EIP712: EIP 712 is a standard for hashing and signing of typed structured data. This contract implements the EIP 712 domain separator (*_domainSeparatorV4*) that is used as part of the encoding scheme, and the final step of the encoding to obtain the message digest that is then signed via ECDSA (*_hashTypedDataV4*). Protocols need to implement the type-specific encoding they need in their contracts using a combination of abi.encode and keccak256.

## Secureum Solidity 201/170

OpenZeppelin Escrow: holds funds designated for a payee until they withdraw them. The contract that uses this escrow as its payment method should be its owner, and provide public methods redirecting to the escrow's deposit and withdraw if the escrow rules are satisfied.

## Secureum Solidity 201/171

OpenZeppelin ConditionalEscrow: Derived from Escrow and only allows withdrawal if a condition is met by providing the *withdrawalAllowed()* function which returns whether an address is allowed to withdraw their funds and is to be implemented by derived contracts.

## Secureum Solidity 201/172

OpenZeppelin RefundEscrow: Derived from ConditionalEscrow and holds funds for a beneficiary, deposited from multiple parties. The owner account (that is, the contract that instantiates this contract) may deposit, close the deposit period, and allow for either withdrawal by the beneficiary, or refunds to the depositors.

## Secureum Solidity 201/173

OpenZeppelin ERC165: In Solidity, it's frequently helpful to know whether or not a contract supports an interface you'd like to use. ERC165 is a standard that helps do runtime interface detection using a lookup table. You can register interfaces using *_registerInterface(bytes4)* and *supportsInterface(bytes4 interfaceId)* returns a bool indicating if that interface is supported or not.

## Secureum Solidity 201/174

OpenZeppelin Math: Standard math utilities missing in the Solidity language:

## Secureum Solidity 201/175

OpenZeppelin SafeMath: provides mathematical functions that protect your contract from overflows and underflows. Include the contract with using SafeMath for uint256; and then call the functions:

## Secureum Solidity 201/176

OpenZeppelin SignedSafeMath: provides the same mathematical functions as SafeMath but for signed integers

## Secureum Solidity 201/177

OpenZeppelin SafeCast: Wrappers over Solidity's uintXX/intXX casting operators with added overflow checks. Downcasting from uint256/int256 in Solidity does not revert on overflow. This can easily result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. `SafeCast` restores this intuition by reverting the transaction when such an operation overflows.

## Secureum Solidity 201/178

OpenZeppelin EnumerableMap: Library for managing an enumerable variant of Solidity’s mapping type. Maps have the following properties: 1) Entries are added, removed, and checked for existence in constant time (O(1)) 2) Entries are enumerated in O(n). No guarantees are made on the ordering. As of v3.0.0, only maps of type uint256 → address (UintToAddressMap) are supported.

## Secureum Solidity 201/179

OpenZeppelin EnumerableSet: Library for managing sets of primitive types. Sets have the following properties: 1) Elements are added, removed, and checked for existence in constant time (O(1)) 2) Elements are enumerated in O(n). No guarantees are made on the ordering. As of v3.3.0, sets of type bytes32 (Bytes32Set), address (AddressSet) and uint256 (UintSet) are supported.

## Secureum Solidity 201/180

OpenZeppelin BitMaps: Library for managing uint256 to bool mapping in a compact and efficient way, providing the keys are sequential.

## Secureum Solidity 201/181

OpenZeppelin PaymentSplitter: allows to split Ether payments among a group of accounts. The sender does not need to be aware that the Ether will be split in this way, since it is handled transparently by the contract. The split can be in equal parts or in any other arbitrary proportion. The way this is specified is by assigning each account to a number of shares. Of all the Ether that this contract receives, each account will then be able to claim an amount proportional to the percentage of total shares they were assigned.

## Secureum Solidity 201/182

OpenZeppelin TimelockController: acts as a timelocked controller. When set as the owner of an Ownable smart contract, it enforces a timelock on all *onlyOwner* maintenance operations. This gives time for users of the controlled contract to exit before a potentially dangerous maintenance operation is applied. By default, this contract is self administered, meaning administration tasks have to go through the timelock process. The proposer (resp executor) role is in charge of proposing (resp executing) operations. A common use case is to position this TimelockController as the owner of a smart contract, with a multisig or a DAO as the sole proposer.

## Secureum Solidity 201/183

OpenZeppelin ERC2771Context: A Context variant for ERC2771. ERC2771 provides support for meta transactions, which are transactions that have been:

## Secureum Solidity 201/184

## Secureum Solidity 201/185

OpenZeppelin MinimalForwarder: provides a simple minimal forwarder (as described above) to be used together with an ERC2771 compatible contract. It verifies the nonce and signature of the forwarded request before calling the destination contract.

OpenZeppelin Proxy: This abstract contract provides a fallback function that delegates all calls to another contract using the EVM instruction delegatecall. We refer to the second contract as the implementation behind the proxy, and it has to be specified by overriding the virtual _implementation function. Additionally, delegation to the implementation can be triggered manually through the _fallback function, or to a different contract through the _delegate function. The success and return data of the delegated call will be returned back to the caller of the proxy.

## Secureum Solidity 201/186

OpenZeppelin ERC1967Proxy: implements an upgradeable proxy. It is upgradeable because calls are delegated to an implementation address that can be changed. This address is stored in storage in the location specified by EIP1967, so that it doesn't conflict with the storage layout of the implementation behind the proxy. Upgradeability is only provided internally through *_upgradeTo*.

## Secureum Solidity 201/187

OpenZeppelin TransparentUpgradeableProxy: implements a proxy that is upgradeable by an admin. To avoid proxy selector clashing, which can potentially be used in an attack, this contract uses the transparent proxy pattern. This pattern implies two things that go hand in hand: 1) If any account other than the admin calls the proxy, the call will be forwarded to the implementation, even if that call matches one of the admin functions exposed by the proxy itself 2) If the admin calls the proxy, it can access the admin functions, but its calls will never be forwarded to the implementation. If the admin tries to call a function on the implementation it will fail with an error that says "admin cannot fallback to proxy target".

## Secureum Solidity 201/188

## Secureum Solidity 201/189

OpenZeppelin ProxyAdmin: This is an auxiliary contract meant to be assigned as the admin of a TransparentUpgradeableProxy.

OpenZeppelin BeaconProxy: implements a proxy that gets the implementation address for each call from a UpgradeableBeacon. The beacon address is stored in storage slot uint256(keccak256('eip1967.proxy.beacon')) - 1, so that it doesn't conflict with the storage layout of the implementation behind the proxy.

## Secureum Solidity 201/190

OpenZeppelin UpgradeableBeacon: is used in conjunction with one or more instances of BeaconProxy to determine their implementation contract, which is where they will delegate all function calls. An owner is able to change the implementation the beacon points to, thus upgrading the proxies that use this beacon.

## Secureum Solidity 201/191

OpenZeppelin Clones: EIP 1167 is a standard for deploying minimal proxy contracts, also known as "clones". To simply and cheaply clone contract functionality in an immutable way, this standard specifies a minimal bytecode implementation that delegates all calls to a known, fixed address. The library includes functions to deploy a proxy using either create (traditional deployment) or create2 (salted deterministic deployment). It also includes functions to predict the addresses of clones deployed using the deterministic method.

## Secureum Solidity 201/192

OpenZeppelin Initializable: aids in writing upgradeable contracts, or any kind of contract that will be deployed behind a proxy. Since a proxied contract cannot have a constructor, it is common to move constructor logic to an external initializer function, usually called *initialize*. It then

## Secureum Solidity 201/193

Dappsys DSProxy: implements a proxy deployed as a standalone contract which can then be used by the owner to execute code. A user would pass in the bytecode for the contract as well as the calldata for the function they want to execute.The proxy will

becomes necessary to protect this initializer function so it can only be called once. The initializer modifier provided by this contract will have this effect.

create a contract using the bytecode. It will then delegatecall the function and arguments specified in the calldata.

## Secureum Solidity 201/194

Dappsys DSMath: provides arithmetic functions for the common numerical primitive types of Solidity. You can safely add, subtract, multiply, and divide uint numbers without fear of integer overflow. You can also find the minimum and maximum of two numbers. Additionally, this package provides arithmetic functions for two new higher level numerical concepts called *wad* (18 decimals) and *ray* (27 decimals). These are used to represent fixed-point decimal numbers. A wad is a decimal number with 18 digits of precision and a ray is a decimal number with 27 digits of precision. These functions are necessary to account for the difference between how integer arithmetic behaves normally, and how decimal arithmetic should actually work.

## Secureum Solidity 201/195

Dappsys DSAuth: Provides a flexible and updatable auth pattern which is completely separate from application logic. By default, the auth modifier will restrict function-call access to the including contract owner and the including contract itself. The auth modifier provided by DSAuth triggers the internal isAuthorized function to require that the msg.sender is authorized ie. the sender is either: 1) the contract owner 2) the contract itself or 3) has been granted permission via a specified authority.

## Secureum Solidity 201/196

Dappsys DSGuard: Manages an Access Control List which maps source and destination addresses to function signatures. Intended to be used as an authority for DSAuth where it acts as a lookup table for the *canCall* function to provide boolean answers as to whether a particular address is authorized to call a given function at another address. The ACL is a mapping of *[src][dst][sig] => boolean* where an address src can be either permitted or forbidden access to a function *sig* at address *dst* according to the boolean value. When used as an authority by

## Secureum Solidity 201/197

Dappsys DSRoles: A role-driven authority for ds-auth which facilitates access to lists of user roles and capabilities. Works as a set of lookup tables for the canCall function to provide boolean answers as to whether a user is authorized to call a given function at given address. DSRoles provides 3 different ways of permitting/forbidding function call access to users:1) Root Users: any users added to the _root_users whitelist will be authorized to call any function regardless of what roles or capabilities might be defined. 2) Public Capabilities: public capabilities are global capabilities which apply to all users and take precedence over any user specific role-capabilities which might be defined. 3) Role Capabilities: capabilities

DSAuth the *src* is considered to be the *msg.sender*, the *dst* is the including contract and *sig* is the function which invoked the auth modifier.

which are associated with a particular role. Role capabilities are only checked if the user does not have root access and the capability is not public.

## Secureum Solidity 201/198

WETH: WETH stands for Wrapped Ether. For protocols that work with ERC-20 tokens but also need to handle Ether, WETH contracts allow converting Ether to its ERC-20 equivalent WETH (called wrapping) and vice-versa (called unwrapping). WETH can be created by sending ether to a WETH smart contract where the Ether is stored and in turn receiving the WETH ERC-20 token at a 1:1 ratio. This WETH can be sent back to the same smart contract to be "unwrapped" i.e. redeemed back for the original Ether at a 1:1 ratio. The most widely used WETH contract is WETH9 which holds more than 7 million Ether for now.

## Secureum Solidity 201/199

Uniswap V2: Uniswap is an automated liquidity protocol powered by a constant product formula and implemented in a system of non-upgradeable smart contracts on the Ethereum blockchain. The automated market making algorithm used by Uniswap is $x*y=k$, where x and y represent a token pair that allow one token to be exchanged for the other as long as the "constant product" formula is preserved i.e. trades must not change the product (k) of a pair's reserve balances (x and y). Core concepts:

## Secureum Solidity 201/200

Uniswap V3: Introduces

## Secureum Solidity 201/201

Chainlink Oracles & Price Feeds: Chainlink Price Feeds provide aggregated data (via its *AggregatorV3Interface* contract interface) from various high quality data providers, fed on-chain by decentralized oracles on the Chainlink Network. To get price data into smart contracts for an asset that isn't covered by an existing price feed, such as the price of a

particular stock, one can customize Chainlink oracles to call any external API.