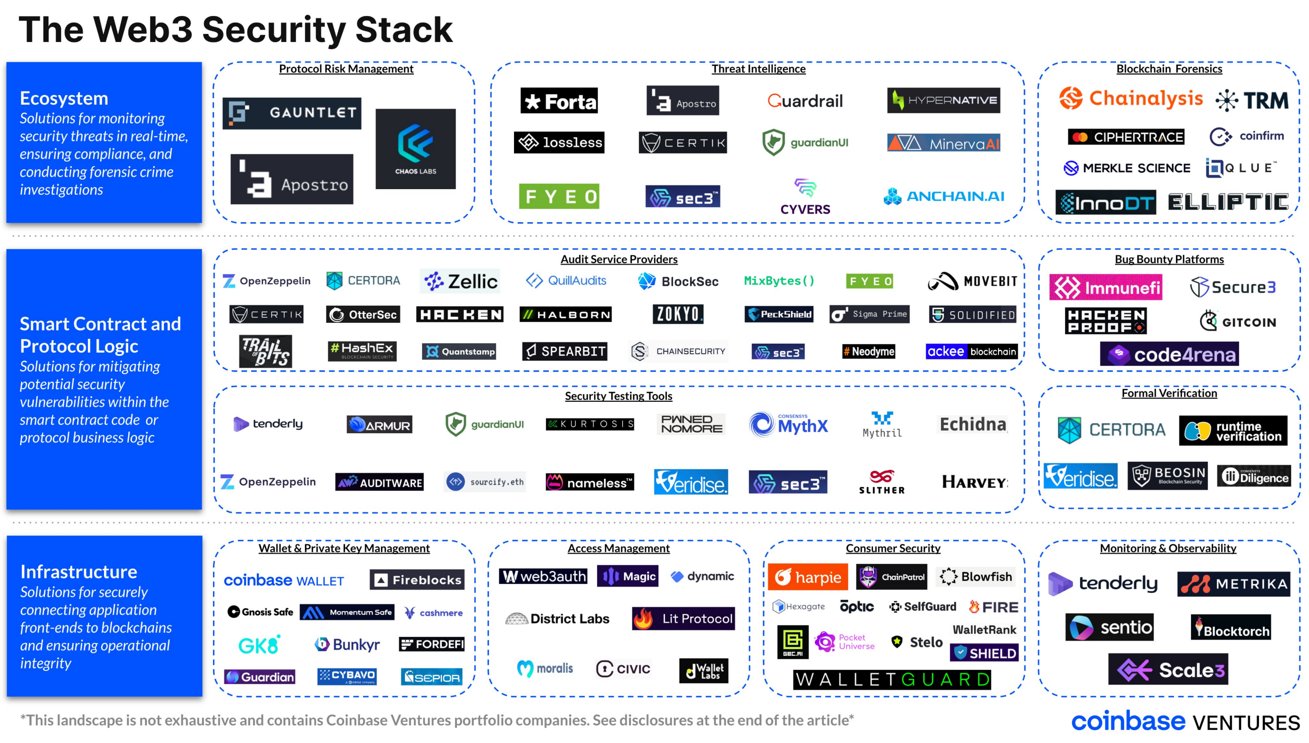
Lesson 19 - Security


The Web3 Security Stack
Ecosystem
Solutions for monitoring security threats in real-time, ensuring compliance, and conducting forensic crime investigations
Protocol Risk Management
Threat Intelligence
Blockchain Forensics
Smart Contract and Protocol Logic
Solutions for mitigating potential security vulnerabilities within the smart contract code or protocol business logic
Audit Service Providers
Bug Bounty Platforms
Security Testing Tools
Formal Verification
Infrastructure
Solutions for securely connecting application front-ends to blockchains and ensuring operational integrity
Wallet & Private Key Management
Access Management
Consumer Security
Monitoring & Observability
This landscape is not exhaustive and contains Coinbase Ventures portfolio companies. See disclosures at the end of the article
coinbase VENTURES

Solidity Specific Exploits

Re entrancy attack

[See]

Integer division

All integer division rounds down to the nearest integer. If you need more precision, consider using a multiplier, or store both the numerator and denominator.

(In the future, Solidity will have a [fixed-point] type, which will make this easier.)

```
// bad uint x = 5 / 2; // Result is 2, all integer division
rounds DOWN to the nearest integer`
```

Using a multiplier prevents rounding down, this multiplier needs to be accounted for when working with x in the future:

```
// good uint multiplier = 10; uint x = (5 * multiplier) / 2;
```

Storing the numerator and denominator means you can calculate the result of `numerator/denominator` off-chain:

```
// good uint numerator = 5; uint denominator = 2;
```

tx.origin

Never use `tx.origin` for authorisation, another contract can have a method which will call your contract (where the user has some funds for instance) and your contract will authorise that transaction as your address is in `tx.origin`.

```
contract MyContract {

    address owner;

    function MyContract() public {
        owner = msg.sender;
    }

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        (bool success, ) = receiver.call.value(amount)("");
        require(success);
    }

}

contract AttackingContract {

    MyContract myContract;
    address attacker;

    function AttackingContract(address myContractAddress)
public {
        myContract = MyContract(myContractAddress);
        attacker = msg.sender;
    }

    function() public {
        myContract.sendTo(attacker, msg.sender.balance);
    }
}
```

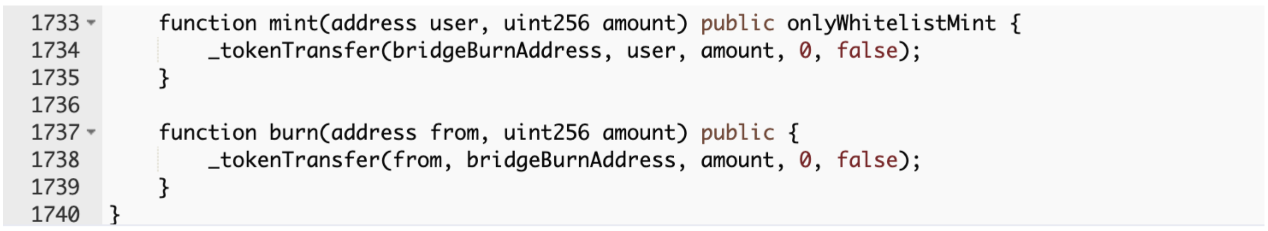
Economic Attacks

Safemoon Exploit

Details from Peckshield, rekt.news

Safemoon lost $8.9M worth of ‘locked LP’ thanks to a bug introduced in the project’s latest upgrade.

The contract was upgraded six hours before the exploit.
In doing so, they introduced public visibility to the burn function.



```
    function mint(address user, uint256 amount) public onlyWhitelistMint {
        _tokenTransfer(bridgeBurnAddress, user, amount, 0, false);
    }

    function burn(address from, uint256 amount) public {
        _tokenTransfer(from, bridgeBurnAddress, amount, 0, false);
    }
}
```

The attacker purchased SFM tokens before the exploit

The attacker was able to burn SFM from the SFM:BNB liquidity pool, pushing up the price of SFM tokens in the pool.

The attacker then sold their SFM tokens at the newly inflated price.

Oracle Manipulation

See

The scenario is simple. A smart contract needs to determine the price of an asset, e.g., when a user deposits ETH into its system. To achieve this price discovery, the protocol consults its respective Uniswap pool as a source.
Exploiting this behaviour, an attacker can take out a flash loan to drain one side of the Uniswap pool. Due to the lack of data source diversity, the protocol's internal price is directly manipulated, e.g., to 100 times the original value. The attacker can now perform an action to capture this additional value. For example, an arbitrage trade on top of the newly created price difference or an advantageous position in the system can be gained.

The problems are two-fold:

1. The use of a single price feed source smart contract allows for easy on-chain manipulation using flash loans.
2. Despite a notable anomaly, the smart contracts consuming the price information continue to operate on the manipulated data.

A more concrete example is provided by the Visor Hack. The [following code]() shows that on deposit, the price feed is fetched directly from Uniswap:

```
uint160 sqrtPrice = TickMath.getSqrtRatioAtTick(currentTick());
uint256 price =
FullMath.mulDiv(uint256(sqrtPrice).mul(uint256(sqrtPrice)),
PRECISION, 2**(96 * 2));
```

Here, `currentTick()` directly fetches the [current price tick]() from a Uniswap pool:

```
// @return tick Uniswap pool's current price tick function
currentTick() public view returns (int24 tick) { (, tick, , , , ,
) = pool.slot0(); }
```

As this price data is fetched from an on-chain dependency, and the price data is determined in the current transaction context, this spot price can be manipulated in the same transaction.

1. An attacker can take out a flash loan on the incoming asset A and on the relevant Uniswap pool, swap asset A for asset B with a large volume.
2. This trade will increase the price of asset B (increased demand) and reduce the cost of asset A (increased supply).
3. When asset B is deposited into the above function, its price is still pumped up by the flash loan.
4. Consequentially, asset B gives the attacker an over-proportional amount of shares.
5. These shares can be withdrawn, giving the attacker equal parts of asset A and asset B from the pool.
6. Repeating this process will drain the vulnerable pool of all funds.

7. With the money gained from the withdrawal of their shares, the attacker can repay the flash loan.

Basic MEV

From https://hackmd.io/@flashbots/quantifying-REV
Maximal (formerly Miner) Extractable Value is the value that can be extracted from a blockchain by any agent without special permissions. Considering this permissionless nature, any agent with transaction ordering rights will be in a privileged position to perform the extraction.

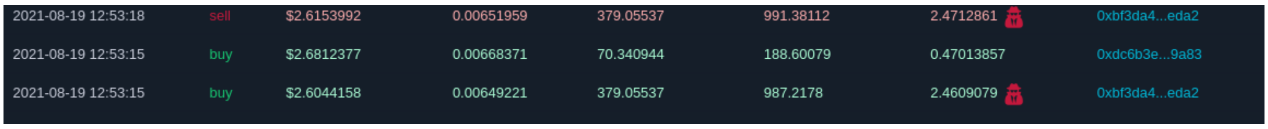
There are features of Ethereum (and other blockchains) that allow front running

1. All transactions are available in a public mempool before they are mined
2. All transaction data is public
3. Transactions can be cloned

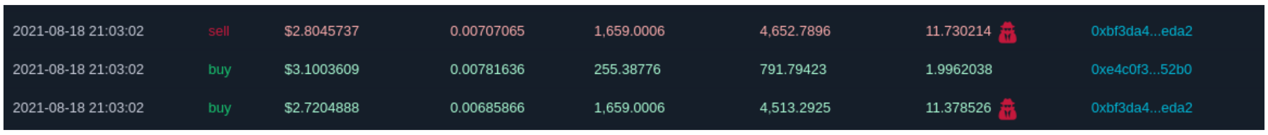
Introductory Video

https://www.youtube.com/watch?v=UZ-NNd6yjFM

Example sandwich attack



2021-08-19 12:53:18	sell	$2.6153992	0.00651959	379.05537	991.38112	2.4712861	0xbf3da4...eda2
2021-08-19 12:53:15	buy	$2.6812377	0.00668371	70.340944	188.60079	0.47013857	0xdc6b3e...9a83
2021-08-19 12:53:15	buy	$2.6044158	0.00649221	379.05537	987.2178	2.4609079	0xbf3da4...eda2



2021-08-18 21:03:02	sell	$2.8045737	0.00707065	1,659.0006	4,652.7896	11.730214	0xbf3da4...eda2
2021-08-18 21:03:02	buy	$3.1003609	0.00781636	255.38776	791.79423	1.9962038	0xe4c0f3...52b0
2021-08-18 21:03:02	buy	$2.7204888	0.00685866	1,659.0006	4,513.2925	11.378526	0xbf3da4...eda2

Storage Layout

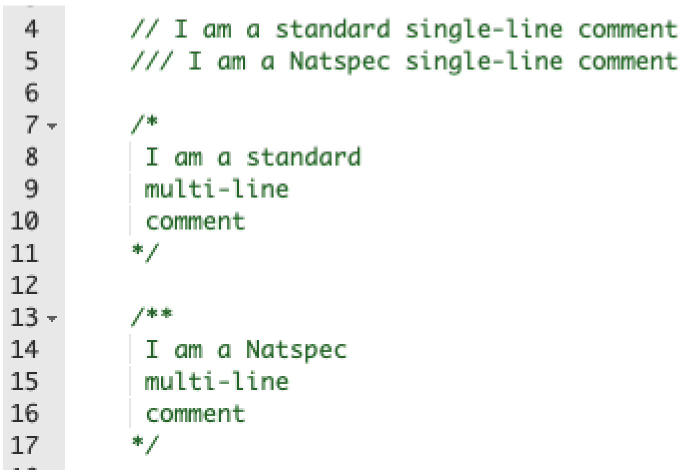
Examples of how different datatypes are stored, including dynamically sized items are illustrated on the sol2uml site

Documentation

Natspec

See Docs

How to comment code in Solidity?
Comments in Solidity can be written in two different ways.



```
// I am a standard single-line comment
/// I am a Natspec single-line comment

/*
 I am a standard
 multi-line
 comment
*/

/**
 I am a Natspec
 multi-line
 comment
*/
```

What are Natspec Comments?
Special form of comments in Solidity contracts
⇒ Machine Readable
Used to documents variables, functions, contracts, etc...
Based on the Ethereum Natural Language Specification Format (NatSpec)

Single line Natspec comment: start with ///
Multi line Natspec comment: start with `/**`, end with `*/`

The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic
simulation
/// @dev All function calls are currently implemented without
```

```
side effects
/// @custom:experimental This is an experimental contract.
contract Tree {
    /// @notice Calculate tree age in years, rounded up, for
live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase
precision
    /// @param rings The number of rings from
dendrochronological sample
    /// @return Age in years, rounded up for partial years
    function age(uint256 rings) external virtual pure returns
(uint256) {
        return rings + 1;
    }

    /// @notice Returns the amount of leaves the tree has.
    /// @dev Returns only a fixed number.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

...

    /// Return the amount of leaves that this specific kind of
tree has
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure
returns(uint256) {
        return 3;
    }
}
```

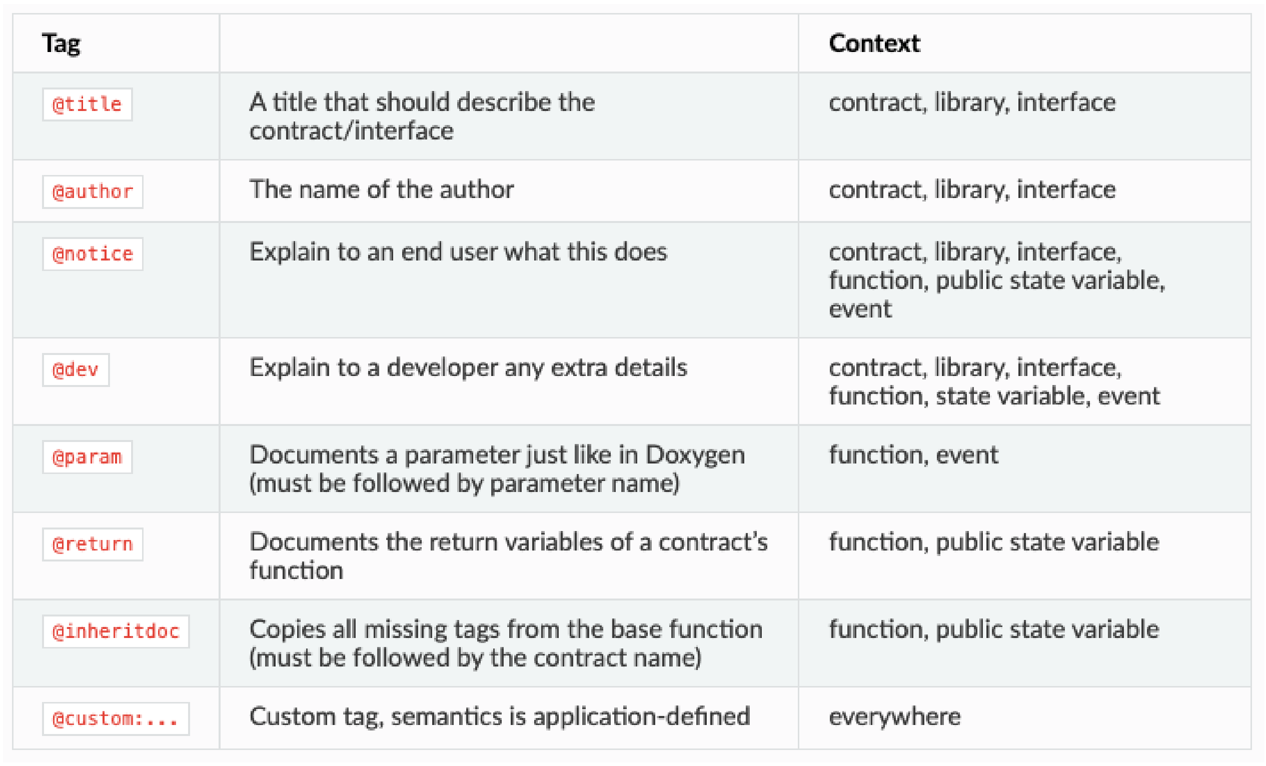
What do Natspec comments do?

Document smart-contracts for developers
Generate documentation for the smart contracts automatically with third-party tools.
Annotate conditions for formal verification.

Using the @dev tag

Notify end-users when interacting with the contract
= more expressive
Show relevant details to end users at the time they will interact with the contract (= sign a transaction.

Using the @notice tag (only in public and external functions).



Tag		Context
`@title`	A title that should describe the contract/interface	contract, library, interface
`@author`	The name of the author	contract, library, interface
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event
`@return`	Documents the return variables of a contract's function	function, public state variable
`@inheritdoc`	Copies all missing tags from the base function (must be followed by the contract name)	function, public state variable
`@custom:...`	Custom tag, semantics is application-defined	everywhere

What should we document in Natspec?

- Contracts
  Including interfaces and libraries

- Functions,
  Including constructors and public state variables (with automatic getter).

- Events

Tags

`@title`

> A title that should describe the contract/interface
>
> context : contract, library, interface
>
> `@author`
>
> The name of the author
>
> context : contract, library, interface
>
> `@notice`
>
> Explain to an end user what this does
>
> context : contract, library, interface, function, public state variable, event
>
> `@dev`
>
> Explain to a developer any extra details
>
> context : contract, library, interface, function, state variable, event
>
> `@param`
>
> Documents a parameter just like in Doxygen (must be followed by parameter name)
>
> context : function, event
>
> `@return`
>
> Documents the return variables of a contract's function
>
> context : function, public state variable
>
> `@inheritdoc`
>
> Copies all missing tags from the base function (must be followed by the contract name)
>
> context : function, public state variable
>
> `@custom:...`
>
> Custom tag, semantics is application-defined

context : anywhere

Documenting Contracts with Natspec

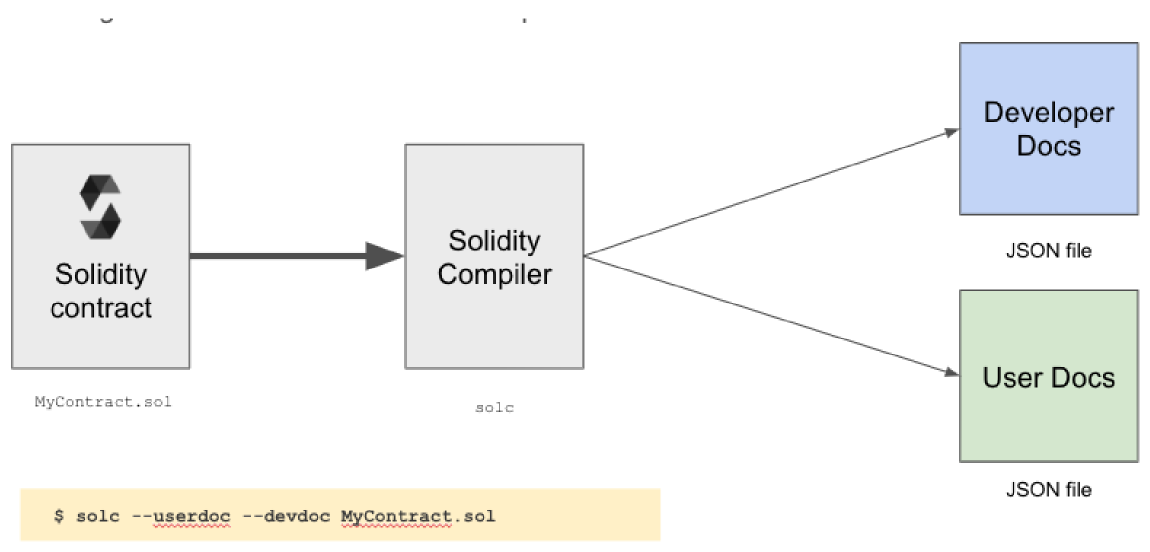
Example: Argent
Example: Buffer Library from Oraclize

@param must be followed by the variable name passed as argument.
@return good practice is to put return type or the name of the variable returned.
@notice only relevant in public + external functions (only for userdocs).

Example: Uniswap
Example: Uniswap

Example where:
A public state variable (= automatic getter function)
Inherit the docs of the parent / base contract (= here the interface)

Documentation Generator



The Solidity compiler generates a JSON file
= artifacts with contract metadata, that contain:

- Compiler version
- ABI
- Contract bytecode

But also the documentation generated by Natspec comments
(in the “output” section at the end of the file)

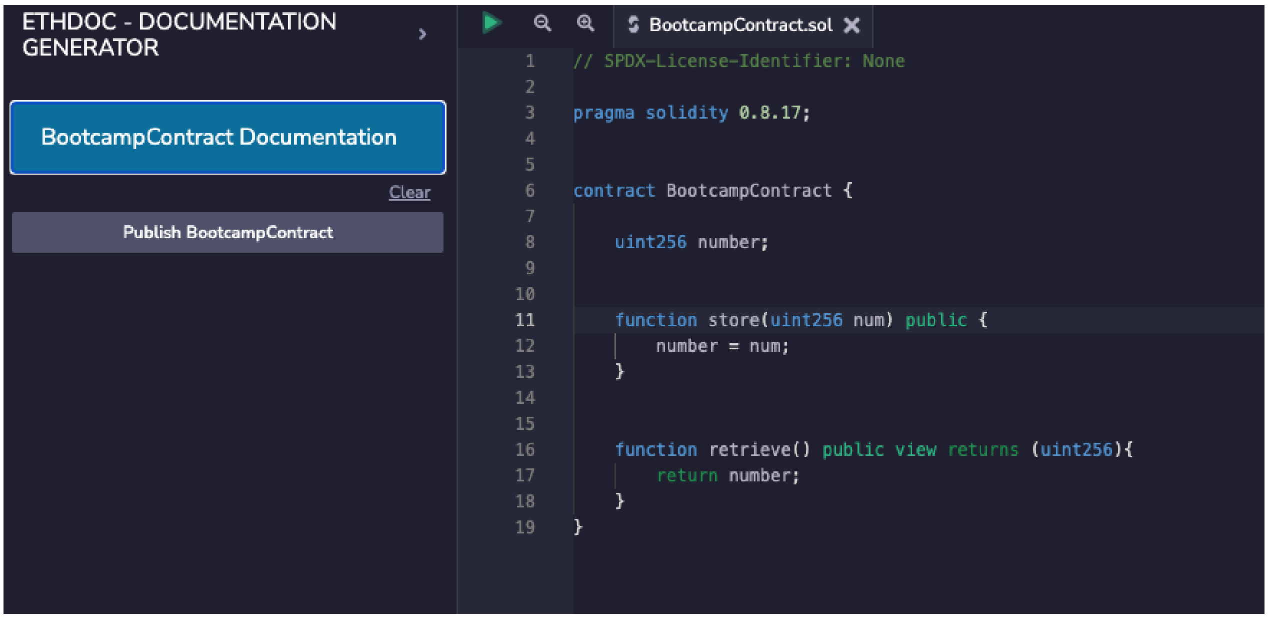
NB: when doing truffle compile, look at the JSON file under the /build folder. If your contract had Natspec comments in it, you will see the “devdoc” and “userdoc” sections.

Developer Docs output

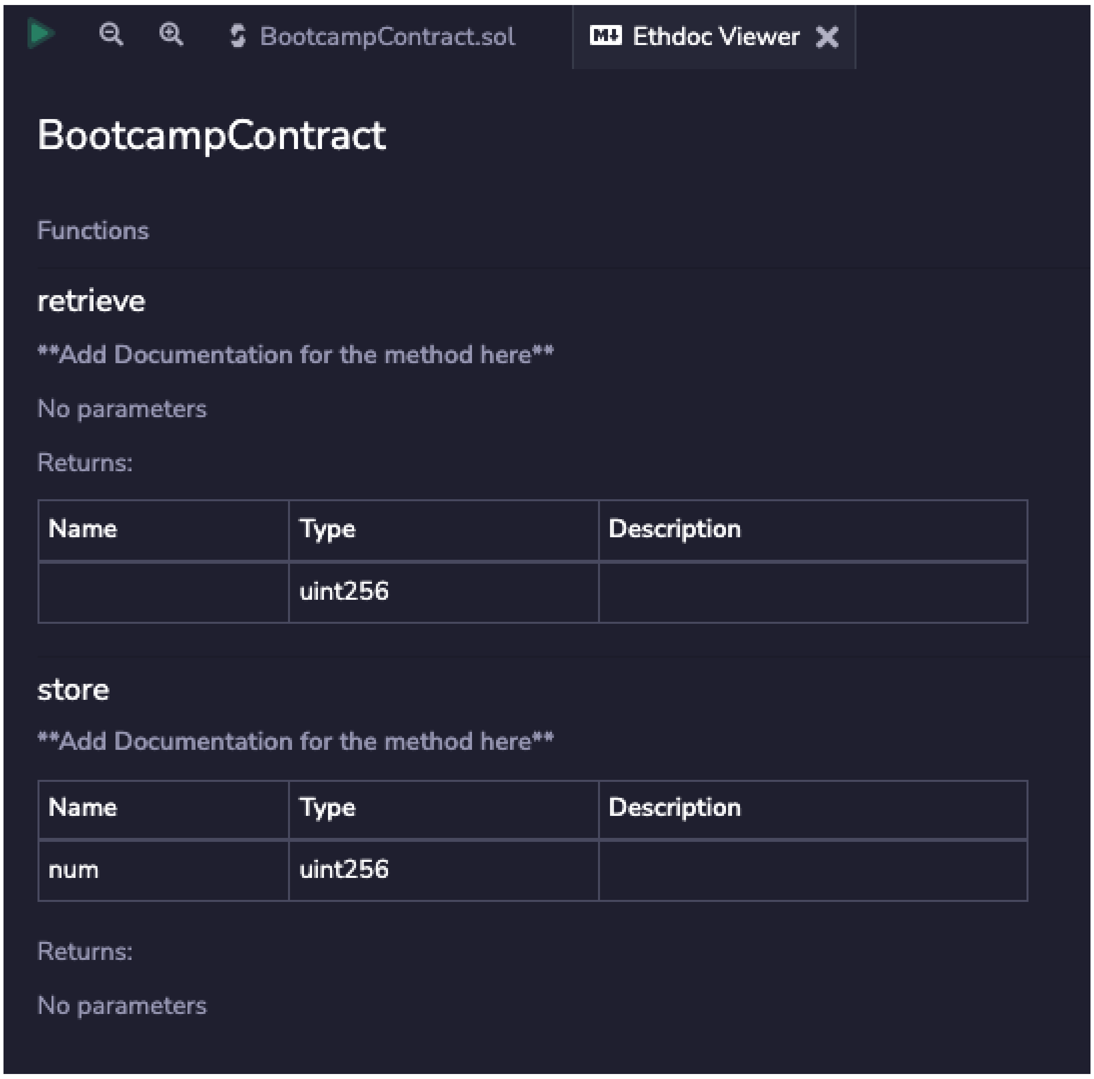
Function signature
= function name + parameter types in parentheses (without spaces)
= hashing this with keccak256 gives you the bytes4 function selector

Only public and external functions are shown
(private and internal functions are not parsed)

User Docs output - using Remix EthDoc plugin
NB: at the current time, the EthDoc - Documentation Generator seems to not be working properly. Only the EthDoc viewer is.


ETHDOC - DOCUMENTATION GENERATOR
BootcampContract Documentation
Clear
Publish BootcampContract
BootcampContract.sol

```
// SPDX-License-Identifier: None

pragma solidity 0.8.17;


contract BootcampContract {

    uint256 number;


    function store(uint256 num) public {
        number = num;
    }


    function retrieve() public view returns (uint256){
        return number;
    }
}
```



BootcampContract

Functions

retrieve

Add Documentation for the method here

No parameters

Returns:

Name	Type	Description
	uint256	

store

Add Documentation for the method here

Name	Type	Description
num	uint256	

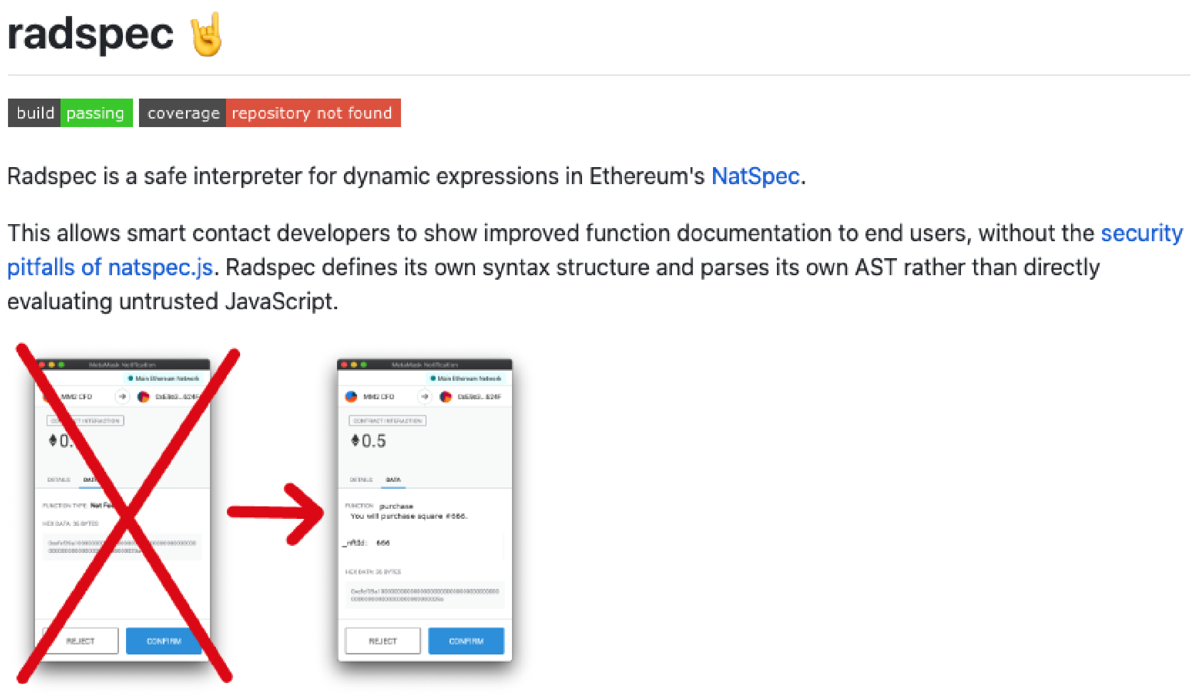
Returns:

No parameters

Solidity compiler ⇒ examine NatSpec comments ⇒ generate JSON. End user software (eg: Metamask) can consume this document.

Parsing @notice tag using Radspec interpreter

From Aragon



radspec 🤘

build passing coverage repository not found

Radspec is a safe interpreter for dynamic expressions in Ethereum's NatSpec.

This allows smart contact developers to show improved function documentation to end users, without the security pitfalls of natspec.js. Radspec defines its own syntax structure and parses its own AST rather than directly evaluating untrusted JavaScript.

Features

- **Expressive**: Show relevant details to smart contract end-users at the time they make transactions.
- **External calls**: Radspec can query other contracts.
- **Safe**: Radspec requires no DOM access or untrusted JavaScript evaluation.
- **Compatible**: Most existing NatSpec dynamic expressions are compatible with Radspec.

Also see Open Zeppelin SolidityDocgen

References
https://docs.soliditylang.org/en/latest/natspec-format.html
https://jeancvllr.medium.com/solidity-tutorial-all-about-comments-bc31c729975a
https://www.bitdegree.org/learn/solidity-syntax#natspec
https://github.com/aragon/radspec