Universität Zürich[UZH]

CAS Blockchain

Overview on Smart Contracts Vulnerabilities and How to Prevent Them

Author Carmel L. Lisser

Affiliation

Date December 01 2019

ABSTRACT

Distributed ledger technologies (DLP) are gaining a lot of traction in many industries today and have the potential to become essential backbones to many platforms in the near future. Although once hailed as safe, blockchain related security incidences where millions in funds were reported stolen are making headlines. The successful adoption of many technologies depends highly on the appropriate understanding and acceptance of security vulnerabilities and how to prevent them. This written assignment dives into the most commonly known vulnerabilities in Ethereum smart contract and is looking for answers on how to prevent them

1 INTRODUCTION

Permission less Blockchain platforms enable the trade of crypto currencies between mutually untrusted parties and without the reliance on trusted third parties. Nakamato's Bitcoin is a peer-to peer network that allows for peers to agree on a trading transaction. Buterin however took this one step further and designed a Permission less Blockchain (Ethereum) that allows for the execution of self-enforcing pieces of software (executable code) that are stored in the Ethereum blockchain. These are called smart contracts and are written in Touring-complete languages. Smart Contracts digitally facilitate, verify or enforce the agreement between a buyer and a seller. They can be invoked from entities within (other smart contracts) or outside (external data source) the Blockchain. So called Oracles inject data that is relevant to the smart contract from the on-chain world to the smart contract information store.

It is noteworthy that the term smart contract is indeed a bit unfortunate, since the smart contracts are neither particularly smart nor should they be confused with a legal contract. A smart contract is only as smart as the people coding them and naturally range from simple to complex. While this has added new opportunities by adding semantically richer applications to Ethereum than Bitcoin, it has also enlarged the threat surface.

Implemented correctly they hold the potential to bring widespread gains across multiple sectors (such as the financial, healthcare, energy and governments).

With the blockchain adoption reaching a new turning point from "Blockchain tourism" and exploration towards building practical business applications, particularly in the financial sector, strong demands for security guaranties have grown in this wake. There is an increased academic and industry interest in the topic of smart contract vulnerabilities and how to best prevent them. This written assignment dives into some of the most commonly known vulnerabilities and outlines methods to prevent them before they are deployed

2 BACKGROUND

Bugs are bad. Unfortunately, every program has one or many of them. In unchangeable, unstoppable programs they can be catastrophic and as Smart contracts are generally designed to move or hold funds in the form of Ether, they present a particularly juicy target. In the short history of Blockchain and even shorter history of Ethereum half of the incidents resulting in financial loss have been Smart contract related where the rest has been related to scams.

Millions of dollars have been stolen or frozen in attacks between 2016 and 2018 due to flaws in design, architecture and code. The most notable hack being The DAO incident where a loophole was exploited to retrieve Ether first and update the balance later.

The Smart contract Programs contain a defined, unchangeable set of instructions that will execute once predetermined conditions are fulfilled with the result of the transaction(s) then written onto the distributed Ethereum infrastructure. Once Smart Contracts are deployed to the Blockchain, they can remain there forever which increases the likelihood of Vulnerability discovery owing to their publicly visible nature.

The Smart contract code is predominantly written in a touring-complete language called Solidity and later compiled to a low-level bytecode called EVM (Ethereum Virtual Machine). In order to execute the smart contract an actor must send a transaction to the smart contract and pay a fee, a cost known as gas. The fee is a measure of the computational cost and complexity of the smart contract and credited to the miner of the respective block. If there is unused gas it will be refunded back to the sender. This is to avoid never-ending programs.

3 VULNERABILITIES

‘It is a truth universally acknowledged, that a smart contract (about to be) in possession of a large amount of money, must be in want of a security audit. ‘

- **Not** Jane Austen, quote found in a short history of smart contract hacks.

By the most common definition Vulnerabilities are weaknesses in an information system, system security procedures, internal controls or implementation that could be exploited or triggered by a threat source. Weaknesses could be accidentally triggered or intentionally exploited and result in financial and/or reputational loss.

Traditionally, information system vulnerabilities that were found after deployment were dealt with by publishing or deploying repaired systems versions (patches) or improved controls at any later point in time.

This is possible because the operator of the vulnerable component has some degree of administrative control over some or all the layers and, more importantly, it is possible to change a system once it is deployed.

The Ethereum Blockchain presents many potentially exploitable components across all layers as well, but in this assignment, I focus on a subset of known smart contract vulnerabilities where it is entirely within the remit of the developer to ensure vulnerable code is identified **before** the smart contract is deployed. Smart contract vulnerabilities found after deployment to the Blockchain are much harder to deal with. Please see figure 1 below to visualize the layers and its components.

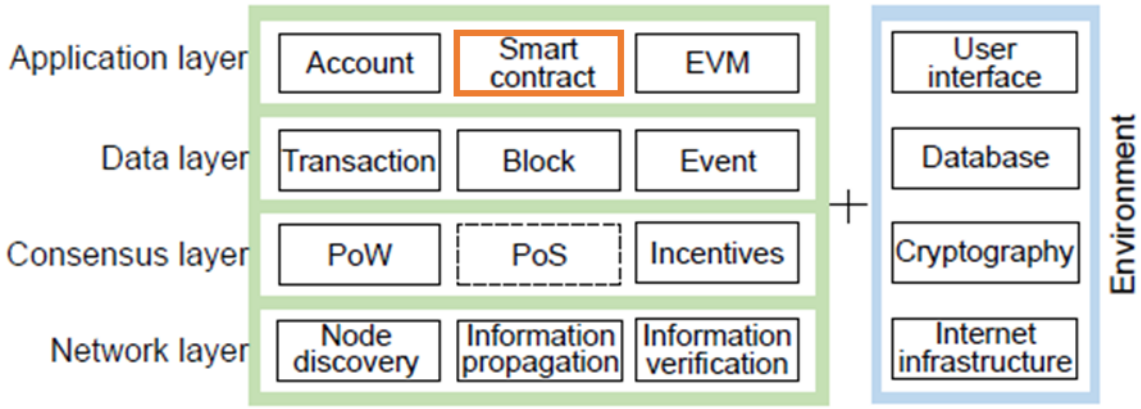

Figure 1 - Ethereum Blockchain Architecture and Components

I have utilized a survey on Ethereum Systems Security (Chen, 2019) and focused on the vulnerabilities associated to the Application Layer. Then I cross referenced the list with the top 10 vulnerabilities listed in the Decentralized Application Security Project (DASP), an open and collaborative project, to capture only the vulnerabilities relevant to the application layer and ordered them according to the DASP numbering in the subsection in the document.

This narrowed the scope of vulnerabilities from 26 to 8. See Figure 2 (The purple dots are the vulnerabilities found in the DASP top 10)

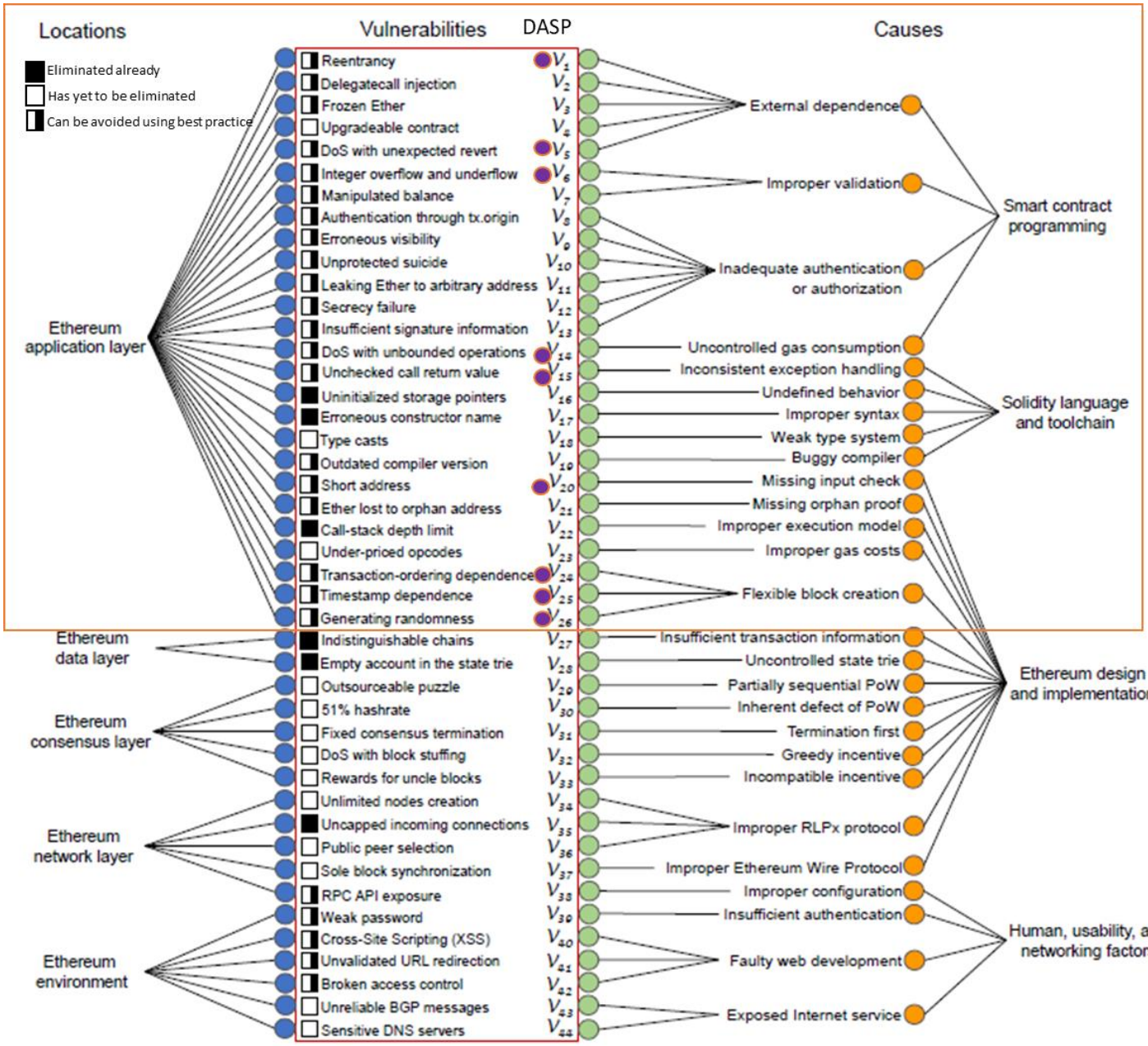

Figure 2 - Ethereum Vulnerabilities and their causes, sourced from A Survey of Ethereum Systems Security: Attacks and Defences

3.1 Re-Entrancy

Real world impact: The DAO attack

Loss: about 60 USD million in Ether theft (source Coinbase)

Functions exploited: public `withdraw()` function, `call.value(amount)()`, `fallback()`

The vulnerability is exploited when an external caller tries to send Ether before the internal state is updated. Meaning the external callee contract calls the execution again and again before the execution of the caller contract is completed. The attacker can bypass the validity check until the contract is drained of Ether (or the transaction runs out of gas)

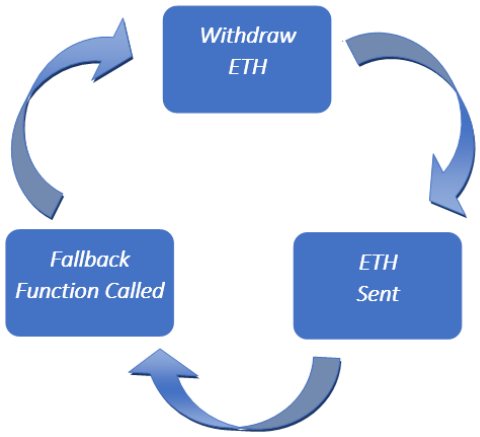

Figure 3 - The recursive loop of a reentrancy attack, Sourced from the internet

Sadly, there is no way to stop this attack once it has started and the withdrawal function repeats itself until the victims gas or the balance has been depleted.

When this vulnerability was used in the DAO heist, the Ethereum community decided to roll back to a previous state using a hard fork.

3.2 Arithmetic Issues (or Integer Overflow and Underflow)

Lorem Real world impact: The DAO, attack against BEC tokens

Functions exploited: `withdraw()` function

This vulnerability is not specific to Ethereum smart contracts and is a common bug in many programming languages. In smart contract, the impact could be severe. If an arithmetic operation falls outside of the range of a Solidity data type it can cause an unauthorized manipulation to the attacker's balance. This can be exploited by an attacker who can find a way to increment the number of iterations of the loop.

Neither the Solidity compiler nor the EVM enforces integer overflow/underflow detection. Therefore, this must be done in the Solidity source code.

```
function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
}
```

Figure 4 - Sample from DASP, Simple example shows how infinite numbers of tokens can be withdrawn

3.3 Unchecked Return Values for Low Level Calls

Real world impact: King of the Ether, Etherpot

Functions exploited: negligence in checking the return value of `send(),`

There are two variants of this vulnerability. Gasless send and unchecked send. These low-level operations in Solidity do not throw an exception on failure but report the status as Boolean value. If an attacker were to exploit this, a contract could continue its execution despite failed payment. This can lead to discrepancies which can further lead to unintended transaction. The caller contract must carefully address and check the discrepancy in order to prevent this.

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

Figure 5 - Sample from DASP, If the return value is not checked the variable etherLeft will end up tracking an incorrect value

3.4 Denial of Service (DoS)

Real world impact: GovernMental, Parity Multi-sig wallet, King of the Ether throne

Loss: about 300M USD

Functions exploited: `bid(uint object)`

This category of attack is a rather wide, but Denial of Service always means (smart contract or not) that the application will be left dysfunctional for a certain time or is rendered useless all together. In smart contracts only 2 are known.

- **DoS with unexpected revert** occur when a caller (or a bidder) contract encounters a failure in an external call. For example if a bidder ensures that refunds to his/her address fail the bidder can remain the leader forever and prevent anyone else from calling the `bid()`function.
  Noteworthy here are Ponzi scheme type smart contracts. They started to show up even in the early days of Ethereum smart contracts. They raised a lot of ethical questions as they aim to solicit new investors with the promise of high returns with little or no risk. Like the pyramid scheme it relies on a constant influx of new investors to carry the top of the pyramid. The promise to get rich quick led to a lot of Ethereum stolen. The KingOfTheEther Throne smart contract is often used as posterchild example on how a `send()` to a contract can revert due to a fallback function thus spending too much gas. The contract became effectively frozen and a user called *Sir Wobblle* is now the eternal king.

Name	Claim Price Paid
Sir Wobblle (0x77a7276609757358a2efcaf65e74d34d4a18acd0)	1.17 ETHER

- **DoS with unbounded operations**. Each block has a limit on how much gas can be spent, the block gas limit. If the gas spent exceeds the limit, the transactions will fail. Even if unintentional this can lead to a DoS attack. Attackers could setup multiple addresses entitled to very small refunds and when the contract exceeds the block gas limit the refunds will be blocked from happening. Therefore, unbounded operations should be avoided (loops over arrays) specially in data structures that can be operated by Externally Owned Addresses (EOA).

3.5 Bad Randomness

Real world impact: SmartBillions Lottery, TheRun

Functions exploited: `private seed.` Even though the seed is private it is presumed to have been set via a transaction of some point in time and is visible on the blockchain.

Many gambling and lottery contracts select winners randomly, which makes sense. The common practice is to use a seed to create a pseudorandom number. Unfortunately, the seeds used are fully controlled by the miners (i.e.

timestamp, block difficulty or blockhash). As the sources of the randomness are somewhat predictable, a malicious miner could manipulate these seed variables to make itself the winner

```
uint256 private seed;


function play() public payable {
        require(msg.value >= 1 ether);
        iteration++;
        uint randomNumber = uint(keccak256(seed + iteration));
        if (randomNumber % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

Figure 6 - Sample from DASP, private seed is used with an iteration number and the keccak256 hash function to determine if the caller wins

3.6 Front Running

Real world impact: Bancor, ERC-20, TheRun

As mentioned earlier, miners always get rewarded in the form of gas fees for running code on behalf of externally owned addresses (EOA). The motivation to reap the rewards lies within the fee itself and miners naturally group and order transactions based on the reward offered by the transaction. As the transactions are publicly visible a malicious EOA can offer a higher gas price in order to have its transaction mined into the blockchain first. In a contents where participants can submit a solution to a puzzle in exchange for reward a malicious EOA could take the solution and submit it as its own with a higher gas price value to ensure the stolen results will get assembled to the blockchain first.

3.7 Time Manipulation

Real world impact: GovernMental

Function: `block.timestamp or Now()`

In centralized networks time is synchronized using on central authoritative source. This does not happen in a decentralized network such as the Ethereum blockchain. Therefore, we turn to the next best thing, The block miners. The miner is bound to only increase the timestamp to be higher than the previous block. As we trust in blocks, we trust in a system that relates to time only within rough intervals (i.e. the cadence at which new blocks are added to the chain) time measured this way can be a fickle thing.

If a smart contract uses a timestamp-based condition to determine whether to transfer money, a malicious miner could manipulate the timestamp to satisfy the condition of the smart contract. As with the generation of random numbers, smart contracts should avoid using values that can be manipulated by miners

```
function play() public {
        require(now > 1521763200 && neverPlayed == true);
        neverPlayed = false;
        msg.sender.transfer(1500 ether);
}
```

Figure 7 - Sample from DASP, the function accepts call from a specific date. The malicious miner could set a timestamp set in the future

3.8 Short Address

Real world impact: Not yet found in the wild

This problem is due to EVM accepting incorrectly padded arguments. Although not yet exploited in the wild, attackers could use this vulnerability to transfer more tokens than specified in the smart contract by using an incorrect short address (provided the exchange is holding a sufficient number of tokens).

For example: should a callee send an address that was missing 1 byte (2 hex digits) and 100 tokens to withdraw it would get encoded with a padding of 00 at the end of the encoding to make up for the short address. When this arrives at the smart contract the value of tokens is now changed to 25600 (from 100).

This can be prevented by validating the length of the transaction input.

4 PRACTICAL PREVENTATIVE MEASURES TO AVOID EXPOSURE

Smart contracts are designed to hold and move funds denominated in Ether. This makes them tempting attack targets where, if successful, the attacker can steal funds from the contract.

NIST the National Institute of Standards and Technology warns that Blockchain networks and their applications are not immune to malicious actors who can conduct network scanning and reconnaissance to discover and exploit vulnerabilities or zero-day attacks. They also stress that in the rush to blockchain-based services, newly coded applications (smart contracts) may contain new and known vulnerabilities. While they are stating these important things, they have not yet produced a formal standard for security in smart contracts. Searches on other known producers of security frameworks have yielded few comprehensive results.

As stated in section 3, vulnerabilities detected before they are deployed are much easier to deal with and, as seen in Figure 2 most if not all the know vulnerabilities could be avoided by using a variety of practices applied at different stages of the Software Development Lifecyle (SDLC) outlined in the following chapter.

The below figure has laid out some principles to apply in the development stage, specifically for the known vulnerabilities listed above. However, it would be pertinent to don the cyber security hat for all phases of the software development lifecycle and maintain vigilant even after the deployment of a smart contract.

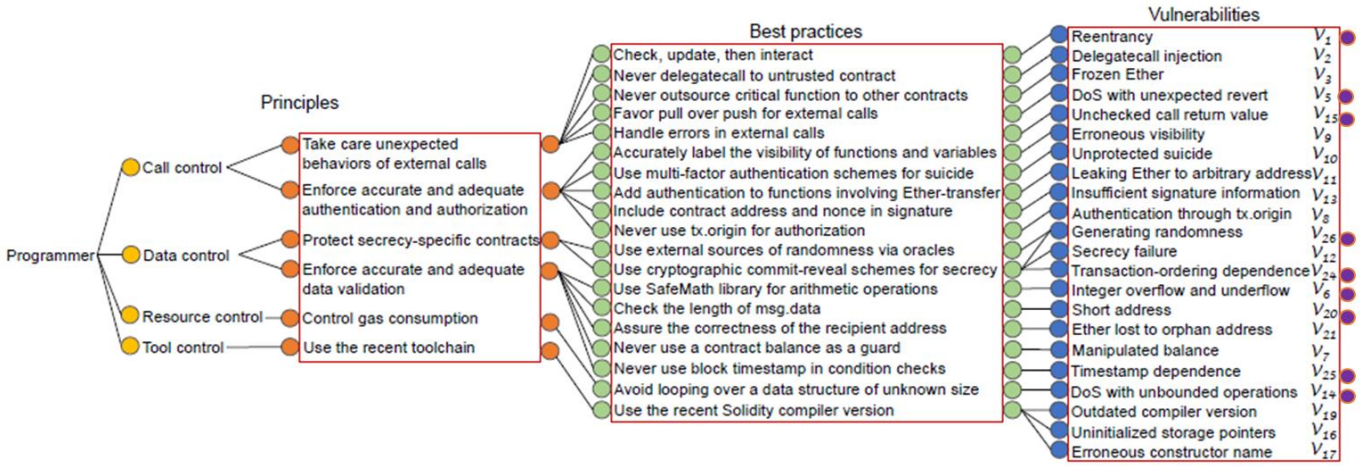

Figure 8 - Guidelines in Assessing a Smart Contract. Sourced from A Survey on Ethereum Systems Security

4.1 Design

As a first step the requirement and intended behaviour of the smart contract should be clearly articulated before programming even begins. This is to ensure that developers can adhere as closely as possible to the principle of minimum function. Requirements can later be used in the audit process where the requirements and documentation are reviewed in conjunction with the code. The following should be considered and documented when gathering the requirements for a smart contract.

- Understand the use case of a smart contract.
- Understand the architecture. Build a diagram or flow chart to understand the interaction between the functions and other smart contracts.

4.2 Architecture

Once the requirements are understood the development team should endeavour to create a basic architecture that clearly depicts the business logic of the smart contract to be. This will help developers to stick to the design path.

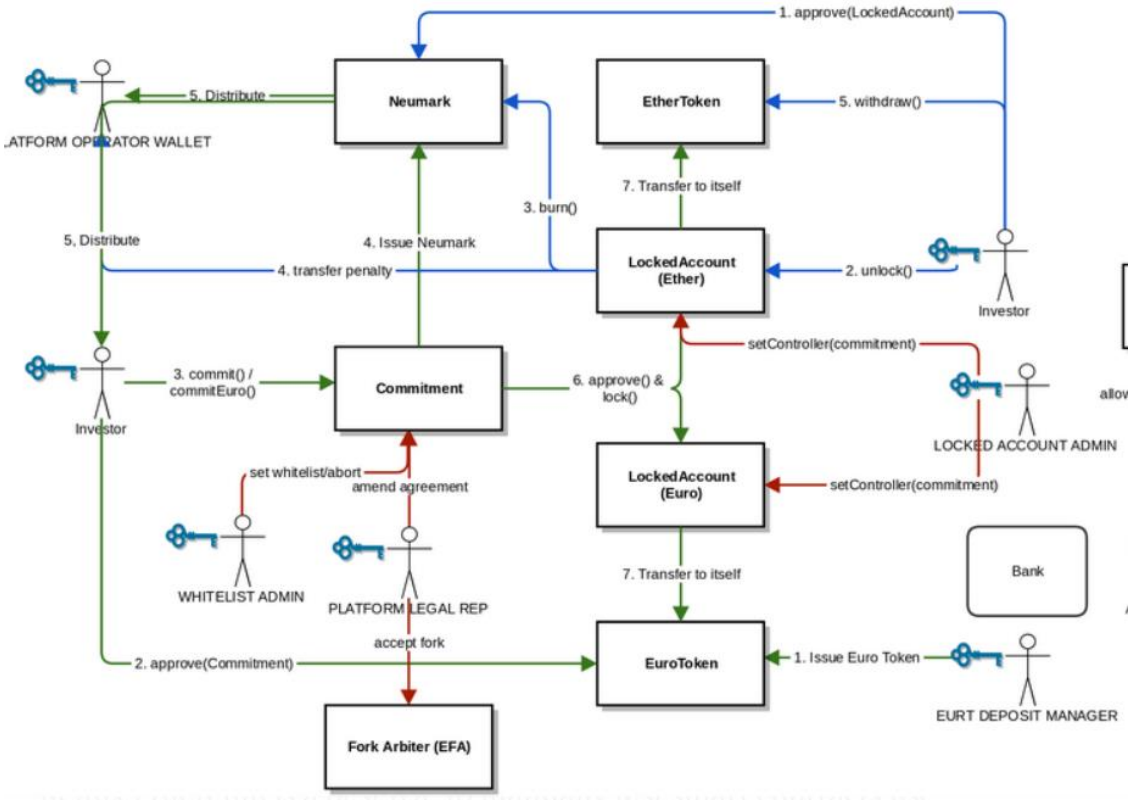

Figure 9 - Sample architecture sourced from the internet

4.3 Develop and Test

A popular development framework for smart contracts is *Truffle*. It allows for unit and integration tests in JavaScript.

- As with any traditional application it is particularly useful in smart contracts to adopt the principle of minimum functionality
- Consider Peer Reviews or a Four-Eye principle to ensure no errors have snuck into the application
- Perform manual testing on private blockchains (such as test-net)
- Record all the transaction while testing on the test-net. Compare the results against the use case and requirements
- Perform Unit testing (can be performed in Truffle)

4.4 Security Code Reviews

The process of auditing code for traditional applications to ensure that proper security controls are present and is quite common and there are many open source or 3rd party products around that can perform these reviews.

There are a quite a few options to make smart contracts more secure as well (See Figure 10). Most of them analyse the source code or its compiled EVM bytecode and look for security issues that are already known (such as re-entrancy). For example:

- **Oyente –** One of the first in action. Executes EVM bytecode and checks for executions traces where
  - transaction order can influence Ether flow
  - the result of a computation depends on the timestamp of the block
  - exceptions raised by calls are not properly caught
- **Zeus -** Takes Solidity code and a so-called policy as inputs and checks whether the code meets the safety property expressed in the policy. The policy must be specified by the user. It can detect six types of common vulnerabilities.
- **MadMax –** disassembles EVM bytecode into an intermediate representation. It can detect gas-related vulnerabilities (DoS vulnerability).
- **Securify-** I found to this to be very helpful as it is immediately available. It defines a set of compliance and violation patterns to identify contract compliance and security violations

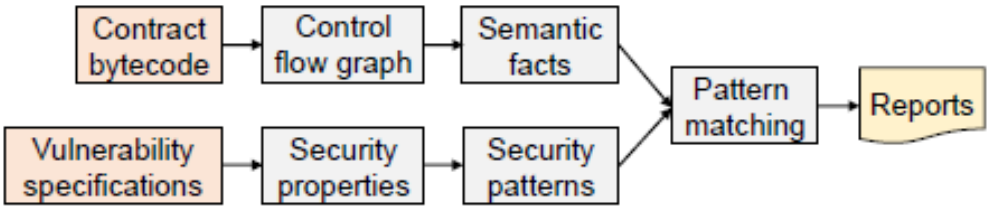

Figure 10 - Securify workflow sourced from A Survey on Ethereum Systems Security

OVERVIEW OF ALL TOOLS INDICATING PURPOSE, CODE LEVEL, TYPE, PRE-PROCESSING, AND METHODS OF ANALYSIS

	Purpose				Level		Type		Code transformation						Analysis method					
Tool	Security issues	Exploits	Formal guarantees	Bulk analysis	Bytecode	Solidity code	Static analysis	Dynamic analysis	Contextualization	Disassembly	Control flow graph	Call graph	AST analysis	Decompilation	Code instrumentation	Symbolic execution	Constraint solving	Abstract interpretation	Horn logic	Model checking
contractLarva	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
E-EVM	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthIR	✗	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗
EtherTrust	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗
FSolidM	✗	✗	✗	✗	form.spec		✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
KEVM	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
MAIAN	✓	✓+	✗	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Manticore	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Mythril	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Osiris	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Oyente	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Porosity	✓	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Remix-IDE	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Securify*	✓	✗	✓	✓	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗
SmartCheck*	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Solgraph	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
SolMet	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Vandal	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗
Ether*	✓	✗	✗	✓	✓+	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Gasper	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
ReGuard	✓	✗	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SASC	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗
sCompile	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
teEther	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Zeus	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗

Figure 11 - Sourced from A survey of Tools for Analysing Ethereum Smart Contracts

4.5 Smart Contract Auditing

Auditing of smart contracts is common industrial practice given that they can hold high monetary value. This step should be performed before the smart contract is deployed and ideally while it's still in the testing phase.

Audits come at a high cost at around 30,000 USD for a standard smart contract (if conducted by an out of house smart contract audit professional) a company may choose to perform its smart contract audits after performing as many internal validations and checks as possible.

The scope of an audit commonly come in 4 stages:

Stage 1.

- Detailed Manual Review to ensure that the requirements specifications are implemented

Stage 2.

- Verify that the contract does not have any behaviour that is not specified in the requirements
- Verify that the contract does not violate original intended behaviour
- Perform security scans to identify known security vulnerabilities
- Security attack playbook analysis based on known incidents
- Manual review of the code to ensure best practices have been considered

Stage 3.

- Gas limit functions will be tested to verify no unnecessary gas is consumed

Stage 4.

- Testing with automated tools (code scanning)

4.6 Bug Bounty Program

A bug bounty for smart contracts is no different than a bug bounty program for any standard web application. The bug bounty program entices people with solidity coding experience and the sneaky understanding of a hacker to find and exploit vulnerabilities. At its core it looks to be no different than the audit program but while the auditors have a point in time view of the current vulnerability landscape, the bounty hunt can remain ongoing long after the smart contract has been deployed and can remain posted throughout the lifecycle of a smart contract.

Entities choosing to run such a program can reward the contributors in fiat or in cryptocurrencies (such as Bitcoin or Ether). The sum of the rewards can be structured around the degree of impact the bug or vulnerability would have had to the business.

Degree	Description	Reward
Mission Critical	Destruction or modification of the blockchain database on all nodes of the network	1000$ - 3000$
Security	Getting access to funds on users' wallets	1000$ - 3000$
Critical	A situation that leads to stopping the generation of blocks and the inability to carry out operations on the platform	500$ - 2000$
Error	Complete failure of one or more platform components on two or more network nodes, which reduces network performance, in particular, the rate of generation of blocks. The platform functioning is not interrupted.	100$ - 500$
Warning	An error that does not stop the functioning of any component	10$ - 150$

* Use "Pull Request" on Credits Github in order to submit a fix on any found bug. In case if it is considered as a viable, you will get a reward 3 times more.

Figure 12 - reward table example by hackernoon

5 DISCUSSION

As with all cyber security issues there is never a 100% guarantee to prevent an attack. In traditional architecture and risk reviews there are usually numerous supporting documents and checklists to can help the security analyst identify and risk rate most known vulnerabilities and weaknesses.

- The community of smart contract and blockchain Engineers is still small and there is a lack of understanding from a broader perspective on how the various components interact.
- Common standards that are referenced and drawn upon by the risk and cybersecurity community are still in draft form or missing. Patterns and attack vectors have been researched and have been documented in some academic publications but there is a lack of evidence of substantial studies from institutions such as MITRE, ISO and NIST.
- With increasing reliance blockchain system there is a gap in knowledge of common attack vectors and control/mitigation steps. Attempts to collaboratively draw these bits of information together on common platforms appear painfully immature and are centered around preventative measures. There or no time proven guidelines on how recovery maneuvers could be performed.

REFERENCES

Hushan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu (2019)
“A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses”.

Daniel Perez, Benjamin Livshits (2019)
“Smart Contract Vulnerabilities: Does Anyone Care?”

Monika di Angelo, Gernot Salzer (2019)
“A Survey of Tools for Analyzing Ethereum Smart Contracts”

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, Martin Vechev (2019)
“Securify: Practical Security Analysis of Smart Contracts”

Bruno Rodrigues, Eder John Scheid, Christian Killer, Muriel Franco, Burkhard Stiller: Blockchain Signaling System (BloSS): Cooperative Signaling of Distributed Denial-of-Service Attacks; Springer, Journal of Network and Systems Management, Vol. 28, No. 3, August 2020, pp 1-27. URL: https://doi.org/10.1007/s10922-020-09559-4.

Various Internet Sources such as:

DASP.co top 10

Consesys.github.io – Ethereum Smart Contract Best Practices

NIST.gov