



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Security Analysis of Applications Using Smart Contracts
Student: Bc. Ondřej Lauer
Supervisor: Mgr. Jakub Růžička
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Information Security
Validity: Until the end of summer semester 2019/20

Instructions

Research the area of applications using smart contracts, including the introduction to the blockchain technology, the Ethereum and Hyperledger platforms and the topic of decentralized applications (dApps).

Based on existing standards and other available resources, develop a general framework for security analysis of components of these applications (such as web applications, mobile applications, infrastructure etc.) according to attack vectors and their impact, with an emphasis on the specifics of analyzing components using smart contracts.

Describe the architecture of the VETRI application, perform a security analysis of the application, discuss potential findings, and evaluate the limitations of your chosen approach.

The deliverables of the thesis are a proposal of a framework for security analysis of applications using smart contracts, listing of vulnerabilities of the VETRI application tested on the basis of the developed framework and suggested remedies.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 10, 2018



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Security Analysis of Applications Using Smart Contracts

Bc. Ondřej Lauer

Department of Information Security
Supervisor: Mgr. Jakub Růžička

April 13, 2019

Acknowledgements

I would like to thank my supervisor, Mgr. Jakub Růžička, for his feedback and guidance that helped me during the creation of this thesis. I would also like to thank Ing. Petr Fojtů for his help with proofreading.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on April 13, 2019

.....

Czech Technical University in Prague
Faculty of Information Technology

© 2019 Ondřej Lauer. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Lauer, Ondřej. *Security Analysis of Applications Using Smart Contracts*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

V této diplomové práci jsem řešil problém bezpečnosti aplikací založených na tzv. smart contractech a vytvořil framework pro testování těchto aplikací. Soustředil jsem se na hlavní zranitelnosti pěti hlavních komponent typických smart contract aplikací - blockchainu, smart kontraktů, webových aplikací, mobilních aplikací a související infrastruktury. Praktická část práce se zabývá aplikováním zmiňovaného frameworku na aplikaci VETRI, její bezpečnostní analýzou a návrhem opravných opatření pro nalezené zranitelnosti.

Klíčová slova blockchain, Ethereum, Hyperledger, chytré kontrakty, kyberbezpečnost, bezpečnostní analýza

Abstract

This Master's thesis is concerned with security of applications utilizing smart contracts and development of framework for security assessment of such applications. I have mainly focused on the most prominent vulnerabilities in five main components of a typical smart contract application - blockchain, smart contracts, web applications, mobile applications and infrastructure. The practical part is about applying the created framework to the VETRI application, its security analysis and summary of remedies for found vulnerabilities.

Keywords blockchain, Ethereum, Hyperledger, smart contracts, cyber security, security assessment

Contents

Introduction	1
0.1 Aim of the thesis	1
0.2 Structure of the thesis	1
1 Analysis	3
1.1 Blockchain	3
1.2 Smart Contracts	8
1.3 Web application	16
1.4 Mobile application	30
1.5 Infrastructure	35
2 Smart contract application testing framework	39
2.1 Framework	39
3 Analysis of the VETRI application	47
3.1 VETRI application overview	47
3.2 VETRI blockchain architecture	49
3.3 Blockchain use-cases	51
3.4 Application analysis	53
3.5 Vulnerabilities and mitigations overview	60
3.6 Application analysis limitations and follow-up	61
4 Conclusion	63
Bibliography	65
A VETRI mobile application screen designs	69
B Full Oyente scan outputs	77
B.1 Identity.sol	77

B.2	ERC20Token.sol	77
B.3	DataExchangeRequest.sol	79
B.4	Migrations.sol	79
C	Full SonarQube scan outputs	81
D	Acronyms	87
E	Contents of enclosed CD	89

List of Figures

1.1	Binary Merkle tree with depth 3	4
1.2	OWASP Top 10 2013 and 2017	16
1.3	Web Application technology stack	27
3.1	VETRI whitepaper architecture	48
3.2	VETRI MVP/MUP system architecture	50
3.3	Full survey usecase	52
3.4	SonarLint bug detection illustrative screenshot [1]	55
3.5	Illustrative SonarQube interface screenshot [2]	57
3.6	Bridge test result screenshot	58
3.7	MobSF static analysis result overview	59
A.1	Welcome screen	70
A.2	Adding data	71
A.3	Token exchange	72
A.4	Data requests	73
A.5	Survey example	74
A.6	Balance overview	75
A.7	Token screen	76
C.1	SonarQube screenshot 1	83
C.2	SonarQube screenshot 2	84
C.3	SonarQube screenshot 3	85

List of Tables

2.1	Framework checklist table, part 1	44
2.2	Framework checklist table, part 2	45

Introduction

Blockchain is gaining momentum in application development. Almost everybody has heard of the word, however only a few people can really explain what a blockchain is and even less people do know about its security challenges.

The purpose of this masters thesis is to study and describe ways in which a smart contract application could be attacked.

I have chosen this topic because I already work in the field of Cyber Security and I'd like to have knowledge about any new security-related technology trend.

This thesis aims to create a security framework which could be used to assess security of a typical smart contract application. The thesis also contains a security analysis of VETRI application (for description please see section 3.1) and a list of mitigations.

0.1 Aim of the thesis

The aim of the research part is to collect and summarize all of the main attack types that could be launched against a smart contract application and to describe their root cause and potential impact.

The aim of the practical part is to use the designed framework to assess the security of the VETRI smart contract application and list remedies for all of the identified vulnerabilities.

0.2 Structure of the thesis

The research part is split into five main sections, one for each of the five main components – blockchain, smart contracts, web application, mobile application and infrastructure.

INTRODUCTION

The practical part is split into two chapters – first one contains the framework and the second contains the analysis of the application itself and suggested mitigations for found vulnerabilities.

Analysis

This chapter discusses the analysis of the most common vulnerabilities in applications based on smart contracts and their main components.

1.1 Blockchain

Blockchain is the new topic of the last few years, however first work on cryptographically secured chain of blocks was written in 1991 by Stuart Haber and W. Scott Stornetta [3] who wanted to create a system that would securely timestamp documents. Blockchain became known to the general public in 2008 after release of a paper from Satoshi Nakamoto [4] who implemented it as a public ledger in cryptocurrency Bitcoin.

1.1.1 Basic principle

Every blockchain network (Bitcoin, Ethereum etc.) requires a network of peer-to-peer (P2P) hosts, meaning that every host has the same importance – unlike client-server networks. Whenever a new transaction is requested it gets broadcast to the whole miner network. The miners validates the transaction and the user who requested it and when confirmed the transaction is combined with other transactions to create new block for the ledger. The newest block is added to the ledger and published to the other miners – it is permanent and unalterable.

The biggest strength of the blockchain is also its biggest weakness – anonymity. Bitcoin came into spotlight around the same time as the WannaCry ransomware, since criminals used it as anonymous bank accounts for the ransom payments. This anonymity is provided by the public key infrastructure (PKI). Public key is used when validating the transaction signed by the user's private key. Parts of the public key along with some other information are used as a blockchain wallet address. Blockchain wallet is a computer program that is used to monitor and use cryptocurrency.

1.1.2 Important concepts

Blockchain (or its implementations) are based on a variety of important ideas which are explained in this section.

1.1.2.1 Hash function

Hash function is a one way algorithm which can transform data of almost any length into a string of constant length. Hash functions have to be one-way functions – the resulting hash cannot be reversed to the original data. When the same data are hashed with the same algorithm the resulting hash is always the same – however, any change (even in one bit of the hashed data) will result in a completely different hash. It has many uses, but the most common ones are for indexing of large data, transfer verification, digital data signatures or password storage.

1.1.2.2 Merkle trees

Merkle trees (sometimes called "hash trees") are data structures used to efficiently and securely verify contents of large data structures. The leaf nodes of Merkle trees contain hash of a data block and every non-leaf node contains hash of its child nodes. We could also recursively define the Merkle tree as a binary tree of hash lists where the parent node is a hash of its children, and the leaf blocks are hashes of the original data blocks. The top of the tree is called **root hash** or **master hash**. This way it is easier to verify any transaction in the tree thanks to the logarithmic complexity – e.g. a binary Merkle tree with depth $N = 3$ (and 2^3 leaf nodes) could verify any transaction with $2 * \log_2 N$ transactions at most.

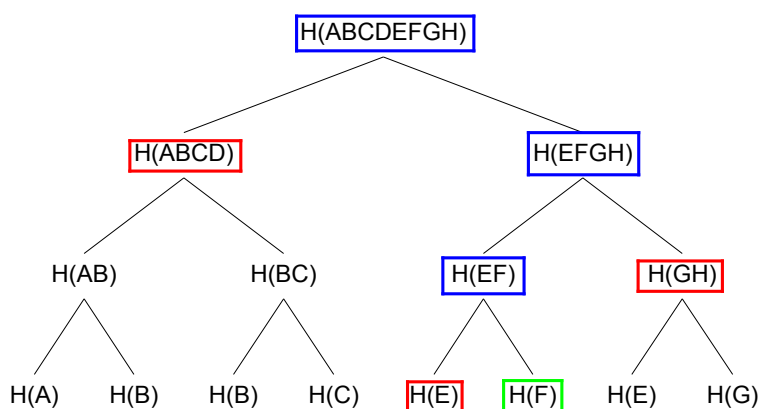


Figure 1.1: Binary Merkle tree with depth 3

Let's say we want to verify transaction F in our Merkle tree 1.1. We would need to take hashes $H(E)$, $H(GH)$, $H(ABCD)$ and compute hashes in this order – $H(F)$, $H(EF)$, $H(EFGH)$ and finally the root hash $H(ABCDEFGH)$. If our computed root hash is the same as the root hash we were provided earlier we can say the transaction is valid.

1.1.2.3 Blockchain proofs

Blockchain proofs are the main way of creating new blocks and keeping the blockchain moving forward. There are many types of blockchain proofs, however in this section only the most common proofs are discussed.

Proof of work Proof of work is a part of the mining process which is used to determine who the successful miner will be. The miners run last block's unique metadata through hashing function while changing only the "nonce" value. If the miner's output hash is lower than the current target he is awarded cryptocurrency and broadcasts the new block across the network. Every other miner can quickly verify the result by hashing it with the correct nonce value. It takes approximately 12-15 seconds to mine a new block in Ethereum. If the mining is too quick or too long the difficulty gets adjusted accordingly – all of the nodes share one formula that takes into consideration how long the last few blocks took to compute in order to adjust the difficulty. This is the way more currency gets distributed to the ecosystem over time and it also incentivises miners to keep mining.

Proof of authority Proof of authority is a possible replacement for the proof of work. Instead of arbitrary difficult mathematical problems a group of nodes (validators) is established and they are explicitly allowed to create new blocks and are responsible for this matter. However, in order to become a validator one must have their identity publicly identified with the option to cross-reference the identities through a reliable database (e.g. public notary database). The process of becoming a validator has to be very selective in order to choose only the people who are of a clear incentive – both financially and reputationally. A few platforms provide financial incentive to validators in order to keep them from being dishonest – but in the end it still depends on one's will to destroy their reputation for a financial gain.

1.1.2.4 Decentralized application

Decentralized applications (also called dApps) are applications that run on blockchain networks and thus don't use the traditional master-slave distributed application architecture. This type of architecture makes them resistant to any kind of modification on the provider's side – one could imagine for example a variation of social network site Twitter where nobody could delete any

tweets that were made – if information is once posted on blockchain it stays there forever¹.

The main difference between a standard client-server application model and a decentralized application model is that a decentralized application doesn't use a single center (e.g. a company's infrastructure, server room or a specific server) to deliver the information to clients. Instead, computation is done at each node and no node is instructing the other – as in standard P2P model. The backend of the decentralized application is limited and it could also be completely substituted by the blockchain. However when some backend is used in a decentralized application a great care has to be taken in order to properly divide functionalities between the blockchain, the backend and eventually devices of the users as well. For example a private key for a blockchain address must definitely be stored locally and as secure as possible. [5]

1.1.3 Most popular blockchain platforms

Blockchain has no particular use by itself – it is but an idea that has to be implemented in order to be useful. These are the most popular blockchain platforms according to the following sources:

- Dev.to - 10 most popular and promising blockchain platforms
- newgenapps.com - 10 Potential blockchain Platforms to Watch Out in 2018

Blockchain platforms:

1. Ethereum – open-source blockchain platform; runs smart contracts; allows for design and issuance of cryptocurrencies; offers Turing-complete languages; command line tools
2. BitCoin – perhaps the most known blockchain platform among the general public; helped create the boom of cryptocurrencies during the WannaCry ransomware attack; open-source; written in C++
3. Hyperledger Fabric – project by company Hyperledger; uses Docker for smart contract implementation; serves as a basis for blockchain-based solutions; very good for permissioned blockchains (only certain users have data access)
4. Hyperledger Cello – blockchain-as-a-Service platform

¹In ideal circumstances – provided the platform is not under any type of attack or doesn't have any kind of vulnerability that would allow block deletion/modification. The blockchain would also have to be public – if a certain group of people or a company controls a blockchain nobody could stop them from modifying it.

5. Hyperledger Sawtooth Lake – enterprise solution for both permissioned and permissionless blockchain; provides smart contracts abstraction layer (allowing to write contract logic in any programming language)
6. Hydrachain – Ethereum blockchain extension; allows for development and deployment of permissioned distributed ledgers; compatible with Ethereum blockchain; provides infrastructure to create smart contracts in Python; offers high level of customization
7. Corda – open-source blockchain platform for building permissioned distributed ledger systems; created by consortium of banks
8. IBM blockchain – progressive ledger for transactions; permissioned network; paid plans; supported languages Go and Java
9. Multichain – open-source distributed ledger system; based on Bitcoin; created for multi-currency financial transactions
10. Openchain – open-source blockchain platform; designed for management of digital assets; includes smart contract modules; every transaction is digitally signed; free to use
11. Chain Core – enterprise-grade blockchain platform; used for digital assets management on permissioned networks

1.2 Smart Contracts

1.2.1 Basic description

Smart contracts are a blockchain equivalent of regular contracts. They are programs written in specialized programming languages (e.g. Solidity for Ethereum blockchain) that use *if-then* and other programming language statements instead of legal terms. A benefit of smart contracts is that they not only define the rules and penalties of the agreement, but also automatically enforce those obligations – if created correctly and safely ².

One could imagine smart contracts as a vending machine – one would send cryptocurrency (along with some other information) to an address where a smart contract is located and would receive goods into their account. And because smart contracts are processed as a regular blockchain transaction they also get recorded on the blockchain and thus are kept securely.

One of the smart contracts disadvantages is its "immaturity" – as the smart contracts are a known tool in a relatively small group of people. There is currently no way to enforce smart contracts in the world outside of blockchain platforms.

1.2.2 Sources

These papers and books were used as a source for smart contracts vulnerabilities.

- A survey of attacks on ethereum smart contracts by Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli [6]
- Making smart contracts smarter by Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena and Aquinas Hobor [7]
- Hawk: The blockchain model of cryptography and privacy-preserving smart contracts by Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen and Charalampos Papamanthou [8]

1.2.3 Execution fees

Users can call the functions by sending ether to the corresponding address. Ether is a necessary element of the Ethereum networks. It is used both as a currency that users can send to each other and also as fuel sent as a payment to the miners by the users for the execution. The fee itself is both an incentive for the miners to supply the computing power and a way to stop DoS (denial-of-service) attacks – which would be costly.

²In case of an exploitable smart contract the attacker might not have to pass the usage requirements of the smart contract – e.g. cryptocurrency deposit – and thus be able to abuse the smart contract for his personal gain.

The fees are defined by *gas* and the *gas price* for each transaction. The gas (also called gas limit) is the maximal cost the user is willing to pay for the transaction – for example if there was no concept of the gas limit and an error would occur the transaction could be cycling endlessly and the user would lose all his currency. The gas price is the amount the user will pay for a unit of gas. This price is set by the users and the miners can decide which transactions to mine into a block – based on the price. The bigger the gas price is, the sooner the transaction will be completed – since most of the nodes favor the more profitable transactions.

If the contract ends successfully the remaining gas would return to the caller. If the gas limit is too low the transaction will end with an "out-of-gas" exception. If the transaction ends with another exception all of the remaining gas will be lost.

1.2.4 Example smart contract

This example [6] is a simple smart contract which implements a personal wallet. The wallet can receive ether from other users and also pay with the function *pay*.

```
1 contract AWallet{
2   address owner;
3   mapping (address => uint) public outflow;
4
5   function AWallet(){ owner = msg.sender; }
6
7   function pay(uint amount, address target) returns (bool){
8     if (msg.sender != owner || msg.value != 0) throw;
9     if (amount > this.balance) return false;
10    outflow[target] += amount;
11    if (!target.send(amount)) throw;
12    return true;
13  }
14 }
```

The function *AWallet()* is a constructor which will run only when the smart contract is first created. In the *pay* function we can see the basic checks – the first check is that the user calling the function has to be the wallet's owner and the amount that is being paid is not zero. The second check is to see whether the owner has got sufficient funds .

We can also see *throw* function that can raise an exception in case the checks fail. The exceptions are not handled in any way – if an exception is caught the contract stops, the miner's fee is lost and all the ether transfers are reverted.

1.2.5 Vulnerabilities

Security of smart contracts is essential, since they are a digital equivalent of real-life transactions. This section describes vulnerabilities from four main groups:

- Solidity vulnerabilities
- EVM bytecode vulnerabilities
- Blockchain vulnerabilities
- Implementation vulnerabilities

1.2.5.1 A call to the unknown

The design of the Solidity language may invoke fallback functions of the callee (recipient).

- If a function of a contract is invoked by *call* function, the ether is transferred to the callee and the function does not exist at the called contract the fallback function of the called contract is executed
- After an amount of ether has been sent by *send* function it also executes the recipient's fallback function
- A function *delegatecall* is similar to the *call* function, however with *delegatecall* the invocation of the called function is run in the caller environment – so if a *this* variable is used in function called by *delegatecall* it would point to the caller's address and not to the callee's address

This means that if a programmer was to mistype a function the fallback function of the recipient (potentially malicious) would be executed and used as an exploit.

1.2.5.2 Exception disorder

Solidity raises exceptions in following situations:

1. If execution runs out of gas (detailed in 1.2.3)
2. If call stack reaches its limit
3. If command *throw* is executed (detailed in 1.2.4)

The exception handling is however not always the same – it depends on how the contracts call each other. For example if we have these two contracts [6]:

```
1 contract Alice {
2   function ping(uint) returns (uint)
3 }
4 contract Bob {
5   uint x=0;
6   function pong(Alice c){
7     x=1; c.ping(42); x=2;
8   }
9 }
```

If someone were to invoke Bob's *pong* and Alice's *ping* was to throw an exception the execution would stop and the side effects of the whole transaction would be reverted. That means the field *x* would contain 0 after the transaction. If Bob was to invoke *ping* with a *call* command and it was to throw exception again the invocation effects would revert and the execution would continue and the field *x* would contain 2 after the transaction.

1.2.5.3 Gasless send

When using the command *send* to transfer ether to a contract it is possible to receive out-of-gas exception. It may be unexpected but when compiled by a compiler of version higher than 0.4.0 the implicit number of gas units is 2300. So if a contract has an expensive call function (reason mentioned in 1.2.5.1) the invocation will probably end with the aforementioned exception. So sending ether succeeds either when the recipient contract has got an inexpensive fallback function or when the recipient is a user (as opposed to a smart contract).

1.2.5.4 Re-entrancy

Smart contract programmers may believe that when invoking a non-recursive function it cannot re-enter before its termination. This however might not be always true – because of the fallback mechanism. Let's take an example of a legitimate contracts Bob and a malicious contract Mallory [6]:

```
1 contract Bob {
2   bool sent = false;
3   function ping(address c) {
4     if (!sent) {
5       c.call.value(2)();
6       sent = true;
7     }
8   }
9 }
10
11
12 contract Mallory {
13   function() {
14     Bob(msg.sender).ping(this);
15   }
16 }
```

The *ping* function in contract Bob sends 2 wei ($1ether = 10^{18}wei$) to an address c with *call* function with no gas limit. If the *ping* function was to be invoked with Mallory's address it would invoke Mallory's fallback function – which is the *ping* function again. This would create a loop which would end when the execution goes out of gas or when stack limit is reached or when Bob has no ether left. An exception would be thrown, however *call* function would not propagate it so all of the previous transactions would stay valid.

1.2.5.5 Keeping secrets

As in object-oriented programming, variables in smart contracts can be either set to private or public. However to publish a transaction it must be sent to miners who would post it on the blockchain. Thus, everybody can see and verify the transaction. This is sometimes undesired – e.g. in games where the next move of the oponent has to be kept secret. In order to ensure appropriate secrecy adequate cryptographic techniques have to be employed (timed commitments [9] etc.).

1.2.5.6 Immutable bugs

All of the contracts once published on the blockchain are permanent – there is no way of altering them after publishing. This fact has both positive and negative effects – if a contract is well written users can expect it to work consistently all the time. However if the contract includes bugs there are no direct ways of changing the contract. Programmers have to implement a way to either change or completely stop a contract.

1.2.5.7 Ether lost in transfer

Ethereum smart contract addresses are a sequence of 160 bits (this equals to theoretically up to 2^{160} possible addresses). When sending any amount of it a recipient address has to be specified. The address has to be thoroughly verified, since it is entirely possible that an address doesn't belong to any user or contract – so called *orphan* address – and there is no way of finding out whether an address is orphaned or not.

1.2.5.8 Transaction ordering dependence

In Ethereum a new block is appended to the blockchain approximately each 12-14 seconds – this is set by design. If a smart contract is state-dependent (dependent on other smart contracts) it could have either malicious or unintentional unexpected consequences. In such scenario, only the miners can decide which transaction goes first and so they are ultimately responsible for the final state of contracts.

If a marketplace is realized by smart contracts and it updates its prices frequently some users might either not be able to buy their requested items or could pay much higher price for the items – this would be unintentional consequence.

Another example – because of the delay between transaction publication and its inclusion in the blockchain a malicious owner of any type of blockchain puzzle game that has to be paid ether for could listen for transactions with a solution. He would immediately publish a transaction lowering the reward amount – if the owner’s transaction would get executed before the one with a solution the owner would gain the fee but would not have to pay out the reward.

1.2.5.9 Timestamp dependency

Some smart contracts can depend on a timestamp of a previous block – for example as a seed for a PRNG (pseudo-random number generator) when choosing a lottery winner. However the miners can manipulate the timestamp. A timestamp can vary by approximately 15 minutes and it would still get accepted by other miners – however it also has to be greater than the timestamp of a block before it. This means that a malicious miner could manipulate the timestamp in order to win a reward from a smart contract that depends on it.

1.2.5.10 Mishandled Exceptions

There are multiple ways of calling other contracts in Ethereum, such as *send* instruction or the function directly. Should the called contract raise an exception (stack limit exceeded, not enough gas, etc.) it terminates, reverts its state and returns *false* return value. When using the *send* function the caller contract should always check the return value to make sure the call was successful. From the first 1,459,999 Ethereum blocks 27.9% of the contracts do not check the return values after calling another contract via *send* function [7].

1.2.5.11 Multiplayer betting vulnerability

Let’s take an example – a two-player game in which both of the players deposit given amount of ether and bet a number. If the sum is even the first player wins, if it is odd then the second player wins. An attacker could wait for a bet from the first player, learn his bet number by examining the blockchain transaction, bet an appropriate number and win the bet.

1.2.6 Past real-world smart contract attacks

1.2.6.1 The DAO attack

The DAO was an implementation of a crowd-funding platform (e.g. Kickstarter) for smart contracts. It had raised approx. \$150M before it was attacked on 18th June 2016. The attacker was able to steal approx. \$60M – but most of the Ethereum community agreed on reverting the state of Ethereum blockchain before the attack. This subsequently led to Ethereum being forked into Ethereum classic and Ethereum. The example [6] shows a simplified version of the DAO contract and a malicious contract Mallory.

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11     }
12   }
13 }
14
15 contract Mallory {
16   SimpleDAO public dao = SimpleDAO(0x354...);
17   address owner;
18   function Mallory(){owner = msg.sender; }
19   function() { dao.withdraw(dao.queryCredit(this)); }
20   function getJackpot(){ owner.send(this.balance); }
21 }
```

The attacker first publishes the malicious contract. He then donates some ether to Mallory and this invokes Mallory’s fallback function. The fallback function will invoke the *withdraw* function – transferring the ether back to Mallory. The *call* function in *withdraw* invokes Mallory’s fallback function once again – which invokes *withdraw* again. The key to the attack is that the *withdraw* function has been stopped before line 10 where it would update the credit field. This sends the amount again to Mallory, invoking the fallback function yet again – resulting in an endless loop of transactions to Mallory that would end with an exception (stack or gas exhaustion) or when the DAO contract runs out of ether. Also the *call* function in *withdraw* function does not specify the gas limit – so an attacker could supply more gas in order to gain more ether from the attack before the out-of-gas exception. The vulnerabilities behind this attack are explained in sections 1.2.5.1 and 1.2.5.4.

1.2.6.2 Ethereum Classic 51% attack

On January 5th, 2019 attackers were able to attack Ethereum Classic blockchain – specifically coin exchanges Coinbase, Bitrue and Gate.io. The attackers were able to double-spend Ethereum Classic cryptocurrency (ETC) by having more computational power than the rest of the miners. This means that they were able to carry out two transactions but only pay for one – thus being able to multiply the amount of the cryptocurrency owned by them. Bitrue’s systems were able to halt the transactions, however the other two exchanges couldn’t stop them.

1.2.7 Smart contract testing

Smart contracts can be tested either by static code review or by using automated tools (e.g. Oyente [7] and OpenZeppelin [10]). There are also compilations of Ethereum best practices for smart contract security, such as the one by ConsenSys [11].

Increasing amounts of resources are being stored in smart contracts, thus putting increased pressure on developers to create safe and secure contracts. This also means that either the developers are trained in security principles or the teams have dedicated security professionals who check all of the contracts.

1.2.8 Summary

Smart contracts are a blockchain equivalent of regular contracts. They are a viable way of creating complex decentralized systems – by being able to theoretically emulate real contracts one could create almost anything with their help. However, they are currently not enforceable outside of the blockchain “ecosystem”.

Their vulnerabilities can be split into four groups : blockchain vulnerabilities, EVM bytecode vulnerabilities, Solidity vulnerabilities and poorly written contracts. We have shown a few examples of those vulnerabilities and two notable carried out attacks.

1.3 Web application

Web applications are the new face of the Internet. Before, with earlier client and server models there had to be a special application installed on every client computer and a bigger change in server code meant the client application had to be updated as well – which was ineffective.

This expansion of web applications brings a lot of security challenges with itself. Users transmit highly important data (e.g. bank account credentials) or their personal data, which could also be used or sold by a potential attacker.

OWASP Top 10 2017 [12] will be used in this thesis as a source material for the most common web application security vulnerabilities. The current Top 10 2017 and the changes from the previous list from year 2013 can be seen in figure 1.2.

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	⊗	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	⊗	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Figure 1.2: OWASP Top 10 2013 and 2017 [12]

1.3.1 Injection

1.3.1.1 SQL injection

SQL injection is an attack on the database layer of a web application. It uses an unsanitized data entry point to insert a specially modified input that contains either SQL commands or special data to modify, view or delete data or gain access to a database. There is also a special type of SQL injection called **blind injection** – with the difference being that during a blind SQL injection the attacker performs true or false actions on the target database and determines the output based on the application response.

SQL injection principle example:


```
1 SELECT * FROM users WHERE username = ' " + inputName + " ' AND
   password = ' " + inputPass + " '
```

This could potentially be a SQL statement in an application login page. An attacker could enter

```
1 'or 1=1 --
```

which would result in a following SQL command :

```
1 SELECT * FROM users WHERE username = ' ' OR 1=1-- '
2 AND password = 'foo'
```

And since the double dash comments out the rest of the query and the

```
1 ' ' OR 1=1--
```

statement will always be true the target application would log the attacker in without a valid password.

The main way of protecting an application against a SQL injection attack is to check and validate the input thoroughly [13].

Injection is a type of attack that could be used on a variety of technologies (e.g. SQL injection, command injection, Lightweight Directory Access Protocol (LDAP) injection, Extensible Markup Language (XML) injection, SMTP header injection). Its main focus is to confuse the target system and either control it, gather information or make it unavailable for legitimate traffic.

1.3.1.2 Command injection

Command injection attacks are possible when application passes user data that have not been properly sanitized to a system shell. The commands supplied by the attacker are usually executed with the privileges of the vulnerable application. These attacks are possible due to insufficient user input check and validation.

Example 1:

```
1 <?php
2 print("Please specify the name of the file to delete");
3 print("<p>");
4 $file=$_GET['filename'];
5 system("rm $file");
6 ?>
```

This is an example of a short PHP program used for file deletion. If the malicious user was to call the program like this:

```
1 http://127.0.0.1/delete.php?filename=bob.txt;id
```

The response would be:

```
1 Please specify the name of the file to delete
2
3 uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

1. ANALYSIS

Allowing the malicious user to carry out reconnaissance, establish backdoor, cause denial-of-service attack or anything the user *www-group* is entitled to.

Example 2:

```
1 int main(int argc, char** argv) {
2     char cmd[CMD_MAX] = "/usr/bin/cat ";
3     strcat(cmd, argv[1]);
4     system(cmd);
5 }
```

This short program written in C programming language could be used as a tool to display system files. Let's say this program is being used by new administrators to view system files and not change or damage them. The code would be run with root privileges and thus the **system()** call will also run with root privileges. A normal use case would be to insert a filename that is to be read. However an attacker could modify the input and insert

```
1 /etc/shadow; rm -rf /
```

which uses the fact that a semicolon is also used to split two commands on one line. And because the input is not sanitized the whole command

```
1 /usr/bin/cat /etc/shadow; rm -rf /
```

would be run with root privileges and the whole root partition would be deleted recursively.

This attack could be potentially used to download malware to the server, gain or maintain access or disrupt the target system.

There are multiple ways of protection against this type of attack – thorough input sanitization, not calling operating system commands directly or using a web application firewall as an additional layer of protection [14].

1.3.1.3 LDAP injection

LDAP stands for Lightweight Directory Access Protocol and is used for storing and accessing data on a remote directory server [15]. LDAP injection is an attack that uses modified input in order to force the application to either reveal or modify sensitive information stored in LDAP data stores.

Example:

```
1 searchlogin= "(&(uid="+user+")(userPassword={MD5}"\
2     +base64(pack("H*",md5(pass))))+"))";
```

In this example we can see a filter searching for user/password pair where the password is sent in Base64 encoding as a MD5 hash. The attacker would then input following data:

```
1 user=*)(uid=*)(|(uid=*
2 pass=password
```

This would result in the filter being

```
1 searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword=\
2 {MD5}X03M01qnZdYdgyfeuILPmQ==))";
```

and being correct and true. The attacker would then be logged in without a password.

The main way of preventing a LDAP injection is either variable escaping, usage of frameworks that would automatically protect from LDAP injection or least privilege principle [16].

1.3.2 Broken Authentication

Authentication is the process of verifying whether the client trying to log in (a person, a device or a software process) is really who he claims to be. The most common way of authenticating today is a password. However, solely using a password is not enough today. Multi-factor authentication is on the rise – the user has to provide additional information in order to access the system. There are three main components of multi-factor authentication – something the user knows (password or personal information), something the user has (an application on a smartphone or a hardware token) and something the user is (a fingerprint or an iris). These types of additional security layers are the only thing the user can do himself to minimize the risk of his account getting stolen – as opposed to a database with e.g. weak hashing algorithm which can only be fixed by the application developers.

Web application developers play the biggest role in the application security because an attacker would only need them to make one mistake to breach the security.

1.3.2.1 User authentication credentials are not protected when stored

Password is one of the most valuable credentials – it should be protected very carefully. The standard way of storing a password is to use a strong hashing algorithm with a salt in order to prevent both collision attacks (creating two different inputs that produce the same hash value) and rainbow tables attacks (having a precomputed hash tables – this type of attack gets harder with proper salt usage).

1.3.2.2 Session IDs are exposed in the URL

Session IDs are a valuable information – with a session identifier an attacker could impersonate a victim even without their credentials. When a session ID is displayed in a URL the user can either send it to someone else (e.g. to show what he bought) – but anyone with the link would be able to see the credit card details as well. The leak of session ID can also happen when it is stored in inappropriate places for a long amount of time or sent without encryption.

1.3.2.3 Session does not invalidate after a logout

Session should always be invalidated as soon as possible. An attacker could acquire a session ID after a moment – if the session is not invalidated server-side they would use it to impersonate the legitimate user and get access to their account.

1.3.2.4 Predictable login credentials

Passwords should be forced to be sufficiently long and complex, however there is a trade off between the password length and complexity and the user's ability to remember passwords. The main idea is that if a password is a relatively complex but short (e.g. 3!1m) it is quite hard for a person to remember – but it is quite easy to be bruteforced.

In a passphrase there could be a problem with a combination of very common words that could be easily guessed. This would be avoided by either using more words and some numbers as well or by using uncommon, slightly changed or unpredictable words. One could also mitigate the risk by using multi-factor authentication.

1.3.2.5 Passwords, other credentials or session IDs are sent over unencrypted connections

All of the traffic between a user and a web server should be encrypted by Transport Layer Security protocol (TLS) or its equivalent – especially the login sequence. More primitive transformations (e.g. hashes) are not enough since after capturing them the attacker could easily resend them for authentication without any knowledge of the plaintext.

1.3.2.6 Login mechanism does not have a logout policy

The login mechanism should have a logout policy implemented. This vulnerability would allow the attacker to carry out bruteforce attacks on the target application. In a combination with a weak password policy this vulnerability could be very dangerous.

1.3.3 Sensitive Data ENTITYxposure

Almost every modern web application is handling some type of sensitive data. Whether it is password, credit card number, sensitive personal information, health information or company information it needs to be properly protected in all stages (during a transfer, client-side or in storage server-side).

1.3.3.1 Cleartext data transmission

All the web applications should use and enforce TLS for all their pages. In case the attacker monitors the victim's network traffic and is able to downgrade the Hypertext Transfer Protocol Secure (HTTPS) traffic to Hypertext Transfer Protocol (HTTP) he could intercept the session token and then replay the token to the application and hijack the user's authenticated session. He could also alter the transported data, for example changing the recipient of a money transfer.

1.3.3.2 Safe data at rest

All of the sensitive data should be encrypted even when not used. An attacker does not have to be an outsider – quite a lot of attacks are carried out by either current or past employees.

1.3.3.3 Cryptographic algorithms

Cryptographic algorithms generally don't stay safe forever. And thus the developers should watch and eventually replace old cryptographic algorithms in case they get deprecated.

1.3.3.4 Cryptographic keys

Even the safest cryptographic algorithms can be bypassed when weak or default cryptographic keys are used. The keys themselves should also be securely stored – either in a HSM (hardware security module) or somewhere where only an administrator can have access.

1.3.3.5 Encryption enforcement

It is very important that encrypted communication is strictly enforced for all web applications. HTTP Strict Transport Security (HSTS) is a mechanism that helps achieve just that. The web server notifies the browser that it is to use only HTTPS instead of HTTP for a period of time. It does so with a header **Strict-Transport-Security** which also contains a time period for which the communication should be HTTPS only. HSTS makes some forms of the man-in-the-middle attack harder to successfully execute.

1.3.3.6 Certificate verification

The user agent should verify the server certificate it receives. Browsers do show the certificate, its information and even a warning when certificate is in disorder, however users have to either check for themselves or employ plugins that can check whether the certificate is really from the target application and it has not been altered. Non-web applications should have a way of

confirming the certificate validity – the certificate has to change in some time so the application should also be prepared for that occasion.

1.3.4 XML External Entities

This vulnerability can be found in servers that parse Extensible Markup Language (XML) input with a poorly configured parser. An attacker would refer an external entity on the system (e.g. /etc/passwd or /etc/shadow files) and the misconfigured XML parser would print the content of the files.

Example 1 [17]:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
3 <!ELEMENT foo ANY >
4 <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
5 <foo>&xxe;</foo>
```

In this code the attacker is trying to access the /etc/passwd file on the remote server. The attacker does not need to explicitly receive a whole answer since he could theoretically exfiltrate the information through any other covert channel - e.g. by DNS subdomains to a DNS server the attacker controls [18].

Example 2 [19]:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE lolz [
3 <!ENTITY lol "lol">
4 <!ELEMENT lolz (#PCDATA)>
5 <!ENTITY lol1 "&lol;&lol.....&lol;&lol;&lol;">
6 <!ENTITY lol2 "&lol1;&lol1;.....&lol1;&lol1;&lol1;">
7 <!ENTITY lol3 "&lol2;&lol2;.....&lol2;&lol2;&lol2;">
8 <!ENTITY lol4 "&lol3;&lol3;.....&lol3;&lol3;&lol3;">
9 <!ENTITY lol5 "&lol4;&lol4;.....&lol4;&lol4;&lol4;">
10 <!ENTITY lol6 "&lol5;&lol5;.....&lol5;&lol5;&lol5;">
11 <!ENTITY lol7 "&lol6;&lol6;.....&lol6;&lol6;&lol6;">
12 <!ENTITY lol8 "&lol7;&lol7;.....&lol7;&lol7;&lol7;">
13 <!ENTITY lol9 "&lol8;&lol8;.....&lol8;&lol8;&lol8;">
14 ]>
15 <lolz>&lol9;</lolz>
```

This attack is called **Billion Laughs Attack**. Its task is to do a denial of service attack. It works by declaring a root element "lolz" that contains the text "&lol9". But "&lol9" entity contains ten "&lol8" entities and each of them contain 10 of "&lol7" entities and so on. This small code would produce 10^9 "lol" entities (thus billion laughs attack) that would take up almost 3 gigabytes of memory.

1.3.4.1 Remediation

All XML processors and libraries should be patched or upgraded regularly, input should be thoroughly sanitized and escaped, less complex formats that

avoid serialization of sensitive data could be used (such as JSON) a server-side whitelist-based input validation should be implemented.

1.3.5 Broken Access Control

Access control – sometimes called authorization – is a way to grant appropriate rights to legitimate users. Attackers typically want to obtain access to accounts with the highest possible rights in order to have greater control over the system. This type of vulnerability is often found in applications that have grown in size over time. Used authorization methods were appropriate only for the small scale of the application in the beginning and were either not changed properly or just added over time.

Example 1:

`http://example.com/app/accountInfo?acct=admin` In this example the account currently logged in is transferred via a HTML parameter. An attacker could log into any account and change his authorization to an administrator account afterwards.

Example 2:

`http://example.com/app/orderInfo?order=123456789` In this example the application would not have restricted access to orders for the customers. An attacker could browse the individual ID's and look for an order that has already been paid for and simply collect the order himself.

1.3.5.1 Remediation

The main way of preventing broken access control is to define the access control rules – with an access control matrix for example. The matrix should document the types of users that would access the system and the functions and content they should be able to access. Also whitelist approach should be preferred – everything should be denied and the access should only be given to appropriate users, not the other way. The developers can also tie the authorization with every request – potential attack would have to bypass the check with every request and not only once.

1.3.6 Security Misconfiguration

The appropriate configuration of both the application and the server it is running on is essential for data safety. Any of the below listed misconfigurations would allow an attacker to either retrieve sensitive data or gain control over the target system. These misconfigurations could happen at any level of the technological stack so an automated tool for constant checking would simplify the task greatly. The most common misconfigurations are:

- Unpatched systems and packages

- Using default login credentials (usernames and passwords)
- Unprotected files and directories
- Enabled directory listing
- Unused web pages
- Misconfigured network devices
- Poor error handling
- Cloud storage with faulty permissions

1.3.6.1 Remediation

The task to configure the system and application properly is extremely specific for each application and technology, so there are also many ways of avoiding the misconfiguration. One could consult CIS Benchmark [20] for platform and application hardening. The available CIS Benchmarks include Apple Safari Browser, Google Chrome, iOS, Android, Checkpoint Firewall, Cisco Firewall Devices, Cisco Routers/Switches, Amazon Linux, CentOS, Debian Linux Server, Microsoft IIS Server, Apache HTTP Server, MySQL Database Server, Docker, Kubernetes, VMware Server and many others. Some of the general tips to prevent misconfiguration are:

- Developing a repeatable patching process
- Keeping software up to date
- Disabling default accounts
- Encrypting data
- Enforcing strong access controls
- Providing administrators with a repeatable process to avoid overlooking items
- Setting security settings in development frameworks to a secure value
- Running security scanners and performing regular system audits
- Disabling administration interfaces
- Disabling debugging
- Configuring server to prevent unauthorized access, directory listing, etc

1.3.7 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a way to disrupt web pages by leveraging vulnerabilities in scripts – unsanitized inputs mostly. An attacker could insert their own malicious (mostly JavaScript) code. They could then use the code to disrupt design of the web page, make it malfunction, gain sensitive information or bypass security measures. It is also used during phishing attacks to show the user different content on a trustworthy website.

There are three basic XSS types.

1.3.7.1 Reflected XSS

Can also be called **non-persistent**. It can be used on a page with generated content. If an attacker adds his code that is not stripped than the page will use it as if it was on the page before.

Example:

If a web application is written in PHP and the title is fetched like this:

```
1 <?php echo $_GET['title']; ?>
```

An attacker could input `http://URL/webpage.php?title=hello<script>alert('1');</script>` which would create an alert with a message "1".

1.3.7.2 Stored XSS

Also called **persistent** because an attacker would typically leave the malicious code somewhere on the target web application in order for it to get stored in a database. This modified code and would get executed when requested by other users. This can be done for example in an unsanitized comment section under an article – where malicious user would post a XSS code and it would get executed by the browsers of other users.

Example:

```
1 Dear user <script>alert('1')</script> welcome!
```

This comment would display alert to every user who visits the malicious website.

1.3.7.3 DOM based XSS

Is also called **local**. It can also be used on static web pages by inserting a variable as a parameter into unsanitized input.

Example:

```
1 <SCRIPT>
2 var pos=document.URL.indexOf("name=")+6;
3 document.write("Hello "+document.URL.substring(pos,document.URL.
4   length));
4 </SCRIPT>
```

If a legitimate user was to visit the website through link `http://URL/webpage.html?name=User` the page would write "Hello User". However an attacker could use a modified URL `http://URL/webpage.html?name=<script>alert('1');</script>` which would create an alert with a message "1".

1.3.7.4 Remediation

The main ways of remediation are HTML context-aware encoding (based on location in the HTML output), a framework which automatically escapes XSS (e.g. Ruby on Rails or React JS) or thorough input sanitization.

1.3.8 Insecure Deserialization

Serialization is the process of storing input into a data format that can be used for storage. Deserialization is the exact opposite – restoring stored data formats into an usable object. The most popular format for data serialization used to be XML and JSON is the most used format today. Deserialization itself is a needed process, however a problem occurs when deserialization is done on unsafe, unsanitized and malicious data.

Two main types of attacks could occur from insecure data serialization – either object or data structure modification and resulting arbitrary remote code execution or data tampering attacks.

Example:

User registration is done by a "super" cookie that is created by PHP object serialization.

```
1 a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
2 i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker could locally modify the cookie and change his username and privilege level as well.

```
1 a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
2 i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

1.3.8.1 Remediation

The best remediation is to either not accept serialized objects from untrusted sources or accept only sources that use primitive data types. If this is not possible then following would be recommended to implement [21]:

- Integrity checks
- Strict data type enforcement before object creation
- Isolating the deserialization code
- Logging of code exceptions and failures

- Network monitoring to and from deserialization servers or containers
- Monitoring of the deserialization itself

1.3.9 Using Component With Known Vulnerabilities

Every web application is dependent on its underlying technologies. Whether an application is hosted on a physical server (as in figure 1.3) or in a datacenter with the help of virtualization technologies every component on both server and client sides has to be updated to their stable versions.

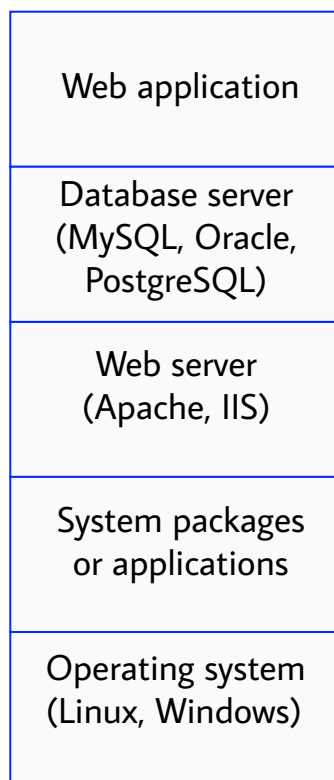


Figure 1.3: Web Application technology stack

NVD (National Vulnerability Database) publishes known vulnerabilities in both open-source and proprietary components – it is not meant for attackers, however from the moment of publication even they can find the individual vulnerabilities and exploit them. Open-source software is widespread – even proprietary applications use a big amount of open-source code [22].

Outdated software components should not be taken lightly, for example the Equifax breach was caused by an Apache Struts vulnerability [23] for which the patch has been issued the very same day it was discovered – on March 7th, 2017. The vulnerability has not been patched and the attackers were able to

exploit the vulnerability and gather sensitive personal data [24]. Exploiting the vulnerability has been relatively easy, since it scores 10.0 CRITICAL on CVSS (Common Vulnerability Scoring System) v3.0 metric. The main reason for this value is that the vulnerability does not require any privileges, the attack itself is not complex and results in high confidentiality, integrity and availability impact [25].

1.3.9.1 Remediation

The main idea is to keep only the required applications and have them always patched to a newest version. More specifically:

- Remove unneeded dependencies
- Keep a list of both client and server-side technologies and their versions
- Monitor NVD or similar sources for new vulnerabilities
- Only receive updates from verified sources
- Patch regularly (perhaps wait a few days in case the new patch is faulty)

1.3.10 Insufficient Logging and Monitoring

It might not be a priority while designing an infrastructure, however the difference between a successful and a blocked attack are adequately set up logging and monitoring systems. A monitoring system would either stop or at least notify responsible people about an incoming attack. Both application and security logs should be collected and analyzed, as both of them could serve as a baseline when creating alerts for unusual situations.

1.3.10.1 Remediation

- Logging should be implemented wherever possible
- High-value transactions should be logged with great detail to leave a sufficient audit trail
- Logs should be generated so they could be processed by a centralized log management system
- Logging servers should be kept secure and separated – both physically and logically
- Incident response and recovery plans and processes should be adopted
- Monitoring and alerting should be implemented so it would detect, notify and respond to a possible threat as soon as possible

- All security-sensitive logs should include enough context in order to identify malicious activities

1.3.11 Summary

Web application testing is often done according to the OWASP Testing Suite. The 10 of the most notorious, severe and potentially dangerous vulnerabilities have been described in this chapter. The mentioned vulnerabilities pose various levels of threat – the consequences could range from a DoS (denial of service) to a sensitive information disclosure (such as passwords or credit card numbers).

Developers are not as well versed in security principles as penetration testers and thus every web application should be thoroughly tested and also designed with security in mind in order to fix and avoid as many vulnerabilities as possible.

1.4 Mobile application

Mobile applications – often referred to as apps – are the main way of using phones today. Everything has its own application – whether it is a chat service, online banking, car controls, smart homes or security systems. This wide spread of applications brings a lot of security challenges as mobile applications tend to be more closed for their users. For example, when logging into a bank web application one could see the HTTPS mark in their browser and theoretically could verify the connection details and the certificate as well. Such things are impossible to look into while using a mobile application for a regular user (without network sniffing, decompiling the application and so on). This introduces a new array of possible security vulnerabilities that require attention.

OWASP Mobile top 10 2016 [26] and Top 10 Vulnerabilities in Mobile Applications [27] article by Don Green from WhiteHat Security will be used as the main sources for this chapter.

1.4.1 Improper Platform Usage

Each mobile platform (Android, Windows Phone, iOS) provides its own set of features or capabilities. The mobile developers either use the feature incorrectly or don't use the feature at all in a lot of cases. These features or capabilities are well known and well documented and developers should use them the way they were meant to be used. This vulnerability is also the only one that is caused purely by developers.

There are several ways this vulnerability can occur:

- Guideline violation – if an app doesn't follow the best practices for each of the specific platforms it will be exposed to this risk (e.g. improper iOS Keychain usage)
- Common practice violation – mobile app development has got its own common practices and the developers should respect them
- Unintentional misuse – despite the best intentions some of the apps can be implemented badly and would not do what they were originally designed to do – this can happen as a result of a bug or a wrong API call flag

Example:

iOS Keychain is secure storage for both system and application data. Application developers should use it to store sensitive information. If the Keychain is not used the data are stored in app local storage and could be found in unencrypted iTunes backups. If a mobile banking application would misuse the Keychain functionality the password would be accessible on the victim's computer.

1.4.2 Insecure Data Storage

Each application that deals with any kind of sensitive information (personal information, passwords, cookies, tokens, encryption keys etc.) should have proper security measures in place to keep the information secure from unauthorized users at all times. App developers assume that the user or malware will not have access to the filesystem – however that does not necessarily have to be true. A new vulnerability can be found (in the operating system, framework, compilers, hardware or the device will be rooted/jailbroken) and the filesystem would suddenly be accessible.

1.4.3 Insecure Communication

Most of the current applications use some type of communication – whether it is a server-client or peer-to-peer – the communication has to be protected from the beginning until the very end. The best way of mitigating such vulnerabilities is to assume that the used channel is not secure and thus adequate encryption should be used at all times – especially when transmitting any type of sensitive data. The encryption should use currently recommended cipher suites with adequate key lengths. The certificates should also be signed by a trusted certificate authority and the server-client connection should be established only after the target servers are verified with trusted certificates. Also the sensitive data should have a separate encryption layer in order to prevent future TLS implementation vulnerabilities.

1.4.4 Insecure Authentication

Authentication in a mobile application can be tricky since its requirements differ from authentication in web applications (see section 1.3.2 for additional information on web application authentication vulnerabilities). Unlike in web applications the users of mobile applications are not expected to stay connected to the Internet at all times during their session – which calls for additional measures, such as offline authentication.

Mobile application will suffer from insecure authentication if it allows for anonymous API execution, stores passwords/secret/private keys locally and outside of secure dedicated storages or when it only supports basic authentication schemes (e.g. only passwords). Attackers could leverage these vulnerabilities during an attack – for example by stealing the phone and reverse-engineering passwords to e-mail addresses, bank accounts or any other online accounts.

There are several ways the developers can enhance the security of their applications, such as :

- When creating a mobile application equivalent of a web application, the mobile version should have the same authentication requirements. This

should be done so the mobile application is not a weaker link and thus is not an easier target for attackers.

- When implementing local authentication the attacker could bypass it with a rooted Android device or jailbroken iOS device. If such need is substantiated then a special care has to be taken to prevent binary attacks against the application.
- The authentication requests should be performed server-side. This ensures that application data will be loaded onto the device only after successful authentication.

1.4.5 Insufficient Cryptography

It is essential that cryptographic algorithms and functions used for encryption are chosen according to reputable sources (e.g. National Institute for Standards in Technology - NIST), are used in a correct manner and with adequately long keys.

Some of the deprecated ciphers and hash algorithms are Message-Digest algorithm (MD5), Secure Hash Algorithm (SHA1) and Data Encryption Standard (DES) and Rivest Cipher 4 (RC4) [28]. An example of a ciphersuite that is considered strong enough in the time of writing is *TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384* [29]. Its components are:

- TLS – Transport layer security protocol
- ECDHE – Elliptic-curve Diffie-Hellman key exchange protocol
- RSA – asymmetric cipher used for authentication
- AES-256 – symmetric cipher for the communication itself
- GCM – Galois/Counter mode of the symmetric block cipher
- SHA384 – a hashing algorithm used for message authentication

1.4.6 Insecure Authorization

The authorization mechanisms face similar challenges as authentication mechanisms do. Authorization takes place right after authentication and when bypassed or modified could have dire consequences, since an attacker could set higher privileges for himself and either create himself another administrator account and gain and/or delete confidential information.

In order to prevent this vulnerability all of the role and permission verification should be done on the backend systems. The reason for this is that the information originating from the mobile device may have been tampered with.

1.4.7 Client Code Quality

A lot of vulnerabilities happen just because of faulty or unrefined code. The imperfections in code can lead to buffer overflows, memory leakages or code execution. But these vulnerabilities are not always found in the application, they can also occur in the operating system's code.

A few ways to remedy these vulnerabilities is to maintain constant coding patterns, write code that is easy to read and well documented and last but not least static analysis and code review should also be conducted regularly.

1.4.8 Code Tampering

Attackers may conduct attack on the application via direct binary changes to the application's core binary or any other resources within the application. The application has to be able to detect and react to such changes at runtime in order to prevent said attacks.

Some of the ways to detect a rooted Android device are to check for known rooted APK's (Android Packages) such as *eu.chainfire.supersu*, check for SU binaries (*/system/sbin/su*, */sbin/su* or */system/su*) or to run an *su* command directly and check UID (user ID) afterwards – if the return value is 0, the command has been successful – thus the application should either exit or disable any potentially dangerous functionality.

1.4.9 Reverse Engineering

An attacker is able to perform analysis (reverse engineering) of the final application binary to determine its source code, algorithms, libraries and resources of which the application consists. There are many tools that can be used for such purposes like IDA Pro, Hopper, strings and many others.

The only way of preventing such intrusions is to use an obfuscation tool. The better ones will be able to adapt the obfuscation to balance performance impact, be able to resist known reverse engineering tools and allow string table obfuscation as well. In order to check the effectiveness of used obfuscation tools one could try the reverse engineering themselves and adjust additional measures accordingly.

1.4.10 Extraneous functionality

An attacker will examine the mobile application within their own environment – either on their device or in a simulator/emulator. All of the found files (log files, configuration files) will be examined. In case they contain any hidden switches or functionality (e.g. login bypass in order to test the application more efficiently) the attackers could use them to conduct an attack on the application.

The best way of preventing such vulnerabilities is to thoroughly remove all test code, remove all unnecessary log information, conduct manual code review and cooperate with mobile application security experts.

1.4.11 Summary

Mobile applications are growing in their popularity each day. There is an application for car control, house security control, banking, social media and almost everything else there is to be done online. This puts tremendous pressure on mobile application developers to write organized, documented and secure code in particular.

There is no one standardized list of what mobile application testing should be, however the main points of interest for both application developers and security testers have been mentioned in this section. And for this reason should mobile application developers be versed in cyber security principles, design the applications with security in mind and have a threat model. In case the developers are not able create or utilize the aforementioned measures they can employ an external security expert to either help with the design and threat model – or to consult their previous work.

1.5 Infrastructure

Every service on the Internet has to be hosted on some type of infrastructure. In former times the companies used to have all their infrastructure in-house or in their own data center. But the growth of virtualization technologies and exponentially larger network throughput allowed cloud computing to develop.

A survey carried out by LogicMonitor in year 2018 [30] showed that 37% of surveyed companies still have their infrastructure on-premises, 31% in private clouds and the remaining 37% in private and hybrid clouds. However the survey expects that by the year 2020, 41% of companies will have their infrastructure in public clouds, 27% still on-premises and 42% in hybrid and private clouds. The same survey also found that 66% of the surveyed companies think that security is the biggest challenge for organizations using public cloud today.

The infrastructure penetration testing is usually done in these main phases [31]:

1. Reconnaissance - information gathering
 - Scanning (nmap, Nikto, SSLyze)
 - Footprinting
2. Exploitation
 - Automated vulnerability testing tools (Nessus)
 - Manual vulnerability testing
 - Wi-Fi war-driving
 - Using exploits on found vulnerabilities (Metasploit)
3. Post-exploitation
 - Creating users
 - Maintaining access
4. Reporting
 - Creating structured report
 - Recommendations

Since there is no current list of most common infrastructure vulnerabilities the author's own infrastructure testing experience and parts from a list by Check Point Software Technologies from year 2016 [32] is used as a source.

1.5.1 Out-of-date Software version

It is crucial that all of the used software is updated regularly. Vulnerabilities in software are found on a daily basis and updates are released quite often – especially for widely-used software. An out-of-date software – Apache Struts – was the main reason the Equifax breach (detailed in section 1.3.9) has happened.

1.5.2 Default Configuration

Software usually requires configuration. Whether it is a Secure Shell (SSH) daemon, web server or cluster computing utility, it has to be configured. Apart from basic modes of functionality (such as password-only authentication or login only for specific users for SSH) basic or easy to guess password are also very important to change since they are well known.

1.5.3 Inappropriate Encryption

Whenever a component (web server, RDP etc.) communicates via an insecure channel or an untrusted network (e.g. Internet) the communication should be protected by encryption. The ciphers become obsolete with time and thus a reputable source (e.g. NIST) should be checked, whether a used cipher hasn't become obsolete.

In the time of writing some of the deprecated ciphers are DES and RC4, hashing algorithms MD5 and SHA1 and also protocols SSL, TLS v1.0 and v1.1 [28].

1.5.4 Self-signed or expired certificates

Self-signed certificates are sometimes used instead of a certificate issued by a certificate authority. Some of the reasons for this may be the price and also the impracticality when setting up any kind of development environment – that will be later dismantled.

However the most popular browsers do show a warning that a used certificate was not issued by a known certificate authority [33] [34] [35] – thus potentially damaging the company's reputation and the trust users have in the application. This kind of warning is also showed when an expired certificate is used, so the renewal in due time should also be done.

Furthermore the manageability of a certificate is hindered, since certificate replacement, update and revocation are handled by the certificate authority. When a self-signed certificate is used the inability to carry out the mentioned actions could bring security risks.

1.5.5 Unnecessary open services

Keeping unnecessary open services should be avoided. When a port is open and not used (and not properly secured) it can leak information to a potential attacker – software version, any type of banners or even confidential information from the service itself – should it be improperly secured.

1.5.6 Lack of Network Segmentation

When an attacker is successful in infiltrating a network segment – either by compromising a machine or simply connecting to a guest network – the segmentation of the network is crucial. For example when creating a guest network a great care should be taken to only allow communication to the Internet and not the rest of the network (stop lateral movement) – e.g. by using VLANs or firewall rules.

1.5.7 DoS/DDoS Vulnerabilities

Denial-of-service attacks can be distributed or not. The main difference is that a distributed denial-of-service attack is led by large amount of different machines from different networks and non-distributed is led by a single machine.

Examples of such attacks:

- SYN flood – the Transmission Control Protocol (TCP) uses a technique called three-way handshake while establishing a connection. Attackers could send an enormous amount of SYN packets to a server, who would send SYN,ACK packet back and wait for a finishing ACK from the attacker. The attacker wouldn't send the ACK back and send many more SYN packets instead – resulting in resource depletion on the target server.
- Address Resolution Protocol (ARP) spoofing – ARP spoofing is a man-in-the-middle (MITM) attack where the attacker would send spoofed ARP messages onto a local area network in order to associate their MAC address with an IP address of a different host (usually default gateway) on the local network in order to intercept the traffic and read or modify it.
- Fork bomb – fork bomb is an attack where a process replicates itself infinitely in order to deplete all resources with the intent of either slowing or crashing the target system.
- Dynamic Host Configuration Protocol (DHCP) starvation – DHCP starvation is an attack where the attacker changes their MAC address frequently and requests a new address from the DHCP server with each

change. If done correctly this attack could deplete all IP addresses allocated for the network segment resulting in a DoS attack. This attack can be paired with a fake DHCP server (controlled by the attacker) that would distribute malicious DHCP information, potentially resulting in a MITM attack where the fake DHCP server would appoint attacker's machine as a default gateway – making them able to either read or change outgoing traffic.

- Billion Laughs (detailed in section 1.3.4)
- Memory/CPU bugs in applications
- Firmware bugs
- Exploits or vulnerabilities in applications

Those attacks could be prevented by either patching both hardware and software components regularly and also by strengthening the supporting infrastructure (e.g. using cloud solutions with auto-scaling, DDoS prevention boxes, load balancing techniques).

1.5.8 Summary

Infrastructure is slowly moving into cloud solutions, however its security will be an ever-growing topic. Secure infrastructure should be well-designed, patched and monitored in order for it to work correctly. However cloud infrastructures do have a shared responsibility architecture – cloud providers are responsible for securing the cloud itself and customers are responsible for security of the content they put into the cloud infrastructure. This makes cloud infrastructure security a non-trivial problem.

Smart contract application testing framework

2.1 Framework

2.1.1 Current standards

None of the commonly used frameworks are taking security of a whole distributed application in account. There is a lot of standards regarding security testing of a "classical" on-premise infrastructure e.g. PTES (Penetration Testing Execution Standard) [31] or OSSTMM (Open Source Security Testing Methodology Manual) [36]. Infrastructure testing uses both automated and manual penetration testing, information gathering, social engineering and red team testing.

Web applications also have their testing standards – e.g. OWASP Web Application Penetration Testing [37] (a part of OWASP Testing Guide v4 [38]). Companies can also have their own framework of testing – the author has used EY web application testing checklist during his work in EY. Web application testing mainly contains penetration testing and source code audit.

Smart contracts do not have exactly a framework or methodology, but they do have thorough descriptions of possible attacks or misuses and how to avoid them during the development phase. As far as testing is concerned one could find security tools for visualization, static and dynamic analysis, test coverage or linters [39]. Smart contract security testing is done mostly with source code auditing – either manually or with the help of automated tools. Smart contracts can also be tested on a test blockchain network in order to save the cost of execution fees. This is also done in order to adequately test both security and functionality, since updating already released smart contracts is not straightforward and the damage may already have been done by the time the creators have found a bug in their smart contract.

2.1.2 Proposed framework

Smart contract application can consist from some (or all) of components mentioned in the theoretical section – mobile application, web application, cloud or physical infrastructure, blockchain as a foundation and smart contracts on top of it. In order to have a safe application as many as possible of the following tests and procedures should be done:

- Blockchain
 - Architecture review
 - * Is the designed architecture overcomplicated ?
 - * Does the architecture store only the necessary data on the blockchain ?
 - Network Penetration Testing
 - * Is the chosen network robust/popular enough ?
 - Static and Dynamic Application Testing including testing wallets, databases, GUI and application logic
 - Integrity Testing
- Smart contracts
 - Code review – manual and automatic – in order to check for vulnerabilities specific for smart contracts
 - * A call to the unknown (1.2.5.1)
 - * Exception disorder (1.2.5.2)
 - * Gasless send (1.2.5.3)
 - * Re-entrancy (1.2.5.4)
 - * Keeping secrets (1.2.5.5)
 - * Immutable bugs (1.2.5.6)
 - * Ether lost in transfer (1.2.5.7)
 - * Transaction ordering dependence (1.2.5.8)
 - * Timestamp dependancy (1.2.5.9)
 - * Mishandled exceptions (1.2.5.10)
 - * An inability to update smart contract code quickly or regularly
 - implying exceptionally thorough testing is required
 - Security check of all the connecting components
- Infrastructure
 - Penetration tests ³

³Heavily depends on the tested infrastructure – whether the tests should be white,gray or black box, whether the infrastructure is virtualized or on "bare metal", whether the infrastructure is behind a load-balancer etc.

- * Pre-engagement actions
- * Reconnaissance
- * Threat Modeling and vulnerability identification
- * Exploitation
- * Post-exploitation
- * Reporting
- * Applying changes and re-testing
- Red team testing
- Configuration review
- Web application
 - Penetration testing
 - * Injection (1.3.1)
 - * Broken authentication (1.3.2)
 - * Sensitive data exposure (1.3.3)
 - * XML external entities (1.3.4)
 - * Broken access control (1.3.5)
 - * Security misconfiguration (1.3.6)
 - * Cross-site scripting (1.3.7)
 - * Insecure deserialization (1.3.8)
 - * Using components with known vulnerabilities (1.3.9)
 - * Insufficient logging and monitoring (1.3.10)
 - Code review
- Mobile application
 - Code review
 - * Improper platform usage (1.4.1)
 - * Client code quality (1.4.7)
 - Penetration testing
 - * Insufficient cryptography (1.4.5)
 - * Code tampering (1.4.8)
 - * Reverse engineering (1.4.9)
 - * Extraneous functionality (1.4.10)
 - * Insecure authorization (1.4.6)
 - * Insecure authentication (1.4.4)
 - * Insecure data storage (1.4.2)

2. SMART CONTRACT APPLICATION TESTING FRAMEWORK

Each of the components requires specific tools – here are the most popular and trusted tools:

- Blockchain
 - Kovan Etherscan – for Ethereum applications
 - BlockCypher – for Bitcoin applications
- Smart contracts
 - Truffle – tool allowing to write automatic tests
 - Populus – tool allowing to write tests in Python
 - Oyente – automatic static analysis tool
 - Mithril – automatic static analysis tool
- Infrastructure
 - Nmap – universal tool for port scanning
 - Metasploit – exploiting tool
 - Nikto – web server analysis tool
 - Nessus – automated scanning tool (paid)
 - OpenVAS – automated scanning tool
 - Wireshark – network analyzer
- Web applications
 - Zed Attack Proxy
 - SQLMap – automatic SQL injection tool
 - Burp Suite
- Mobile applications
 - Quick Android Review Kit – source code analysis tools
 - Drozer – security and attack framework
 - MITM proxy

This many tests on the VETRI application would however take an immense amount of time and thus the actual testing will be limited only to smart contracts and their implementation in the mobile application.

2.1.3 Framework checklist table

This framework is mainly created for developers of applications using smart contracts – specifically their web and mobile applications – who are not versed enough in cyber security and need a checklist which can guide them through main security issues that can be present in the developed application. This checklist also summarizes which test have been carried out on the VETRI application (column *Tested*).

Component	Vulnerability	Detection method	Detection tool	Tested
Blockchain	Blockchain application integration	Architecture review		x
	Network/transactions	Integrity testing		x
Smart contracts	A call to the unknown	Pentest/Code review	Oyente/Mithril	✓
	Exception disorder	Pentest/Code review	Oyente/Mithril	✓
	Gasless send	Pentest/Code review	Oyente/Mithril	✓
	Re-entrancy	Pentest/Code review	Oyente/Mithril	✓
	Keeping secrets	Pentest/Code review	Oyente/Mithril	✓
	Immutable bugs	Pentest/Code review	Oyente/Mithril	✓
	Ether lost in transfer	Pentest/Code review	Oyente/Mithril	✓
	Transaction ordering dependence	Pentest/Code review	Oyente/Mithril	✓
	Timestamp dependency	Pentest/Code review	Oyente/Mithril	✓
	Mishandled exceptions	Pentest/Code review	Oyente/Mithril	✓
Infrastructure	Inability to update	Pentest/Code review	Oyente/Mithril	✓
	Out-of-date software version	Pentest/Automatic scan	Nessus/OpenVAS	✓
	Default configuration	Pentest/Automatic scan	Nessus/OpenVAS	✓
	Inappropriate encryption	Pentest/Automatic scan	Nessus/OpenVAS	✓
	Self-signed or expired certificates	Pentest/Automatic scan	Nessus/OpenVAS	✓
	Unnecessary open services	Pentest/Automatic scan	Nessus/OpenVAS	✓
	Lack of network segmentation	Infrastructure review	-	x
	DoS/DDoS vulnerabilities	Volumetric testing	-	x

Table 2.1: Framework checklist table, part 1

Component	Vulnerability	Detection method	Detection tool	Tested
Web applications	Injection	Pentest/Code review	Burp Suite Pro	✗
	Broken authentication	Pentest/Code review	Burp Suite Pro	✗
	Sensitive data exposure	Pentest/Code review	Burp Suite Pro	✗
	XML external entities	Pentest/Code review	Burp Suite Pro	✗
	Broken access control	Pentest/Code review	Burp Suite Pro	✗
	Security misconfiguration	Pentest/Code review	Burp Suite Pro	✗
	Cross-site scripting	Pentest/Code review	Burp Suite Pro	✗
	Insecure deserialization	Pentest/Code review	Burp Suite Pro	✗
	Using component with vulnerabilities	Pentest/Code review	Burp Suite Pro	✗
	Insufficient logging and monitoring	Pentest/Code review	Burp Suite Pro	✗
	Improper platform usage	Pentest/Code review	ZAP proxy/Drozer	✓
	Client code quality	Code review	-	✗
	Mobile applications	Insufficient cryptography	Pentest/Code review	ZAP proxy/Drozer
Code tampering		Reverse engineering	MARA framework	✗
Reverse engineering		Reverse engineering	MARA framework	✗
Extraneous functionality		Pentest/Code review	ZAP proxy/Drozer	✓
Insecure authentication		Pentest/Code review	ZAP proxy/Drozer	✓
Insecure authorization		Pentest/Code review	ZAP proxy/Drozer	✓
Insecure data storage		Pentest/Code review	ZAP proxy/Drozer	✓

Table 2.2: Framework checklist table, part 2

Analysis of the VETRI application

3.1 VETRI application overview

VETRI is a digital identity platform from a company Procivis in partnership with CreativeDock. It enables users to "take privacy into their own hands" by allowing them to store their personal data securely and allowing them to trade data in a controlled fashion where they are rewarded for their data instead of middlemen. The currency used in the VETRI ecosystem is called VLD token – based on ERC20 Ethereum token standard [40]. The data consumers (companies or individuals seeking to buy data) can either buy the tokens directly from VETRI Foundation or on cryptocurrency exchanges ⁴. Data owners (VETRI users willing to share their data) can receive VLD tokens by participating in surveys. Afterwards the data owners can exchange the accumulated VLD tokens for rewards or gift cards – users well versed in cryptocurrencies could sell the VLD tokens themselves on aforementioned exchanges.

The difference between most of the current online services – e.g. Facebook or Google – and VETRI is that both of them are free for the users but the first two make money by either selling user data or with ads. VETRI, on the other hand, allows users to manage their own personal data and decide who they want to sell them to – and the reward for doing so goes back to them. Another important thing to mention is that the user's personal data are fully decentralized – the data are stored solely in their device until they decide to sell them for a price they deem reasonable. The data user's data are not currently decentralized – they have to currently access VETRI via a web application. This is however about to change in the future.

VETRI consists of a mobile wallet (VETRI wallet – screenshots shown in appendix chapter A) and a web application (VETRI marketplace). The web

⁴VLD tokens are registered on Bitfinex and Ethfinex at the time of writing

3. ANALYSIS OF THE VETRI APPLICATION

application will be used by both users and data consumers. The architecture from user and data consumer point of view is shown in figure 3.1. This architecture, present in the VETRI whitepaper [41], has been later modified for both technological design and testing. The main technologies that enable VETRI application are JavaScript, TypeScript, React Native, React, Node.js, GraphQL, Postgres, Kubernetes, Docker, Terraform, blockchain, Ethereum smart contracts and web3.js.

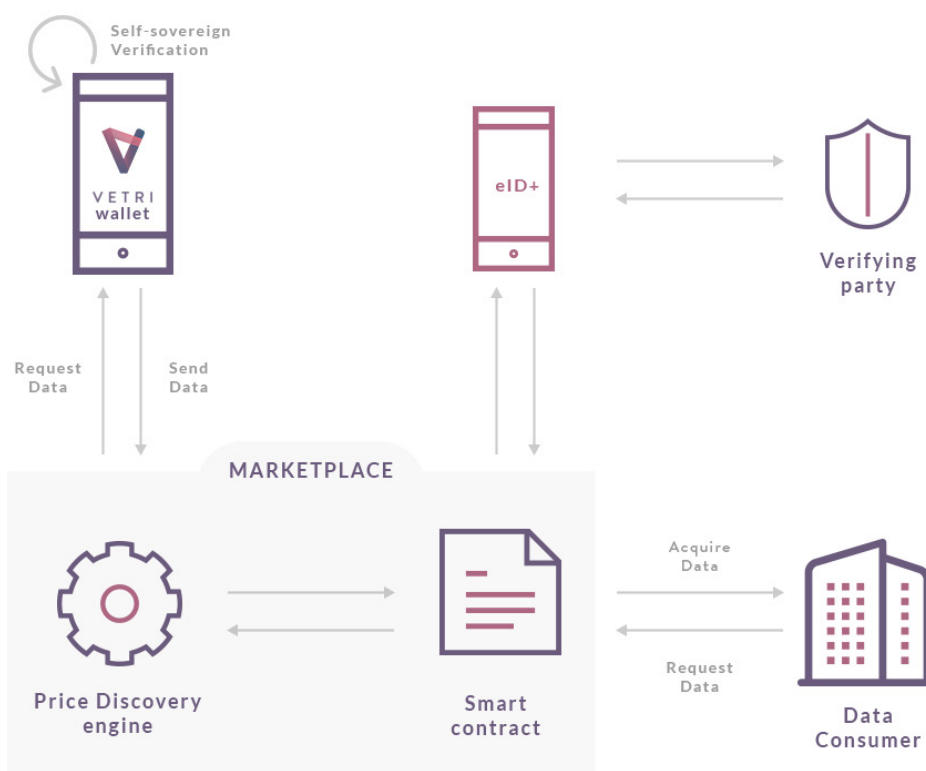


Figure 3.1: VETRI whitepaper architecture

The VETRI ecosystem is planned to have more usecases in the future, but the first usecase that is being worked on is a survey enriched with media data. Another important component is eID+ [42]. It is a product for identity management also made by ProCivis and it shares technological foundation and best practice with VETRI – since it’s also an application for digital identity and personal data management and both of them are blockchain-based. The eID+ application is used by the Swiss Canton of Schaffhausen. It also has open APIs, meaning that third-party companies can create applications using eID+. Main usecases for the eID+ application are:

- e-Data – A marketplace for personal data

- e-Voting – A fully digital and blockchain-secured e-Voting application
- e-Document – Importing, exporting and managing documents with eID+
- e-Signature – Signing digital documents with eID+
- e-Authentication – eID+ is the universal authentication tool
- e-KYC – eID+ simplifies KYC⁵ processes (coming soon)
- e-License – eID+ allows users to store and receive licenses and permits digitally (coming soon)
- e-Company – eID+ simplifies the founding process of companies (coming soon)
- e-Health – Direct access to e-Health platforms (coming soon)

3.2 VETRI blockchain architecture

Information about the current VETRI blockchain architecture design has been taken from the documentation received from the application developers and simplified.

The blockchain architecture is split into three main parts: main-net/test-net, bridge and side (private) chain.

3.2.1 Private chain

The main part of the VETRI application as far as blockchain is concerned is the VETRI private chain. It is a private blockchain run by VETRI and it is based on proof-of-authority (see section 1.1.2.3 for more information). Private chain contains deployed data exchange requests, usable VLD tokens and also deployed identity contracts – a list of verified wallet addresses maintained by verifier service.

The main reasons why the chain is private are decreased fees and ability to have the whole VETRI ecosystem in a secure environment. By the time of writing the VETRI chain is tested on 4 nodes – AWS, Google Cloud, Azure and a server owned by CreativeDock. The usage of the private chain can be seen in figure 3.2.

⁵Know your customer

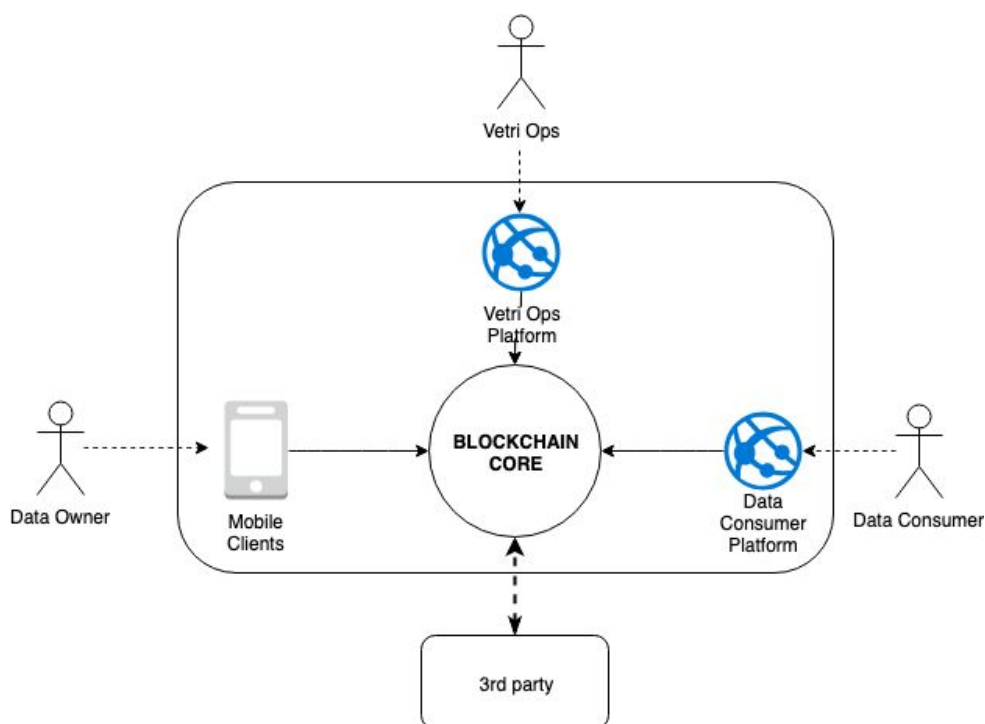


Figure 3.2: VETRI MVP/MUP system architecture

3.2.2 Bridge

Bridge is the connecting point between main-net and private chain. It is responsible for blocking VLD tokens on the Ethereum main-net and releasing them into the private chain.

At the time of writing the chosen bridge technology is *token-bridge* [43] – an interoperability solution between Ethereum networks for ERC-20-to-ERC20 token bridging.

3.2.3 Public chains

Both main-net and test-net (both on Ethereum networks) have been used during the development phase of the VETRI application. This separation allows the developers to create the smart contracts in a safe environment without any substantial worries about stolen resources.

3.2.3.1 Test-net

Test-net was mainly being used during the minimum viable product phase, since not all of the vulnerabilities that could occur in the smart contracts can

be found out with a static (source code) analysis.

3.2.3.2 Main-net

Main-net – Ethereum network – is used almost exclusively for token trading in the finished product since the essential processes are happening on the private chain maintained by authoritative nodes.

3.3 Blockchain use-cases

In this section the most prominent use cases in the VETRI ecosystem that use blockchain will be described and explained.

3.3.1 Data consumer wants to allocate VLD tokens for a data exchange request

The main way of receiving data from data owners is called DER (data exchange request). Its content can be for example a survey to be filled by the data owners or gathering of "data points" via data plugins the user might decide to import because he sees an interesting offer from a company. This use case assumes that a data exchange request has been deployed and that the data consumer has enough tokens in his VETRI side chain.

Data consumer has to first transfer tokens to a smart contract in data consumer interface (this happens automatically if he uses the web application /data consumer interface). He then starts the data exchange request.

The whole process of a data consumer creating a DER – specifically a survey – is detailed in figure 3.3. The technical process of survey creation starts by generating a JSON definition on the backend, sending it to DER of a smart contract and into the backend. The mobile application will fetch the survey JSON definition and additional metadata directly from the smart contract or, in a cached form, from the VETRI backend. After the user fills the survey a JSON file with responses and data points from data plugins that were requested as part of the data exchange request will be sent to data pickup service by the mobile application.

3. ANALYSIS OF THE VETRI APPLICATION

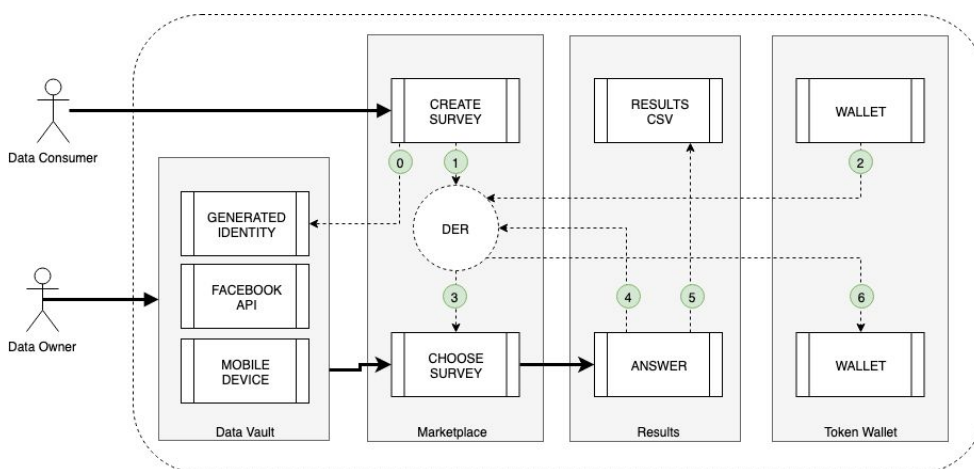


Figure 3.3: Full survey usecase

3.3.2 Identity verification

Since the true identity of individual users is paramount to the application a great care has to be taken in order to verify whether the user is indeed the person he is claiming to be. However, the data owner also needs to remain anonymous. When the VETRI Foundation (or any other trusted identity provider) verifies the user this information is stored on the blockchain and the user can subsequently sign DER (data exchange request) smart contracts – the smart contracts wouldn't allow the user to sign them otherwise.

3.3.3 Token transfer

When an user uses the VETRI application he receives 'internal' tokens by participating in surveys and other data exchange requests. These tokens can be either exchanged for VLD tokens on Ethereum main-net (this exchange is done via the bridge – section 3.2.2) or they can be exchanged for various gift cards (this exchange does not require any action on bridge or main-net). This allows both technically adept and regular users to use the VETRI application and redeem their rewards however they see fit.

3.3.4 Data consumer wants to top up the VETRI account

This use case assumes that the data consumer has an account in data consumer interface and his VLD token wallet has been created.

When data consumers want to top up their VETRI accounts, they have to buy VLD tokens on an exchange ⁶ or from the VETRI Foundation or any of

⁶VLD tokens are registered on Bitfinex and Ethfinex at the time of writing

the current VLD token holders. Data consumer then instructs data consumer interface to transfer tokens to VETRI private chain. Data consumer interface moves the VLD tokens from user's wallet to VETRI bridge – this is done via a bridge contract. The amount of tokens is decreased by bridged amount.

Once the data consumer has the tokens on VETRI private chain he can top up the data exchange request via the data consumer interface.

3.3.5 Data owner gets a reward when he confirms (accepts) the Data contract

This use case assumes that data consumer has deployed a data contract, allocated tokens in said contract and enabled it.

Data owner has to accept a data exchange request currently via the wallet application. The wallet submits a transaction accepting the data exchange request – the transaction contains a footprint hash of the user's data. When the transaction has been accepted the wallet sends the data to the data pickup service alongside with the wallet address of the data owner. After the data pickup service checks whether the data exchange request has been accepted it responds that everything is correct and tells the data exchange request to release VLD token reward for the data owner.

3.4 Application analysis

3.4.1 Smart contract analysis using automated tools

The automated static smart contract code analysis had been done with the help of tool Oyente [7]. In order to run the tool a Docker image *oyente* by user *luongnguyen* [44] was downloaded and run with the following command:

```
1 docker pull luongnguyen/oyente && docker run -i -t luongnguyen/oyente
```

However its version of components and used libraries wasn't compatible with the version of Solidity used by the smart contract developers. A docker image with the same name but from an user *tobykendall* [45] was used successfully. The second Docker image was downloaded with following command:

```
1 docker pull tobykendall/oyente && docker run -i -t tobykendall/oyente
```

The automated tests were run for the following smart contract files from the VETRI code repository *blockchain-smart-contracts*:

- Identity.sol
- ERC20Token.sol
- DataExchangeRequest.sol

3. ANALYSIS OF THE VETRI APPLICATION

- Migrations.sol

The Oyente tool was searching for the following vulnerabilities:

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug 2
- Callstack Depth Attack Vulnerability
- Transaction-Ordering Dependence (section 1.2.5.8)
- Timestamp Dependency (section 1.2.5.9)
- Re-Entrancy Vulnerability (section 1.2.5.4)

The tool tested the aforementioned smart contract files and also the smart contract files they were using – e.g. SafeMath.sol⁷ from OpenZeppelin library in the ERC20Token.sol smart contract file.

The test outputs are in appendix chapter B.

3.4.1.1 Output analysis

The tool hasn't found any major vulnerabilities, except an instance of integer overflow in DataExchangeRequest.sol on line 188:

```
1 ...
2 function getFields() public view returns (string memory) {
3   return fields; % <-- this line
4 }
5 ...
```

The integer overflow would occur if variable *fields* equals 1.

3.4.2 Code testing for libraries shared between the mobile and web applications

The next step in testing were TypeScript files⁸. Those files are used in both mobile and web applications for further interaction between the smart contracts and the applications.

The testing was done by SonarLint [46] and SonarQube [2] applications.

⁷Library with unsigned math operations containing safety checks and reverting on error

⁸TypeScript is an open-source programming language developed by Microsoft and it is a superset of JavaScript programming language

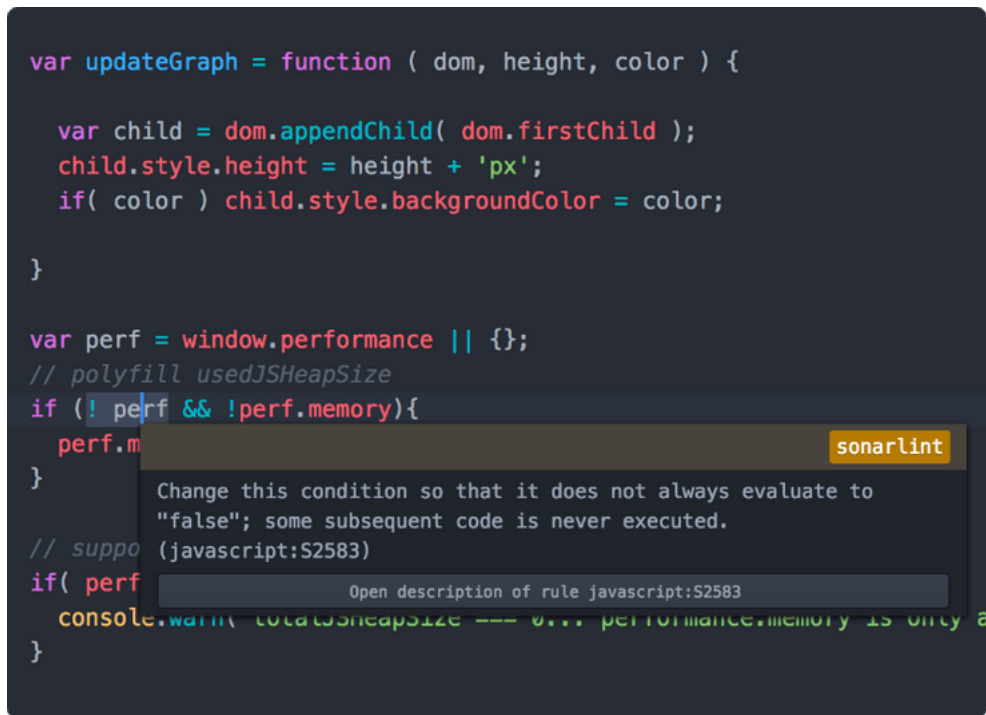


Figure 3.4: SonarLint bug detection illustrative screenshot [1]

3.4.2.1 SonarLint

SonarLint is a linter add-on that can be added to following IDEs : Eclipse, IntelliJ IDEA, Visual Studio, VS Code and Atom. For these tests the linter has been used with the Atom open-source IDE. Its features are [1] :

- Bug detection
- Instant feedback
- Fix recommendations
- Coding best practice documentation
- List of already existing issues

The linter was installed via the Atom IDE package manager and consulted for each individual file.

3.4.2.2 SonarQube

SonarQube is a CI (Continuous Integrity) tool that can be downloaded for all the main platforms or downloaded as a Docker image. The application

3. ANALYSIS OF THE VETRI APPLICATION

has multiple pricing options set up – the Community version was used in this instance. For these tests the application has been used as a Docker image. Its features are [2]:

- Continuous inspection
 - Overall health
 - Code quality enforcement
 - Pull request analysis
 - Project history visualization
- Issue detection
 - Bug detection
 - Difficultly maintainable code detection
 - Security vulnerability detection
 - Custom rule support
 - Analysis of all execution parts
- Support for multiple programming languages
 - C/C++
 - Python
 - Typescript/JavaScript
 - PHP
 - Swift
 - Ruby
 - And others
- DevOps integration
 - Build systems – Maven, MSBuild, Makefile, etc.
 - CI Engines – Bamboo, Jenkins, Azure DevOps, etc.
 - Pass/fail notification
 - Web API
 - High availability
- Centralizations

SonarQube has been run from the file repository directory with the following command:

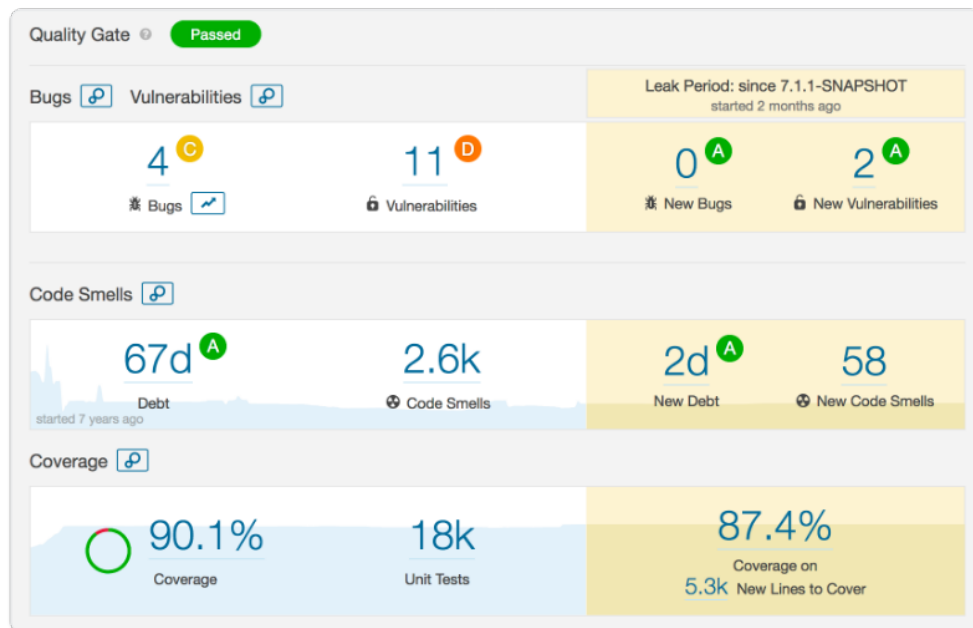


Figure 3.5: Illustrative SonarQube interface screenshot [2]

```

1 sonar-scanner \
2 -Dsonar.projectKey=Typescript \
3 -Dsonar.sources=. \
4 -Dsonar.host.url=http://localhost:9000 \
5 -Dsonar.login=SECRET_KEY

```

3.4.2.3 Test results

Neither of the used tools have found any vulnerabilities in the tested source code files. SonarQube output along with result screenshots can be found in appendix chapter C.

3.4.3 Token bridge test

The next test subject is the bridge that connects the private VETRI chain and the Ethereum chain. The used solution is called *token-bridge* [43]. SonarQube was again used as a testing tool.

No vulnerabilities have been found, although the tools identified that 15,2% of the code was duplicate – 10 blocks, to be precise. This can be seen in figure 3.6.

3. ANALYSIS OF THE VETRI APPLICATION

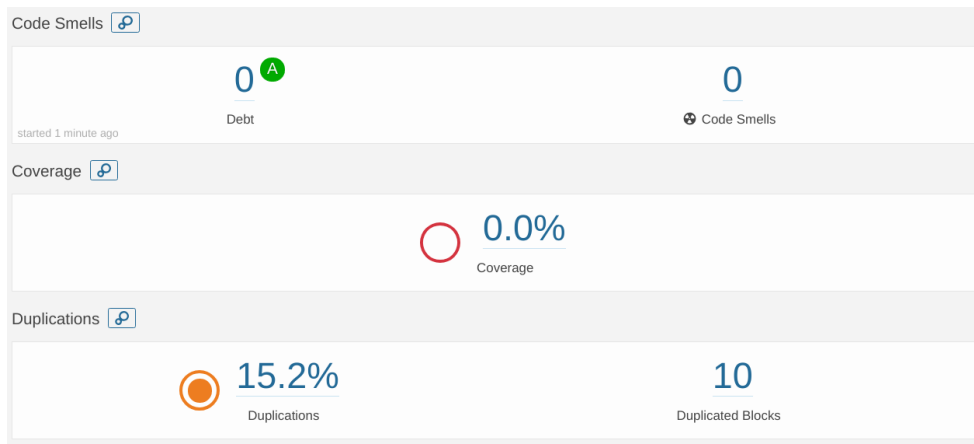


Figure 3.6: Bridge test result screenshot

3.4.4 Analysis of the Android mobile application using automated tools

The mobile application (Android version, specifically ⁹) was the next component to be tested. Mobile Security Framework (MobSF) [47] has been chosen for this task, since it is both open-source and available as a Docker image. It was downloaded and run with following commands:

```
1 docker pull opensecurity/mobile-security-framework-mobsf
2 docker run -it -p 8000:8000 opensecurity/mobile-security-
  framework-mobsf:latest
```

The author has been given tester access to the VETRI application on Google Play store and was subsequently able to retrieve the VETRI Android package file (APK). This was done with Android Debug Bridge (ADB) with following commands ¹⁰:

```
1 adb shell pm list packages
2 adb shell pm path io.vetri.wallet.dev
3 adb pull /data/app/io.vetri.wallet.dev-ugs_saNK2a7N9HW8UWZUiQ==/
  base.apk
```

This file has been uploaded to the Mobile Security Framework application running on the testing laptop. The MobSF application is a disassembler and a static analysis tool and it provides information about the APK file, such as:

- Code nature

⁹Most of the code is shared between Android and iOS application and the iOS application testing – dynamic analysis mainly – requires Apple-specific test environment that was not available to the author

¹⁰With the testing phone connected via USB and with ADB allowed and appropriately configured

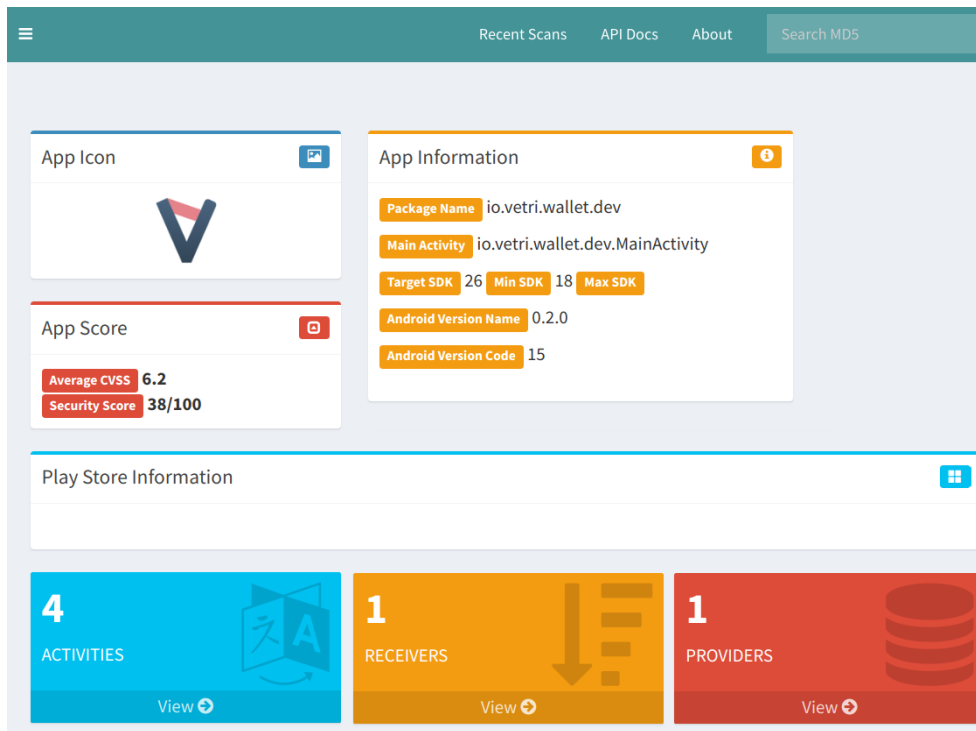


Figure 3.7: MobSF static analysis result overview

- Signer certificate details
- Required permissions
- Android API details
- Components
- Security analysis

The result overview can be seen in figure 3.7.

The security analysis of the APK file revealed a few vulnerabilities, resulting in average Common Vulnerability Scoring System (CVSS) score of 6.2¹¹ and security score of 38/100. However it is important to mention that all of the found vulnerabilities were found in external dependencies and libraries. It is also important to mention that external libraries of mobile application are not in the testing scope and thus no further testing regarding the found vulnerabilities has been done – this implies that the received result should not be taken as definitive and necessarily correct.

¹¹CVSS scores range from 0.0 – informative – up to 10.0 – extremely critical

The found vulnerabilities were exclusively present in the used external files or libraries (full list of the vulnerable files can be found in the attached CD). These vulnerabilities can be fixed by either upgrading the used components to their newest versions (either once or continuously) or using other similar components with similar functionalities. The second solution might not be very viable since most of the used components are linked via their function names and changing these could result in many errors.

3.4.5 OpenVAS automated security scan of deployment servers

The VETRI development team already performed an internal OpenVAS vulnerability scan which was provided to the author of the thesis. These are the vulnerabilities that have been found – both of them were assigned medium severity.

3.4.5.1 Vulnerable Cipher Suites for HTTPS

Insecure cipher suites have been found, specifically *TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA* and *TLS_RSA_WITH_3DES_EDE_CBC_SHA*. Both of those ciphers are vulnerable to SWEET32 attack (CVE-2016-2183).

The test also revealed that TLS versions 1.0 and 1.1 are supported – these versions are not secure anymore and thus should not be used.

The remediation is to disable the vulnerable cipher suites and TLS versions.

3.4.5.2 Untrusted Certificate Authorities

A self-signed certificate has been found – the remediation is to replace said certificate with one signed by valid certificate authority.

See 1.5.4 for further information.

3.5 Vulnerabilities and mitigations overview

3.5.1 Vulnerable dependencies in mobile application

The vulnerable dependencies (see 3.4.4 for details) could generally be solved by either upgrading the used versions or choosing other dependencies. However a risk analysis has to be done for either of these mitigations – since both of them could theoretically break the developed application (e.g. different function names, different return values, different data types) and hinder the whole development.

3.5.2 OpenVAS security scan of deployment servers

An OpenVAS security scan has already been run by the VETRI development team. These are the vulnerabilities that have been found – both of them were assigned medium severity. Please see section 3.4.5 for details.

- Vulnerable Cipher Suites for HTTPS - Remove outdated cipher suites and insecure TLS versions
- Untrusted Certificate Authorities - Sign the certificates with a trusted certificate authority

3.6 Application analysis limitations and follow-up

3.6.1 Testing limitations

The limitations of testing were that almost no dynamic or manual analysis has been done since they were out-of-scope and would require strong knowledge of Solidity, TypeScript and other programming languages.

Furthermore, penetration testing is not the only method of security assessment – security audit, architecture review, bug bounty or hiring test users can also be done.

This thesis is also not concerned with security of blockchain networks as such – e.g. 51% attack or any kind of "conspiracy" of the authoritative nodes on the sidechain.

And the last important thing to keep in mind when designing any kind of decentralized application is that the **all** of the used components have to be secure and tested. For example a vulnerable web application allowing a malicious user to carry out illegal actions would nullify all the effort that has been made in order to secure the other components. This is also true for any application or system – especially one with broad userbase, since 90-95% of all successful cyber attacks begin with a phishing e-mail [48] and thus with the human component of the application.

3.6.2 Follow-up

In order to improve the usability of the proposed framework it is recommended to include smart contract (Solidity) rules to popular vulnerability scanners – such as Nessus or OpenVAS. The framework would also be more user-friendly if it was scripted and the script would install the required dependencies and run the tests automatically. Supporting more programming languages would help as well.

Conclusion

The aim of this thesis has been to describe the most common vulnerabilities in all of decentralized application's components, creating a framework that should help developers of such applications that are not as versed in cyber security and also to test specific sections of the VETRI distributed application.

The analysis of the five basic components (blockchain, smart contracts, web application, mobile application and infrastructure) has been performed while using the industry-standard sources (OWASP Top 10 for web and mobile applications), newest research for smart contracts or author's personal experience in infrastructure testing.

The framework has been created in a checklist-style form in order to allow developers to simply check each of the vulnerabilities during their quest for a secure application.

The analysis of the VETRI application has been performed – automatic code scans for smart contracts in Solidity language and libraries shared between the web and mobile applications in TypeScript language. Automatic code scans have also been performed for the entire Android mobile application – finding several vulnerabilities caused by outdated dependencies. The last test was an automated OpenVAS scan of deployment servers that also revealed several minor vulnerabilities – both of which were regarding cipher suites or certificate signature.

Bibliography

- [1] SonarSource. SonarLint Features. 2018. Available from: <https://www.sonarlint.org/features/>
- [2] SonarSource. SonarQube. 2018. Available from: <https://www.sonarqube.org/>
- [3] Haber, S.; Stornetta, S. W. How to time-stamp a digital document. *Journal of Cryptology*, January 1991.
- [4] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [5] Raval, S. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. " O'Reilly Media, Inc.", 2016.
- [6] Atzei, N.; Bartoletti, M.; et al. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, Springer, 2017, pp. 164–186.
- [7] Luu, L.; Chu, D.-H.; et al. Oyente - Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 254–269.
- [8] Kosba, A.; Miller, A.; et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, IEEE, 2016, pp. 839–858.
- [9] Boneh, D.; Naor, M. Timed commitments. In *Annual International Cryptology Conference*, Springer, 2000, pp. 236–254.
- [10] OpenZeppelin. OpenZeppelin. 2018. Available from: <https://openzeppelin.org/>

BIBLIOGRAPHY

- [11] ConsenSys. Ethereum Smart Contract Security Best Practices. 2018. Available from: <https://consensys.github.io/smart-contract-best-practices/>
- [12] Wichers, D. Owasp top-10 2017. *OWASP Foundation, February, 2017.*
- [13] Enumeration, C. C. W. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). 2018. Available from: <https://cwe.mitre.org/data/definitions/89.html>
- [14] Enumeration, C. C. W. CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection'). 2018. Available from: <https://cwe.mitre.org/data/definitions/77.html>
- [15] RFC 4511 - Lightweight Directory Access Protocol. 2006. Available from: <https://tools.ietf.org/html/rfc4511>
- [16] Foundation, O. Testing for LDAP Injection (OTG-INPVAL-006). 2018. Available from: [https://www.owasp.org/index.php/Testing_for_LDAP_Injection_\(OTG-INPVAL-006\)](https://www.owasp.org/index.php/Testing_for_LDAP_Injection_(OTG-INPVAL-006))
- [17] Foundation, O. Top 10-2017 A4-XML External Entities (XXE). 2018. Available from: [https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_\(XXE\)](https://www.owasp.org/index.php/Top_10-2017_A4-XML_External_Entities_(XXE))
- [18] Bromberger, S. DNS as a covert channel within protected networks. *National Electronic Sector Cyber Security Organization (NESCO)(Jan., 2011)*, 2011.
- [19] Sullivan, B. XML Denial of Service Attacks and Defenses. 2009. Available from: <https://msdn.microsoft.com/en-us/magazine/ee335713.aspx>
- [20] for Internet Security, C. CIS Benchmark. 2018. Available from: <https://www.cisecurity.org/cis-benchmarks/>
- [21] Foundation, O. Top 10-2017 A8-Insecure Deserialization. 2018. Available from: https://www.owasp.org/index.php/Top_10-2017_A8-Insecure_Deserialization
- [22] Zorz, Z. The percentage of open source code in proprietary apps is rising. 2018. Available from: <https://www.helpnetsecurity.com/2018/05/22/open-source-code-security-risk/>
- [23] Equifax. Frequently Asked Questions. 2017. Available from: <https://www.equifaxsecurity2017.com/frequently-asked-questions/#general-faqs>

- [24] Goldstein, A. The Equifax Breach: Who's to Blame? 2017. Available from: <https://resources.whitesourcesoftware.com/blog-whitesource/the-equifax-breach-who-s-to-blame>
- [25] NIST. CVE-2017-5638 Detail. 2017. Available from: <https://nvd.nist.gov/vuln/detail/CVE-2017-5638>
- [26] Foundation, O. OWASP Mobile top 10 - 2016. 2016. Available from: https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10
- [27] Green, D. Top 10 Vulnerabilities in Mobile Applications. 2017. Available from: <https://www.whitehatsec.com/blog/top-10-vulnerabilities-in-mobile-applications/>
- [28] IBM. Deprecation: weaker cryptographic algorithms. 2018. Available from: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.pro.doc/q123425_.html
- [29] Qualys. SSL report: google.com. 2019. Available from: <https://www.ssllabs.com/ssltest/analyze.html?d=google.com&s=216.58.194.206&latest>
- [30] LogicMonitor. Cloud Vision 2020: The Future of the Cloud Study. 2018.
- [31] Standard, P. T. E. Main page. 2014. Available from: http://www.pentest-standard.org/index.php/Main_Page
- [32] Point, C. Critical infrastructure and SCADA/ICS cybersecurity vulnerabilities and threats. 2016. Available from: <https://www.checkpoint.com/downloads/products/top-10-cybersecurity-vulnerabilities-threat-for-critical-infrastructure-scada-ics.pdf>
- [33] ConsenSys. Usage share of web browsers. 2019. Available from: https://en.wikipedia.org/wiki/Usage_share_of_web_browsers
- [34] Google. Check if a site's connection is secure. 2019. Available from: <https://support.google.com/chrome/answer/95617?hl=en>
- [35] Corporation, M. How to troubleshoot security error codes on secure websites. 2018. Available from: <https://support.mozilla.org/en-US/kb/error-codes-secure-websites>
- [36] ISECOM. Open Source Security Testing Methodology Manual (OS-STMM). 2017. Available from: <http://www.isecom.org/research/>
- [37] OWASP. Web Application Penetration Testing. 2014. Available from: https://www.owasp.org/index.php/Web_Application_Penetration_Testing

BIBLIOGRAPHY

- [38] OWASP. OWASP Testing Guide v4 Table of Contents. 2016. Available from: https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents
- [39] ConsenSys. Ethereum Smart Contract Security Best Practices - Security Tools. 2018. Available from: https://consensys.github.io/smart-contract-best-practices/security_tools/
- [40] Wiki, T. E. ERC20 Token Standard. 2018. Available from: https://theethereum.wiki/w/index.php/ERC20_Token_Standard
- [41] AG, P. VETRI whitepaper. 2018. Available from: <https://vetri.global/static/WP-VETRI.pdf>
- [42] Procivis. eID+. 2019. Available from: <https://procivis.ch/eid/>
- [43] poanetwork. GitHub - poanetwork/token-bridge. 2018. Available from: <https://github.com/poanetwork/token-bridge>
- [44] luongnguyen. Docker Hub - luongnguyen/oyente. 2018. Available from: <https://hub.docker.com/r/luongnguyen/oyente/>
- [45] tobykendall. Docker Hub - tobykendall/oyente. 2018. Available from: <https://hub.docker.com/r/tobykendall/oyente/>
- [46] SonarSource. SonarLint. 2018. Available from: <https://www.sonarlint.org/>
- [47] MobSF. GitHub - MobSF/Mobile-Security-Framework-MobSF. 2019. Available from: <https://github.com/MobSF/Mobile-Security-Framework-MobSF>
- [48] Cision. New Email Security Report from IRONSCALES Identifies Email Phishing Attack Detection, Mitigation and Remediation as Biggest Challenge for Security Teams. 2017. Available from: <http://www.prweb.com/releases/2017/09/prweb14742215.htm>

VETRI mobile application screen designs



Your personal data storage

Connect services you use online as data sources, and store your data on your device

Figure A.1: Welcome screen



Select data source

Add data

Personal



Data About You



Online Services



Facebook data

add

Device



Device Information Data

add

Figure A.2: Adding data

A. VETRI MOBILE APPLICATION SCREEN DESIGNS

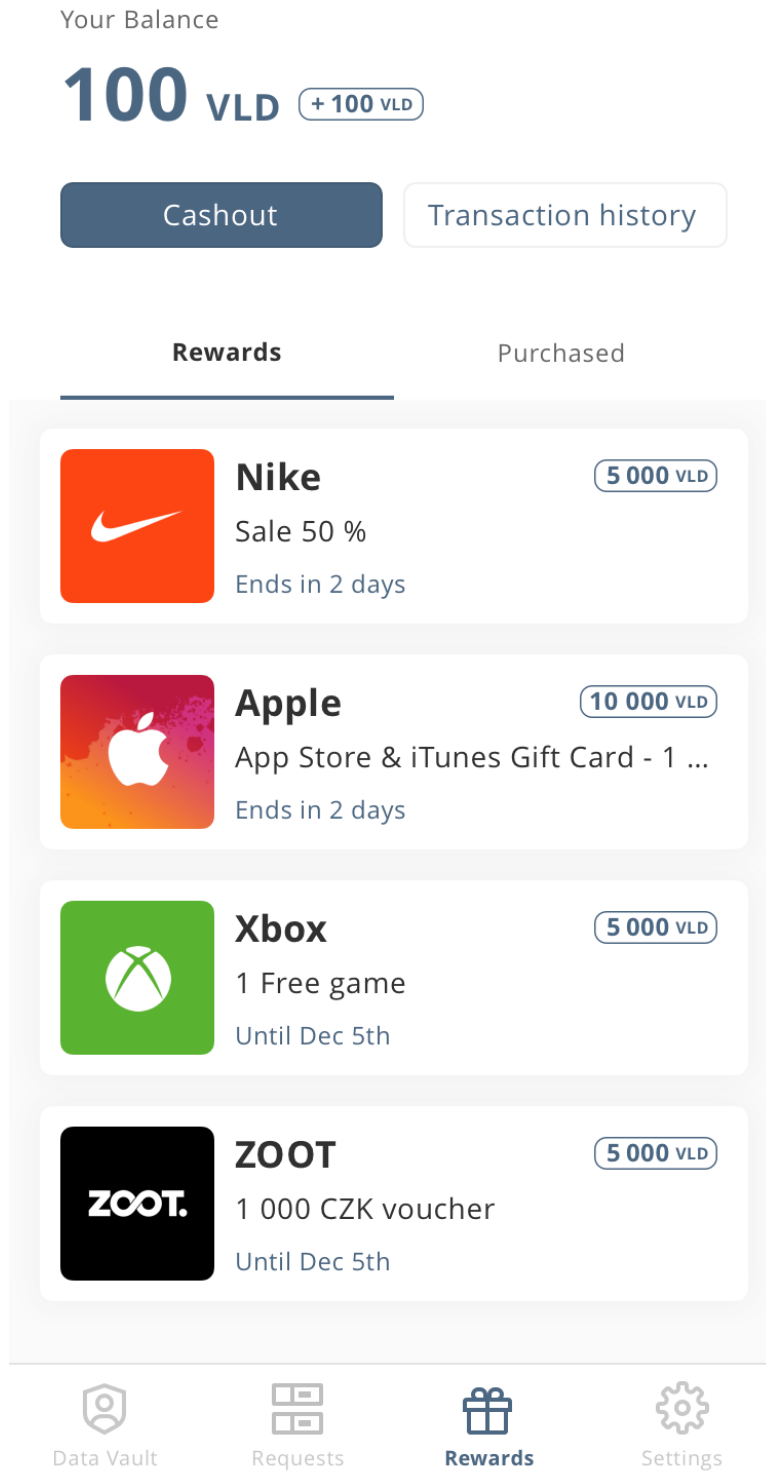


Figure A.3: Token exchange

Data Requests

Available

Archive

Health data for survey 5000 VLD
University of Zurich
Time to complete 45 min


Free time activities 5000 VLD
Trust Square
Time to complete 5 min

Gender + Birthdate 50 VLD
Vetri
Time to complete 5 min

Facebook data ✕
Establish facebook connection and get more requests.


Requests


Data Vault


Wallet


Settings

Figure A.4: Data requests



Try your first data req...

4. In the last 12 months, when you needed care right away, how often did you get care as soon as you thought you needed?

- Never
- Sometimes
- Usually
- Always



5 of 15 answered



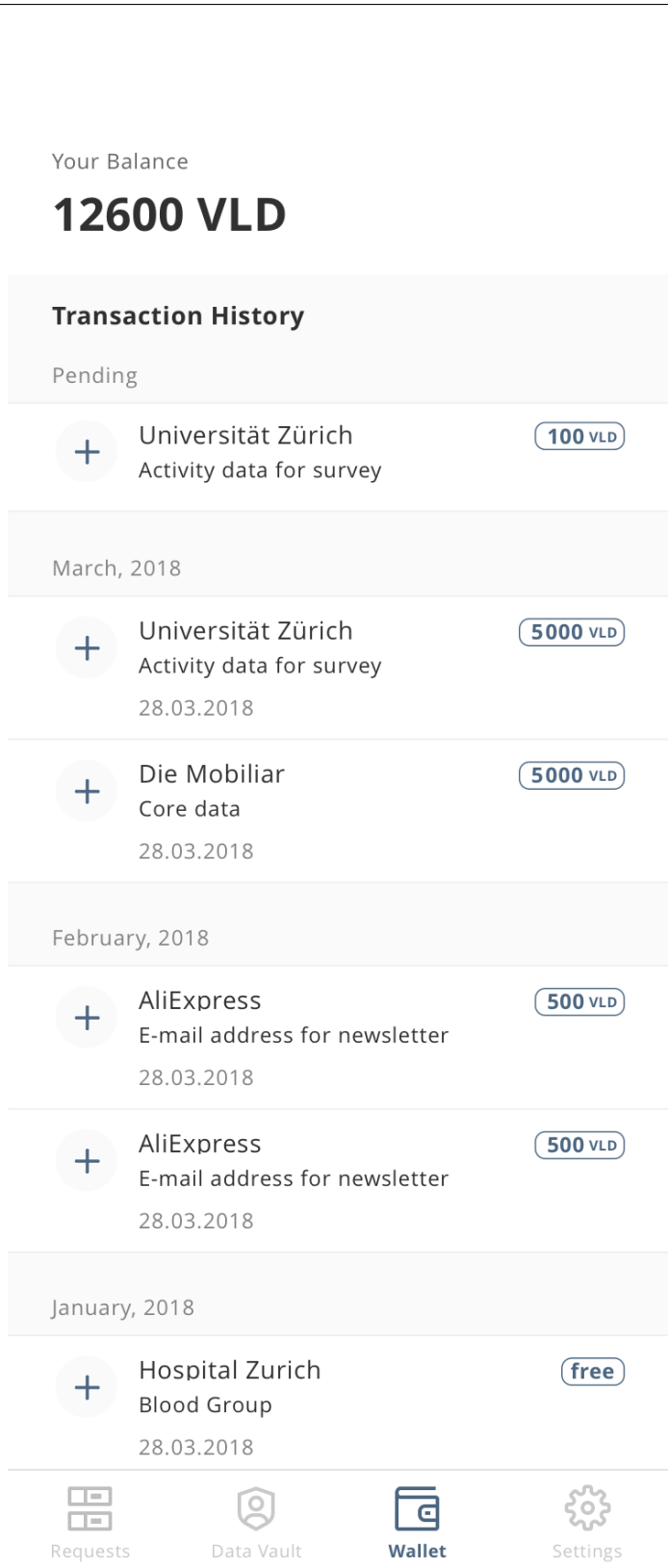


Figure A.6: Balance overview

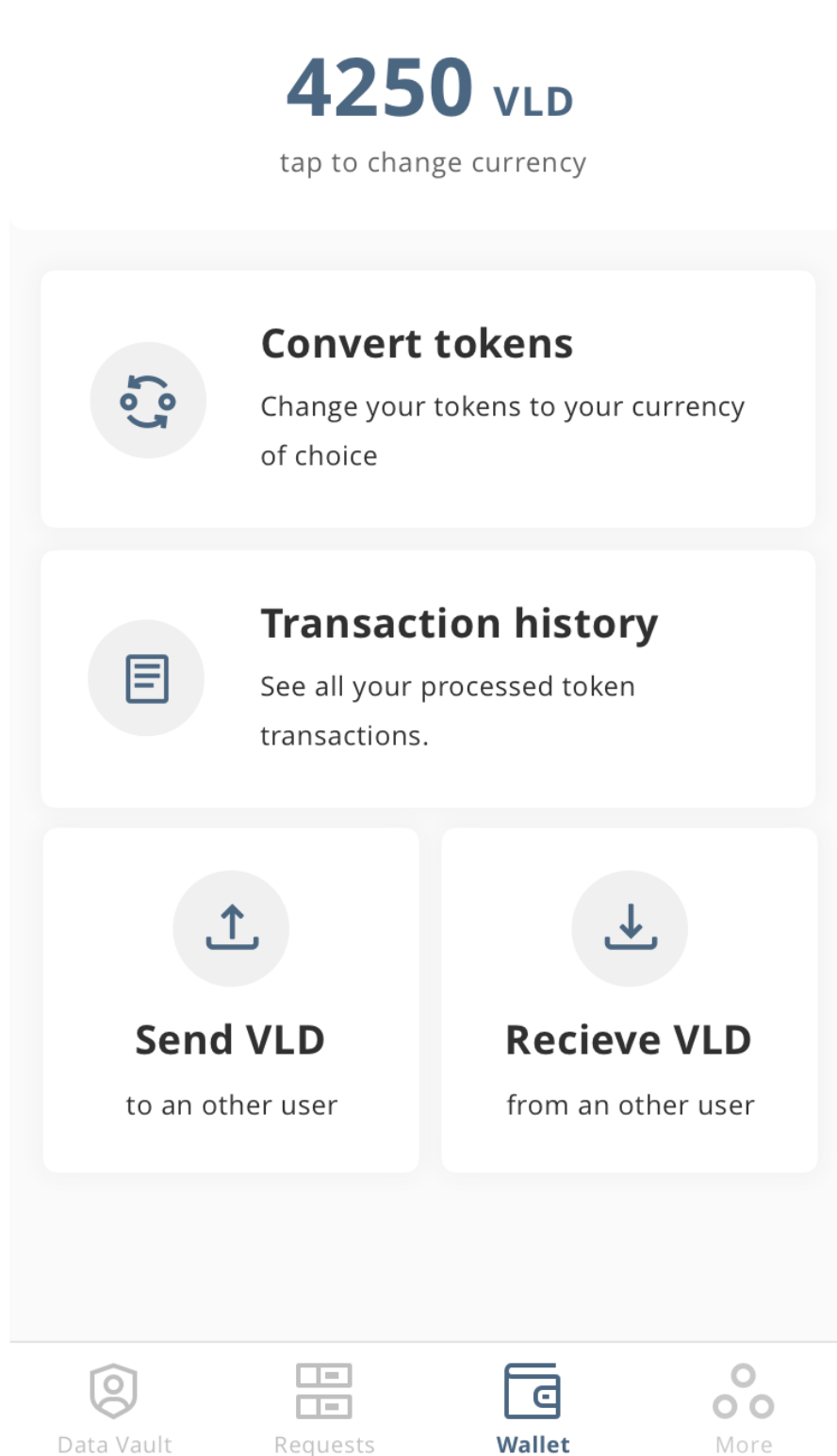


Figure A.7: Token screen

Full Oyente scan outputs

B.1 Identity.sol

```
Identity.sol:Identity:
===== Results =====
  EVM Code Coverage:          44.9%
  Integer Underflow:         False
  Integer Overflow:          False
  Parity Multisig Bug 2:     False
  Callstack Depth Attack Vulnerability: False
  Transaction-Ordering Dependence (TOD): False
  Timestamp Dependency:      False
  Re-Entrancy Vulnerability: False
===== Analysis Completed =====
openzeppelin-solidity/contracts/math/SafeMath.sol:SafeMath:
===== Results =====
  EVM Code Coverage:         100.0%
  Integer Underflow:         False
  Integer Overflow:          False
  Parity Multisig Bug 2:     False
  Callstack Depth Attack Vulnerability: False
  Transaction-Ordering Dependence (TOD): False
  Timestamp Dependency:      False
  Re-Entrancy Vulnerability: False
===== Analysis Completed =====
```

B.2 ERC20Token.sol

```
ERC20Token.sol:ERC20Token:
===== Results =====
  EVM Code Coverage:         85.3%
```

B. FULL OYENTE SCAN OUTPUTS

```
Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:  False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:   False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====
openzeppelin-solidity/contracts/access/Roles.sol:Roles:
===== Results =====
EVM Code Coverage:      100.0%
Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:  False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:   False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====
openzeppelin-solidity/contracts/math/SafeMath.sol:SafeMath:
===== Results =====
EVM Code Coverage:      100.0%
Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:  False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:   False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====
openzeppelin-solidity/contracts/token/ERC20/ERC20.sol:ERC20:
===== Results =====
EVM Code Coverage:      99.9%
Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:  False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:   False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====
openzeppelin-solidity/contracts/token/ERC20/ERC20Mintable.sol:ERC20Mintable:
===== Results =====
EVM Code Coverage:      85.3%
```

```

Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:   False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:    False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====

```

B.3 DataExchangeRequest.sol

```

DataExchangeRequest.sol:DataExchangeRequest:
===== Results =====
EVM Code Coverage:      56.3%
Integer Underflow:      True
Integer Overflow:       False
Parity Multisig Bug 2:   False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:    False
Re-Entrancy Vulnerability: False
DataExchangeRequest.sol:188:9: Warning: Integer Underflow.
    return fields
Integer Underflow occurs if:
    return fields = 1
===== Analysis Completed =====

```

B.4 Migrations.sol

```

Migrations.sol:Migrations:
===== Results =====
EVM Code Coverage:      65.2%
Integer Underflow:      False
Integer Overflow:       False
Parity Multisig Bug 2:   False
Callstack Depth Attack Vulnerability: False
Transaction-Ordering Dependence (TOD): False
Timestamp Dependency:    False
Re-Entrancy Vulnerability: False
===== Analysis Completed =====

```

Full SonarQube scan outputs

```
INFO: Scanner configuration file: /home/archer/Downloads/
\sonar-scanner-3.3.0.1492-linux/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarQube Scanner 3.3.0.1492
INFO: Java 1.8.0_121 Oracle Corporation (64-bit)
INFO: Linux 4.4.0-139-generic amd64
INFO: User cache: /home/archer/.sonar/cache
INFO: SonarQube server 7.6.0
INFO: Default locale: "en_US", source code encoding: "UTF-8"
\ (analysis is platform dependent)
INFO: Load global settings
INFO: Load global settings (done) | time=173ms
INFO: Server id: BF41A1F2-AWlTvcjQyBG9mJtObXbt
INFO: User cache: /home/archer/.sonar/cache
INFO: Load/download plugins
INFO: Load plugins index
INFO: Load plugins index (done) | time=127ms
INFO: Load/download plugins (done) | time=180ms
INFO: Process project properties
INFO: Execute project builders
INFO: Execute project builders (done) | time=15ms
INFO: Project key: Typescript
INFO: Base dir: /home/archer/VETRI/TSanalysis
INFO: Working dir: /home/archer/VETRI/TSanalysis/.scannerwork
INFO: Load project settings
INFO: Load project settings (done) | time=21ms
INFO: Load project repositories
INFO: Load project repositories (done) | time=134ms
INFO: Load quality profiles
INFO: Load quality profiles (done) | time=58ms
```

C. FULL SONARQUBE SCAN OUTPUTS

```
INFO: Load active rules
INFO: Load active rules (done) | time=3008ms
INFO: Load metrics repository
INFO: Load metrics repository (done) | time=85ms
WARN: SCM provider autodetection failed. Please use "sonar.scm.provider"
\to define SCM of your project, or disable the SCM Sensor in the
\project settings.
INFO: Indexing files...
INFO: Project configuration:
INFO: 32 files indexed
INFO: Quality profile for ts: Sonar way
INFO: ----- Run sensors on module Typescript
INFO: Sensor JaCoCo XML Report Importer [jacoco]
INFO: Sensor JaCoCo XML Report Importer [jacoco] (done) | time=9ms
INFO: Sensor JavaXmlSensor [java]
INFO: Sensor JavaXmlSensor [java] (done) | time=3ms
INFO: Sensor HTML [web]
INFO: Sensor HTML [web] (done) | time=41ms
INFO: Sensor SonarTS [typescript]
INFO: No tsconfig.json file found for 8 file(s) (Run in debug mode to
\see all of them). They will be analyzed with a default configuration.
INFO: Analyzing 8 typescript file(s) with the following configuration\
file
DEFAULT_TSCONFIG
INFO: 8 files analyzed out of 8
INFO: Sensor SonarTS [typescript] (done) | time=2669ms
INFO: Sensor Zero Coverage Sensor
INFO: Sensor Zero Coverage Sensor (done) | time=46ms
INFO: ----- Run sensors on project
INFO: No SCM system was detected. You can use the 'sonar.scm.provider'\
property to explicitly specify it.
INFO: 3 files had no CPD blocks
INFO: Calculating CPD for 5 files
INFO: CPD calculation finished
INFO: Analysis report generated in 202ms, dir size=118 KB
INFO: Analysis report compressed in 41ms, zip size=33 KB
INFO: Analysis report uploaded in 79ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://localhost:9000\
dashboard?\id=Typescript
INFO: Note that you will be able to access the updated dashboard
\once the server has processed the submitted analysis report
INFO: More about the report processing at http://localhost:9000\
/api/ce/task?id=AWlT5JixyBG9mJt0bZh3
INFO: Analysis total time: 10.701 s
```

INFO: -----
INFO: EXECUTION SUCCESS
INFO: -----
INFO: Total time: 13.166s
INFO: Final Memory: 15M/256M
INFO: -----

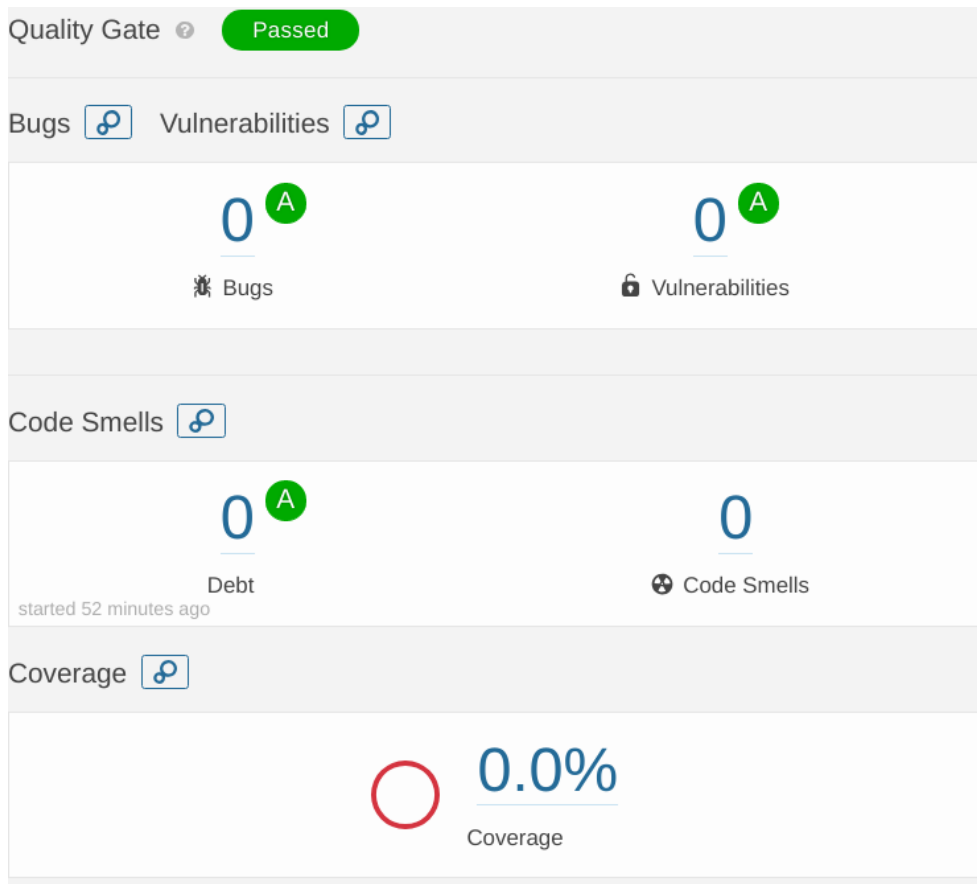


Figure C.1: SonarQube screenshot 1

C. FULL SONARQUBE SCAN OUTPUTS

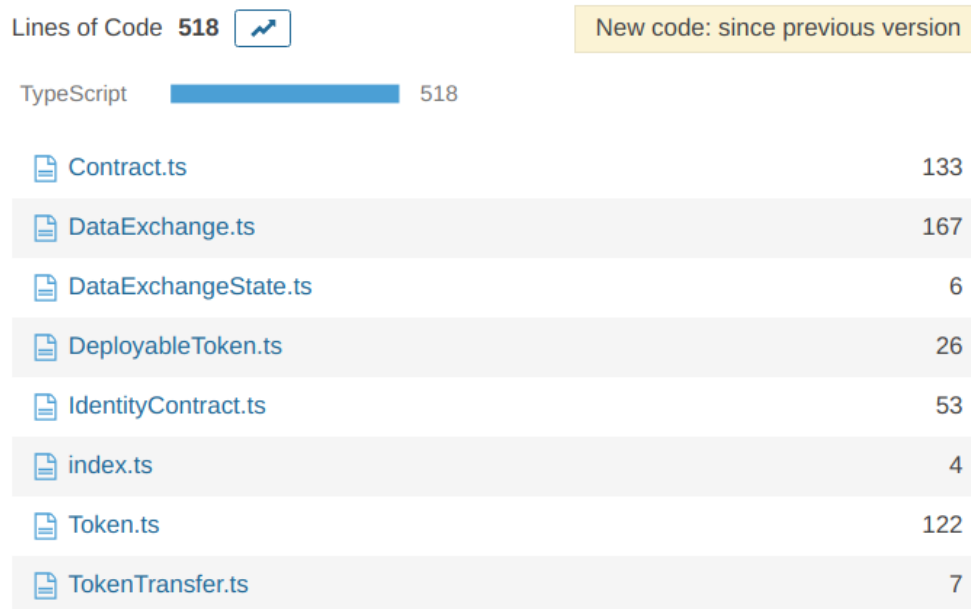


Figure C.2: SonarQube screenshot 2

New Lines	0
Lines of Code	518
Lines	708
Statements	166
Functions	63
Classes	8
Files	8
Directories	0
Comment Lines	98
Comments (%)	15.9%

Figure C.3: SonarQube screenshot 3

Acronyms

P2P	Peer-to-peer
PKI	Public key infrastructure
HTTP	Hypertext transfer protocol
HTTPS	Secure hypertext transfer protocol
WAF	Web application firewall
OS	Operating system
LDAP	Lightweight directory access protocol
TLS	Transport layer security
HSM	Hardware security module
HSTS	HTTP strict transport security
XML	Extensible markup language
NVD	National Vulnerability Database
CVSS	Common Vulnerability Scoring System
DoS	Denial of Service attack
APK	Android Package
UID	User ID
KYC	Know Your Customer
DER	Data Exchange Request
APK	Android Package

Contents of enclosed CD

readme.txt	the file with CD contents description
thesis	the directory of L ^A T _E X source codes of the thesis
├─ mybibliographyfile.bib	BibTeX source file
├─ source.tex	source .tex file of the thesis
text	the thesis text directory
├─ thesis.pdf	the thesis text in PDF format
├─ MobSF.pdf	complete report from MobSF analyzer