Web3 Transaction Simulation White Paper

ecosystem support program

zengo

Version 1.0
February 2025

Web3 Transaction Simulation Paper

Intro

The introduction of Web3's smart contracts to the blockchain environment has opened virtually unlimited opportunities for Dapps and users. Prior to smart contracts, cryptocurrency blockchain functionality was limited to transfer of funds. With smart contracts, anything that can be coded can be done on the blockchain, enabling innovative new verticals such as Decentralized Finance (DeFi) protocols and new asset types (e.g. NFTs).

The flip side of this expressiveness is that while a traditional financial transaction is very simple to comprehend, in a Web3 environment, the users' blockchain transactions are actually remote procedure calls (RPCs) for Smart Contracts, and it is almost impossible to analytically know what would be the result of a general function. This observability gap can, and in fact is, abused by attackers to trick users into signing transactions that are actually harmful for them.

A common and the current de-facto standard solution for this transaction intelligibility gap is Transaction Simulation, commonly abbreviated as Simulation. With Simulation the candidate transaction is evaluated by a node, as if it was sent to the blockchain, but **without actually** sending the transaction it. As a result, users can now see the expected results of their transaction and decide whether or not they want to actually approve it.

However, Simulation solutions have their own limitations, whether theoretical or practical and just like any other solution may suffer from implementation bugs and attackers exploitation of them. In this paper we explore Simulation limitations and suggest solutions.

This paper is organized as follows:

Section 1:
Ethereum Smart Contracts Enabling Web3

Bitcoin, the first blockchain launched in 2009, was primarily designed as a decentralized digital currency and store of value, aiming to serve as an alternative to traditional fiat currencies. Its main purpose is to facilitate peer-to-peer electronic transactions without the need for intermediaries.

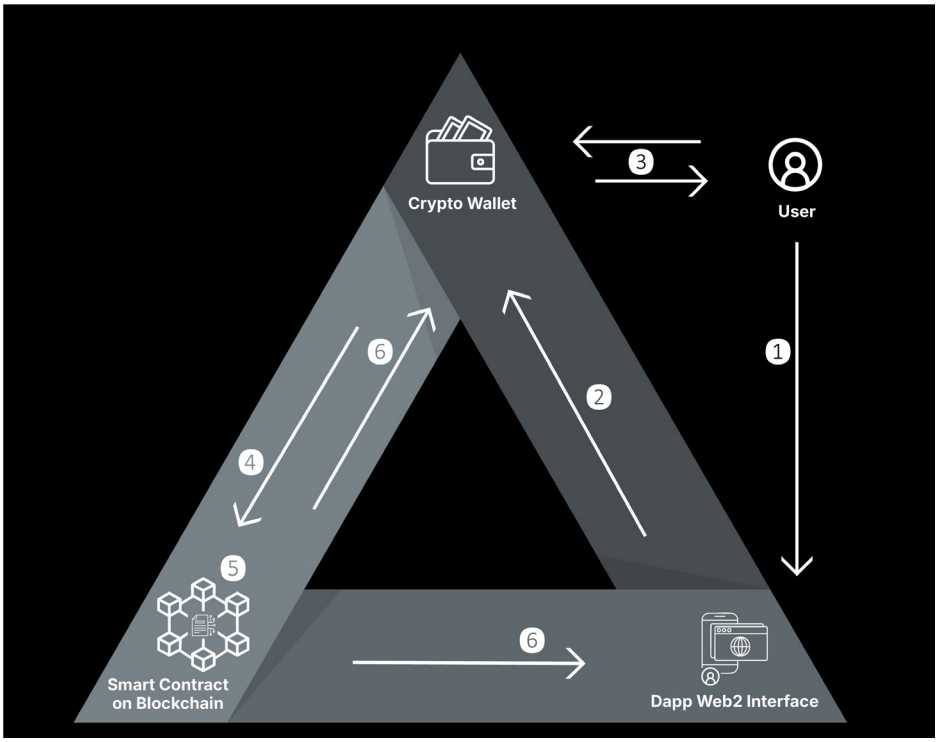
Conversely, Ethereum, launched in 2016, was developed as a versatile platform that supports not only digital currency (Ether) transfers, but also code execution with Smart Contracts. Smart Contracts can serve as the backend computing layer for Web3, the infrastructure being built to decentralize the internet.

Smart Contract developers implement their ideas in Solidity code and deploy it to the Ethereum blockchain. Ethereum users interact with Smart Contract functions via blockchain transactions. In such transactions, the transaction recipient address is the executed Smart Contract and the transaction data includes encoded function name and parameters.

The verbosity of such transactions created a user experience challenge: How can a common user know which functions to call and what are the right parameter values to encode in the transaction?

For some limited yet highly important use cases, such as the ERC-20 tokens, the solution was to create standards and have wallets implement the relevant user experience for them. As a result, when users send a token, the wallet displays a human-readable user interface and encodes the relevant Smart Contract transaction with the relevant function and parameters encoded within it for the user to sign.

However, when developers want to innovate and create new types of apps, standards do not scale. Developers need to create dynamic user frontend interfaces that can serve as intermediaries between the user and backend code deployed as a smart contract. These frontend intermediaries are called Decentralized Applications, and commonly abbreviated as Dapps.



The Web3 Triangle: Wallet, Blockchain, Dapp

When users want to interact with a Web3 Dapp:

1. User interacts with dapp (e.g. UniSwap)
    a. The user browses to the dapp's Web2 interface hosted on Web2
    b. The Dapp presents the users with a user interface
    c. The user interacts with the Dapp's interface
    d. Finally, the Dapp creates a transaction according to the users' choices
2. The transaction is transferred to the user's wallet to sign
3. The user interacts with the wallet to sign the Dapp's suggested transaction
4. The wallet transmits the transaction to the blockchain.
5. The blockchain executes the transaction according to the Smart Contract code
6. The Wallet and the Dapp polls on the blockchain to fetch the transaction results and present them to the user. The user can continue to interact with the Dapp (back to step 1)

Section 2:
Web3 Security Challenges

Prior to Web3, in the Bitcoin model, users needed to trust two elements for transaction execution, namely their wallet and blockchain:

- The wallet needed to present the transaction details appropriately
- The blockchain needed to execute the transaction according to its content

Although there could be problems within both of these elements (rogue, buggy and hacked wallets and blockchains) users have found ways to identify the trustworthy blockchains and wallets.

However, Web3 introduced additional elements within the blockchain ecosphere: the Dapp and the Smart Contract. Unlike wallets and blockchains, in which users typically have a long-lasting relationship, Dapps and Smart Contracts can represent a very short-term relationship, so trust is hard to establish.

How can users protect themselves from Dapps that present attractive promises on their human-readable Web2 interface, but actually encode a transaction that would send all their money away?

There could be many reasons for discrepancies between the expected result as shown on the Web2 interface and the actual result as recorded on the blockchain. The Dapp interface can be hacked by attackers, spoofed by attackers, rogue by design, or just buggy and (almost) same goes for Smart contracts that can be buggy, or rogue by design.

A Prominent example of such issues is the BadgerDAO hack, in which hackers changed the Dapp's Web2 interface to target high–worth individuals and steal $120M of cryptocurrency.[1]

1 See: https://zengo.com/the-badgerdao-hack-what-really-happened-and-why-it-matters/

Section 3:
Solving Web3 Security Challenges with Transaction Simulation

To promote a trustworthy Web3 ecosystem, Web3 users must know that the result of the transaction they are about to sign, created by Dapp and evaluated by a Smart Contract, will yield the expected result as advertised by the Dapp.

To do so, they need to rely on their wallet (which they already trust and have a long-lasting relationship with) to help them, and present the expected results of the transaction, suggested by the Dapp.

While it is very hard to analyze the results of a piece of code for a given input, it is relatively straightforward to execute it in a contained environment and observe the results. This is what Transaction Simulation does: It executes the Transaction **locally** on the Ethereum Virtual Machine (EVM) and presents its results to the user.

Specifically, the transaction function and parameters are decoded from the transaction and executed by the Smart Contract code as deployed on the blockchain against the state of the blockchain (e.g. the user balance is taken from the blockchain's state) at the time of simulation. The results of the transaction is determined by standard events emitted by the Smart Contract, mostly the tokens' Transfer and Approval event.

However, there are limitations to this practice, as we will demonstrate in Section 4 (below).

Section 4:
Limitations of Simulation

The design of current mainstream Transaction Simulation solutions may be exposed to the following issues:

- **Event dependency issues:**
  - Mainstream Simulation solutions cannot detect operations that are not transfers or approvals: e.g. the simulation will not detect some proprietary function in a Dapp although it may have important consequences
  - Simulation will not detect non-standard events or when events are not emitted: for example, some early implementations of ERC-20 tokens (e.g. CryptoKitties) and NFTs were created before the standard was finalized and require a specific tailoring for their events to be interpreted
  - Rogue contracts may emit false events to deceive the Simulation

- **Time-of-Check, Time of Use (TOCTOU) issues:** Simulation is run against a given state of the blockchain, the results may vary when the actual transaction is executed. E.g. the simulation of a uniswap transaction with a changing rate might be different from the actual result when the transaction actually gets executed.

- **Evaluation of non-transactions ("offline signatures"):** Dapps may suggest users to sign buffers that are not immediately executed on the blockchain. These signed buffers are used later on as parameters for such Dapps. Examples for such offline signatures are the Permit offline signature and OpenSea List offline signature[2]. Since Simulation is based on execution of the buffer on the EVM, it cannot evaluate in the same manner.

- **Red pill attacks:** If the Simulation environment parameters are different than the actual blockchain environment, then a rogue contract can abuse that to detect it is running a

2 https://zengo.com/offline-signatures-can-drain-your-wallet-this-is-how-part-1-2/

Simulation and present benign results - delaying its actual malicious functionality until realtime execution on the real blockchain. We'll expand on this attack vector in Section 5 (below).

While these issues may appear to be very problematic, it is not to say that Transaction Simulations are irrelevant. On the contrary, Transaction Simulation can be a highly relevant solution to protect users against **rogue Dapps** suggesting bad function calls to **good contracts** controlling Tokens and NFTs.

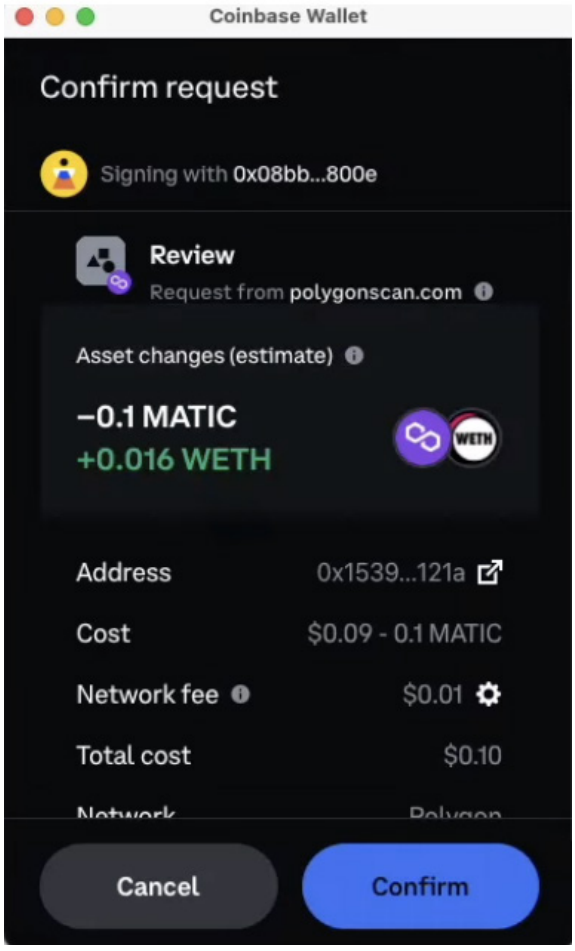
The **rogue Dapps** abusing **good contracts** scenario constitutes a large portion of the issues that users are faced with. It is important to note that Transaction Simulation is not a panacea and must be augmented with additional solutions to provide a more extensive user attack surface coverage.

Section 5:
Red Pill Attacks

If the Simulation environment parameters are different than the actual blockchain environment, then a rogue contract can abuse that to detect it is running under simulation and present benign results. Later, it will run its malicious functionality only when executed on the real blockchain.

Sieving through all of Solidity's (Ethereum smart contract programming language) instructions, we set our sights on "Special-Variables." These variables carry some general information on blockchain functionality (e.g. the timestamp of the current block, the address of current block miner) or some information on the user controlled parameters of the transaction (e.g. paid fees).
These instructions looked like potential candidates to be "red pills." Since these variables may take a range of values, there is no "correct" value for them. Therefore, it's tempting for simulation implementers to "take a shortcut" and set them to some constant value. While these constant values are technically valid, they can serve as a "red pill" by savvy attackers.

40	BLOCKHASH	`blockNum`	`blockHash(blockNum)`	
41	COINBASE	·	`block.coinbase`	address of miner of current block
42	TIMESTAMP	·	`block.timestamp`	timestamp of current block
43	NUMBER	·	`block.number`	number of current block
44	PREVRANDAO	·	`randomness beacon`	randomness beacon
45	GASLIMIT	·	`block.gaslimit`	gas limit of current block
48	BASEFEE	·	`block.basefee`	base fee of current block
3A	GASPRICE	·	`tx.gasprice`	gas price of tx, in wei per unit gas



Coinbase Simulation exploited with COINBASE Red Pill

For example, the "COINBASE" instruction contains the address of the current block miner. Since in simulation there is no real block and hence no miner, some simulation implementations just set it to the null address (all zeros address).

Therefore, a malicious smart contract may weaponize this "COINBASE" red pill as follows:
Ask users to send some native coin to the contract, if COINBASE is zero (which means simulation in Polygon) the contract will send back some coins in return, thus making the transaction potentially profitable to the user when its wallet simulates it. However, when the user sends the transaction on-chain, COINBASE is actually filled with the non-zero address of the current miner, and the malicious contract just takes the sent coins.

This is exactly what is shown in the screenshot[3], in which we show how the "COINBASE" exploit can be used against Coinbase wallet users.

The victim users are shown by simulation that the transaction is highly profitable: If they send 0.1 MATIC (~$0.1), they will get 0.016 WETH (~$30) in return. However, when they actually send their MATIC they actually get nothing in return!

3 Full video https://www.youtube.com/watch?v=Krp8qtVpGpQ

As a result, this vulnerable simulation implementation actually helps the malicious smart contract persuade its victims into sending their funds to it.

Testing for Red Pill attacks:

We tested a few simulation implementations and found out they were vulnerable to one or more variants of the red pill attack.

Vendor	Special variable opcode		
	Coinbase	Basefee	Gasprice
Zengo Wallet* with Alchemy	✓	✓	✓
Coinbase Wallet	✗	✗	✗
Rabby Wallet	✓	✓	✗
Blowfish	✓	✓	✗
Pocket Universe	✓	✗	✗
File Extension	✓	✓	✗
Unnamed Extension	✓	✓	✗

The above table shows vulnerable extensions, wallets and simulation vendors. All vendors were very receptive to our reports, and most of them were quick to fix their faulty implementations. **To help Simulation providers, wallet developers and users to test against such attacks, we are sharing our code and deployed Smart Contract as open source (See Appendix below for details).**

Section 6: Recommendations

For users:

1. Make sure the wallet you are using for Web3 transactions includes a Transaction Simulation capability as part of a holistic security solution that consists of multiple additional elements.

2. Security mechanisms are not a replacement for common-sense. If something is too good to be true, it probably is, even if your advanced security mechanisms do not alert.

For Wallets and Simulation Implementors:

1. Transaction Simulation capabilities are an important and even mandatory security tool for modern wallets and should be part of any modern Web3 wallet.

2. Transaction Simulation is not easy to get right, so it is probably better to use a battle tested implementation or service and not “roll your own”.

3. Test your Transaction Simulation implementation against our red pill attack reference implementation to make sure your solution is not vulnerable to such attacks.

4. Create a comprehensive Web3 security solution which includes Transaction Simulation along with additional elements (e.g. security reputation).

5. It is very important to put Transaction Simulation results in the necessary context to enable users to make informed decisions. E.g. the user experience for “Approval” transaction simulation should include explanation on the implications of approval and the reputation of the Approval spender’s address. E.g. approval for an EOA address should be a red flag.

Conclusions

Transaction Simulation is a highly relevant solution to protect users against **rogue Dapps** suggesting bad function calls to **good contracts** controlling Tokens and NFTs. The **rogue Dapps** abusing **good contracts** scenario, constitutes a very large portion of the issues that users are faced with.

However, it is important to note that Transaction Simulation is not a panacea and must be augmented with additional solutions to provide a more extensive user attack surface coverage. Additionally, Simulation and similar security solutions should be thoroughly tested, because of the negative impact of the false sense of security that may facilitate further abuse.

Appendix

Ethereum assembly opcodes analysis

The analysis showed the following potentially interesting opcodes

40	BLOCKHASH	`blockNum`	`blockHash(blockNum)`	
41	COINBASE	·	`block.coinbase`	address of miner of current block
42	TIMESTAMP	·	`block.timestamp`	timestamp of current block
43	NUMBER	·	`block.number`	number of current block
44	PREVRANDAO	·	`randomness beacon`	randomness beacon
45	GASLIMIT	·	`block.gaslimit`	gas limit of current block
48	BASEFEE	·	`block.basefee`	base fee of current block
3A	GASPRICE	·	`tx.gasprice`	gas price of tx, in wei per unit gas

Onchain infrastructure

A contract to test these behaviors was deployed to Ethereum Mainnet
0x53eeac329df155f07f327d13867fc93b0c7ebbf3#writeContract

The code is shared on https://github.com/ZenGo-X/simulation-checker

Supporting the following functionalities, that corresponds to tested opcodes above

1. claimBlockBasefee (0xe167211f)
2. claimBlockDiff (0x344c81b5)
3. claimCoinbase (0xd9d38244)
4. claimDifficulty (0x1f675d34)
5. claimTxGasprice (0x2a2b90b1)
6. renounceOwnership (0x715018a6)
7. testBlockBasefee (0xa4a557be)
8. testBlockCoinbase (0xa4f45e8b)
9. testBlockDifficulty (0xb0d49a89)
10. testGasLimit (0xd1123dde)
11. testTxGasPrice (0x54ca3ee5)
12. transferOwnership (0xf2fde38b)
13. updateTestingToken (0xab7e821e)
14. updateValidBlockDiff (0xe07ed431)
15. withdraw (0x3ccfd60b)

Authors and Acknowledgements

This report was assembled by the Zengo Research Team led by Zengo Co-founder and Chief Technical Officer Tal Be'ery, and contains contributions from many members across the company.

A special thank you to @0xVazi for his deep research contribution and to the Ethereum Foundation's Ecosystem Support Program (ESP) for their grant to pursue this critical research and share it with the wider community.

Questions or comments? Email us at security@zengo.com