Evasion Techniques
State Of The Art

Web3

Apehex
11/08/2023

CONTEXT

Prepared by the community of Forta as part of its Threat Research Inititative.

See here to apply to the TRi.

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document creation	01/08/2023	Apehex
0.2	First draft	11/08/2023	Apehex

CONTACTS

CONTACT	MAIL	?
Apehex	apehex@protonmail.com	t.me/apehex
?	?	?

OVERVIEW

0.1. INTRODUCTION

Smart contracts brought forth a new era of decentralized finance, with increasing value being funneled into DEFI platforms. In turn, they have become attractive tools for scammers and protocol attackers to steal digital assets.

As there is growing scrutiny by both users and security tools, malicious actors are answering with deception. This cat-and-mouse game between malware detection mechanisms and evasive tatics has been relentless in the binary and web spaces.

Now, this history can be analyzed to improve the current detection tools and anticipate future threats in the web3 ecosystem. We will delve into the code of each evasion technique, highlight their distinctive features and propose countermeasures.

0.2. METHODOLOGY

This state-of-the-art is grounded in both past and present research.

A literature review on traditional malware evasion forms the basis for the study's taxonomy and framework. Studying these historical evasion techniques gives insights into potential trends for the blockchain ecosystem.

In addition to the lessons from the past, the study also incorporates findings from current research in the web3 space. This research is sourced from academic papers, conferences, tools, and watch groups focused on blockchain security.

The report's practical aspect is backed by an analysis of selected smart contract samples. These samples were chosen for two reasons: their association with recent hacks and their ability to slip past detection mechanisms, especially those of the Forta network. Contracts from other platforms such as chainabuse, web3rekt, and rekt.news provided the necessary data for this analysis.

0.3. RELATED WORK

The closest research topic in mid 2023 is the honeypots, smart contracts which bait code reviewers into sending funds. [**the-art-of-the-scam**] shows how malicious actors craft fake vulnerabilities, e.g. code that behaves unexpectedly.

KNOWN TECHNIQUES

1. SPOOFING

Spoofing is the art of disguising malicious code as well-known or vulnerable code (honeypots) to bait users.

1.1. FAKE STANDARD IMPLEMENTATION

1.1.1. Overview

This contract borrows the function & class names from the ERC standards, but the code inside is actually different.

The malicious contracts generally pretend to be:

proxies but the implementation is either not used or different from the ERC-1967 proxy
tokens but the transfer and / or approve functions behave differently than ERC-20 / 721 / 1155

1.1.2. Evasion Targets

Etherscan the interpretation of proxy is fixed, it can easily be fooled
users few users actually check the code, having a valid front is enough

1.1.3. Samples

Fake EIP-1967 Proxy

Standard EIP-1967 [**eip-1967**] has pointers located in specific storage slots:

0x360894a13ba... location of the logic contract address
0xa3f0ad74e54... location of the beacon contract address

These can be kept null or point to any contract, while the proxy actually uses another address.

A minimal example was given at DEFI summit 2023 [**video-masquerading-code**]:

```
1  function _getImplementation() internal view returns (address) {
2      return
3          StorageSlot
4              .getAddressSlot(bytes32(uint256(keccak256("eip1967.fake")) - 1)).
5              .value;
6  }
```

Etherscan will show some irrelevant contract, giving the impression it is legit.

Fake ERC20 Token

Many phishing operations deploy fake tokens with the same symbol and name as the popular ones.

For example, this contract is spoofing the USDC token. It was used in this phishing transaction.

1.1.4. Detection & Countermeasures

Several sources can be monitored, depending on the standard that is being spoofed:

Storage comparing the target of `delegateCall` to the address in the storage slots of the standards

Events changes to the address of the logic contract should come with an `Upgraded` event

Bytecode the implementation of known selectors can be checked agains the standard's reference bytecode

1.2. OVERRIDING STANDARDS IMPLEMENTATION

1.2.1. Overview

Like the previous technique 1.1, the goal is to have a malicious contract confused with legitimate code.

It is achieved by inheriting from standardized code like `Ownable`, `Upgradeable`, etc. Then, the child class overwrites key elements with:

redefinition an existing keyword is defined a second time for the references in the child class only

polymorphism an existing method can be redined with a slightly different signature

From the perspective of the source code, a single keyword like `owner` can refer to different storage slot depending on its context. It is only in the bytecode that a clear difference is made.

1.2.2. Evasion Targets

This technique is a refinment of the previous one: it will work on more targets.

Etherscan blockchain explorers lack even more flexibility to detect these exploits

Users the source code is even closer to a legitimate contract

Reviewers the interpretation of the source code is subtle, and reviewing the bytecode is very time consuming

1.2.3. Samples

Attribute Overwriting

In section 3.2.2, the paper [**paper-art-of-the-scam**] shows an example of inheritance overriding with `KingOfTheHill` :

```
1  contract KingOfTheHill is Ownable {
2      address public owner; // different from the owner in Ownable
3
4      function () public payable {
5          if(msg.value > jackpot) owner = msg.sender; // local owner
6          jackpot += msg.value;
7      }
8      function takeAll () public onlyOwner { // contract creator
9          msg.sender.transfer(this.balance);
10         jackpot = 0;
11     }
12 }
```

In the modifier on `takeAll`, the `owner` points to the contract creator. It is at storage slot 1, while the fallback function overwrites the storage slot 2.

In short, sending funds to this contract will never make you the actual owner.

1.2.4. Detection & Countermeasures

While subtle for the human reader, tools can rather easily detect it in:

source code the sources can be checked for duplicate definitions & polymorphism
bytecode

Since the whole point is to advertize for a functionality with the sources, they will be available.

1.3. BUG EXPLOITS

1.3.1. Overview

A more vicious way to mask ill-intented code is to exploit bugs and EVM quirks.

By definition, these bugs trigger unwanted / unexpected behaviors.

They can be:

EVM quirks in particular, some operations are implied and not explicitely written

bugs the Solidity language itself has numerous bugs, depending on the version used at compilation time [**changelog-solidity-bugs**]

They are usually leveraged in honeypots, where the attackers create a contract that looks vulnerable. But the "vulnerability" doesn't work and people who try to take advantage of it will lose their funds.

1.3.2. Evasion Targets

tools honeypots are meant to trigger alerts in popular tools and mislead their users

reviewers successfully used in honeypots, these tricks can fool security professional

1.3.3. Samples

All the samples below come from the paper The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts [**paper-art-of-the-scam**].

Impossible Conditions

Attackers can craft a statement that will never be true.

A minimal example was given at DEFI summit 2023 by Noah Jelic [**video-hacker-traps**]:

```
1  function multiplicate() payable external {
2      if(msg.value>=this.balance) {
3          address(msg.sender).transfer(this.balance+msg.value);
4      }
5  }
```

This gives the illusion that anyone may-be able to withdraw the contract's balance.

However, at the moment of the check, this.balance has already been incremented: it can never be lower than msg.value.

In reality, the contract would have exactly the same behavior if the multiplicate function was empty.

Skip Empty String Literal

The Solidity encoder skips empty strings: the following arguments in a function call are shifted left by 32 bytes.

In the following snippet, the call to `this`.loggedTransfer ignores `msg`.`sender` and replaces it with `owner`. In other words the sender cannot actually receives the funds, it is a bait.

```
1  function divest ( uint amount ) public {
2      if (investors[msg.sender].investment == 0 || amount == 0) throw;
3      investors[msg.sender].investment -= amount;
4      this.loggedTransfer(amount, "", msg.sender, owner);
5  }
```

Type Deduction Overflow

The compiler uses type deduction to infer the the smallest possible type from
its assignment. For example, the counter is given the type uint8, and the loop
actually finishes at 255 instead of 2*msg.value:

```
1  if (msg.value > 0.1 ether) {
2      uint256 multi = 0;
3      uint256 amountToTransfer = 0;
4      for (var i=0; i < 2*msg.value; i++) {
5          multi = i * 2;
6          if ( multi < amountToTransfer ) {
7              break;
8          }
9          amountToTransfer = multi;
10     }
11     msg.sender.transfer(amountToTransfer);
12 }
```

Since the caller must have sent 0.1 ether he loses money.

Uninitialised Struct

Non initialized structs are mapped to the storage. In the following example,
the struct GuessHistory overwrites the "private" random number.

```
1  contract GuessNumber {
2      uint private randomNumber = uint256(keccak256(now)) % 10+1;
3      uint public lastPlayed;
4      struct GuessHistory {
5          address player;
6          uint256 number;
7      }
8      function guessNumber (uint256 _number) payable {
9          require (msg.value >= 0.1 ether && _number <= 10);
10         GuessHistory guessHistory;
11         guessHistory.player = msg.sender;
12         guessHistory.number = _number ;
13         if (number == randomNumber)
14             msg.sender.transfer(this.balance);
15         lastPlayed = now;
16     }
17 }
```

in the check (number == randomNumber), the randomNumber is now an address which is
highly unlikely to be lower than 10.

1.3.4. Detection & Countermeasures

testing symbolic testing & fuzzing will show the actual behavior; the issue is rather to formulate what is expected for any arbitrary contract

CVEs known vulnerabilities can be identified with pattern matching; in traditional malware detection, YARA rules are written

There's a tool aimed specifically at detecting honeypots, HoneyBadger.

2. MORPHING

Morphing contracts change their behavior depending on the context. In particular they replicate benign functionalities when they're under scrutiny.

2.1. RED-PILL

2.1.1. Overview

The red-pill technique detects simulation environment to disable its exploits upon scrutiny.

The contract detects simulation environments by checking:

globals these variables have special values in test environments:
- block.basefee: 0
- block.coinbase: 0x0000000000000000000000000000000000000000
- tx.gasprice: large numbers, higher than 0xffffffffffffffff

Then it triggers legitimate code in simulation contexts and malicious code on the mainnet.

2.1.2. Evasion Targets

tests wallets often perform a simulation of the transaction before committing
tools automatic tools may not go further than basic dynamic analysis

On the other hand it is rather obvious when reviewing the code.

2.1.3. Samples

The contract FakeWethGiveaway mentioned in [**article-red-pill**] checks the current block miner's address:

```solidity
function checkCoinbase() private view returns (bool result) {
    assembly {
        result := eq(coinbase(), 0x0000000000000000000000000000000000000000)
    }
}
```

When null (test env), it actually sends a reward:

```solidity
bool shouldDoTransfer = checkCoinbase();
if (shouldDoTransfer) {
    IWETH(weth).transfer(msg.sender, IWETH(weth).balanceOf(address(this)));
}
```

Otherwise, on the mainnet, it just accepts transfers without doing anything.

2.1.4. Detection & Countermeasures

opcodes looking for unusual opcodes: typically `block`.`coinbase`
fuzzing the transactions can be tested with blank data and compared with results
      behavior on data

2.2. LATERAL MOVEMENT

2.2.1. Overview

New: - EOA addresses - logic contracts -

Evade transactions history parsers.

2.2.2. Evasion Targets

2.2.3. Samples

2.2.4. Detection & Countermeasures

3. OBFUSCATION

Obfuscation is the process of making (malicious) code hard to find and understand.

3.1. HIDING IN PLAIN SIGHT

3.1.1. Overview

By stacking dependencies, the scammer grows the volume of the source code to thousands of lines.

99% of the code is classic, legitimate implementation of standards.

And the remaining percent is malicious code: it can be in the child class or hidden inside one of the numerous dependencies.

This technique is the most basic: it is often used in combination with other evasion methods.

3.1.2. Evasion Targets

users wallets often perform a simulation of the transaction before committing
reviewers the goal is to overwhelm auditors with the sheer volume of code
tools unrelated data also lowers the efficiency of ML algorithms

3.1.3. Samples

Hidden among 7k+ lines of code:

```
1  // no authorization modifier `onlyOwner`
2  function transferOwnership(address newOwner) public virtual {
3      if (newOwner == address(0)) {
4          revert OwnableInvalidOwner(address(0));
5      }
6      _transferOwnership(newOwner);
7  }
```

3.1.4. Detection & Countermeasures

bytecode the size of the bytecode is a low signal
tracing the proportion of the code actually used can be computed by replaying
      transactions

3.2. HIDING BEHIND PROXIES

3.2.1. Overview

Malicious contracts simply use the EIP-1967 [**eip-1967**] specifications to split the code into proxy and logic contracts.

3.2.2. Evasion Targets

Etherscan the proxy contracts are often standard and will be validated by block explorers

users most users rely on block explorers to trust contracts

reviewers the source code for the logic contract may not be available: reversing and testing EVM bytecode is time consuming

3.2.3. Samples

This phishing contract has its proxy contract verified by Etherscan.

While its logic contract is only available as bytecode.

3.2.4. Detection & Countermeasures

Since it comes from Ethereum standards, this evasion is well-known and easy to detect.

However it is largely used by legitimate contracts, it is not conclusive by itself.

proxy patterns proxies can be identified from the bytecode, function selectors, storage slots of logic addresses, use `delegateCall`, etc

block explorer the absence of verified sources is a stronger signal (to be balanced according to contract activity and age)

bytecode the bytecode of the logic contract can still be further analyzed

3.3. HIDDEN STATE

3.3.1. Overview

The storage slots are not explicitely listed: it is easy to stash data without trace.

initialization the constructor code is not in the available bytecode, it can fill slots without raising any flag
delegation a delegate contract could also modify the state

3.3.2. Evasion Targets

Actually, this method is effective against all the detection agents:

everyone the data is not visible in the sources nor in the bytecode

3.3.3. Samples

The contract can be entirely legitimate, and compromising the storage is enough.

It has been demonstrated by Yoav Weiss [**video-masquerading-code**] with a Gnosis Safe. The constructor injected an additional owner into the storage, allowing a hidden address to perform administrative tasks.

3.3.4. Detection & Countermeasures

4. POISONING

Poisoning techniques hijack legitimate contracts to take advantage of their authority and appear trustworthy.

4.1. EVENT POISONING

4.1.1. Overview

By setting the amount to 0, it is possible to trigger `Transfer` events from any ERC20 contracts.

In particular, scammers bait users by coupling two transfers:

- a transfer of 0 amount of a popular token, say USDT
- a transfer of a small amount of a fake token, with the same name and symbol

4.1.2. Evasion Targets

users many users don't double check events coming from well-known tokens

4.1.3. Samples

In this batch transaction, the scammer pretended to send USDC, DAI and USDT to 12 addresses.

The Forta network detected the transfer events of null amount.

4.1.4. Detection & Countermeasures

These scams are easily uncovered:

logs the transactions logs contain the lsit of events, whose amounts can be parsed

5. REDIRECTION

These techniques reroute the execution flow from legitimate functions to hidden and malicious code.

5.1. HIDDEN PROXY

5.1.1. Overview

Here, the contract advertises functionalities through its sources but actually redirects to another contract.

One common way to achieve this is to performs `delegateCall` on any unknown selector, via the fallback.

The exposed functionalities are not meaningful, the logic is located at a seemingly unrelated & hidden address.

The target address can be hardcoded or passed as an argument, making it stealthier.

5.1.2. Evasion Targets

This technique stacks another layer of evasion on top those mentioned in 3.1:

tools testing visible code does not bring out the malicious part
reviewers the proxy address may not even be in the byte / source code

5.1.3. Samples

A malicious fallback can be inserted into an expensive codebase:

```
1  fallback () external {
2    if (msg.sender == owner()) {
3      (bool success, bytes memory data) = address(0
           x25B072502FB398eb4f428D60D01f18e8Ffa01448).delegateCall(
4        msg.data
5      );
6    }
7  }
```

5.1.4. Detection & Countermeasures

In addition to the sources & indicators mentioned in 3.1:

history the hidden proxy address can be found in the trace logs

upgrades replaying transactions before / after upgrades may show significant differences

KNOWN TECHNIQUES

5.2. SELECTOR COLLISIONS

5.2.1. Overview

Because the function selectors are only 4 bytes long, it is easy to find collisions.

When a selector in the proxy contract collides with another on the implementation side, the proxy takes precedence.

This can be used to override key elements of the implementation.

5.2.2. Evasion Targets

tools this subtle exploit evades most static analysis
reviewers the sources don't show the flow from legitimate function to its malicious collision

5.2.3. Samples

As Yoav Weiss showed at DSS 2023 [**video-masquerading-code**], this harmless function:

```
1  function IMGURL() public pure returns (bool) {
2      return true;
3  }
```

Collides with another function:

```
1  Web3.keccak(text='IMGURL()').hex().lower()[:10]
2  # '0xbab82c22'
3  Web3.keccak(text='vaultManagers(address)').hex().lower()[:10]
4  # '0xbab82c22'
```

And this view is used to determine which address is a manager, e.g. it is critical:

```
1  mapping (address=>bool) public vaultManagers;
```

5.2.4. Detection & Countermeasures

The collisions can be identified by comparing the bytecodes of proxy and implementation:

selectors the hub section of the bytecode has the list of selectors
debugging dynamic analysis will trigger the collision; still it may not have an obviously suspicious behavior

The article deconstructing a Solidity contract [**article-deconstructing-contract**] has a very helpful diagram [**image-deconstruction-diagram**].

FORESEEN TECHNIQUES

6. SPOOFING

Spoofing is the art of disguising malicious code as well-known or vulnerable code (honeypots) to bait users.

6.1. FAKE USERS

6.1.1. Overview

Much like social networks, the blockchain has a network of interconnected users. Their activity in and out of the blockchain gives weight to a project.

So scammers could creates bots and enroll people to build a legitimate history on their contracts.

6.1.2. Evasion Targets

6.1.3. Samples

6.1.4. Detection & Countermeasures

FORESEEN TECHNIQUES

7. OBFUSCATION

Obfuscation is the process of making (malicious) code hard to find and understand.

7.1. PAYLOAD PACKING

7.1.1. Evades

Pattern matching on the bytecode.

7.1.2. How

Encryption / encoding / compression can be leveraged to make malicious code unreadable.

7.1.3. Detection & Countermeasures

1. Scanning for high entropy data

DETECTION TOOLS

8. MALWARE DETECTION IN WEB3

In this section, we'll consider each the parts and processes of the detection tools. This serves both the purpose of imagining how they can fail as well as designing the next tools.

8.1. DATA SOURCES

The data available for analysis depends on the execution stage and scope. There are three main contexts.

8.1.1. Static Analysis

Outside execution, the blockchain and the surrounding services act as a cold storage.

contract metadata details like the contract's creator, the balance, the creation timestamp and associated Ether provide a context to the whole analysis

bytecode representation similarly to the traditional binaries, smart contracts are compiled into bytecode which can be parsed: headers

opcode sequences bytecode can be interpreted as a language, giving a level of abstraction to the analysis

function signatures more specifically, functions can be extracted and compared to the reference implementations of the standards for example

source code when available, this layer can hold deceptive measures for the human reader; hence, it is very informative + creation code (not in bytecode)

8.1.2. Dynamic Analysis

When a transaction is committed to the blockchain, trace data is generated:

transaction metadata the global variables `block`, `tx` and `msg` hold valuable informations

state changes storage slots and balance may change

function calls this can be insightful in determining the contract's behavior

external calls identify if the contract interacts with other contracts or addresses

events the list of events is always available

8.1.3. Network Analysis

Statistics and graph theory can be applied on the blockchain as a whole:

transaction data the blockchain can be interpreted as a graph, with addresses as nodes and transactions as vertexes

historical behavior the activity of a single address over time can be broken-down with statistics

8.2. DETECTION TECHNOLOGIES

8.2.1. Signature Based

Signature-based malware detection extracts distinctive information known as signatures to detect malware. then compared against references to determine whether the unclassified binary's signature indicates malicious behavior.

YARA

YARA separates the detection logic from the software that scans files. It has a rule-based language extending the regular expressions. It was originally designed for traditional binaries, but the YARA rules could be applied to smart contract bytecode or even source code.

Semgrep

Similarly, Semgrep patterns can help prototype detection ideas.

Solidity Static Analysis Tools

Tools like Mythril, Slither, and Securify can analyze Solidity smart contract code for known vulnerabilities. While they primarily focus on vulnerabilities rather than malware, their methodologies could inspire a signature-based approach to detect malicious intent in smart contracts.

9. DESIGNING THE FORTA AGENT

9.1. TAKING ADVANTAGE OF THE NETWORK

APPENDICES

J. SAMPLES

J.I. RED PILL