Evasion Techniques

Report On The Continuous Monitoring

Web3

Forta Community
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. To achieve their end goals, they first have to appear legitimate and circumvent the security tools. This involves specific tricks, which we refer to as "evasion" and are the focus of this document.

Exploit detection mechanisms and evasive tactics have played a relentless cat-and-mouse game 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 report 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.

Forta being a network of independent scanning agents, each of them is free to implement a different approach. Since it is not bound by a systemic choice of detection, the countermeasures are centered on each evasion technique. Static, dynamic, hybrid, graph analysis are all mentioned when it is relevant to a given target.

The analysis is meant as a reference guide for the development of future bots on the Forta network. It will be a continuous feeedback loop: the report will be updated regularly as progress is made.

DETECTION IN WEB3

We'll transpose the traditional malware analysis to the smart contracts.

This serves both the purpose of designing detection tools as anticipating their shortcomings.

1. DATA SOURCES

The data available for analysis depends on the execution stage. For smart contracts, there are three main contexts: static, dynamic and hybrid.

1.1. STATIC ANALYSIS

Outside of execution, the blockchain acts as a cold storage. In this first context, the detection methods are called "static analysis".

1.1.1. Creation Metadata

The block and transaction objects hold a lot of data related to the infrastructure of the blockchain. These informations, like `block.difficulty` or `block.gaslimit`, can be ignored when considering the smart contracts.

Still, the global variables `block`, `msg` and `tx` hold valuable informations.

Contract's creator

The values `msg.sender` & `tx.origin` of the transaction that created the contract tell us who did it!

This would be like having an IP: the addresses can be indexed to follow the activity of known attackers.

In turn, bad actors can simply use new "external owned accounts" (EOA) and redeploy / upgrade their contracts.

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

Creation Cost

The product of the gas price and gas used gives the cost of the smart contract deployment.

This gas consumption is directly related to the intensity of the processing involved. The historical data can be compared to a local replay to determine if all the operations are accounted for in the deployment code.

1.1.2. Bytecode

Similarly to the traditional binaries, smart contracts are compiled into bytecode. It has several sections which can be parsed: OpenZeppelin wrote an in-depth

article on the structure of smart contract bytecode.

In itself, providing only the bytecode (and not the sources) is already a layer of obfuscation. But it is always available and has all the logic of the smart contract.

Function Selectors

Functions are not called by name, but by their selector. And the selectors are hashes computed on the signature, like transfer(address,uint256):

```
1  Web3.keccak(text='transfer(address,uint256)').hex().lower()[:10]
2  # '0xa9059cbb'
```

The list of selectors for all the function in the bytecode is found in its hub.

Keeping an updated index of all known selectors allows to go back from hash to signature. It gives a lot of insight on the expected behavior of a contract.

On the other hand, nothing prevents malicious actors from naming their functions as they please.

Function Bodies

Of course, execution requires instructions: the function bodies implement the logic of the contract.

Just like binaries, they can be reversed and analysed statically. This opens the way for pattern matching and manual reviews of the code.

However, these processes can be hindered with code stuffing and other techniques like packing (encryption, compression, etc).

Constructor

It may sound counterintuitive, but the smart contract constructor is not included in the bytecode deployed on the blockchain. It is called once to initialize the contract state and generate the final code that will sit on the blockchain.

It can be found:

· in the data of the transaction that created the contract
· or in the source code, if provided as is discussed below

The storage slots hold values that can totally change the behavior of the contract. Admin privileges can at as a backdoor and enable rug pulls for example.

Attackers will try and sneak data into the contract's state.

Opcode Sequence

Bytecode can be interpreted as a language, giving a level of abstraction to the analysis.

Indeed, different hex bytecodes can achieve the same result. It is easier to get the high level logic from the sequences of opcodes than from raw and specific hex chunks.

So the analysis mentioned above can be performed on opcodes, after disassembling the binary. But disassembling is not an exact science and it can be made even harder by classic techniques like anti-patterns.

1.1.3. Source code

First, source code is not always available: the blockchain itself doesn't hold it, it has to be supplied to third party services, e.g. block explorers.

When it is, it provides human readable code, which makes reverse engineering easier by hand. It is important to understand new attacks, but orders of magnitude too time consuming to provide live intelligence.

Furthermore, the underlying language has many ambiguities and bugs. They can be leveraged to mislead analysts and tools.

1.2. DYNAMIC ANALYSIS

When a transaction is committed to the blockchain, the targeted smart contract is executed. The actual behavior of the contract can be witnessed first hand in this "dynamic" analysis, rather than infered.

1.2.1. Execution Metadata

First, the execution can be monitored on the blockchain nodes, with the actual live data.

Transaction Origin

Just like the the contract's creator, every address the contract interacts with can be indexed. This way, one suspicious occurrence can be correlated with others to increase the accuracy.

Again, the attackers can answer with lateral movement.

Transaction Recipient

Here the `to` field can only be the contract under inspection. However it can call other addresses as part of its processing, as seen below.

Transaction Gas

As mentioned earlier, gas is directly linked to the intensity of the operations in the transaction.

Like CPU and RAM overloading, intensive computation can be the sign of unwanted activity. Or it can be exploited for its own value: similarly to CPU / GPU mining, gas can sometimes be redeemed by attackers.

Still, the blockchain always has its "task manager" open, so it is hard to fly these tricks under the radar.

Transaction Value

High value transactions are not necessarily bad, but they are bound to attract attention.

Bad actors will lower the noise levels by mixing / scattering the cash flow for example.

1.2.2. Event Logs (Topics)

The events triggered by a given transaction are encoded in the logs, more specifically in their topics and data fields. The type and arguments of the events hold a lot of information by themselves. Also the emitting address tells what external contracts were called if any.

Sometimes the presence of events is suspicious: in case of a high number of transfers for example.

Other times their absence has implications: upgrading the implementation of a proxy without triggering an `Upgraded` event is at least weird.

1.2.3. Execution Traces

Execution traces can be obtained either by replaying locally a transaction or by querying a RPC node with tracing enabled.

Internal Function Calls

The flow of internal calls can be debugged locally, which may be the most insightful analysis tool.

Just like traditional malware, smart contracts have means to evade debugging: tests can be detected, the logic of the contract can be cluttered...

External Function Calls

A given smart contract can redirect the execution flow to external addresses. `address`.`call` will segregate the contexts of the contracts, while `address`.`delegatecall` allows the target contract to modify the state of the origin address.

These external calls may be aimed at:

· EOAs, for example to bait them into performing unsafe actions
· legitimate contracts, to loan, launder, exploit, etc
· malicious contracts, to split and layer the suspicious activity

Splitting the logic over several contracts is a way to make local debugging harder too.

1.2.4. State Changes

State changes cover:

· modification of the data in the storage slots
· changes to the balance of the address

In particular, the storage of ERC contracts hold a lot of financial information, which is valuable in itself: token holders, exchange rates, administrative privileges, etc.

Because of the way data is encoded and positioned in the storage slots, there is no way to tell which slots are used without context. This context can come from the transaction history or local debugging.

In any case the storage is stealthy by design.

1.3. HYBRID ANALYSIS

Zooming out from the perspective of a single smart contract, the blockchain can be considered as a whole. This is a mix of the static data over all addresses and the dynamic data generated over time and addresses.

Rather than going over all the data sources again, this section offers new angles from which they can be considered.

1.3.1. Statistics

The activity of a single address over time can be broken-down with statistics

Cash flow.

Creation time

The age of the contract can be learnt from `block`.`timestamp` and `block`.`number`.

It can be used to weight other temporal data

1.3.2. Graph Theory

The blockchain archive can be seen as a temporal graph. While the exact structure can vary depending on the application, generally the nodes are the addresses and the vertices are transactions.

1.3.3. Machine learning

1.3.4. Symbolic Fuzzing

Tools like HoneyBadger leverage symbolic testing to explore all the execution paths.

2. TAXONOMY

Having looked over the sources of data available, many avenues for detection and evasion emerged. You can see them classified in figure 2.1 below.

This taxonomy was made by analogy with the malware space: a good overwiew can be found in this survey from Applied Sciences.

The evasion tactics leave traces on all the levels: static, dynamic and hybrid analysis all have a role to play in their detection.

So the specifics of the detection methods depend entirely on their target: the rest of the document will focus on each evasion mechanism and draw specialized indicators of compromise.
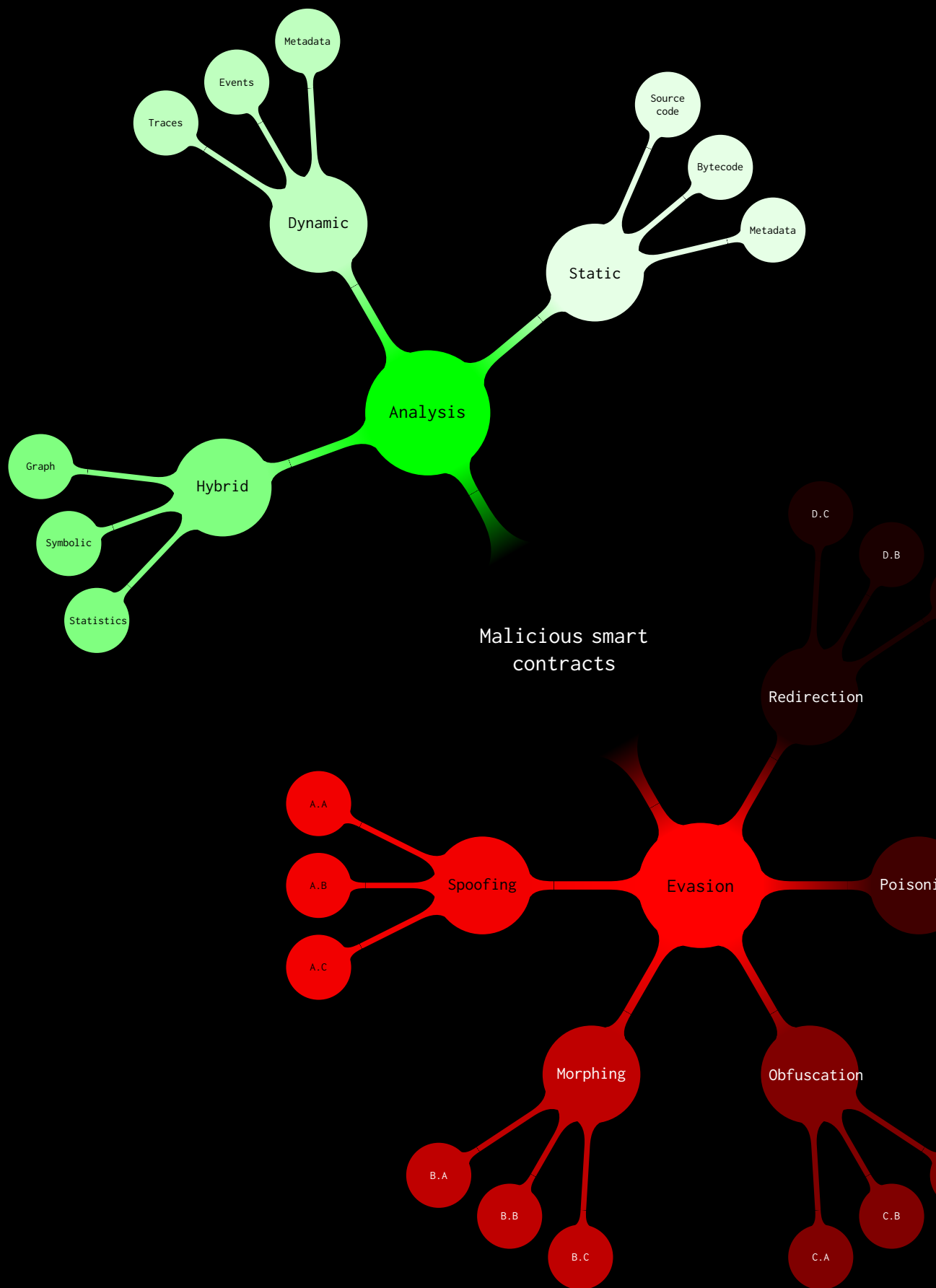
Figure 2.1: Taxonomy of the detection & evasion techniques

KNOWN EVASION TECHNIQUES

3. SPOOFING

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

3.1. FAKE STANDARD IMPLEMENTATION

3.1.1. 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

3.1.2. Evasion Targets

block explorers the interpretation of proxies is fixed, it can easily be fooled

users few users actually check the code, having a valid front is enough

3.1.3. 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.

3.1.4. 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

3.2. OVERRIDING STANDARDS IMPLEMENTATION

3.2.1. Overview

Like the previous technique 3.1, 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.

3.2.2. Evasion Targets

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

block explorers 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

3.2.3. 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.

3.2.4. 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.

3.3. BUG EXPLOITS

3.3.1. 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.

3.3.2. Evasion Targets

tools honeypots are meants to trigger alerts in popular tools and mislead their users
reviewers successfully used in honeypots, these tricks can fool security professional

3.3.3. 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.

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

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`:

```solidity
if (msg.value > 0.1 ether) {
    uint256 multi = 0;
    uint256 amountToTransfer = 0;
    for (var i=0; i < 2*msg.value; i++) {
        multi = i * 2;
        if ( multi < amountToTransfer ) {
            break;
        }
        amountToTransfer = multi;
    }
    msg.sender.transfer(amountToTransfer);
}
```

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.

```solidity
contract GuessNumber {
    uint private randomNumber = uint256(keccak256(now)) % 10+1;
    uint public lastPlayed;
    struct GuessHistory {
        address player;
        uint256 number;
    }
    function guessNumber (uint256 _number) payable {
        require (msg.value >= 0.1 ether && _number <= 10);
        GuessHistory guessHistory;
        guessHistory.player = msg.sender;
        guessHistory.number = _number ;
        if (number == randomNumber)
            msg.sender.transfer(this.balance);
        lastPlayed = now;
    }
```

```
17 }
```

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

3.3.4. 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.

4. MORPHING

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

4.1. RED-PILL

4.1.1. 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.

4.1.2. 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.

4.1.3. Samples

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

```
1  function checkCoinbase() private view returns (bool result) {
2      assembly {
3          result := eq(coinbase(), 0x0000000000000000000000000000000000000000)
4      }
5  }
```

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

```
1  bool shouldDoTransfer = checkCoinbase();
2  if (shouldDoTransfer) {
3      IWETH(weth).transfer(msg.sender, IWETH(weth).balanceOf(address(this)));
4  }
```

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

4.1.4. 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

4.2. LATERAL MOVEMENT

4.2.1. Overview

New: - EOA addresses - logic contracts

Evade transactions history parsers.

4.2.2. Evasion Targets

4.2.3. Samples

4.2.4. Detection & Countermeasures

5. OBFUSCATION

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

5.1. HIDING IN PLAIN SIGHT

5.1.1. 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.

5.1.2. 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

5.1.3. 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  }
```

5.1.4. 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

5.2. HIDING BEHIND PROXIES

5.2.1. Overview

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

5.2.2. 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

5.2.3. Samples

This phishing contract has its proxy contract verified by Etherscan.

While its logic contract is only available as bytecode.

5.2.4. 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

5.3. HIDDEN STATE

5.3.1. 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

5.3.2. 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

5.3.3. 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.

5.3.4. Detection & Countermeasures

6. POISONING

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

6.1. EVENT POISONING

6.1.1. 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

6.1.2. Evasion Targets

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

6.1.3. 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.

6.1.4. Detection & Countermeasures

These scams are easily uncovered:

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

7. REDIRECTION

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

7.1. HIDDEN PROXY

7.1.1. 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.

7.1.2. Evasion Targets

This technique stacks another layer of evasion on top those mentioned in 5.1:

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

7.1.3. 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  }
```

7.1.4. Detection & Countermeasures

In addition to the sources & indicators mentioned in 5.1:

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

upgrades replaying transactions before / after upgrades may show significant
differences

7.2. SELECTOR COLLISIONS

7.2.1. 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.

7.2.2. 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

7.2.3. 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;
```

7.2.4. 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 EVASION TECHNIQUES

8. SPOOFING

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

8.1. SYBILS

8.1.1. Overview

Much like social networks, the blockchain is made 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. - create a legitimitae service to hijack it later

8.1.2. Evasion Targets

8.1.3. Samples

8.1.4. Detection & Countermeasures

9. MORPHING

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

9.1. LOGIC BOMB

9.1.1. Overview

As Wikipedia states it: a logic bomb is a piece of code intentionally inserted into a software system that will set off a malicious function when specified conditions are met. These conditions are usually related to:

· the execution time: it can check the `block`.`timestamp` or `block`.`number` for example
· the execution environment: actually, the technique from section 4.1 is a subclass of the logic bomb

Some logic bombs are meant to counter symbolic testing. These bombs nest conditional statements without actually caring about the tests themselves. The simple chaining of conditions has the effect of exponantially increasing the number of execution paths. In the end, it may overload the testing process.

9.1.2. Evasion Targets

Tests

9.1.3. Samples

Executing Raw Bytecode

9.1.4. Detection & Countermeasures

10. OBFUSCATION

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

10.1. PAYLOAD PACKING

10.1.1. Overview

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

10.1.2. Evasion Targets

Pattern matching on the bytecode.

10.1.3. Samples

10.1.4. Detection & Countermeasures

1. Scanning for high entropy data

11. POISONING

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

11.1. LIVING OFF THE LAND

11.1.1. Overview

Living off the land means surviving on what you can forage, hunt, or grow in nature. For malware, it means using

- callbacks (flashloans) - special contract

11.1.2. Evasion Targets

Pattern matching on the bytecode.

11.1.3. Samples

Executing Raw Bytecode

11.1.4. Detection & Countermeasures

APPENDICES

L. SAMPLES

L.I. RED PILL