Evasion Techniques

State Of The Art

Web3

Apehex

04/08/2023

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	09/08/2023	Apehex

CONTACTS

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

Overview

0.1. Introduction

Smart contracts are core tools for scammers and protocol attackers to steal digital assets.

As there is now more scrutiny by both users and security tools, scammers are answering with deception.

There is a long history of malware detection and evasion growing side-by-side in the binary and web2 spaces.

It is very likely web3 will follow the same path: this report details the latest known developments as well as potential evolutions hinted at by the history of malware.

0.2. Methodology

This state of the art is the product of:

litterature review there's a wealth of information on malware in:

- **Web2 history** by analogy, new techniques can be predicted for the blockchain
- **Web3 community** the community is very active: papers, conferences, tools, watch groups are great sources of information

sample collection with the analysis of contracts collected by:

- the Forta network
- chainabuse
- web3rekt
- rekt.news

Known Techniques

1. Faking

1.1. Fake Standard Implementation

1.1.1. Technical Details

This technique takes on 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**]:

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

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

1.1.4. Detection & Countermeasures

Several sources can be monitored:

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

KNOWN TECHNIQUES

1.2. Overriding Standards Implementation

1.2.1. Technical Details

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` :

```
contract KingOfTheHill is Ownable {
    address public owner; // different from the owner in Ownable

    function () public payable {
        if(msg.value > jackpot) owner = msg.sender; // local owner
        jackpot += msg.value;
    }
    function takeAll () public onlyOwner { // contract creator
        msg.sender.transfer(this.balance);
        jackpot = 0;
    }
}
```

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.

Method Overwriting

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. Technical Details

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 meants 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

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**]:

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

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.

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

2.1. Red-Pill

2.1.1. Technical Details

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

The contract detects simulation environments by checking:

globals the global variables have special values in test environments:

- block.basefee:
- block.coinbase: 0x0000000000000000000000000000000000000000
- tx.gasprice:

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:

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

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

```
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

3. Obfuscation

3.1. Hiding In Plain Sight

3.1.1. Technical Details

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:

```
// no authorization modifier `onlyOwner`
function transferOwnership(address newOwner) public virtual {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}
```

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. Technical Details

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. Technical Details

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

4.1. Event Poisoning

5. Redirection

5.1. Hidden Proxy

5.1.1. Technical Details

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:

```
fallback () external {
  if (msg.sender == owner()) {
    (bool success, bytes memory data) = address(0
        x25B072502FB398eb4f428D60D01f18e8Ffa01448).delegateCall(
      msg.data
    );
  }
}
```

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

5.2. Selector Collisions

5.2.1. Technical Details

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:

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

Collides with another function:

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

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

```
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. Obfuscation

6.1. Payload Packing

6.1.1. Evades

Pattern matching on the bytecode.

6.1.2. How

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

6.1.3. Detection & Countermeasures

1. Scanning for high entropy data

Detection Tools

7. Static Analysis

8. Dynamic Analysis

Appendices

I. Samples

I.I. Red Pill

APPENDICES