Evasion Techniques

Report On The Continuous Monitoring

Web3

Apehex
24/01/2024

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
1.0	First draft	31/08/2023	Apehex
1.1	Processing flowcharts	29/09/2023	Apehex
1.2	Metamorphic contracts	20/10/2023	Apehex
1.3	Obfuscation techniques	24/01/2024	Apehex

CONTACTS

CONTACT	MAIL	MORE
Apehex	apehex@protonmail.com	Github: apehex
Christian Seifert	christian@forta.org	Twitter: cseifert

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 feedback 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 safely ignored when considering the smart contracts.

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

Contract's creator

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

Creation Cost

The product of the gas price and gas used gives the cost of the smart contract deployment. This gas consumption is related to the intensity of the processing involved.

EVM operation costs have a wide range and some key operations are especially pricy. Contract activity can be differentiated based on this value: it is a built-in high level profiling tool.

1.1.2. Compilation Metadata

Similarly to traditional binaries, smart contracts are compiled into bytecode. The settings used during the compilation are described in **a JSON file**.

The hash of these metadata may be appended to the bytecode: it is actually an ID, which can be used to retrieve the metadata and possibly the sources from a IPFS.

In particular, the configuration of the optimizer is specified: the exact binary output of the compilation will vary according to these settings. These informations can be used to adapt static analysis patterns to a specific target.

1.1.3. Bytecode

The main product of the compilation is the bytecode deployed on the mainnet. 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

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.

So it can be found in the data of the **transaction that created the contract**. Or in the source code, if provided (discussed below).

The constructor sets storage slots, which hold values that can totally change the behavior of the contract. Admin privileges can act 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.

But disassembling is not an exact science and it can be made even harder by classic techniques like **anti-patterns**.

1.1.4. 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.

With it, code review is humanly possible and reverse engineering becomes easier. Sources help to understand new attacks, but are too time consuming to provide live intelligence.

Solidity can be misleading because of its ambiguities and **bugs**. Attackers will take advantage of the imprecision in the tools and the limited resources of human reviewers.

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

The records on the blockchain show every address that interacted with a given contract. Just like the contract's creator, these addresses can be saved and used to correlate different events on the blockchain.

Again, the attackers can answer with lateral movement: new EOAs, new contract instances.

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

Smart contracts have many tools to reach third party addresses: transfers, function calls, `selfdestruct`, etc.

Function calls can be executed in the context of the recipient contract with `address.call`. Or modify the state of the origin contract with `address.delegatecall`.

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 setup, 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 design of the storage makes it stealthy.

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 across all addresses and the dynamic data generated across 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.

They can combine static and dynamic analyses by bringing out which functions / events are actually triggered and filtering out irrelevant code.

It will add to the previous analyses and weight all the smart contract actions with their frequency. This temporal profile can be compared with other known contracts.

1.3.2. Graph Theory

Graph theory will perform the same type of analysis than statistics while retaining more of the structure of the blockchain.

Indeed, the blockchain can be viewed as a graph with addresses as the nodes and transactions as the vertices. The tricky part is to decide which specific metric will be used on nodes and vertices.

Even simple labeling schemes, like the transaction amount, will help to inspect the flow of cash & tokens. Graph analysis can also be used to cluster the address space and show the similarities between contracts.

To fool these meta indicators, attackers may add legitimate use & traffic to their contracts.

1.3.3. Symbolic Fuzzing

Standard dynamic analysis will explore only a few execution paths during fuzzing. Even the historical log of transactions will not show all the possible interactions with a contract.

The goal of symbolic analysis is to test all the execution branches and make the other detection techniques more exhaustive.

Symbolic testing has been adapted to Ethereum **HoneyBadger** leverage symbolic testing to explore all the execution paths.

This technique has known flaws: in particular, the number of conditional branches can be exponantially increased, leading to **path explosion**.

1.3.4. Machine learning

Machine learning can be used to achieve all of the above.

The ML models add a layer of abstraction that make the detection inherently more robust to small variations and improvements from the attackers. They will also find new samples even when they were not exactly accounted for.

Tricky attackers may try and poison the models or flood the inputs with irrelevant data.

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`.

This categorization is very generic: since the evasion tactics leave footprints on all the data, all the analysis tools 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.

Malicious smart contracts

Analysis

- **Dynamic**
  - Execution Traces
  - Event Logs
  - Execution Metadata
  - State Changes
- **Hybrid**
  - Statistics
  - Graph Theory
  - Symbolic Analysis
  - Machine Learning
- **Static**
  - Source code
  - Bytecode
  - Creation Metadata

Evasion

- **Redirection**
  - Selector Collisions
  - Hidden Proxy
- **Obfuscation**
  - Packing
  - Hidden State
  - Hiding Behind Proxies
  - Hiding In Plain Sight
- **Poisoning**
  - Living Off The Land
  - Sleep Minting
  - Event Poisoning
- **Morphing**
  - Lateral Movements
  - Red Pill
  - Logic Bomb
- **Spoofing**
  - Faking Stds
  - Variable Shadowing
  - Bug Exploits
  - Sybils

Taxonomy of the detection & evasion techniques

EVASION TECHNIQUES

3. SPOOFING

Spoofing is the art of disguising malicious entities to appear common and harmless.

3.1. FAKE STANDARD IMPLEMENTATION

3.1.1. Overview

Such contracts borrow the function and class names from industry standards (OpenZeppelin, ERC, etc), 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 functions behave differently than ERC-20 / 721 / 1155

Most often, the code will be very close to correct and vary only on key aspects.

3.1.2. Evasion Targets

Users

Few users actually check the code, so having a valid front is enough.

Block explorers

For now, block explorers have fixed models for proxies: they will show the address matching the ERC standards even if the contract actually uses another address.

3.1.3. Samples

Fake EIP-1967 Proxy

The **standard EIP-1967** has pointers located in specific storage slots. In particular, slot number `0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc` holds the address of the logic contract.

These pointers can be kept null or target a random contract, while the proxy actually uses another address. A minimal example was given at **DEFI summit 2023**:

```
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 ERC storage slots
- Events: changes to the address of the logic contract should come with an `Upgraded` event
- Bytecode: the implementation of known selectors can be checked against the standard's reference bytecode

3.2. VARIABLE SHADOWING

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: a keyword is defined a second time for 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. Only the bytecode makes a clear difference.

3.2.2. Evasion Targets

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

Users

The source code is even closer to a legitimate contract: even with a large userbase, it is likely to fly under the radar.

Reviewers

The interpretation of the source code is subtle, and reviewing the bytecode is very time consuming. So even people with the required skills may not have enough incentive to check the code in depth, outside of paid audits.

3.2.3. Samples

Attribute Overwriting

In section *3.2.2*, the paper **The 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

Source Code

While subtle for the human reader, tools can easily scan the sources for duplicate definitions and polymorphism.

Since the whole point is to advertize for a functionality with the sources, they will be available. However, the bytecode does not provide any information on this class of evasion.

This is where static analysis tools like **Slither** shine. It has a **specialized detector for keyword shadowing**.

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

Security Tools

Honeypots are meants to trigger alerts in popular tools to make their users think they found a vulnerable contract.

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;
    }
}
```

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**.

3.4. SYBILS

3.4.1. 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 normal service to hijack it later

Bots have been leveraged to generate trading activity for several tokens: DZOO, oS-HIB, oDOGE, GPT, and SHIBP at least. For instance, the **case study of the DZOO campaign** shows how it used bot EOAs to pump the price of its token.

These techniques are an **active area of research** and would require an entire study. They will not be covered in this document.

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:

- global variables: these variables may have special values in test environments
  - block.basefee: 0
  - block.coinbase: 0x0000000000000000000000000000000000000000
  - tx.gasprice: large numbers, higher than 0xffffffffffffffff
- other contracts: by probing third party contracts, it can assess whether the blockchain mainnet is available

4.1.2. Evasion Targets

Wallets

Wallets often perform a simulation of the transaction before committing. The whole point of this method is to pass these tests and bait the end-user.

Security Tools

Automatic tools will likely not fuzz the coinbase or other global variables. So the dynamic analysis may follow the "harmless" branch and not inspect the actual behavior of the contract on the mainnet.

On the other hand these unusual checks stand out when reviewing the code.

4.1.3. Samples

The contract FakeWethGiveaway mentioned in **the Zengo article** 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

Control Flow Graph

The flow of each function, especially the conditional jumps, can be parsed from the bytecode. And further analyzed to find the conditional jumps that depend on the global variables mentioned above.

Differential Testing

Another approach is to use the historic transactions as invariants. Replay them and fuzz the global variables with blank data, to see if it changes the behavior.

The transactions can be compared at a high level thanks to their gas consumption.

4.2. LATERAL MOVEMENT

4.2.1. Overview

After being detected, attackers can either improve their scheme... Or just rinse and repeat! This is a very basic and widespread method.

More specifically, attackers can just:

- create new EOA addresses
- deploy several instances of their contracts

4.2.2. Evasion Targets

Block Explorers

Many block explorers allow users to **tag addresses**, especially scams.

This is a manual process, so new addresses have to be discovered and tagged, even exact duplicates.

User Tools

This simple trick will get attackers past the blacklists of wallets and firewalls, for a time.

4.2.3. Samples

Fake tokens have been deployed in numerous phishing scams. This particular USDT variant has **412 siblings in ETH**.

4.2.4. Detection & Countermeasures

Bytecode

Signatures of the attacking contract can be indexed in a database, so that when a new sample surfaces it will be instantly found.

Graph Analysis

The secondary addresses will most likely interact with their siblings / parent at some point. In particular the collected funds may be redirected to a smaller set of addresses for cashout.

Graph analysis would propagate its suspicions from parent to child nodes.

4.3. METAMORPHISM

4.3.1. Overview

So far, we've seen how to (re)deploy contracts at random addresses. Given the size of the address space, there is close to no chance a contract is deployed in place of another one. So contracts are usually thought as immutable.

However, the more recent opcode `CREATE2` has a deterministic / known outcome. It can be computed offline with:

```
1   address(
2       uint160(                        // downcast to match the address type.
3           uint256(                    // truncate the upper digits.
4               keccak256(              // compute the CREATE2 hash using 4 inputs.
5                   abi.encodePacked(   // pack all inputs to the hash together.
6                       hex"ff",        // start with 0xff to distinguish from RLP.
7                       sender,         // address of the contract calling CREATE2.
8                       salt,           // some arbitrary salt.
9                       hash            // the Keccak hash of the code to deploy.
10                  )
11              )
12          )
13      )
14  );
```

This exerpt is adapted from **0age's API**. This tool allows to redeploy contracts at the same address in 3 transactions (A, B and C in the diagram):
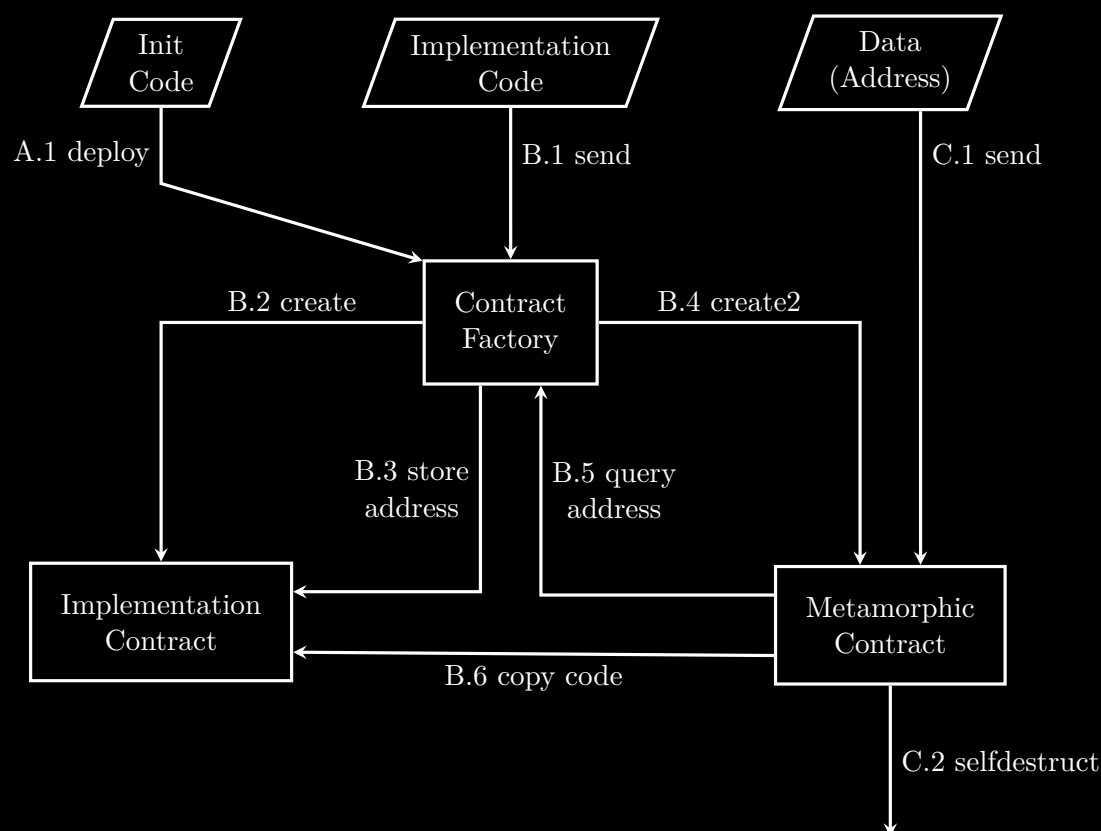
These steps have been outlined by Michael Blau in the article following the **publication of his detection tool**. This technique has many variants, in particular the init code in the API can be tweaked.

This init code is a small creation bytecode that is deployed using `CREATE2`. The same init code is used on every deployment of the metamorphic contract, which guarantees that it will be published at a fixed address. The init code then copies the actual implementation code from another location giving birth to a new variant of the metamorphic contract.

4.3.2. Evasion Targets

Users

Few users are aware that contracts can change. Checking a project once is already demanding, it is even less likely users will double check later.

Reviewers

Metamorphism allows to turn a totally legitimate contract into anything. The first version may pass all the security checks, even though the presence of `CREATE2` and `SELFDESTRUCT` may raise some concern.

4.3.3. Samples

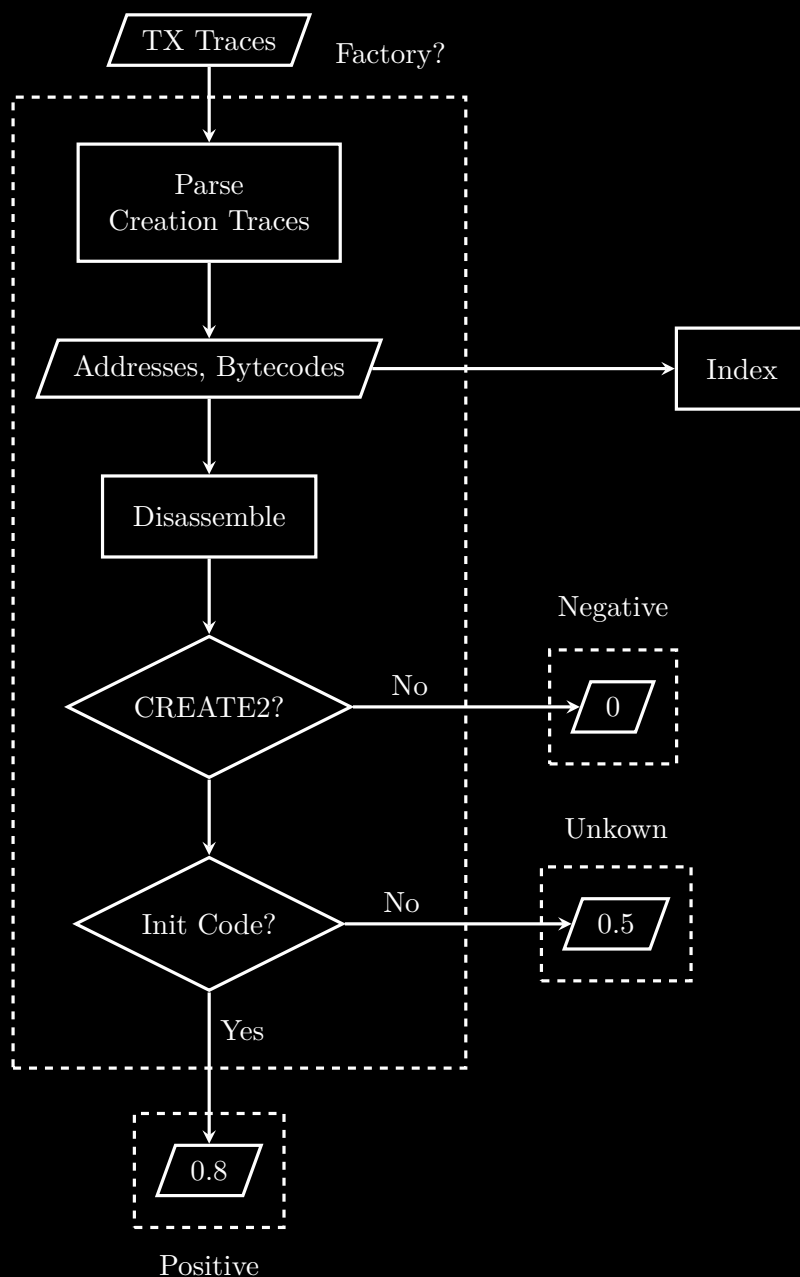Metamorphism is commonly used by `MEV bots`.

0age has also deployed several demonstration contracts, even **on the mainnet**.

4.3.4. Detection & Countermeasures

Instead of characterizing a given address, it is more efficient to detect the mutation as it happens on each of the 3 key transactions depicted in the overview 4.3.

On Factory Deployment

The process can be detected as soon as the **factory deployment**. The bytecode of the deployed contracts can be disassembled and scanned for signs of metamorphic code.

TX Traces

Factory?

Parse
Creation Traces

Addresses, Bytecodes → Index

Disassemble

CREATE2? — No → 0

Negative

Init Code? — No → 0.5

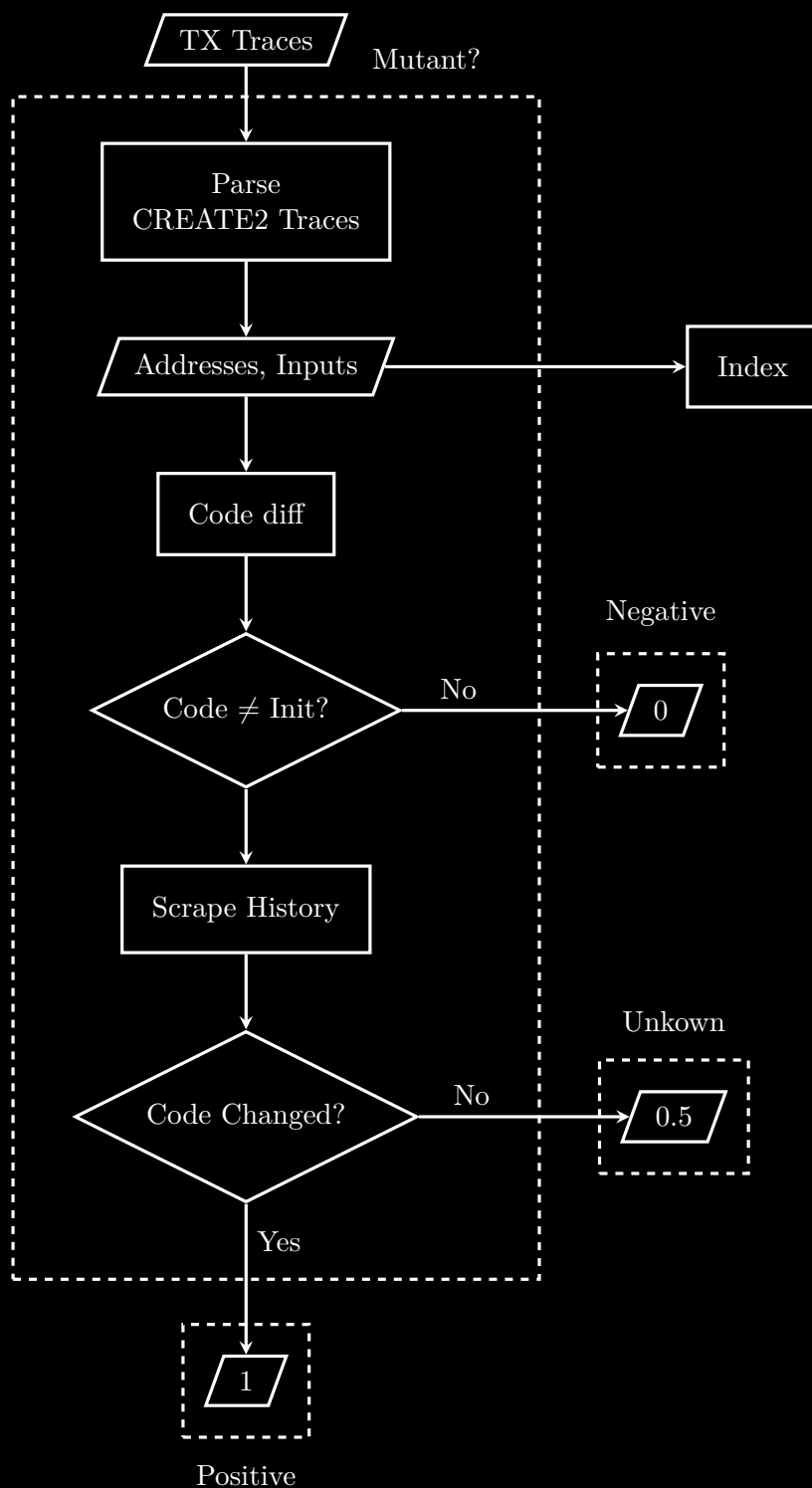Unkown

Yes → 0.8

Positive

On Mutant Metamorphing

The mutation is the most telling transaction, as you can see **in this example**.

The detection is very similar to the process described in the previous section. Instead of looking for the indicators in the bytecode, they can directly be viewed in the execution traces.

At this stage the address of the morphing contract is known, a simple code-diff will give a definitive answer.

Mutant?

TX Traces → Parse CREATE2 Traces → Addresses, Inputs → Index

Addresses, Inputs → Code diff

Code diff → Code ≠ Init?

Code ≠ Init? → No → Negative (0)

Code ≠ Init? → Scrape History

Scrape History → Code Changed?

Code Changed? → No → Unkown (0.5)

Code Changed? → Yes → Positive (1)

On Mutant Suicide

Self-destructions are fairly rare and can be the triggering factor of further analyses. They can be directly detected using traces.

4.4. LOGIC BOMB

4.4.1. 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
- patterns in the input data: typically, the execution can depend on the address of the sender

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.

4.4.2. Evasion Targets

User Tools

Just as the red-pill bypassed wallets 4.1, logic-bombs may fool other tools.

For example, the past transactions listed in a block explorer may give a false sense of security. There is no guarantee that similar calls will result in the same results in a different context (different sender, later time, etc).

Honeypots tend to fail once there is enough transaction records to show that the vulnerability is not exploitable. However, a malicious smart contract may only need to perform it's evil actions in a fraction of the transactions it processes. These failed attempts could be flooded in attractive promises of gain as shown by other past transactions.

Security Tools

Most likely the fuzzing of security tools will remain in the space where the malicious functionalities are disabled. **Path explosion** is also designed specifically to break the symbolic analysis of code in general.

4.4.3. Samples

To our knowledge, this technique is a speculation and has not yet been witnessed in Web3.

4.4.4. Detection & Countermeasures

Fuzzing

Here, the probability of detecting such tricks depends of the extent of the input space covered by the tests. Security tools should fuzz the metadata of the transactions too.

Opcodes

Scanning the bytecode for unusual opcodes may be enough to uncover logic-bombs.

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

Code Reviewers

A single line can compromise the whole codebase, so the reviewing process is very laborious and slow. Attackers stuff the code to overwhelm security auditors with the sheer volume of code.

Security Tools

Unrelated data also lowers the efficiency of ML algorithms: adding valid code will increase the chances of the contract to be classified as harmless.

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, but:

- it is easy to measure, with certainty
- the codebase is always large when this technique is used
- reference implementations are shared and reused, they provide a solid range

Execution Traces

The proportion of the code actually used can be computed by replaying transactions.

It is important to *replay the past transactions* and **not** perform new tests. Indeed, testing all the functions would skew the statistics on mainnet usage.

5.2. HIDING BEHIND PROXIES

5.2.1. Overview

Malicious contracts simply use proxy standards like the **EIP-1967** specifications to split the code into proxy and logic contracts.

5.2.2. Evasion Targets

Block Explorers

Even when redirecting to malicious contracts, the proxy contracts themselves are often standard. Block explorers will validate them and give a false sense of legitimacy.

Users

Most users rely on block explorers to trust contracts.

Reviewers

The source code for the logic contract will most likely 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 of `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 should be further analyzed

5.3. HIDDEN STATE

5.3.1. Overview

The used storage slots are not explicitly listed: data can be slipped in the huge address space of the storage without leaving a public handle.

- initialization: the constructor 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, since 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** 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

Gas Consumption

Storing data on the blockchain is a **very costly operation**. If nothing else, changes to the storage can be detected through gas consumption, especially when writing to empty / unsued slots.

5.4. PAYLOAD PACKING

5.4.1. Overview

For software executables, packing applies a combination of encryption / encoding / compression on a binary. These operations are reversed during execution. Originally, the purpose was to spare secondary memory and make software more compact.

This motivation **still stands on the blockchain**, where processing and storage are especially costly. Several schemes for compression are being studied, even on the **EVM level**.

These techniques could also be leveraged to harden contracts against reverse-engineering. Both data and / or code can be packed, by the contract itself, a proxy or a web app.

Unpacking can be performed either by the contract itself or by a proxy.

5.4.2. Evasion Targets

Block Explorers

With the input data packed in the transaction history, making sense of past events is harder.

Security Tools

All the known patterns and signatures will fail on the packed data.

Security Reviewers

Interacting with a packed contract may require additional layers of (un)packing to handle the input and outputs. If the (byte)code is packed, static analysis will be significantly slowed too.

5.4.3. Samples

To our knowledge, this technique is a speculation and has not yet been witnessed in Web3.

5.4.4. Detection & Countermeasures

Entropy

Usually, these obfuscation schemes can be detected by measuring the entropy. This is harder to implement in this context because the blockchain makes extensive use of hashing algorithms, which are high entropy.

5.5. ANTI-REVERSING

5.5.1. Overview

Smart contracts hold and process key information they may want to keep private:

selectors as a whole, the interface reveals the parents of a given contract and gives an idea of its use

addresses addresses can designate the target of a hack, or recipients of a coin mixing operation for example

calldata it is a rich source of information that be used to understand the behavior of a contract

Anti-reversing techniques aim at concealing or distorting these informations. And they can go further by obfuscating the contract logic and generally interfering with RE tooling.

The application of such anti-reversing methods serves multiple purposes. It can protect secret data, make it harder to find and / or exploit vulnerabilities, or hide the contract's intended functionality.

However, obfuscation is rarely used with good intentions: it is a red flag for security researchers. Even though these methods hinder analysis, they may in turn motivate it!

5.5.2. Evasion Targets

Tools

Static analysis often employ common and imperfect methods that are both known and can be gamed.

Reviewers

Malicious agents can try can and delay / dissuade security researchers by obfuscating their smart contracts.

MEV Researchers

MEV researchers are very protective of their secret sauce: to prevent their competition from gaining insights, the bots are sometimes obfuscated.

5.5.3. Samples

Scrambling Constants

A basic technique to extract data from a contract is to disassemble it and look for `PUSH`. `PUSH4` for selectors, `PUSH20` for addresses, etc.

The MEV searcher DeGatchi mentions a few tricks **in his serie on obfuscation**.

- several values can be packed into larger types, for example selectors into a `bytes32`
- data can be split in several parts and later reconstructed
- real values can be computed from shifted / offset / arithmetically modified values, which are the only ones directly visible

For example an address could be hidden in a bytes32:

```
1  PUSH32 0x699694baab565c1557ace8cade5a7c71deadbeef00
2  PUSH1 0x08
3  SHR
```

Selector Obfuscation

The naïve parsing of the contract of **this MEV bot** will only yield `6c0ffcda` and `cabcfc90`.

But the selector `0xf5c58cda` called to **front-run the hack on Wise Lending** does not appear in clear. Actually it's totally ignored, the fallback is called and the value `0xf5c58cda` is not used.

Keeping Key Data Out Of The Bytecode

The exploit contract of DappSocial didn't have the address of its target in the bytecode.

Even if someone identified the contract as malicious, it was very hard to guess the target in the 108s before the **attack transaction**.

6. POISONING

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

6.1. EVENT POISONING

6.1.1. Overview

Events have the underlying implication that some change happened and the blockchain state evolved accordingly.

It is actually possible to trigger events without their side effects or with mismatching effects. For instance, by setting the amount to 0 on the ERC20 `tranfer` it is possible to trigger `Transfer` events without moving any token!

Actually, all standards and events could potentially be hijacked.

6.1.2. Evasion Targets

Users

Many users don't double check events, especially not when they come from well-known tokens / contracts.

6.1.3. Samples

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

The Forta network **detected the transfer events of null amount**.

6.1.4. Detection & Countermeasures

Overall process
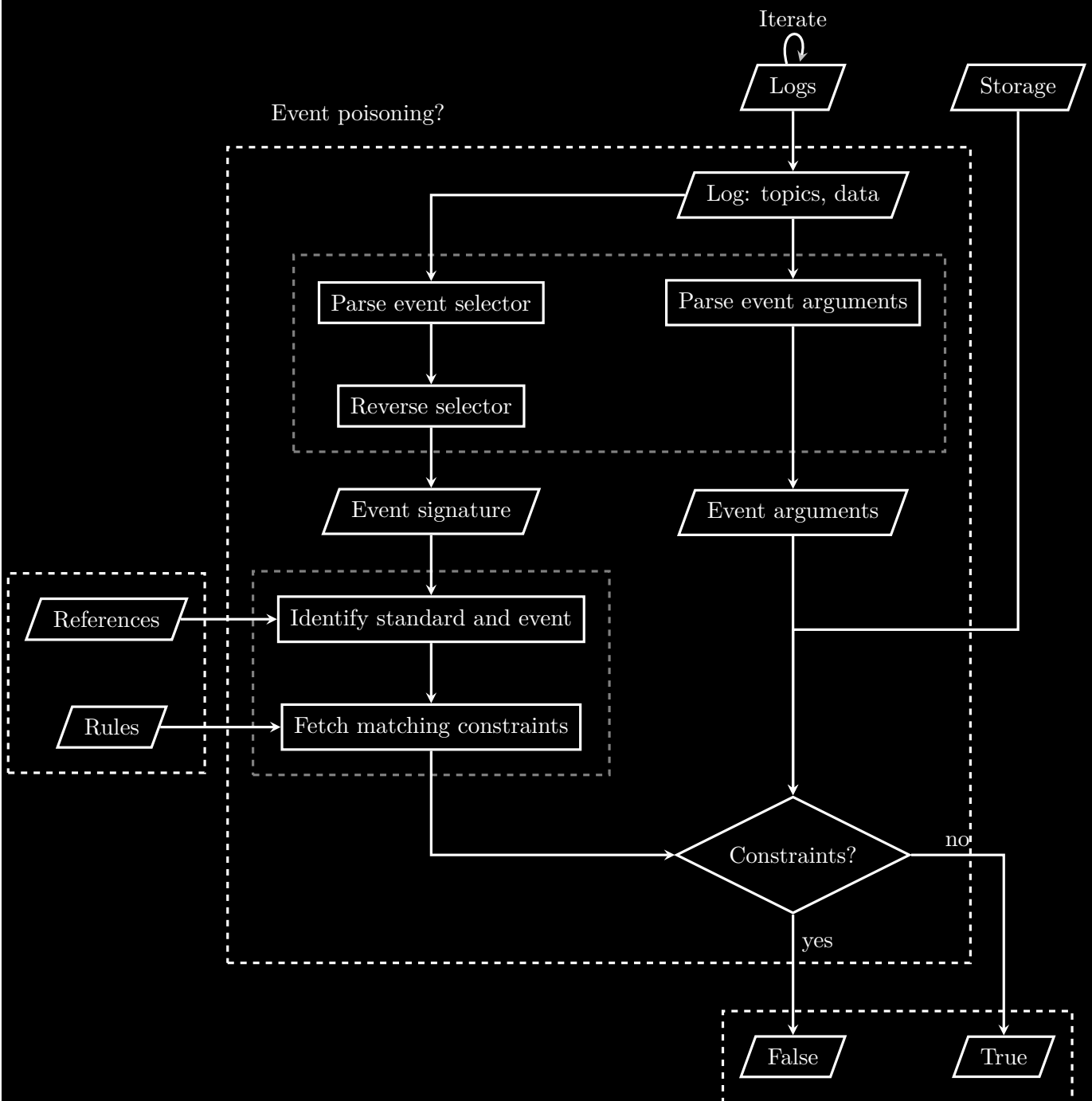
Parsing the logs

First the logs have to be parsed and decoded from the transaction topics and data. Like functions, events have selectors which can be reversed with rainbow tables.

Constraints on the events

The idea is to define constraints on the arguments of all the standard events. For example, valid ERC20 `Transfer` events would have a constraint *strictly greater than zero* on the `amount` argument. A change to the implementation address of a proxy should trigger a `Upgraded` event, etc.

Here the decision block has a black & white output, but in reality it would be a probability. This probability depends on the event and the constraint that was broken, so it cannot be shown in this diagram as a generic metric for all the cases.

The overall output of the process is the **conflation of the probabilities on each event**.

Building the referential

The reference is a database indexing the selectors of all the standards and matching them with the event signature.

6.2. LIVING OFF THE LAND

6.2.1. Overview

Living off the land means surviving on what you can forage, hunt, or grow in nature. For malware, it means using generic, OS-level, tools to compromise a target.

For smart contracts, it could mean:

- taking advantage of callbacks to run malicious code
- using factory contracts to deploy evil variants

The more complex the protocol, the more facilities they will offer for attackers.

6.2.2. Evasion Targets

Potentially, this category of evasion could bypass many layers of defense: since a significant part of the exploitation runs in legitimate contracts, their authority will most likely escape detection.

6.2.3. Samples

To our knowledge, this technique is a speculation and has not yet been witnessed in Web3.

7. REDIRECTION

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

7.1. HIDDEN PROXY

7.1.1. Overview

Hidden proxies redirect the execution to another contract just like standard proxies, except that they pretend not to.

Apart from this redirection trick, the rest of the contract code can be anything: a token, ERC-1967 proxy, etc. There are two cases:

- the contract inherits from a reference proxy contract: the expected implementation will serve as a bait and another logic contract is used in practice
- otherwise the delegate contract adds hidden functionalities, like a backdoor

Just like proxies, a common way to achieve this is to performs `delegatecall` on any unknown selector, via the fallback function. In its simplest form, the fallback would just use another address for the logic contract. More sophiscated attackers will chain proxies or combine this trick with other evasion techniques like 3.2.

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.

Block Explorers

Block explorers can detect standard proxy patterns and show the corresponding logic contract. Here, the shown implementation is not used and the explorers are actually misinforming their users.

Security Tools

The malicious code is not directly accessible and the tools may end up analysing the legitimate implementation instead.

The actual logic address can be obfuscated or even missing from the bytecode. Transaction tracing is the most reliable inspection tool in this case, and it is not always available.

7.1.3. Samples

A malicious fallback can be inserted into an expensive codebase:
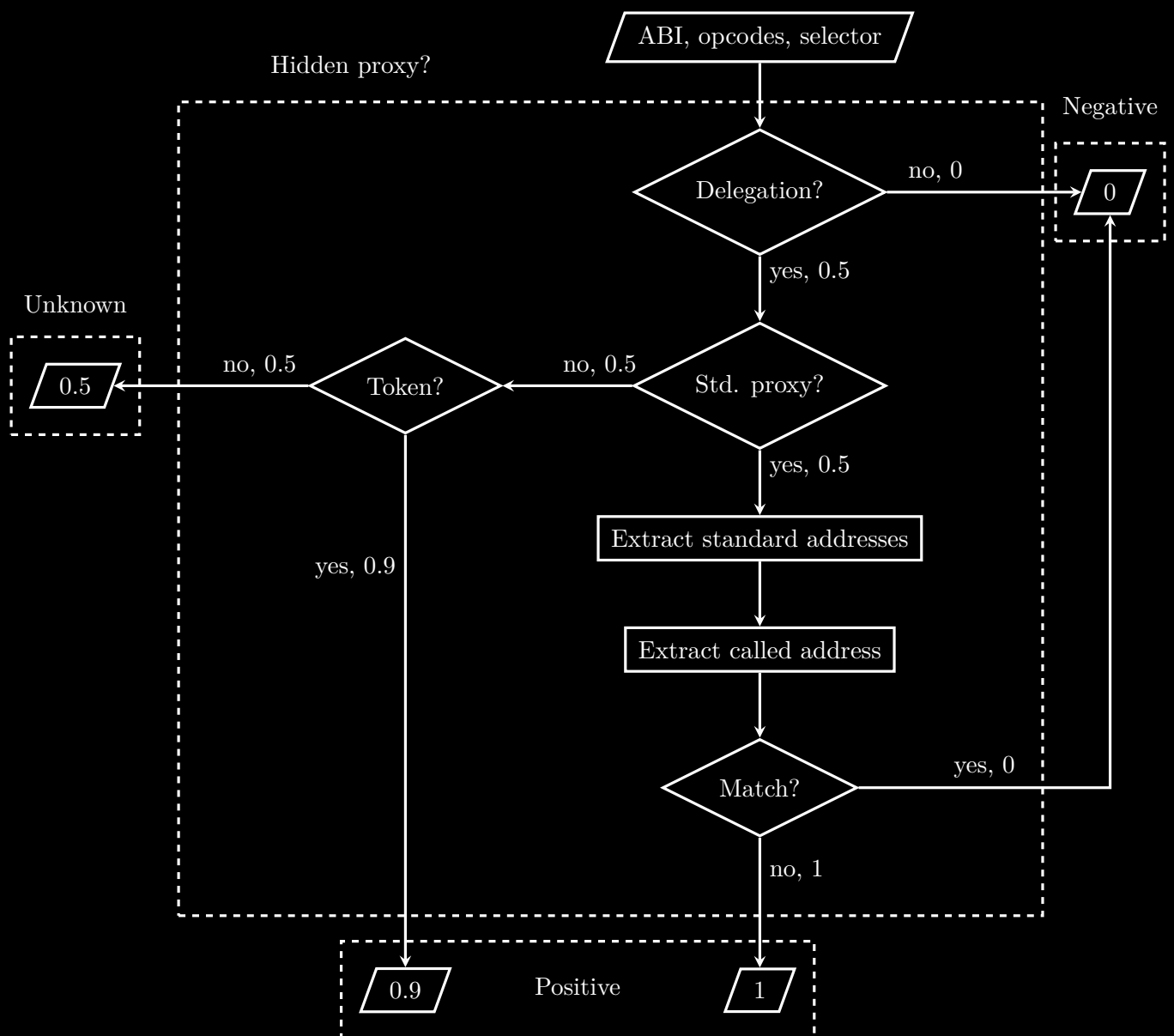
```
1  fallback () external {
2      if (msg.sender == owner()) {
3          (bool success, bytes memory data) = address(0
               x25B072502FB398eb4f428D60D01f18e8Ffa01448).delegateCall(
4              msg.data
5          );
6      }
7  }
```

7.1.4. Detection & Countermeasures

Overall process

This scheme is restricted to two subclasses of the hidden-proxies: standard proxies that don't follow the specifications and tokens that act as proxies. It can be extended and improved upon.

Delegation

Delegation can be detected by comparing the selector from the transaction data with contract's interface.

The contract's interface itself can be extracted from the bytecode, in the hub section of the contract.

Standard Proxy & Token

Once the interface is extracted from the bytecode, it can be compared with known interfaces. In particular tokens and proxies have well-known and constant interfaces.

Implementation addresses

The implementation address can be retrieved from the storage of standard addresses. It is stored at a fixed slot for each standard:

```
1  LOGIC_SLOTS = {
2      # bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)
3      'erc-1967': '360894
           a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc',
4      # keccak256("org.zeppelinos.proxy.implementation")
5      'zeppelinos': '7050
           c9e0f4ca769c69bd3a8ef740bc37934f8e2c036e5a723fd8ee048ed3f8c3',
6      # keccak256("PROXIABLE")
7      'erc-1822': '
           c5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7',}
```

The address to which the transaction call was redirected can be identified in the traces. Or it can be parsed from the bytecode and / or the transaction data depending on the logic of the contract.

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**, 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** has a **very helpful diagram**.

APPENDICES

8. DETECTION MODULES

8.1. PREPROCESSING

Before being processing by the detection bot, the mainnet data has to be parsed and augmented.

All the algorithms detailed in the part III take preprocessed data as input.