

Matildapp: Multi Analysis Toolkit (by IdeasLocas) on DAPPs

Ideas Locas – Discovery Unit
CDO - Telefónica
ideaslocas@telefonica.com

Summary

Web3 is a paradigm that has burst into the digital world with force. Millions of transactions are performed on different types of blockchain. The importance of smart contracts on blockchain, such as Ethereum, is gaining greater relevance due to the finances being managed. A single error or security breach can cost millions of dollars, making cybersecurity a vital factor. This paper introduces the tool Madildapp, which enables conducting DAST and SAST tests to evaluate the security of smart contracts and other elements within the Web3 value chain (such as DAPPs themselves). It is an innovative tool that aggregates different types of tests oriented towards different types of elements, including bytecode, pure source code, and the contract in its own execution through dynamic tests. Madildapp is an all-in-one modular implementation that will help the community to improve the tool.

1. Context tool

In our modern, interconnected world, the concept of Web3, also known as the decentralized web, represents the next significant shift in Internet technology. Web3, underpinned by blockchain technology and smart contracts, offers unprecedented decentralization, transparency, and user sovereignty possibilities. However, with these new possibilities come new challenges – one of the most crucial is security. Web3's decentralized nature eliminates central points of failure typical in Web2 applications, leading many to view it as inherently more secure. However, the security dynamics in Web3 are different, and a unique set of vulnerabilities has emerged. The secure design, development, and operation of Web3 applications and platforms have become crucial skills in the rapidly evolving digital landscape. It's no longer sufficient to build on top of blockchain technologies; developers, cybersecurity professionals, and even end-users must grasp the principles of securing these systems.

This paper presents Madildapp, a novel tool designed to evaluate the security of smart contracts and other components within the Web3 value chain, including decentralized applications (DAPPs). Madildapp enables the execution of Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST) to assess the security of smart contracts. This innovative tool aggregates various testing modalities, catering to different elements, including bytecode, pure source code, and contract execution through dynamic testing.

2. Web3 Vs Web2

The Web3, also known as the decentralized web, is a paradigm that seeks to revolutionize the way we interact with the Internet and online services. The Web2, which is the web we know today, has several limitations and problems that Web3 aims to resolve. Some of the reasons why Web3 is needed are:

Centralization: Web2 is controlled by a small group of companies and governments that have access and control over our online data and activities. Web3 seeks to decentralize the web, giving users more control and reducing dependence on intermediaries. Privacy: Web2 has proven to be vulnerable to surveillance and personal data theft. Web3 uses advanced cryptography and blockchain technologies to protect user privacy. Interoperability: Web2 is fragmented into data silos and applications that do not communicate with each other. Web3 aims to create a more interconnected and open ecosystem, allowing users to benefit from interoperability between applications and services. Monetization: Web2 is based on business models that exploit user data and turn them into products. Web3 seeks to create fairer and more transparent business models, giving users more control over their data and rewarding them for their participation.

3. Matildapp

'Matildapp' (Multi Analysis Toolkit -by IdeasLocas- on DAPPs) is an Open Source project providing a framework for Web3 environments in the field of cybersecurity and pentesting. The tool offers

modules to interact with different types of blockchains and to conduct SAST and DAST evaluations of potential vulnerabilities in smart contracts.



```
python sc.py

[~*~] matildapp: Multi Analysis Toolkit (by IdeasLocas) on DAPPs  [~*~]
[~*~]   Created by: IdeasLocas(With Love!)   [~*~]

[+] Starting the console...
[*] Console ready!

sc $ > hello world!!
```

Figure 1: 'Matildapp'

The architecture of 'matildapp' is modular. It has over 15 modules that allow detecting various vulnerabilities in smart contracts by providing the code in Solidity language or, in some cases, by providing only the contract's bytecode. The tool's architecture is shown in the following diagram:

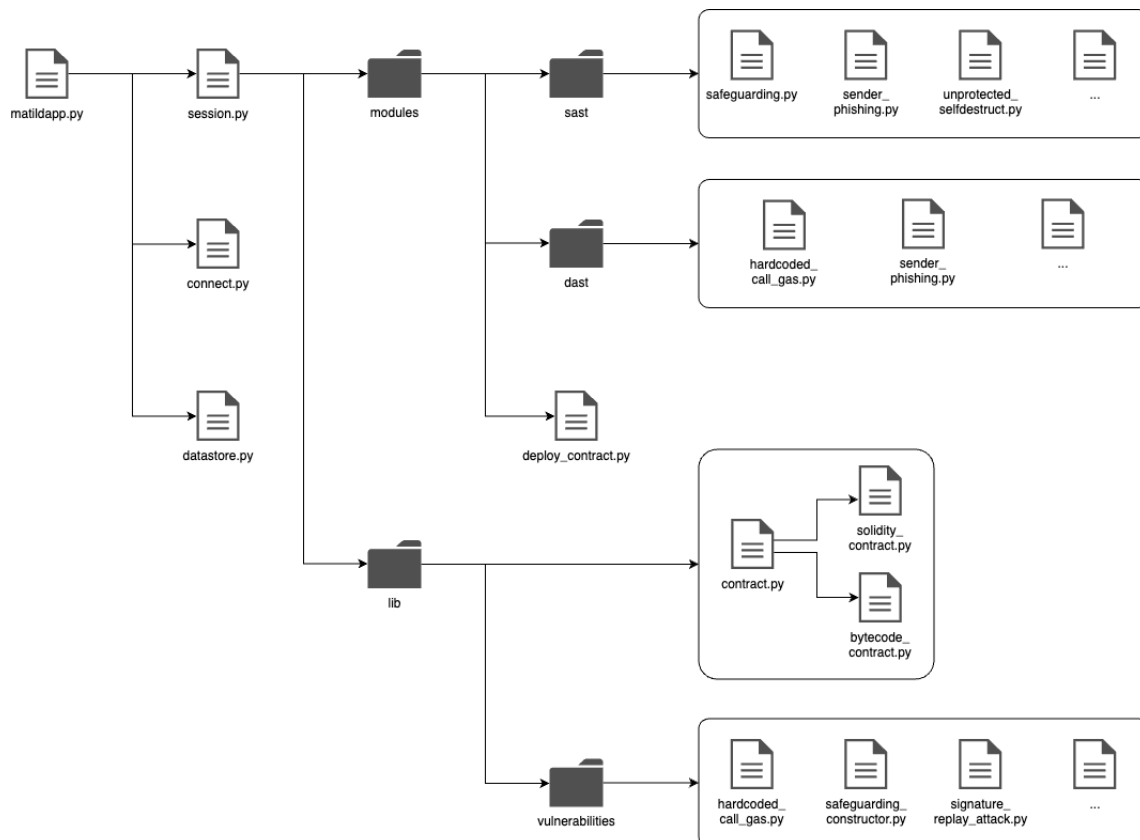


Figure 2: Modular architecture of 'matildapp'

During the execution of 'matildapp', an interactive session is started where you can:

- Connect to a blockchain to interact directly with it, for example, compiling and deploying contracts.
- Load different modules for SAST and DAST analysis.
 - SAST analysis modules require the smart contract code, though some modules can also work with just the contract bytecode.
 - DAST analysis modules connect directly to a specified contract address and proceed with the analysis.
- Access a directory with example modules to allow anyone to create their own modules.
- Use a vulnerability catalog based on the *Smart Contract Weakness Classification* (SWC) to facilitate classification.
- Utilize a datastore object for data storage, allowing management of various information, such as wallet management or adding results from the modules of each contract being analyzed.

The different modules that make up 'matildapp' can be loaded using the *load* command.

```

sc $ > [bytecode]> load modules/
modules/dast/          modules/examples/          modules/utlis
modules/deploy_contract modules/sast/
sc $ > [bytecode]> load modules/dast/
modules/dast/hardcoded_gas_call      modules/dast/timestamp_dependence
modules/dast/sender_phishing         modules/dast/unprotected_selfdestruct
sc $ > [bytecode]> load modules/sast/
modules/sast/array_length_manipulation/      modules/sast/signature_replay_attacks/
modules/sast/hardcoded_gas_call/             modules/sast/timestamp_dependence/
modules/sast/right_to_left_override_character/ modules/sast/unprotected_selfdestruct/
modules/sast/safeguarding_constructor/       modules/sast/untrusted_delegatecall/
modules/sast/sender_phishing/
sc $ > [bytecode]> load modules/sast/sender_phishing/
modules/sast/sender_phishing/bytecode      modules/sast/sender_phishing/solidity
sc $ > [bytecode]> load modules/utlis
modules/utlis
sc $ > [bytecode]> load modules/utlis

```

Figure 3: 'matildapp' Modules

The commands available in the tool are as follows:

```

[~*~] matildapp: Multi Analysis Toolkit (by IdeasLocas) on DAPPs [~*~]
[~*~] Created by: IdeasLocas(With Love!) [~*~]

[+] Starting the console...
[*] Console ready!

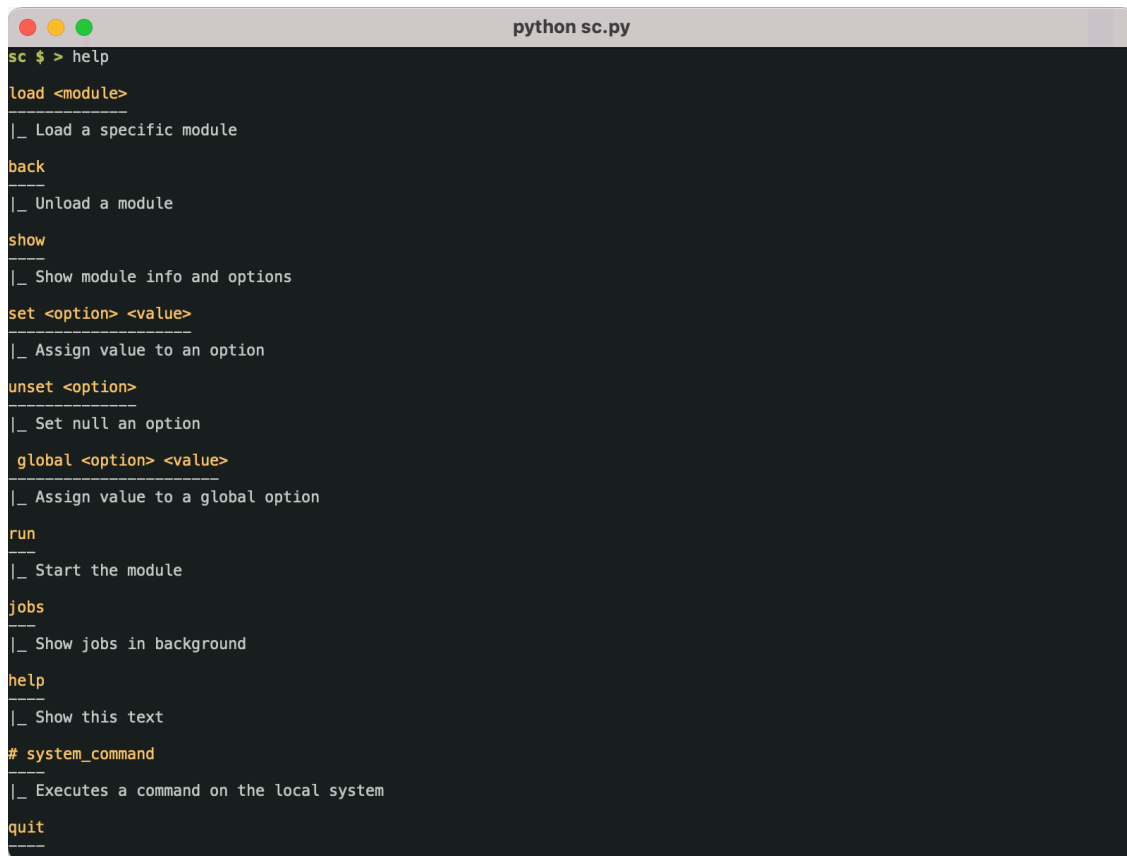
sc $ >
back      datastore  global      jobs      quit      set      unset
connect   disconnect  help       load      run       show
sc $ >

```

Figure 4: Available commands in the tool

- *help*: Provides guidance on what the commands do.
- *load*: Loads a module for configuration and execution.
- *back*: Reverses the previous operation, returning control to the main prompt.

- *jobs*: Shows the modules that generate threads and are in execution.
- *run*: Executes a module.
- *set*: Assigns values to a module's attribute.
- *show*: Displays the options of a module before executing it.
- *connect*: Connects to a blockchain.
- *disconnect*: Disconnects from the blockchain to which it was connected.
- *datastore*: Manages data storage for wallets and contracts.



```
python sc.py
sc $ > help
load <module>
-----
|_ Load a specific module
back
-----
|_ Unload a module
show
-----
|_ Show module info and options
set <option> <value>
-----
|_ Assign value to an option
unset <option>
-----
|_ Set null an option
global <option> <value>
-----
|_ Assign value to a global option
run
-----
|_ Start the module
jobs
-----
|_ Show jobs in background
help
-----
|_ Show this text
# system_command
-----
|_ Executes a command on the local system
quit
-----
```

Figure 5: Description of 'matildapp' commands.

4. Requirements

'Matildapp' is written in Python and uses libraries to work with Web3 such as *pyweb3* and *py-solc-x*. Version management of these libraries is crucial, as well as the Python version used. It has been tested to work in a Python environment with version 3.10.14 and *pyweb3* version 5.31.4 and *py-solc-x* version 2.0.2.

Other versions of Python and the libraries could cause issues preventing the tool from running correctly.

A *requirements.txt* file should be executed the first time the tool is started using *pip install -r requirements.txt*. Again, the *pip* version should correspond to a tested Python version like 3.10.14.

For working in a blockchain test environment, it is also necessary to have a utility that provides this service. Tools like Ganache or Hardhat can be used for this purpose.

5. How to create a 'matildapp' module

Creating a module is straightforward, and an example can be found in the *examples* folder (within the modules directory). This file implements the 'CustomModule' class, which inherits from the 'Module' class in module.py.

This class has a constructor that facilitates the display of the module's information through the 'information' variable. The 'options' variable defines the necessary attributes for the class to be implemented. The 'super' function is then used to transfer the values of 'information' and 'options' to the main class.

When running a 'show' command in the application with this module loaded, the specified options will be displayed and can be assigned with the 'set' command.

```
class CustomModule(Module):
    def __init__(self):
        information = {"Name": "My hello module",
                      "Description": "Test module",
                      "Author": "author"}

        # -----name-----default_value--description--required?
        options = {"message": ["hello world!", "Message for you", True],
                  "option2": [None, "Text description", False],
                  "option3": [None, "Text description", False]}

        # Constructor of the parent class
        super(CustomModule, self).__init__(information, options)

        # Class attributes, initialization in the run_module method
        # after the user has set the values
        self._option_name = None
```

Figure 6: 'CustomModule' class and its constructor

The 'CustomModule' class also implements a method called 'run_module'. This method must be implemented in each module to provide functionality. Functions or even methods can be created in our classes, but the 'run_module' method will be executed when the 'run' command is issued.

For example, a simple action like printing the content of the 'message' variable specified earlier in the module's options can be seen below.

```
# This module must be always implemented, it is called by the run option
def run_module(self):

    print(self.args["message"])
```

Figure 7: run_module' method

Once the module is created, it will be copied to its corresponding category. For example, if it is a static analysis module and requires the contract's Solidity code, it will be placed under `'sast/module_name/solidity.py'`. If it is developed to find the vulnerability within the bytecode, the path would be `'sast/module_name/bytecode.py'`. Similarly, if it is a dynamic analysis module, it would be placed under the `'dast'` modules.

After it is placed in its appropriate directory, the tool will automatically load it during execution.

6. Use cases

Below are several examples demonstrating the tool's capabilities. All use cases have been tested in our own lab environment.

a) Use case: Detecting a possible phishing vulnerability by analyzing source code

In the following example, we will demonstrate the use of the `'sender_phishing'` module. This module identifies improper uses of the `'tx.origin'` variable.

The `'tx.origin'` variable is a global variable in Solidity that returns the address of the wallet that initiated the original transaction. Unlike the `'msg.sender'` variable, which represents the address of the sender of the current call (i.e., the contract or account invoking the function), `'tx.origin'` always shows the address of the user who started the transaction, regardless of how many intermediate contracts have been called.

Below is the code for a smart contract that we want to check, and we proceed to load the module `'sast/sender_phishing/solidity.py'`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.1;

import '@openzeppelin/contracts/access/Ownable.sol';
import '@openzeppelin/contracts/token/ERC20/ERC20.sol';

contract Token is Ownable, ERC20 {
    function Token() ERC20("HOLA","H") public {
        _mint(tx.origin, 10000 * (10**uint256(decimals())));
    }

    function mint(address _to, uint256 amount) public onlyOwner {
        _mint(_to, amount * (10**uint256(decimals())));
    }

    function burn(address _from, uint256 amount) public onlyOwner {
        _burn(_from, amount * (10**uint256(decimals())));
    }
}
```

Figure 8: Example of a smart contract making improper use of `'tx.origin'`

In `'matildapp'`, there is a folder with several examples, and the code used for this example can be found in `'examples/sol/phishing.sol'`. Here, within the `'Token()'` function, you can see the use of `'tx.origin'` when calling the `'_mint()'` function.

Once the module is loaded, we proceed to view its configuration options using the `show` command. In the options, you can see that the only variable to be assigned is the path to the smart contract, as this is a SAST module and will analyze the source code.

```
python sc.py
sc $ > load modules/sast/sender_phishing/
modules/sast/sender_phishing/bytecode modules/sast/sender_phishing/solidity
sc $ > load modules/sast/sender_phishing/solidity
[+] Loading module...
[+] Module loaded!
sc $ > [solidity]> show

Name
-----
|_Sender Phishing Solidity check

Description
-----
|_
SWC: 115
CWE RELATED: CWE-477
Title: Authorization through tx.origin
Description: tx.origin is used for finding the wallet that triggered the original transaction.
              Can be used for Phishing if a wallet calls a malicious smart contract.
              Can be catastrophic if the wallet is a multisig wallet, as it will only store the
              reference to the person who triggered the last vote.

Detection cases: Can occur when tx.origin is used.
1. tx.origin appears in the code
2. tx.origin is in a comment
Final conclusion: 1 && !2

Author
-----
|_@chgara

Options (Field = Value)
-----
|_[REQUIRED] contract = None (Contract path, should be a solidity file)

sc $ > [solidity]> set contract examples/bytecode/sender_phishing.txt
contract >> examples/bytecode/sender_phishing.txt
sc $ > [solidity]>
```

Figure 9: Configuration of the `sast/sender_phishing/solidity.py` module

Once this variable is defined, the module can be launched using the `run` command and we wait to see the result. In this example, the contract is vulnerable.

```
python sc.py
sc $ > [solidity]> run
[+] Running module...
Vulnerability found (surely)
- SWC: 115
- CWE-related: CWE-477
- Title: Authorization through tx.origin
- Effect: Critical
- Description:
  > X Phishing alert
  > The contract uses tx.origin, which can be used for phishing attacks
  > More info: https://swcregistry.io/docs/SWC-115

Vulnerability found (surely)
- SWC: 115
- CWE-related: CWE-477
- Title: Authorization through tx.origin
- Effect: Critical
- Description:
  > X Phishing alert
  > The contract uses tx.origin, which can be used for phishing attacks
  > found in line 9 of contract Token
  > _mint(tx.origin, 10000 * (10**uint256(decimals())));
  > More info: https://swcregistry.io/docs/SWC-115

sc $ > [solidity]>
```

Figure 10: Result of the `sast/sender_phishing/solidity.py` module

b) Use case: Detecting a selfdestruct vulnerability by analyzing bytecode

In the following example, we will demonstrate the use of the `unprotected_selfdestruct` module. This module finds unprotected uses of the `selfdestruct` function.

The `selfdestruct` function (formerly known as `suicide`) is an instruction that allows a smart contract to be removed from the blockchain and transfer the remaining balance to a specified address. By executing `selfdestruct`, the contract is completely removed from the blockchain state, and all future interactions with the contract will be impossible.

In this case, we have the bytecode of the smart contract, and the module to load is ``sast/unprotected_selfdestruct/bytecode.py``.

[illegible]

Figure 11: Bytecode fragment of a smart contract

The example can be found within the examples folder:
`examples/bytecode/unprotected_selfdestruct.txt`.

Similar to the previous use case, here we need to assign the ``bytecode`` variable, which allows inputting the bytecode directly or providing the path to a .txt file containing the bytecode.

```
python sc.py
sc $ >[solidity]> load modules/sast/unprotected_selfdestruct/
modules/sast/unprotected_selfdestruct/bytecode modules/sast/unprotected_selfdestruct/solidity
sc $ >[solidity]> load modules/sast/unprotected_selfdestruct/bytecode
[+] Loading module...
[+] Module loaded!
sc $ >[bytecode]> show

Name
----
|_Unprotected Selfdestruct Solidity check

Description
-----
|_
SWC: 106
CWE RELATED: CWE-284
Title: Unprotected Selfdestruct
Description: SELFDESTRUCT is a dangerous opcode that will destroy the contract when called.
            Also it will transfer all aviable funds to an specified address.
            If not protected this instruction in your code it will be used to destroy the contract.

Detection cases: If the opcode SELFDESTRUCT is present with no protection.
1. selfdestruct(address) appears in the code
2. No access control to the function using it
Final conclusion: 1 && !2

Author
-----
|_@chgara

Options (Field = Value)
-----
|_[REQUIRED] bytecode = None (Bytecode or path to .txt containing the bytecode)

sc $ >[bytecode]> |
```

Figure 12: Configuration of the ``sast/unprotected selfdestruct/bytecode.py`` module

Once this variable is defined, the module can be launched using the `run` command and we wait to see the result. In this example, the contract is vulnerable.

```
python sc.py
sc $ >[bytecode]> run
[+] Running module...

Vulnerability found (potentially)
- SWC: 106
- CWE-related: CWE-284
- Title: Unprotected Selfdestruct
- Effect: Critical
- Description:
  > Δ The contract uses selfdestruct() without any protection. This can lead to the contract being destroyed.
  - More info: https://swcregistry.io/docs/SWC-106

sc $ >[bytecode]> █
```

Figure 13: Result of the 'sast/unprotected_selfdestruct/bytecode.py' module

c) Use Case: Deploying a Contract and Using the Datastore

The following example will demonstrate the deployment of a contract and its storage in the `datastore` object of `matildapp`.

To begin, it is possible to connect to a known blockchain, such as Ethereum, Polygon, or to set up our own development blockchain using tools like Ganache or Hardhat. The `connect` command is used to establish the connection.

```
python sc.py
sc $ > connect
[!] Arguments incorrect: connect provider <rpc_server>
sc $ > connect provider http://127.0.0.1:7545
Connection created with Blockchain
sc $ > █
```

Figure 14: Connecting to a blockchain

To ensure a successful connection, the provider must be specified. In this example, the provider is located on the same machine. The message indicating a successful connection can be observed.

The next step is to load the module for deploying the contract. This is located under the `modules/deploy_contract` module. The available configuration options can be listed, and several are required: the path to the smart contract, the public address, and the private key of the account that will deploy the contract. This account must have sufficient balance to cover the gas fees for the deployment.

```
python sc.py
sc $ > load modules/deploy_contract
[+] Loading module...
[+] Module loaded!
sc $ >[deploy_contract]> show

Name
|_Deploy contract module

Description
|_This module compiles a contract (solidity) and deploys it on a blockchain. You need private key, public address and chain_id.

Author
|_@pablogonzalezpe

Options (Field = Value)
|_ [REQUIRED] contract = None (Contract path in order to compile)
|_ [REQUIRED] address = None (Public address)
|_ [REQUIRED] pkey = None (Private key)
|_ _show_bytecode = true (Show bytecode value)
|_ _show_opcode = true (Show opcodes value)
|_ _show_abi = true (Show ABI value)
|_ _compiler_version = 0.8.4 (Solidity version)

sc $ >[deploy_contract]> █
```

Figure 15: Deploying the smart contract

Another important option to consider is the compiler version, as using an incompatible version could cause issues.

Once all necessary configurations are completed, the module is executed, and we observe the output.

Figure 16: Output of the smart contract deployment

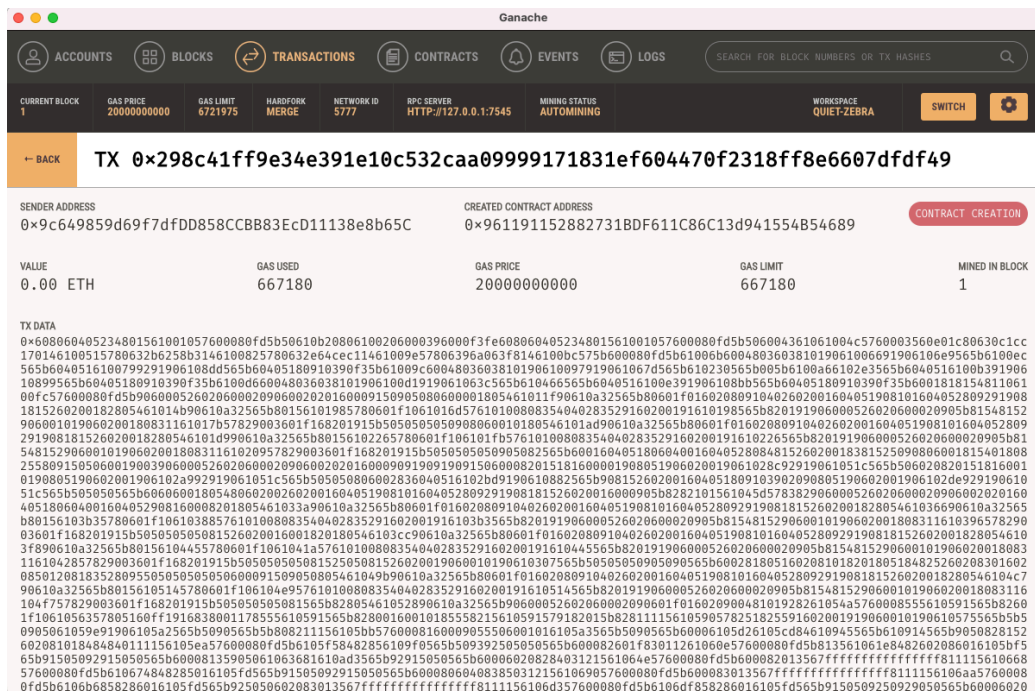


Figure 17: Deployment details in Ganache GUI

Finally, the *datastore* object can be used to check the stored data and utilize this data as input parameters for the *'matildapp'* modules.

To begin working with the *datastore*, using the command without any parameters will display the help with the various options and actions available for its use. The options distinguish whether you are working with a wallet or a contract. The actions are as follows:

- *create*: Creates a new contract or wallet record
- *delete*: Deletes a contract or wallet record
- *show*: Displays the information of a contract or wallet
- *name*: Assigns a new name to a contract or wallet
- *add*: Adds new information to an existing contract or wallet
- *del*: Deletes information from an existing contract or wallet
- *mod*: Modifies the information of an existing contract or wallet
- *set*: Assigns a value to a module parameter
- *save*: Saves a copy of the datastore to disk

```
python sc.py

sc $ >[deploy_contract]> datastore

Datastore help

Usage: datastore <option> <action> <name contract | wallet> [<param> <value>]

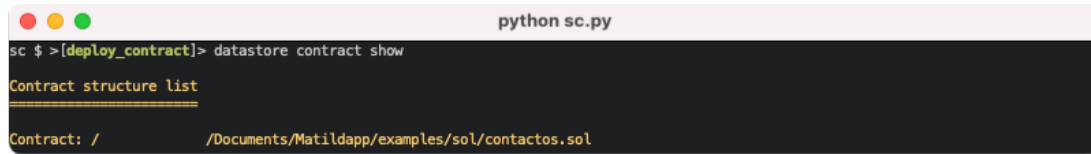
Options:
contract - Manage your (smart) contracts
wallet - Manage your wallets

Actions:
create - create your [contract | wallet] info - Ex: datastore contract create <name>
delete - delete your [contract | wallet] info - Ex: datastore contract delete <name>
show - show your [contract | wallet] info - Ex: datastore contract show [contract_name]
name - modify name [contract | wallet] - Ex: datastore contract name <old_name> <new_name>
add - add info about [contract | wallet] - Ex: datastore contract add <contract_name> <param_name> <value>
del - delete info about [contract | wallet] - Ex: datastore contract del <contract_name> <param_name>
mod - modify info about [contract | wallet] - Ex: datastore contract mod <contract_name> <param_name> <value>
set - set value on module params (fast link) - Ex: datastore contract set <contract_name> <param_module_name> <param_name> (only with module loaded)
save - save your datastore info on filesystem - Ex: datastore save

sc $ >[deploy_contract]> 
```

Figure 18: Using the datastore command

To display a list of stored information about smart contracts, the command ``datastore contract show`` is used.



```
python sc.py
sc $ >[deploy_contract]> datastore contract show

Contract structure list

Contract: / /Documents/Matildapp/examples/sol/contactos.sol
```

Figure 19: List of contracts stored in the datastore

Once the names of the available contracts are known, specific information about each one can be accessed, for example: ``datastore contract show my_contract.sol``.



```
python sc.py
sc $ >[deploy_contract]> datastore contract show / /Documents/Matildapp/examples/sol/contactos.sol
({'address': '0x96119115288273180F611C86C13d941554854689', 'abi': [{'inputs': [{'internalType': 'string', 'name': '_name', 'type': 'string'}, {'internalType': 'string', 'name': '_phoneNumber', 'type': 'string'}], 'name': 'addContact', 'outputs': [], 'stateMutability': 'nonpayable', 'type': 'function'}, {'inputs': [{'internalType': 'uint256', 'name': '', 'type': 'uint256'}], 'name': 'contact', 'outputs': [{'internalType': 'string', 'name': 'name', 'type': 'string'}, {'internalType': 'string', 'name': 'phoneNumber', 'type': 'string'}], 'stateMutability': 'view', 'type': 'function'}, {'inputs': [{'internalType': 'string', 'name': '', 'type': 'string'}], 'name': 'nameToPhoneNumber', 'outputs': [{'internalType': 'string', 'name': '', 'type': 'string'}], 'stateMutability': 'view', 'type': 'function'}, {'inputs': [], 'name': 'retrieve', 'outputs': [{'components': [{'internalType': 'string', 'name': 'name', 'type': 'string'}, {'internalType': 'string', 'name': 'phoneNumber', 'type': 'string'}], 'internalType': 'struct ContactList.Contact[]', 'name': '', 'type': 'tuple[]'}, 'stateMutability': 'view', 'type': 'function'}], 'opcode': 'PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH2 0x820 DUP1 PUSH2 0x20 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH2 0x4C JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4 0xC1CC170 EQ PUSH2 0x51 JUMPI DUP1 PUSH4 0x2B6258B3 EQ PUSH2 0x82 JUMPI DUP1 PUSH4 0x2E64CEC1 EQ PUSH2 0x9E JUMPI DUP1 PUSH4 0x96A063F8 EQ PUSH2 0xBC JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH2 0x6B PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1 PUSH2 0x66 SWAP2 SWAP1 PUSH2 0x6E9 JUMP JUMPDEST PUSH2 0xEC JUMP JUMPDEST PUSH1 0x40 MLOAD PUSH2 0x79 SWAP3 SWAP2 SWAP1 PUSH2 0x8DD JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH2 0x9C PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1 PUSH2 0x97 SWAP2 SWAP1 PUSH2 0x67D JUMP JUMPDEST PUSH2 0x230 JUMP JUMPDEST STOP JUMPDEST PUSH2 0xA6 PUSH2 0x2E3 JUMP JUMPDEST PUSH1 0x40 MLOAD PUSH2 0xB3 SWAP2 SWAP1 PUSH2 0x899 JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH2 0xD6 PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1 PUSH2 0xD1 SWAP2 SWAP1 PUSH2 0x63C JUMP JUMPDEST PUSH2 0x466 JUMP JUMPDEST PUSH1 0x40 MLOAD PUSH2 0xE3 SWAP2 SWAP1 PUSH2 0x8BB JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH
```

Figure 20: Details of a contract stored in the datastore

Similarly, the information of the stored wallets is handled in the same way.