Collateral Use of Deployment Code for Smart Contracts in Ethereum

Monika di Angelo
Eurecom, Sophia Antipolis, France
Technische Universität Wien, Vienna, Austria
monika.diangelo@tuwien.ac.at

Gernot Salzer
Eurecom, Sophia Antipolis, France
Technische Universität Wien, Vienna, Austria
gernot.salzer@tuwien.ac.at

***Abstract*—Ethereum is still the most prominent platform for smart contracts. For the deployment of contracts on its blockchain, the so-called deployment code is executed by Ethereum's virtual machine. As it turns out, deployment code can do a lot more than merely deploying a contract.**

This paper identifies less-anticipated uses of contract deployment in Ethereum by analyzing the available blockchain data. In particular, we analyze the specifics of deployment code used beyond actually deploying a contract in a quantitative and qualitative manner. To this end, we identify code patterns in deployment code by distilling recurring code skeletons from all external transactions and internal messages that contain deployment code. Tracking the use of these patterns reveals a set of vulnerabilities in contracts targeted by skillfully crafted deployment code. We summarize the encountered exploitative cases of collateral use of deployment code and report respective quantities. Example scenarios illustrate the recent usage.

Collateral use of deployment code starts to appear in the middle of 2018 and becomes dominant among contract creations in autumn of 2018. We intend to raise awareness about the less obvious uses of deployment code and its potential security issues.

***Index Terms*—analysis, deployment code, exploit, Ethereum, smart contract**

I. Introduction

Smart contracts on a blockchain can be described as computer programs that run on a peer-to-peer network with the purpose of automating the exchange of digital assets without the need for an external trusted authority. Such assets may be linked to non-digital objects or values.

Fraud. Taking the radical stance that *code is law* in the trustless environment of blockchain-based cryptocurrencies, any feature usage must be deemed rightful. The question arises as to what extent it is acceptable to do whatever can be done, and to utilize available features in any way possible.

Intention or *fair use* are notions the affected community might care about. The rules of the game can be changed if the community decides that specific user actions misuse some (possibly unintentionally) available features. The means to overrule user actions ex post include a hard fork to revert unintended actions, and protocol changes to stop further abuse.

Setting. The most prominent platform for smart contracts still is Ethereum. It has its own mechanism for deploying contracts via so-called deployment code that is executed once to initialize a contract. The mechanics of such a deployment are usually expected to just accomplish that: initialize the new contract and write its bytecode to its address.

Focus. We focus on 'unintended use' of deployment code of smart contracts in Ethereum. Our goal is to determine the actual usage of deployment code, and to clarify what it can do in general.

Audience and Added Value. This paper may be interesting for contract developers to avoid the presented issues, for contract users to understand how secure the contracts are that they use, and for the community to be informed about what to watch out for or where to take precaution.

To this end, we elaborate on deployment mechanics, on vulnerabilities of contracts that are attackable by deployment code, and on found exploit patterns that are summarized in exemplary scenarios. This may contribute to a refined design of new VMs for smart contracts.

Methods. As data set, we extracted all deployment codes that exist on the Ethereum blockchain. We analyze these deployment codes in a quantitative and qualitative manner. Bytecode was analyzed automatically for all available deployment codes, as well as manually for representative patterns we encountered (see section III for details on our code analysis). For the analysis of interaction patterns in deployment code, we first quantify the interaction of deployment codes with other addresses. Then, we distill usage scenarios that make 'unintended use' of deployment code.

Analyzed Data. The activities on the Ethereum chain are usually described in terms of transactions clustered into blocks. This view is too coarse for our purpose since transactions may be composed of several internal messages. These internal messages are caused by contract activity: they are calls, creates or selfdestructs initiated by contracts. Internal messages are essential to understand the activities on the Ethereum blockchain.

The messages were extracted from the traces provided by the Ethereum client `parity` in archive mode in version 2.1.4-beta, with a patch applied to fix a bug regarding `CALLCODE` traces. Our analysis is based on data up to Ethereum block 7 280 000 (Feb 28, 2019).

Roadmap. Section II explicates the mechanics of contract deployment, emphasizing unusual deployment code. We explain

our methods of bytecode analysis in section III. Section IV lists vulnerabilities that we found exploited by deployment code, while exploit scenarios are discussed in section V. We present related work in section VI and our conclusions in section VII.

II. Deployment of Contracts in the EVM

A. Contract-related Terms

EVM. In Ethereum [1], smart contracts are executed by its virtual machine, the EVM.

Accounts. Ethereum distinguishes between externally owned accounts often called *users*, and contract accounts or simply *contracts*. For any account, its data is stored at its address.

Contracts additionally have bytecode stored at their respective addresses. Its code is executed only when receiving a call.

Transactions are signed data packages sent from users to other users or contracts. They are recorded on the blockchain.

Messages are data packages sent from contracts to other contracts or users. They only exist in the execution environment of the EVM and are reflected in the execution trace and potential permanent data changes.

B. Mechanics of a Deployment

For a contract to exist, it needs to be created by another address via the so-called deployment code, which is executed once by the EVM. As part of this deployment, the so-called *deployed code* is written to the code section of the contract's address. The contract exists upon the successful completion of the create operation.

1) User Deploys a Contract: "There are two types of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as 'contract creation'). ...fields of a transaction ...Additionally, a contract creation transaction contains: $init$: An unlimited size byte array specifying the EVM-code for the account initialisation procedure. ...$init$ is executed only once at account creation and gets discarded immediately thereafter." [1]

2) Contract Deploys Another Contract: Similarly, contracts can create other contracts by sending a create message. "Finally, the account is initialised through the execution of the initialising EVM code $i[nit]$ according to the execution model. Code execution can effect several events that are not internal to the execution state: the account's storage can be altered, further accounts can be created and further message calls can be made." [1]

3) Deployment Code: is the EVM bytecode called $init$ in Ethereum's yellow paper [1]. It is used mainly to deploy a contract (by writing bytecode to the new contract's address).

C. Collateral Aspects of Deployment Code

Interestingly, deployment code need not write any code to the new contract's address. It may even contain arbitrary EVM bytecode including calls, creates, and a selfdestruct. All executed code will seem to originate from the address of the new contract, although the affected account does not contain any code yet. "During initialization code execution, `EXTCODESIZE` on the address should return zero, which is the length of the code of the account." [1] The contract code is available at its address only upon successful completion of the create operation.

Note, that the contract and its code do exist between the successful execution of the create operation and the 'delayed' destruction at the completion of the initiating transaction. Between creation and self-destruction, the contract may even receive successful calls that are executed as intended, if the message that created and selfdestructed the new contract is part of the same transaction that contains the messages/calls to the new contract.

In case of an actually executed *selfdestruct* in the deployment code, no contract will be created because it halts the execution of the deployment without returning the values necessary for a successful deployment.

Recent scenarios for these collateral aspects as used by some addresses (users as well as contracts) are shown in section V.

D. Messages Caused by Deployment Codes

Since deployment code can contain any EVM operation including create, selfdestruct, and diverse types of calls (which result in respective internal messages), we were interested in a quantification of the usage of this feature so far.

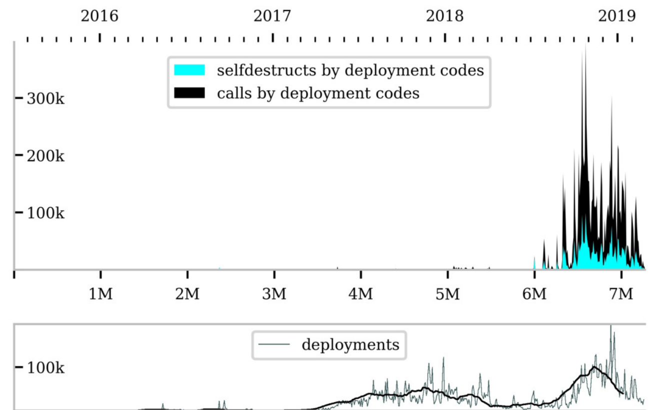

Fig. 1. Deployments and their messages. The upper plot depicts the number of messages caused by deployment codes, differentiated between selfdestructs and calls. The lower plot depicts the number of deployments in grey and a moving average over 25 values in black.

Fig. 1 effectively depicts the occurrence of collateral use of deployment code. More precisely, it depicts the number of messages caused by deployment codes in the upper plot, and the number of deployments in the lower plot. The horizontal axis shows the time line, in years and months on top and in million blocks at the bottom of the upper plot. In both plots, each value is aggregated over consecutive 10 000 blocks.

Looking at the messages that are caused by deployment codes, we find an unusual increase in the number of such messages in the second half of 2018. Fig. 1 illustrates this

recent phenomenon, which starts around block 6 000 000. This increase in messages is not paralleled by the number of deployments. In 2017, when the number of deployments increased, no such phenomenon occurred. The high number of messages caused by deployment codes thus indicates a recent usage pattern, which we analyze in the following sections.

Deployment codes caused a total of slightly over 10 M messages, of which virtually all occurred after block 6 M. The remaining 131 k messages were distributed more or less evenly throughout the first 6 M blocks.

When comparing our findings with data from Etherscan[1], we notice that Etherscan does not show all internal messages, especially when they do not transfer any Ether. This is usually the case with messages caused by deployment codes.

III. Code Analysis

Here, we present our methods to analyze deployment code.

Skeletons. To detect functional similarities between distinct bytecodes we consider their *skeletons*. The skeleton of a deployment code is obtained by replacing all 20-byte-addresses occurring literally as the argument of a `PUSH` operation by place-holders, and by removing trailing swarm hashes (metadata not affecting the functionality) as well as constructor arguments.

Contract interfaces. Most contracts, in particular those obtained from Solidity code, follow the convention that the first four bytes of the input data specify the called function. These four bytes are so-called *function signatures*, computed as the first four bytes of the hashed function headers (consisting of function name and argument types). They mark entry points in contracts.

By identifying function signatures at appropriate locations in the bytecode, it is possible to partially reconstruct the interface. Obviously, it is not possible to regain the header from the function signature, which is a hash fragment. We use a directory of about 275 000 signature-header pairs that we extracted from available Solidity source codes. This way, one often obtains translations of function signatures to possible headers that help in understanding the meaning of messages and contracts.

For all deployed contracts, we extracted their interfaces together with possible translations of the contained function signatures.

Call message analysis. We collected all calls (internal messages) that deployment code made to other contracts. Then we analyzed these messages using our partially translated contract interfaces in order to derive statistics about interaction patterns of deployment code: which deployment code calls which function of which contract.

Message viewer. For an easier tracing of subsequent messages in a transaction, we developed a simple message viewer to display all internal messages that belong to an external transaction. It includes message type, value, return value, and decoded data per message.

[1] https://etherscan.io/

Manual code inspection. Using the tool Vandal [2] [3], we disassemble EVM bytecode and display the control flow graph.

Scenario extraction. First, we grouped the deployment codes according to the skeletons they employed. Additionally, the functionality of frequently used skeletons was determined by manual inspection, by tracing its usage in the message viewer, and by looking at its call statistics. Then, highly similar usage patterns were grouped into a scenario.

IV. Vulnerabilities Exploited by Deployment Code

Even though the vulnerabilities below have been known for some time, code containg them is still being deployed. We see a marked increase in actually exploiting them starting around block 6 M which was mined in July 2018. Apparently, contract developers are still not aware or do not care.

A. Existing Exploits

isHuman: To check that the message sender is not a contract, a function asserts that `extcodesize(msg.sender) == 0`. This once was a recommendation by OpenZeppelin [4], but has been removed for the reasons explained in section II.

RNG: Reliable random number generation (RNG) in a deterministic system like consensus-based blockchains proves difficult. An excellent review of approaches to RNG is given in [5] including its vulnerabilities and incentive misalignments.

Using current block data for RNG is one of the attackable, but still used practices. Adequately crafted deployment code is one (but not the only) possibility to exploit it. The crafting includes that the relevant messages are in the same block to render the random number predictable.

Blacklist: Free tokens should only be received once by each user. With black- or whitelists one can restrict certain functions of a smart contract by excluding or including addresses respectively.

Again, adequately crafted deployment code is one (but not the only) possibility to exploit it, as multiple addresses (new contracts) can be created with little effort.

B. Structures of Vulnerabilities

Regarding the structures of the encountered vulnerabilities, we can conclude that affected contracts falsely assume that *Users*

- do not have code (on or off-chain)
- do not know how to collaterally use deployment code

or *Contracts*

- cannot access relevant block data
- cannot ‘change’ address
- have a code size greater than zero when making a call.

V. Usage Patterns

In this section, we take a closer look at unconventional usage patterns of deployment code. The intended purpose of deployment code is to initialize a new contract and write its bytecode to the storage at its address. We therefore expect that

interactions with the environment (like calls) are rare during deployment. This is indeed the prevailing pattern until the summer of 2018. After block 6 000 000, however, we see a marked increase in deployments that apparently serve a different purpose. Of 6.2 M deployments, 2.5 M (41 %) destruct themselves upon completion of the creating transaction, thus not giving rise to a new contract. Instead, the deployment code itself becomes active. Compared to regular deployment code, it is twice as fertile (0.7 % vs. 0.35 % creates per deployment) and 130 times more talkative (2.97 vs. 0.023 calls per deployment). This phenomenon is not entirely new, but the blocks since 6 M account for 98 % of such cases.

The 2.5 M self-destructing deployments were issued from just 8 697 distinct accounts, which were mostly contracts (99 %). This explains the uniformity of the deployment code: we count just 7 126 different skeletons, with the most frequent one occurring more than 800 000 times.

To understand the purpose of these short-lived contracts we applied the methods described in section III. By iterating from the most frequent skeletons down to the rarer ones we succeeded in classifying 99.99 % of the 2.5 M deployments (96 % of the 7 126 skeletons). We ended with 325 deployments (256 skeletons) remaining unclassified, as we discovered no more new patterns. The last codes we analyzed were more likely to be one-of-a-kind and often seemed like experimental versions of code we had seen before.

Our analysis resulted in an unexpectedly clear picture. The vast majority of deployments (97.5 %) involving two thirds of the skeletons (67 %) aimed at *token harvesting*. Almost the entire remainder (2.4 % of the deployments, 30 % of the skeletons) targeted the Ponzi lottery *Fomo3D* or one of its many clones. The few other deployments and skeletons we came across took advantage of weaknesses in other games.

A. Scenario Token Harvesting

In this scenario, typically several dozens of contracts are created in a row that claim free tokens, transfer them to a fixed address, and destruct themselves while still in the deployment phase. These multi-creations can be implemented just with deployment code or via a deployed contract that can be called multiple times (both usually with an immediate selfdestruct).

To understand the popularity of this coding pattern, we take a look at token contracts and airdrops. For funding a business idea in the blockchain era, you could deploy a token contract with the promise that token holders will profit once your enterprise sky-rockets. To create a community for your idea and advertise it, you distribute parts of your tokens for free. A common implementation of this so-called airdrop is to provide a function that transfers, for a certain period of time, free tokens to any address that calls it. To counter-act hoarding of tokens, addresses are blacklisted once they got their share.

This is where deployment code comes in. Token harvesters aim at collecting large quantities of free tokens. To circumvent blacklisting, each request for free tokens has to be issued from a new address. As deployment code is executed as being from the new contract, the token contract will see a new address.

Figure 2 shows the start of such a harvesting sequence in block 6 409 724. The external user 43132826 calls an existing contract at address 45884811, which creates new contracts in a loop. The first one is deployed at address 68649799. From this new address, the deployment code calls the fallback function of the token contract at address 42687408 (*NewIntelTechMedia*) and receives free tokens. The amount often diminishes with each airdrop, so the code queries the precise amount with `balanceOf`. Finally, 1285.6 tokens are transferred to the account of the external user, and the deployment code self-destructs. (Note that for reasons of efficiency and readability, 20-byte-addresses were mapped to 4-byte-integers.)

```
TRANSACTION 6409724/70
43132826
└──45884811 call('')=''
   ├──68649799 CREATE(174439375)=''
   │  ├──42687408 call('')=''
   │  ├──42687408 call(balanceOf(68649799))='1285.6'
   │  ├──42687408 call(transfer(43132826,'1285.6'))='1'
   │  └──46429544 SELFDESTRUCT
   ├──68649800 CREATE(174439375)=''
   │  ├──42687408 call('')=''

ACCOUNTS
  42687408: 0xe30a76ec9168639f09061e602924ae601d341066
  43132826: 0xcb44e6ac67fca11e46a0e4d0758455a06846fb5b
  45884811: 0x955074147610509817a092db3a7e353478cd1bf1
  46429544: 0x808284ad89f21c512f9a5d43ad24aaf79780849a
  68649799: 0x837f4c693c134014811524b0f96fd1d1ee41910b

DATA
 174439375: 0x6080604052348015610010576000080fd5b50 ...
```

Fig. 2. Token Harvester. In total, 40 selfdestructing contracts are created during this transaction, targeting the same token.

This practice has even become a business model. Replicators of token harvesters have been deployed that can be called by anyone. The replicator will generate as many harvesters as the gas supplied with the call permits. All but the last harvester collect tokens for the caller, while the last one transfers the tokens to the owner of the replicator.

Depending on the variation of the token contract, functions like `getTokens`, `Mine`, or `register` need to be called in place of the fallback function, or the airdrop functionality is integrated into `balanceOf` or `transfer`. The harvesters also differ regarding where the selfdestruct occurs. Until March 2019, 2.5 M harvesters collected 178 different token types for 8 434 beneficiaries.

B. Scenario Fomo3D and its Clones

The game suite of Fomo3D combines a Ponzi scheme with a lottery. Many players use bots in their attempt to win. The first round was won by someone outsmarting the bots with contracts coordinated by hand. For a realistic chance to win one has to be a skilled (and playful) developer.

Fomo3D uses information of the current block (like block number and timestamp) and the sender's address to determine whether the sender wins an airdrop. If the sender is a contract it has access to the same information and will only play if it is sure to win. Therefore, Fomo3D checks whether the address *isHuman* (see section IV).

Again, this can be circumvented by deployment code, which has access to the block information and knows its own address. The addresses of new contracts are computed as the hash of the creator's address and a nonce that initially equals one. Thus, the deployment code can check whether a winning address is in reach. If so, it deploys several self-destructing contracts to increment the nonce. Then a final contract is deployed that now resides under a winning address. It calls Fomo3D from its deployment code, so that it passes *isHuman*.

Against this background, it is of little surprise that we find 17 775 deployments of the smallest selfdestructing contract with bytecode `0x33ff`. In total, we count 62 000 selfdestructing contracts that interact with Fomo3D or one of its 50 clones.

C. Scenario Poker

Figure 3 shows one of the few scenarios unrelated to tokens or Fomo3D. An external account creates a contract with an initial endowment of 0.2 Ether at address 62660975, whose deployment code plays two rounds of a betting game called *Poker* at address 62357440. In each round, the code queries the minimal and maximal value for a bet, calls `play` while going all-in, and miraculously doubles the bet each time. The deployment code then selfdestructs and returns almost 0.8 Ether to the user.

```
TRANSACTION 6505536/42
61392828
└──62660975 CREATE(213123879)='' [Ξ0.2]
    ├──62357440 call(maxBetVal())='1.0'
    ├──62357440 call(minBetVal())='0.01'
    ├──62357440 call(play(1,[1,2]))='37' [Ξ0.2]
    │   └──62660975 call('')='' [Ξ0.394]
    ├──62357440 call(maxBetVal())='1.0'
    ├──62357440 call(minBetVal())='0.01'
    ├──62357440 call(play(1,[1,1]))='12' [Ξ0.394]
    │   └──62660975 call('')='' [Ξ0.77618]
    └──61392828 SELFDESTRUCT [Ξ0.77618]

ACCOUNTS
  61392828: 0x839f04e6e570d52bf86943a86f0b8d9cc696098f
  62357440: 0x23b8e1a594b18c450852454d72a5cd20e1ba63ad
  62660975: 0x3a7ee583ac6235bdd7569e785a2ff97c6a9eeb70

DATA
 213123879: 0x60806040526040516020806106a83398101 ...
```

Fig. 3. Selfdestructing contract betting two rounds.

The explanation for this lucky streak sounds familiar. The *Poker* contract relies on block number, player address and similar values to determine a winning player. It protects itself by requiring *isHuman*. Again, the deployment code has access to all relevant numbers and circumvents *isHuman*.

VI. Related Work

The examinations of Ponzi schemes deployed on Ethereum in [6] and [7] are related to some extent. Many academic analyses of smart contracts work at bytecode level, more precisely they analyze the deployed code, but disregard deployment code. Forums like reddit.com, medium.com, or hackernoon.com report and discuss security issues, including a few involving deployment code. However, the massive use of deployment code is a recent phenomenon that to our knowledge has not been investigated before.

Ethernaut [8] challenges are built around known vulnerabilities and their exploits. The challenge 'Coin Flip' involves a vulnerable random number generator relying on block data, while 'Gatekeeper Two' addresses the usage of `EXTCODESIZE` during contract creation.

Regarding reverse engineering and code analysis we found [9], [10] particularly inspiring.

VII. Discussion and Conclusions

We discussed selfdestructing deployment code that started to appear in large numbers after block 6 000 000, and by the end of 2018 it accounted for half of the contract creations. We gave a quantitative and qualitative assessment of this new phenomenon. Its prime application area is token harvesting, followed by advantage gains in gambling and games. Its purpose is to bypass the intended interaction patterns of certain contracts. This is made possible by the unsuspecting design of these contracts on the one hand, and the barely limited expressiveness of contract deployment on the other hand.

The design of the EVM admits arbitrary bytecode during contract deployment. This guarantees maximum flexibility, while it introduces further complexity that developers may miss to consider. Contract development and deployment is less error prone when it is routine, unsurprising, and transparent.

So far, the extra flexibility was mainly used for exploits, for these less obvious features of deployment code are not yet widely known. One remedy is to spread the news.

Future work may be the integration of the presented analysis into existing tools, and generating a warning when deployment code contains a *selfdestruct* or *create* operation since most of its usage so far was malicious.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf, accessed 2019-02-02.

[2] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv:1809.03981*, 2018.

[3] Smart Contract Research at USYD, "Vandal," https://github.com/usyd-blockchain/vandal, accessed 2019-02-02.

[4] OpenZeppelin, "Library for secure smart contract development," https://github.com/OpenZeppelin/openzeppelin-solidity, accessed 2019-02-02.

[5] K. Chatterjee, A. Goharshady, and A. Pourdamghani, "Probabilistic smart contracts: Secure randomness on the blockchain," in *EEE International Conference on Blockchain andCryptocurrency (ICBC)*, 2019.

[6] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting ponzi schemes on ethereum: identification, analysis, and impact," *arXiv:1703.03779*, 2017.

[7] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology," in *Proceedings of the 2018 World Wide Web Conference (WWW '18)*, 2018.

[8] OpenZeppelin, "Ethernaut – Solidity security challenges," https://github.com/OpenZeppelin/ethernaut, accessed 2019-02-02.

[9] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, vol. 1, 2018.

[10] B. Mueller, "Smashing smart contracts," in *9th HITB Security Conference*, 2018.